



C2 VHDL描述的结构

Yann Douze

VHDL



VHDL描述的结构

- VHDL描述由两个不可分割的部分组成，即：
 - 实体，它定义输入和输出。
 - 结构体，它包含允许执行预期操作的VHDL指令。
- 参见课程资料。



库声明

- 任何用于综合的VHDL描述都需要库。
- IEEE对它们进行了标准化，特别是IEEE 1164库。
- 它们包含信号类型的定义，元件、函数和子程序允许执行的算术和逻辑操作等。

```
Library ieee;  
Use ieee.std_logic_1164.all;  
Use ieee.numeric_std.all;
```



实体声明

- 语法:

```
entity NOM_DE_L_ENTITE is  
    port ( Description des signaux d'entrées /sorties ...);  
end entity;
```

- Exemple :

```
entity SEQUENCEMENT is  
    port (  
        CLOCK : in std_logic;  
        RESET : in std_logic;  
        Q : out std_logic_vector(1 downto 0),  
    );  
end entity;
```

注意: 在PORT语句的最后一个信号定义之后, 不要使用**分号**。

I/O信号声明

- 端口语句:

Syntax: *NOM_DU_SIGNAL* : *sens* *type*;

Exemple: *CLOCK*: *in* *std_logic*;

BUS : *out* *std_logic_vector* (7 *downto* 0);

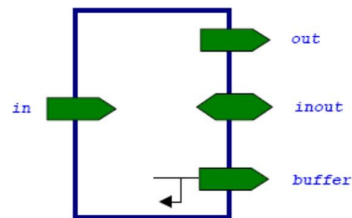
- 端口的方向:

in : 用于输入信号。

out : 用于输出信号。

inout : 用于输入输出信号

buffer : 用于输出但可读取的信号（不推荐）。





类型

输入/输出信号类型为:

- 单个信号用 ***std_logic***。
- 由多个信号组成的总线用 ***std_logic_vector***。

例如, 5位双向总线将表示如下:

LATCH: ***inout std_logic_vector (4 downto 0)***;

其中 ***LATCH(4)*** 对应于 **MSB**, ***LATCH(0)*** 对应于 **LSB**。

std_logic 类型的信号可以取的值为:

- “0”或 “L”: 表示低电平。
- “1”或 “H”: 表示高水平。
- “X”或 “W”: 未知级别。
- ‘U’: 表示未初始化。
- “Z”: 高阻抗状态。
- “-”: 任意, 即任意值。



结构体说明

- 结构体是相对于一个实体的。
- 它描述了设计的主体，它的行为，并通过指令建立了输入和输出之间的关系。
- 示例：

```
-- Opérateurs logiques de base
entity PORTES is
    port (A,B :in std_logic;
          Y1,Y2,Y3,Y4,Y5,Y6,Y7:out std_logic);
end entity;
architecture DESCRIPTION of PORTES is
begin
    Y1 <= A and B;
    Y2 <= A or B;
    Y3 <= A xor B;
    Y4 <= not A;
    Y5 <= A nand B;
    Y6 <= A nor B;
    Y7 <= not(A xor B);
end architecture;
```



VHDL: 并行语言?

```
architecture DESCRIPTION of DECOD is  
begin
```

```
-- instructions concurrentes
```

```
D0 <= (not(IN1) and not(IN0));      -- première instruction
```

```
D1 <= (not(IN1) and IN0);           -- deuxième instruction
```

```
end architecture;
```

- 在结构体的开始和结束之间，处于一个相互并行的指令结构中。
- 并行指令：
 - 编写指令的顺序并不重要。。
 - 所有指令都被评估，并同时影响输出信号的时序。
 - 这是与计算机语言的主要区别。

下面的体系结构是等效的：

```
architecture DESCRIPTION of DECOD is  
begin
```

```
D1 <= (not(IN1) and IN0);           -- deuxième instruction
```

```
D0 <= (not(IN1) AND not(IN0));      -- première instruction
```

```
end architecture;
```


VHDL描述示例:

```
library ieee;  
Use ieee.std_logic_1164.all;  
Use ieee.numeric_std.all;
```

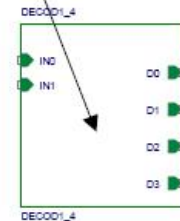
Déclaration des bibliothèques

Commentaires, en VHDL ils commencent par --

```
-- décodeur  
-- Un parmi quatre
```

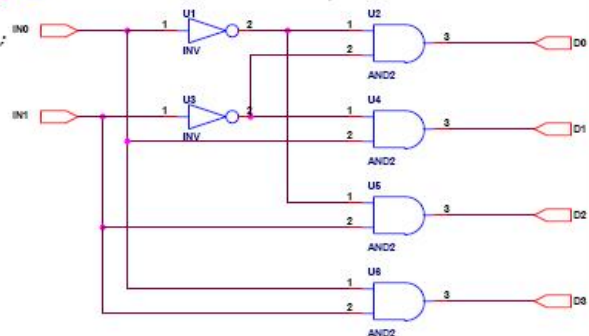
Déclaration de l'entité du décodeur
Correspondance schématique

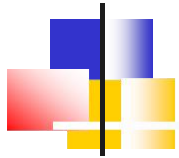
```
entity DECOD1_4 is  
    port(IN0, IN1: in std_logic;  
          D0, D1, D2, D3: out std_logic);  
end DECOD1_4;
```



Déclaration de l'architecture du décodeur
Correspondance schématique

```
architecture DESCRIPTION of DECOD1_4 is  
begin  
    D0 <= (not(IN1) and not(IN0));  
    D1 <= (not(IN1) and IN0);  
    D2 <= (IN1 and not(IN0));  
    D3 <= (IN1 and IN0);  
end DESCRIPTION;
```





C3-基本运算符

Yann Douze

VHDL



简单赋值:<=

Examples :

S <= *E2* ;

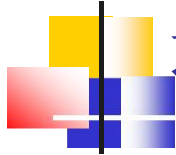
S <= '0' ;

S <= '1' ;

对于多位信号，使用省略号“...”，

BINARY , 例如: *BUS* <= "1001" ; - *BUS* = 9 十进制

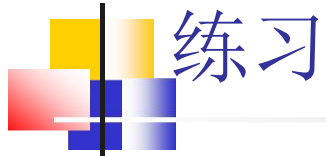
HEXA , 例如: *BUS* <= x"9" ; - *BUS* = 9 in 十六进制



逻辑运算符

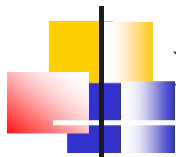
非	→	not
与	→	and
与非	→	nand
或	→	or
或非	→	nor
异或	→	xor

Exemple : **S1** <= (**E1** and **E2**) or (**E3** nand **E4**);



练习

做练习1和2。



关系运算符

- 它们允许根据测试结果或条件改变信号的状态。

等于

€ =

不等于

€ /=

小于

€ <

小于等于

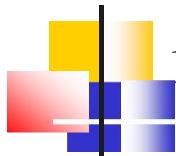
€ <=

大于

€ >

大于等于

€ >=

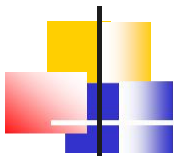


条件赋值语句

- 根据一个或多个信号、值、常量之间的逻辑条件的结果改变信号的赋值。

```
SIGNAL <= expression when condition  
[else expression when condition]  
[else expression];
```

注意：指令 [**else** expression] 允许定义默认情况下为 **SIGNAL**。



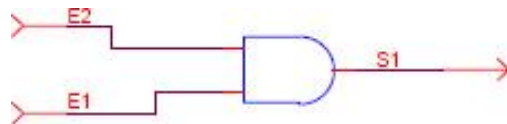
条件赋值（2）

例1:

--当 (E1= '1') 时, S1 取 E2 的值, 否则 S1 取值 '0' ,

S1 <= **E2** when (**E1**= '1') else '0';

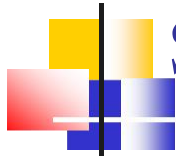
对应电路:与逻辑



例2:

--2选1数据选件器的行为描述

Y <= **A** when (**SEL**= '0') else
B when (**SEL**= '1') else '0';

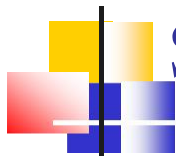


Selection赋值语句

该指令允许根据被选择信号的取值，为信号分配不同的取值。

```
with SIGNAL_DE_SELECTION select  
  SIGNAL <= expression when valeur_de_selection,  
    [expression when valeur_de_selection,]  
    [expression when others];
```

注意:[Expression when others]语句不是必需的，但强烈建议使用，它允许设置为默认值。



Selection赋值语句 (2)

Exemple : *Multiplexeur 2 vers 1*

with *SEL select*

Y <= *A* when '0',

B when '1',

'0' when others;

注意:在多路复用器的情况下, *when others*是必须的, 因为对于所选择的信号其定义包含了其他的取值情况, 所以应考虑其所有可能的值。

with *SEL select*

Y <= *A* when '0',

B when '1',

'-' when others;

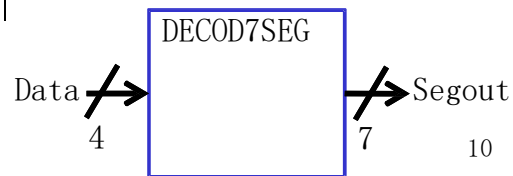
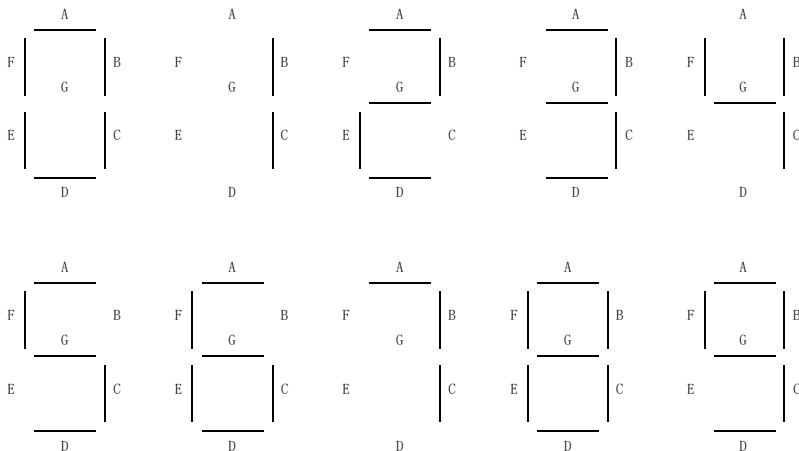
—对于 *SEL* 的其他情况, 它将取任何值

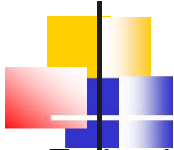
—优化综合



示例: 7段解码器 (1)

7段译码器-仅BCD (0..9)





示例:7段译码器 (2)

Entity decod7Seg is

```
    port ( Data : in std_logic_vector(3 downto 0); -- Expected within 0 ..9
          Segout : out std_logic_vector(1 to 7) ); -- Segments A, B, C, D, E, F,
    G end entity;
```

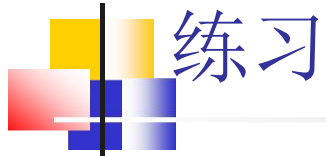
Architecture behavior of decod7seg is

begin

 with Data select

```
        Segout <= "1111110" when x"0",
                  "0110000" when x"1",
                  "1101101" when x"2",
                  .....
                  "1111011" when x"9",
                  "-----" when others;
```

End architecture;



-
- 练习3 使用选择性或条件赋值。

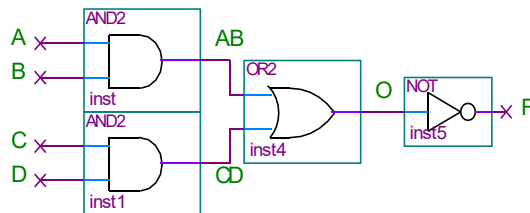
内部信号

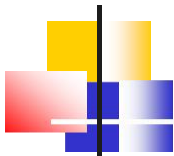
- Syntax : **signal** **NOM_DU_SIGNAL** : **type**;
- Exemple : **signal** **I** : **std_logic**;
- signal** **BUS** : **std_logic_vector** (**7 downto 0**);

下面的方案可以用两种不同的方式来描述：

■无内部信号

```
architecture V1 of AOI is
Begin
    F <= not ((A and B) or (C and D));
end architecture V1;
```

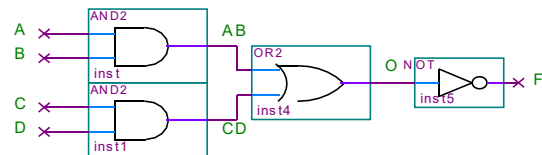




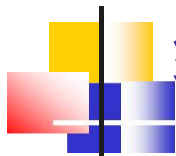
内部信号（2），

- 具有内部信号

```
architecture V2 of AOI is
    --zone de declaration des signaux
    signal AB,CD,O: STD_LOGIC;
begin
    --instructions concurrentes
    AB <= A and B;
    CD <= C and D;
    O <= AB or CD;
    F <= not O;
end V2;
```

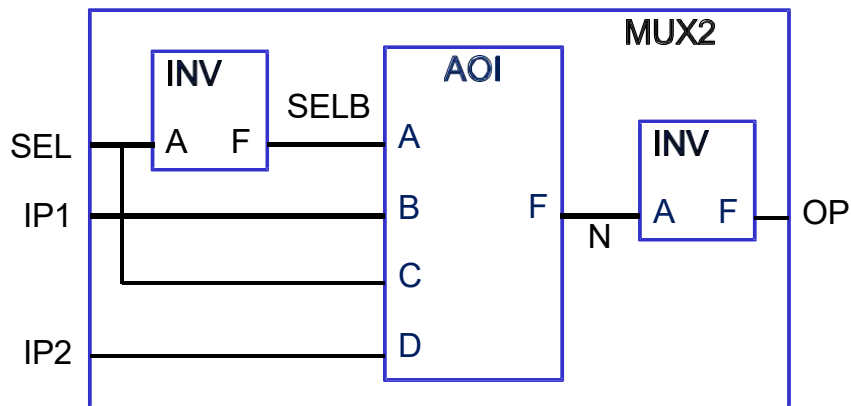


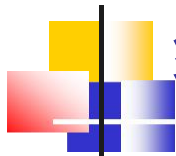
- 并发指令：
 - 写指示的顺序并不重要。
 - 所有指令都被评估并同时影响输出信号。
 - 与计算机语言的主要区别。



结构化描述

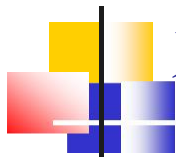
- 这是层次化连接的描述类型（NetList）
- 如果描述包含一个或多个元件，则该描述是结构性的。
- 示例：



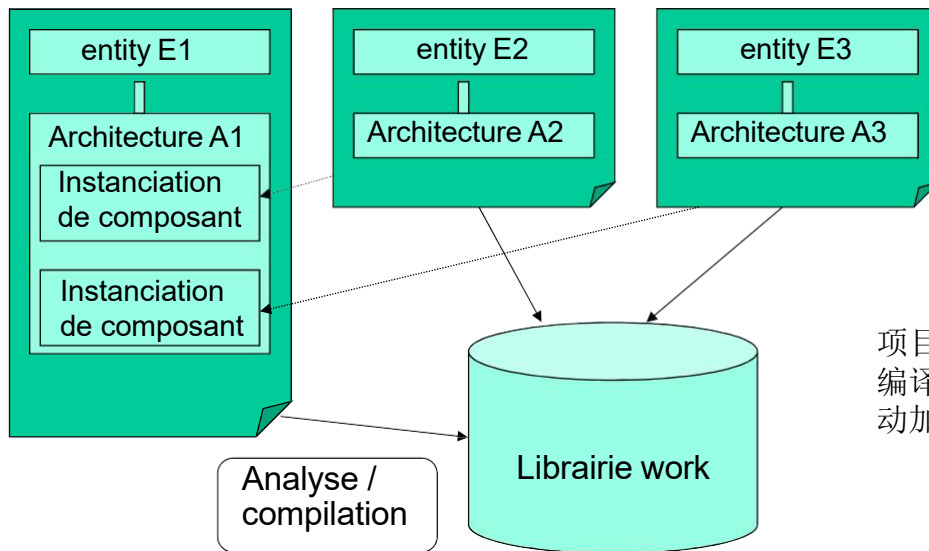


结构化描述

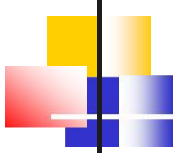
- 应遵循的步骤：
 - 绘制要实例化的元件图。
 - 声明必要的内部信号列表： **SIGNAL...**
 - 实例化每个元件并指定其连接列表： **PORT MAP...**



元件例化



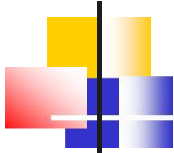
项目设计的元件经过编译综合后，会被自动加入到work库中



INV和AOI元件的声明

```
entity INV is
    port ( A   : in  STD_LOGIC;
           F   : out STD_LOGIC);
end entity;
architecture V1 of INV is
    ...
end architecture;

entity AOI is
    port ( A,B,C,D   : in  STD_LOGIC;
           F           : out STD_LOGIC);
end entity;
architecture V1 of AOI is
    ...
end architecture;
```



直接实例化

```
entity MUX2 is
    port ( SEL, IP1, IP2    : in  STD_LOGIC;
           op               : out STD_LOGIC);
end entity;
architecture DIRECTE of MUX2 is
    signal SELB, N: STD_LOGIC;
begin
    G1: entity WORK.INV(V1) port map (A => SEL, F => SELB);
    G2: entity WORK.AOI(V1) port map (
        A => SELB, B => IP1,
        C => SEL,  D => IP2,
        F => N);
    G3: entity WORK.INV(V1) port map (A => N, F => OP);
end architecture;
```



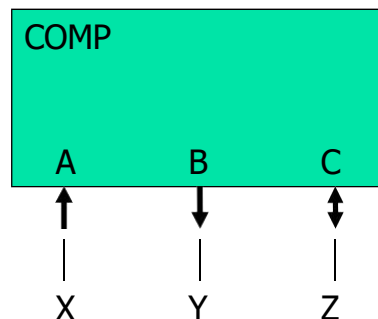
通过名称或位置进行端口的关联

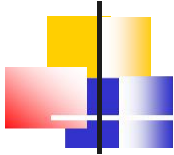
```
entity COMP
port(  A: in      STD_LOGIC;
      B: out     STD_LOGIC;
      C: inout   STD_LOGIC);
end entity;
```

```
Architecture V1 of COMP is
Signal X,Y,Z: STD_LOGIC;
begin
```

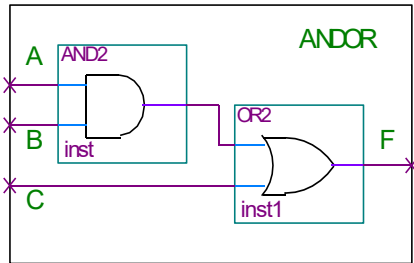
```
C1: entity work.COMP port map (A => X, B => Y, C => Z);
--association par nom
```

```
C1: entity work.COMP port map (X, Y, Z); --association par position
```





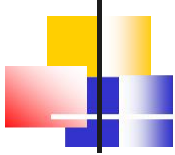
要完成的练习



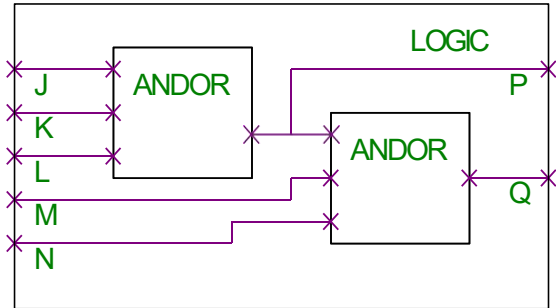
```
library IEEE;  
use IEEE.STD_LOGIC_1164.all;
```

```
entity ANDOR is  
    port (  
        A,B,C : in std_logic;  
        F : out std_logic);  
end entity;
```

```
Architecture dataflow of ANDOR is  
Begin  
    F <= (A and B) or C;  
End architecture;
```

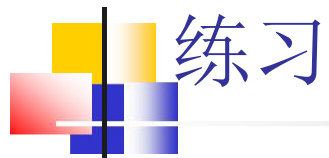


要完成的练习

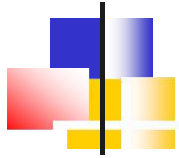


```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
```

```
entity LOGIC is
    port (
        J,K,L,M,N : in std_logic;
        P,Q : out std_logic);
end entity LOGIC;
Architecture STRUCT of LOGIC is
```

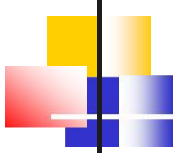


-
- 做练习4和5。

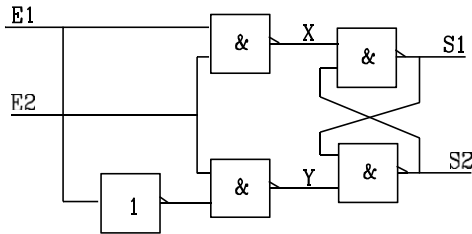


C4-顺序指令

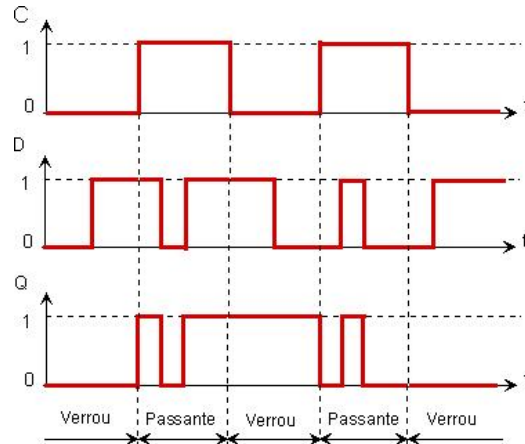
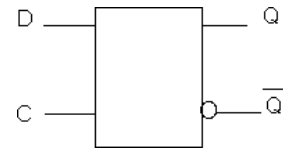
Yann Douze
VHDL

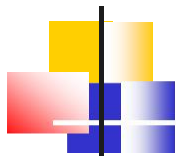


复习：D锁存器-电平触发

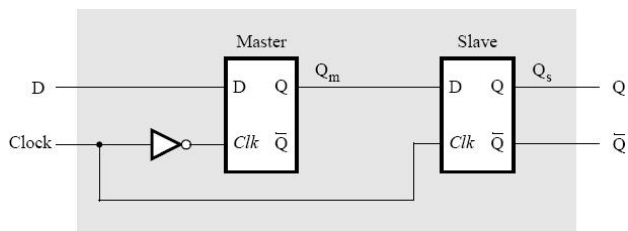


Entrées		Sorties	
C	D	Q_{n+1}	\overline{Q}_{n+1}
0	X	Q_n	\overline{Q}_n
1	0	0	1
1	1	1	0

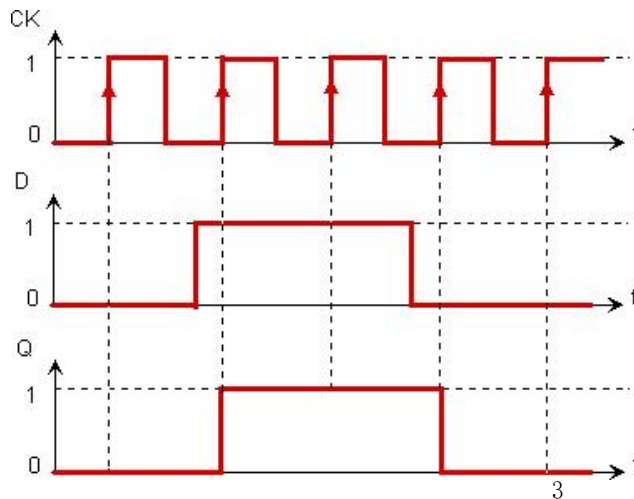
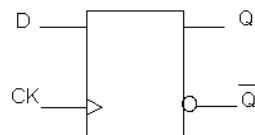




复习：D触发器-边沿出发



Entrées		Sorties	
CK	D	Q_{n+1}	\overline{Q}_{n+1}
0	X	Q_n	\overline{Q}_n
1	X	Q_n	\overline{Q}_n
↓	X	Q_n	\overline{Q}_n
↑	0	0	1
↑	1	1	0



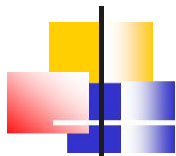


process 定义

- **process** 是电路描述的一部分，其中指令依次执行：一个接一个。
- 它允许使用结构化编程的标准指令对信号进行操作，就像在微处理器系统中一样。

语法：

```
[Nom_du_process :]process(Liste_de_sensibilité_nom_des_signaux)  
Begin  
    -- instructions du process  
end process [Nom_du_process] ;
```



Process运行规则

- 当敏感列表中信号的状态发生变化时，就会执行一个进程。
- 进程指令按顺序执行。
- 进程指令对信号状态的改变，在进程结束时才被执行。

在进程中，可以允许同一信号有 多个驱动源（赋值源），即在同 一进程中存在多个同名的信号被赋值，其结果只有最后的赋值语句被启动，并进行赋值操作。



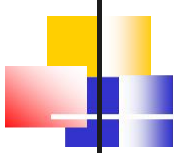
进程的作用

- 进程有三种不同的方法：
 1. 用结构化编程的指令描述组合电路: `if`, `case`等。
 2. 用于描述具有一个或多个存储单元（DFF触发器）的同步电路。
 3. 描述不可综合的功能，如 `testbench` 或建模。



顺序指令

- 注意！
- if和case语句只存在，并且只能存在于进程中。



IF指令

语法:

```
if condition then instructions  
[elsif condition then instructions]  
[else instructions]  
end if ;
```

Exemple:

```
Process (A,B,E1,E2,E3)  
Begin  
  if A='1' then SORTIE <= E1;  
  Elsif B = '1' then SORTIE <= E2;  
  Else SORTIE <= E3;  
  end if ;  
end process ;
```

注意:if语句只能在进程中使用



case 指令

语法:

CASE expression IS

WHEN choix => instructions;

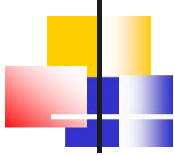
[WHEN choix => instructions;]

[WHEN OTHERS => instructions;]

END CASE;

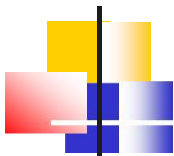
除非所列条件的取值能完整覆盖case语句所有取值，否则other不应省略。
综合器会插入不必要的锁存器。

注意: 与if语句一样，CASE语句只能在进程中使用。



示例

```
process (TEST,A,B)
begin
    case TEST is
        when "00" => F <= A and B;
        when "01" => F <= A or B;
        when "10" => F <= A xor B;
        when "11" => F <= A nand B;
        when others => F <= '0';
    end case;
end process;
```



组合进程

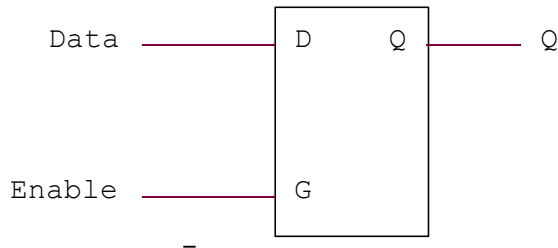
```
process (SEL, A, B, C)
begin
    if SEL = '1' then
        OP <= A and B;
    else
        OP <= C;
    end if;
end process;
```

- 敏感信号列表必须是完整的：进程中读取的所有信号都必须出现在敏感信号列表中。
- 在任何情况下都应输出赋值，以避免综合出D锁存器。



不完全条件赋值

```
process (Enable, Data)
begin
    if Enable = '1' then
        Q <= Data;
    end if;
end process;
```



- 不完全条件赋值生成的D锁存器（透明锁存器），这并不是所期待的设计。
- 有些FPGA没有D锁存器，因此会在组合逻辑上创建异步循环（这是要避免的！）



默认赋值**

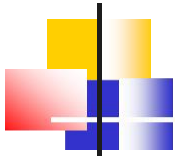
- 为了避免锁定透明片，建议使用默认赋值。

```
Process (SELA, SELB, A, B)
begin
  OP <= '0';
  if SELA = '1' then
    OP <= A;
  end if;
  if SELB = '1' then
    OP <= B;
  end if;
end process;
```

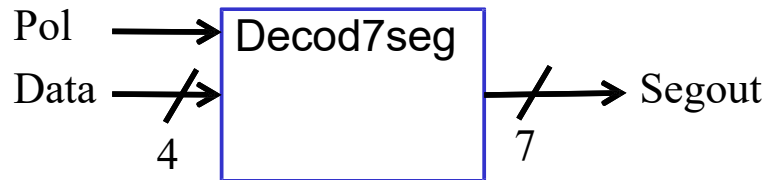
Assignement par défaut

Remplace l'événement par défaut

Remplace l'événement à nouveau

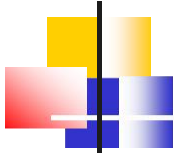


示例:7段译码器

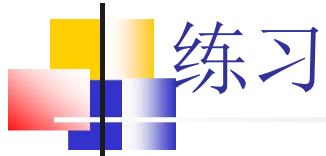


```
Entity SEVEN_SEG is
port(
  Data : in  std_logic_vector(3 downto 0); --Expected within 0...9
  Pol   : in  std_logic;                  -- '0' if active LOW
  Segout: out std_logic_vector(1 to 7)); --Segments A,B,C,D,E,F,G
end entity;
```

7段译码器组合架构

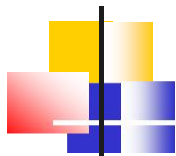


```
architecture COMB of SEVEN_SEG is
    signal sevseg : std_logic_vector(1 to 7);
begin
    Process(Data, Pol, sevseg)
    Begin
        case(Data) is
            when x"0"    => sevseg <= "1111110";
            when x"1"    => sevseg <= "0110000";
            when x"2"    => sevseg <= "1101101";
            when x"3"    => sevseg <= "1111001";
            when x"4"    => sevseg <= "0110011";
            when x"5"    => sevseg <= "1011011";
            when x"6"    => sevseg <= "1011111";
            when x"7"    => sevseg <= "1110000";
            when x"8"    => sevseg <= "1111111";
            when x"9"    => sevseg <= "1111011";
            when others => sevseg <= (others => '-');
        end case;
        if (Pol='1') then Segout
            <= sevseg;
        else
            Segout <= not(sevseg);
        end if;
    End process;
End architecture;
```

练习

- 练习1:8选1数据选择器



同步进程

```
process (CLK)
begin
  if RISING_EDGE (CLK) then
    Q1 <= D;
  end if;
end process;
```

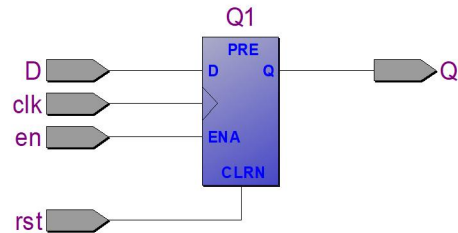
```
process (CLK)
begin
  if FALLING_EDGE (CLK) then
    Q2 <= D;
  end if;
end process;
```

- 同步进程是在时钟的每个上升沿执行。
- 要测试时钟的上升沿，请使用std_logic_1164包中定义的rising_edge()函数。
- 当时钟从状态‘0’变为状态‘1’时，RISING_EDGE为true。
- 用于描述D触发器(DFF)的翻转。

如何添加异步重置？

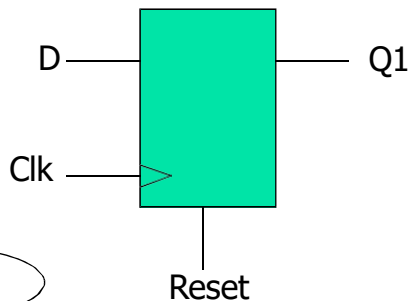


异步复位

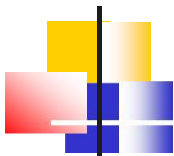


```
process (Reset, Clk)
begin
  if Reset = '1' then
    Q1 <= '0';
  elsif RISING_EDGE(Clk) then
    Q1 <= D;
  end if;
end process;
```

•在时钟上升沿之前，检测异步复位。



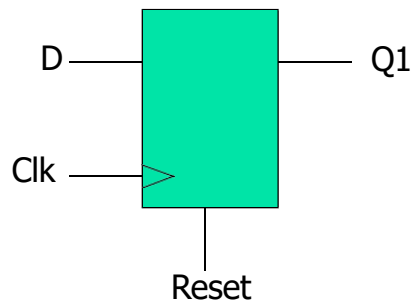
如何进行同步重置？

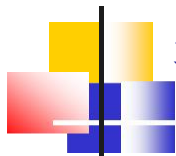


同步复位

```
process(Clk)
begin
  if RISING_EDGE(Clk) then
    if Reset = '1' then
      Q1 <= '0';
    else
      Q1 <= D;
    end if;
  end if;
end process;
```

- 在时钟上升沿之后，检测同步复位信号。





寄存器传输级 (RTL)

同步数字电路的
抽象模型

- 在同步进程中，每个赋值，都创建一个触发器。

描述信号在硬件寄存器、存储器、组合逻辑与总线等逻辑单元之间流动。

```
process (Clock)
```

```
begin
```

```
  if Rising_edge(Clock) then
```

```
    P <= A nand B;
```

```
    Q <= P;
```

```
    R <= not Q;
```

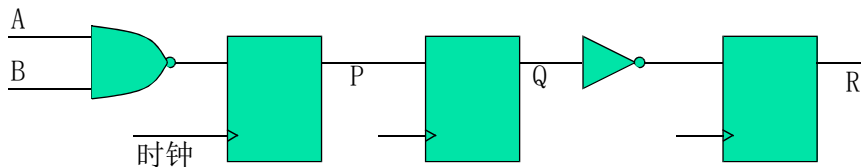
```
  end if;
```

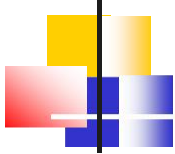
```
end process;
```

Clock

Registers

Logique combinatoire

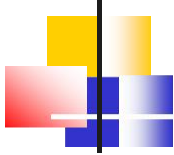




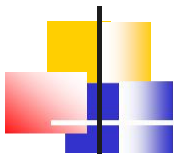
同步进程的规则

- 敏感信号列表应该包括时钟和复位信号，而不是其他的。
- 在进程中赋值的所有信号都必须通过复位信号来初始化。
- 要描述同步进程，请遵循以下结构：

```
process(clk,  
rst) begin  
  if rst = '1' then  
    -- valeur initiale  
  elsif rising_edge(clk) then  
    --  
    instruction  
  end if;  
end process;
```



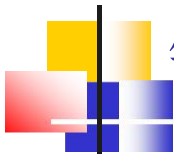
- 注意，信号只有在进程结束时才被赋值。
- 完成4级移位寄存器的代码，并绘制时序图？



移位寄存器

```
-- Registre à décalage
entity registre_decalage is
port( Qin, rst, clk : in std_logic;
      Qout           : out std_logic);
end entity;
architecture RTL of registre_decalage is
    signal Q1,Q2,Q3 : std_logic;
begin
    process(Rst,Clk)
    begin
        if Rst = '1' then
            Qout <= '0'; Q1<='0';Q2<='0';Q3<='0';
        Elsif rising_edge(clk) then
            Qout <= Q3;
            Q3 <= Q2;
            Q2 <= Q1;
            Q1 <= Qin;
        End if;
    End process;
End architecture;
```

产生4个DFF。



错误的进程形式（1）？

```
process(Clock,Reset)
Begin
    if Rising_edge(Reset) then
        ...
    elsif Rising_edge(Clock) then
        ...
    end if;
end process;
```

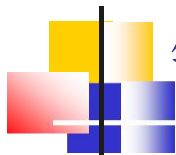
一个进程中引入了
两个边沿检测语句。

```
process(Clock,Reset)
begin
    if Reset = '1' then
        -- Actions asynchrone
    elsif Rising_edge(Clock) then
        -- Actions synchrone
    end if;
end process;
```

```
process(Clock,Reset,Ena)
begin
    if Reset = '1' then
        -- Actions asynchrone
    elsif Rising_edge(Clock) and Ena = '1' then
        -- Actions synchrone
    end if;
end process;
```

将用于产生寄存器的信号或变量的赋值语句放在了一个ELSE条件分支上。相当于检测，如果没有时钟信号时，则赋新值。显然不可能有这样的硬件电路与之对应。

```
process(All_Inputs)
begin
    -- Logique purement combinatoire
end process;
```



错误的进程形式（2）？

如果一个变量已在IF的边沿检测语句结构中作了赋值操作，就不能在同一进程中再作读操作。

一种错误是将边沿表达式当成了操作数。

IF NOT(clock'EVENT AND clock = '1') THEN ...

```
process (Clock, Reset)
begin
    if Reset = '1' then
        -- Actions asynchrone
    elsif Rising_edge(Clock) then
        -- Actions synchrone
    end if;
    -- d'autres actions
end process;
```

```
process (Clock)
begin
    if Rising_edge(Clock) then
        -- Actions synchrone
    end if;
end process;
```

```
process (Clock, Reset)
begin
    if Reset = '0' then
        if Rising_edge(Clock) then
            -- Actions synchrone
        end if;
    else
        -- Actions asynchrone
    end if;
end process;
```

锁存器的引入是为了能使某种状态必须保持到下一次时钟沿到来。在时序中由于变量具有局部特性，所有在锁存器中的变量都要被初始化。否则，锁存器的输出量不可能引出锁存器。不

在边沿检测语句之外，要特别注意不要对已赋值的信号做读操作。

错误将输出赋值信号放在了进程内部，将导致综合后的电路中多了两个不必要的寄存器。

PROCESS (CLK)

BEGIN

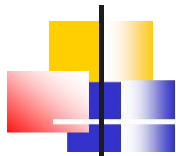
IF (CLK'EVENT AND CLK='1') THEN B <= C;

A <= B; H <= I;

I <= J XOR K;

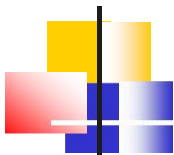
END IF;

END PROCESS ;



练习

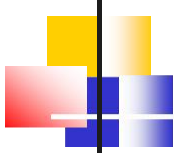
练习2： 串行OR或线性反馈移位寄存器LFSR



不可综合的进程

- 不可综合的进程用于：
 - 描述模型
 - 编写testbench
- 不可综合的进程没有敏感列表，这些是WAIT指令，可同步进程。

- 可以组合三种形式的**WAIT**
 - **WAIT ON** 事件； 示例： `wait on A,B,C,D`
 - 替换常见进程的敏感信号列表
 - **WAIT FOR** 时间； 示例： `wait for 100 ns`
 - 产生时间延迟
 - **WAIT UNTIL** 条件； 示例： `wait until rst = '1'`
 - 阻塞条件
 - **WAIT**;
 - 无限循环，类似C语言中的while (1)

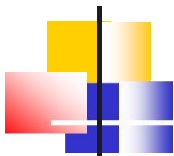


不可综合进程示例

```
-- Génération d'un reset
STIMULUS: process
begin
    reset <= '1';
    wait for 50 NS;
    reset <= '0';
    wait;
end process STIMULUS;
```

```
-- Génération d'une horloge
ClockGenerator: process
begin
    Clock <= '0';
    wait for 5 NS;
    Clock <= '1';
    wait for 5 NS;
end process;
```

两个进程同时使用的。



循环指令

```
for PARAMETER in LOOP_RANGE loop
    ...
end loop;
```

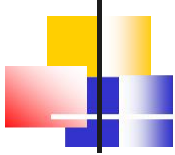
```
While CONDITION loop
    . 在进程内部。
    while 需要定义变量。
end loop;
```

```
boucle1: -- étiquette optionnelle
FOR i IN 0 TO 10 LOOP
    -- calcul des puissances de 2
    b := 2**i;
    WAIT FOR 10 ns; --toutes les 10 ns
END LOOP;
```

```
I := 0;
WHILE b < 1025 LOOP
    b := 2**i;
    i := i+1;
    WAIT FOR 10 ns;
END LOOP;
```

FOR后的循环变量是一个临时变量，属**LOOP**语句的局部变量，不必事先定义。这个变量只能作为赋值源，不能被赋值。它由**LOOP**语句自动定义，使用时应当注意在**LOOP**语句范围内不要再使用其它与此循环变量同名的标识符。

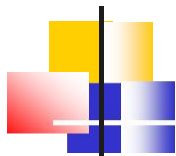
循环次数范围规定**LOOP**语句中的顺序语句被执行的次数。循环变量从循环次数范围的初值开始，每执行完一次顺序语句后递增1，直至达到循环次数范围指定的最大值。



循环示例

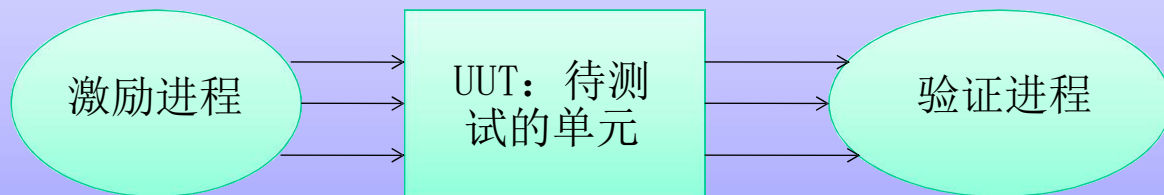
```
ClockGenerator: process  
begin  
    while not Stop loop  
        Clock <= '0';  
        wait for 5 NS;  
        Clock <= '1';  
        wait for 5 NS;  
    end loop;  
    wait;  
end process;
```

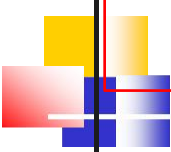
这里**stop**可能是自己定义的，



testbench

试验台

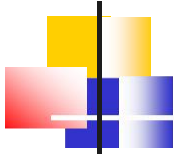




验证=断言

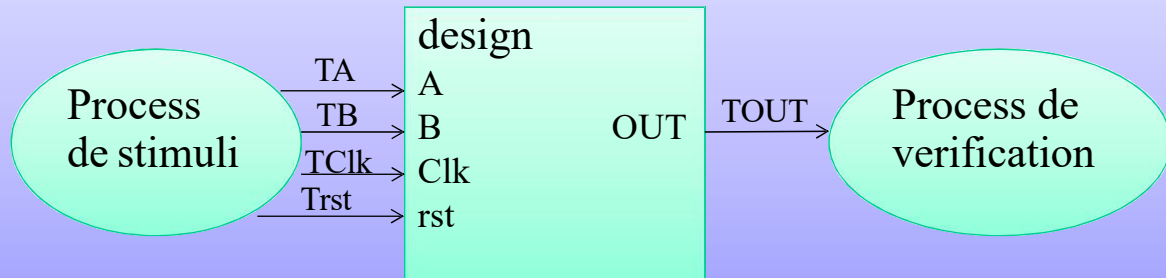
- `assert`: 允许在屏幕上写入消息。
- 语法:
 - `ASSERT condition REPORT message <SECURITY ...>`
- 强制消息:
 - `ASSERT FALSE REPORT "Toujours à l'écran " SEVERITY note;`
 - `REPORT "Toujours à l'écran " SEVERITY note;`
- 顺序内容测试:
 - `ASSERT s = '1' REPORT "la sortie vaut '0' et ce n'est pas normal " SEVERITY warning;`
--提出一个警告
 - `ASSERT s = '1' REPORT "la sortie vaut '0' et ce n'est pas normal " SEVERITY failure;`
--报告错误并停止执行

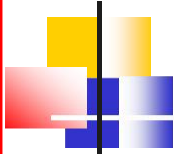
FAILURE 用在发生了致命错误，仿真过程必须立即停止的情况。



示例

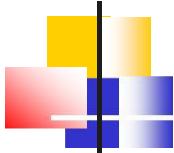
design_tb





testbench (1)

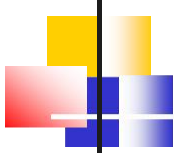
```
Entity design_tb is
End entity;
Architecture bench of design_tb is
    Signal Tclk : std_logic := '0';
    Signal Trst : std_logic;
    signal TA,TB,TOUT : std_logic_vector(3 downto 0);
    signal Done : boolean := False;
Begin
-- instanciation du composant à tester
UUT: entity work.design port map (
    A => TA, B => TB,
    OUT => TOUT,
    clk => Tclk, rst => Trst);
-- Génération d'une horloge
Tclk <= '0' when Done else not Tclk after 50 ns;
-- Génération d'un reset au début
Trst <= '1', '0' after 5 ns;
```



testbench (2)

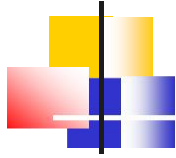
```
Stimuli: process
begin
    TA <= "0000";
    TB <= "0000";
    wait for 10 NS;
    TA <= "1111";
    wait for 10 NS;
    TB <= "1111";
    wait for 10 NS;
    TA <= "0101";
    TB <= "1010";
    wait;
end process;
```

```
Verification: process
begin
    wait for 5 NS;
    assert TOUT = "0000" report "erreur"
    severity warning;
    wait for 10 NS;
    assert TOUT = "1111" report "erreur"
    severity warning;
    wait for 10 NS;
    assert TOUT = "1111" report "erreur"
    severity warning;
    wait for 10 NS;
    assert TOUT = "1010" report "erreur"
    severity warning;
    Done <= True;
    wait;
end process;
End architecture;
```



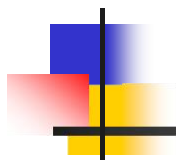
Testbench: 备选方案

```
Stimuli_&_verification: process
Begin
TA <= "0000";
TB <= "0000";
wait for 5 NS;
assert TOUT = "0000" report "erreur" severity warning;
wait for 5 NS;
TA <= "1111";
wait for 5 NS;
assert TOUT = "1111" report "erreur" severity warning;
wait for 5 NS;
TA <= "0101";
TB <= "1010";
wait for 5 NS;
assert TOUT = "1010" report "erreur" severity warning;
Done <= True;
wait;
end process;
End architecture;
```



练习

- 练习3: MinMax电路



C5-文件和库

Yann Douze

VHDL

分析(Analysis)、精细化(Elabortion)、综合(Synthesis)、适配(Fitter)、汇编(Assembler)等。

- 分析——检查语法；
- 精细化——建立数据库、为综合进行初始化；
- 综合——将高层次描述转化为低层次描述并优化代码、
- 适配——布局 and 布线、
- 汇编 ——产生配置数据。

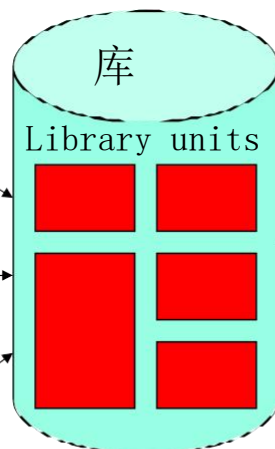
编译

包含“设计单元”的
VHDL文件

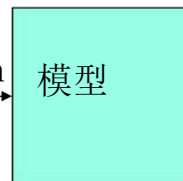
VHDL

VHDL

VHDL



Elaboration



综合

仿真

Synthesis

During Synthesis, the design is mapped from RTL generic constructs to gate-level components. On an FPGA the building blocks are the device primitives: LUTs, flip-flops, memories, PLLs, input-output pins, etc.

Elaboration

The compiler explores the top-down design hierarchy and builds an interconnection table until it reaches the building blocks of the design.

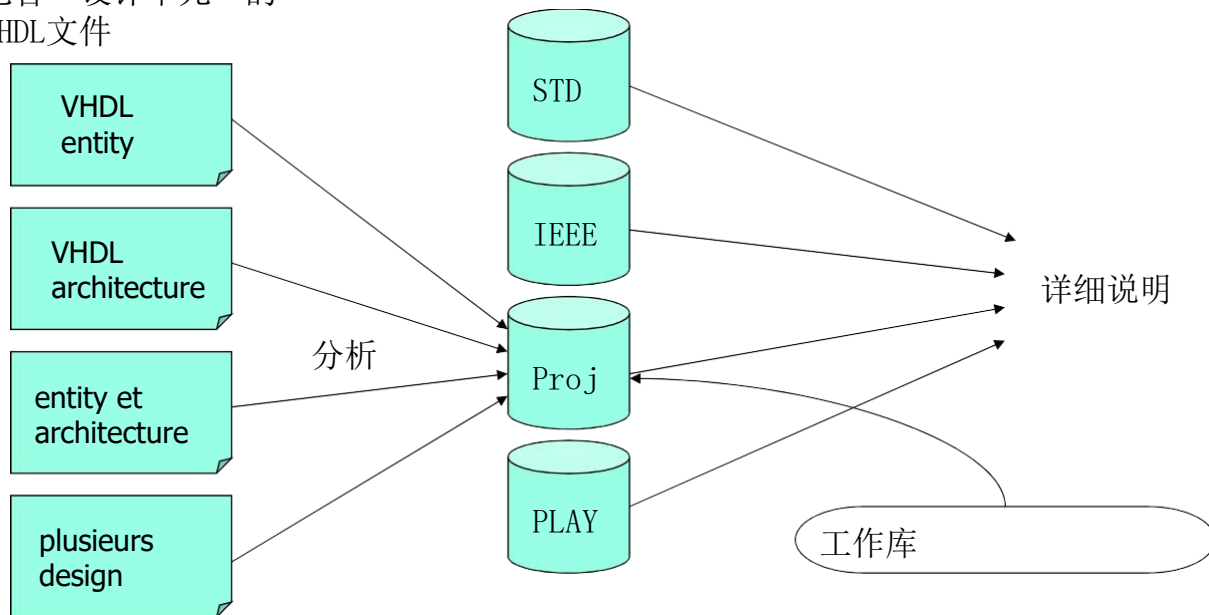
VHDL designs are hierarchic by nature. The top entity instantiates signals, components and processes. Each component instantiates additional signals, components and processes. At this step the building blocks are still generic RTL constructs: Logic gates, registers, memories, etc.

Analysis

Analysis is the process where the design files are checked for syntactic and semantic errors. The syntactic rules are dictated by the language.

工作库

包含“设计单元”的
VHDL文件



上下文子句

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity TEST_BENCH1 is
end entity;

architecture TB of TEST_BENCH1 is
...
end architecture;

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity ENTITY2 is
  port( ... );
end entity;

architecture RTL of ENTITY2 is
...
end architecture;
```

空实体=>上下文子句在
体系结构前面引用

每个实体/包/配置都需要
一个上下文子句



隐含库调用

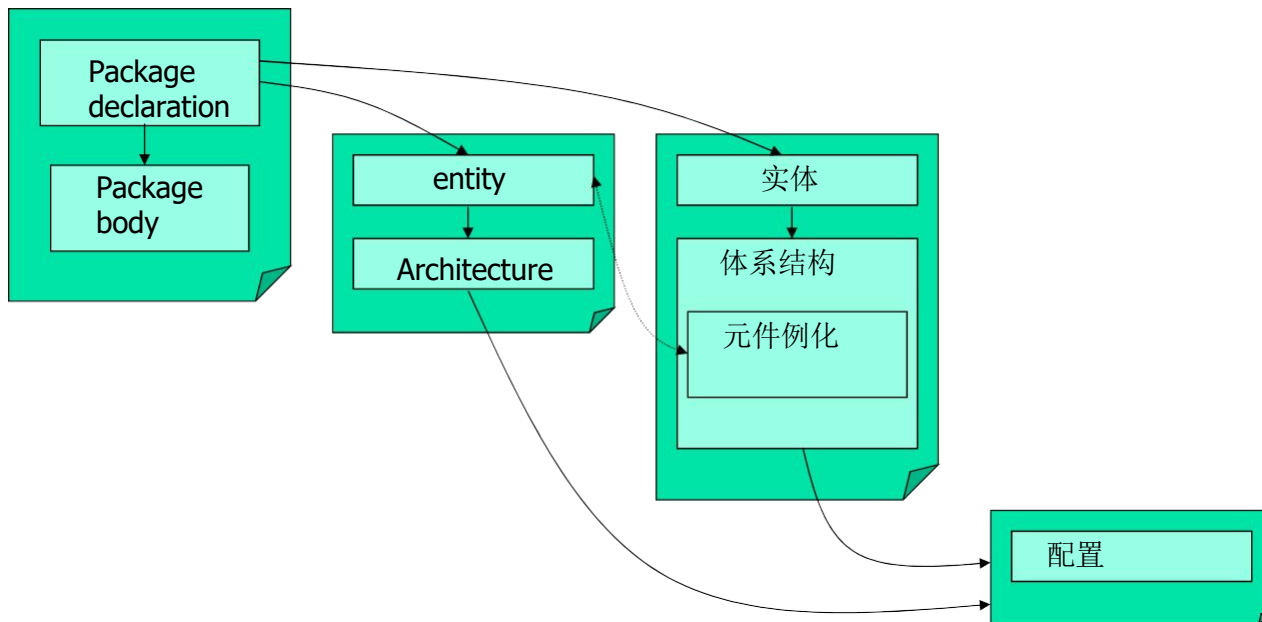
```
library STD, WORK;  
use STD.STANDARD.all; ←
```

默认情况下包括

```
INTEGER    0 1 99  
BIT        '0'  '1'  
BOOLEAN    FALSE  TRUE  
STRING     "Hello World"  
TIME       10 NS
```

Std.Standard包类
型

编译顺序





推荐样式

VHDL 93

缩进

直接实例化

```
architecture BENCH of NOR2_TB is

    signal A, B, F: STD_LOGIC;

begin

    stimulus: process is
    begin
        A <= '0';
        B <= '0';
        wait for 10 NS;
        B <= '1';
        wait;
    end process stimulus;

    UUT: entity work.NOR2 port map
        (   A => A,
          B => B,
          F => F);

end architecture BENCH;
```

对齐

空间

可理解的标注

关联名称

标签名称

标识符 = 字母、数字和下划线

G4X6 Gate_45 \extended! "\$%^&* () \
TheState The_State

不同名称

标识符大小写不敏感

INPUT Input input

同样的名字

许多保留的关键词标识符

And buffer bus function register select

非法标识符。。。

4plus4 A\$1 V-3 The__State _State_

双下划线



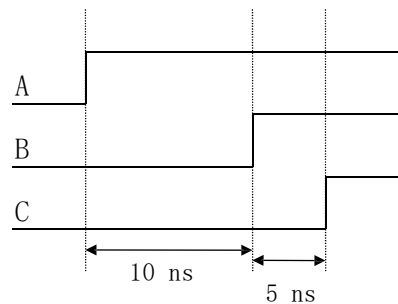
C6-延迟、Delta延迟（增量延迟）和变量

Yann Douze

固有延迟

```
process (A,B)
begin
    B <= A after 10 ns;
    C <= B after 5 ns;
end process;
```

A, B和C的时序图?



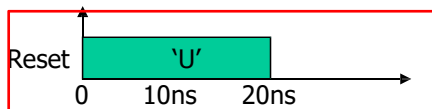
- 延迟用于：
 - 仿真过程中产生激励
 - 由于技术造成的延迟建模
- 综合时会忽略延迟。

产生脉冲 (1)

1. 没有敏感信号列表（只用于仿真），只执行一遍；
2. `process`结束后完成一次赋值；
3. 多重赋值，只执行最后一次赋值语句。

```
process
begin
    reset <= '0';
    reset <= '1' after 10 ns;
    reset <= '0' after 20 ns;
    wait;
end process;
```

错误语法！

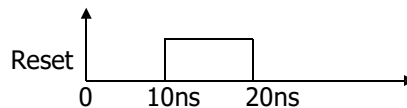


产生脉冲 (2)

```
Reset <= '0', '1' after 10 ns, '0' after 20 ns;
```

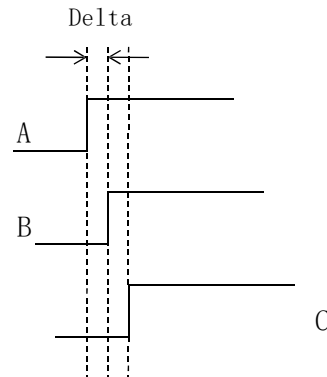
```
process
begin
    reset <= '0';
    wait for 10 ns;
    reset <= '1'
    wait for 10 ns;
    reset <= '0';
    wait;
end process;
```

正确语法!



Delta延迟

```
process (A,B)
begin
  B <= A;
  C <= B;
end process;
```

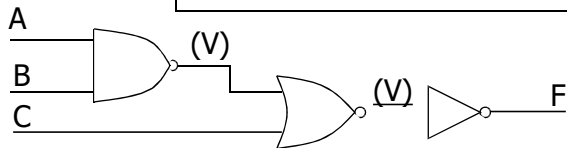


- 在VHDL程序中，delta延迟 = 非零无穷小延迟
- 信号在delta延迟后取新值。
- 变量会立即采用其新值。

变量

- 变量只能声明，并且只存在于进程中。
- 变量的赋值是立即的：在第一行赋值给v的值，可以在第二行直接重用。
- 如果V表示为信号，第一行赋值不起作用。执行第二行赋值。

```
process (A, B, C)
-- zone de déclaration d'une variable
  variable V: STD_LOGIC;
begin
  V := A nand B;
  V := V nor C;  F
  <= not V;
end process;
```



如果你认为V是一个信号，你能重新绘制图表吗？



例：偶校验

检验偶数。初始P为1；
来一个1，设为0；
来第二个1，设为1；

```
Entity parite is
PORT ( a   : IN std_logic_vector(0 TO 3) ;
      s   : OUT std_logic );
END entity;
architecture behaviour of parite is
begin
    process(a)
        variable parite : std_logic ;
    begin
        parite := '1' ;
        FOR i in 0 to 3 LOOP
            if a(i) = '1' then
                parite := not parite;
            end if;
        END LOOP;
        s <= parite;
    end process;
END architecture;
```



练习

```
process (A, S)
    variable V: STD_LOGIC;
begin
    V := A;
    S <= V;
    V := S;
    T <= V;
end process;
```

变量立即赋值。

当在同一进程中，同一信号赋值目标有多个赋值源时，信号赋值目标获得的是最后一个赋值源的赋值，其前面相同的赋值目标不作任何变化。

本应Process结束赋值。
最终V=S，S再次被赋给自己。

立即赋值，使得之前V:=A无用。

process结束时，T被赋为V。

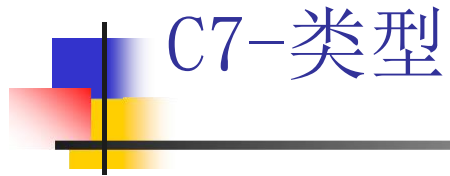
- 假设S为‘0’，A从‘0’状态变为‘1’状态。



问题

1. 在delta延迟之前，进程结束时的s值是多少？
2. 在delta延迟之前，进程结束时，v的值是多少？
3. 在进程执行之后，在delta延迟之后，S和T取它们的新值，S和T是什么？
4. 在进程第一次运行之后，在delta延迟之后，会发生什么？





C7-类型

Yann Douze
VHDL



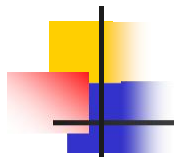
自定义类型（枚举类型）

- 定义新的数据类型

```
type Opcode is (Add, Neg, Load, Store, Jump, Halt);  
signal S: Opcode;
```

```
S <= Add;
```

```
process(S)  
begin  
  case S is  
    when Add =>  
      ...
```

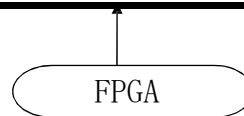
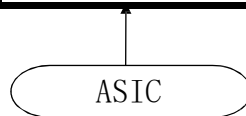


自定义类型的综合

- 用于综合的枚举类型编码

虽然编程是会被自定义，但是在综合后，系统会分配二进制编码，在FPGA内会被实现为独热码。

	二进制	一个热的
Add	000	100000
Neg	001	010000
Load	010	001000
Store	011	000100
Jmp	100	000010
Halt	101	000001





默认类型

```
library STD, WORK;  
use STD.STANDARD.all;
```

默认情况下包括

```
INTEGER      0 1 99  
BIT           '0'  '1'  
BIT_VECTOR   "101011"  
BOOLEAN      FALSE  TRUE  
STRING       "Hello World"  
TIME         10 NS  
REAL         1.345
```

标准包类型。

STD.STANDARD



多值逻辑类型

在STD.STANDARD包中

```
type BOOLEAN is (False, True);  
type BIT is ('0', '1');
```

在IEEE.STD_LOGIC_1164包中

```
type STD_ULOGIC is (  
  'U',  -- non initialisé (par défaut)  
  'X',  -- état inconnu  
  '0',  -- 0 puissant  
  '1',  -- 1 puissant  
  'Z',  -- Haute impédance  
  'W',  -- État inconnu mais faible  
  'L',  -- 0 faible  
  'H',  -- 1 faible  
  '-'); -- indifférent (pour la synthèse)  
  
subtype STD_LOGIC is RESOLVED STD_ULOGIC;
```

Std_ulogic不允许两个以上的驱动器

判决函数

决断函数，用于在多驱动信号时解决信号竞争问题。

决断子类型：设置优先级

驱动器的决断函数

转换函数，用于从一种数据类型到另一种数据类型的转换。如在元件例化语句中，利用转换函数可允许不同数据类型的信号和端口间，进行映射。

决断函数，用于在多驱动信号时解决信号竞争问题。

```
subtype STD_LOGIC is RESOLVED STD_ULOGIC;
```

```
Signal BUSS,ENB1,ENB2,D1,D2: STD_LOGIC;
```

```
...
```

```
TRISTATE1: process (ENB1,D1)
```

```
Begin
```

```
if ENB1 = '1' then
```

```
    BUSS <= D1;
```

```
else
```

```
    BUSS <= 'Z';
```

```
end if;
```

```
End process;
```

```
TRISTATE2: process (ENB2,D1)
```

```
Begin
```

```
    if ENB2 = '1' then
```

```
        BUSS <= D2;
```

```
    else
```

```
        BUSS <= 'Z';
```

```
    end if;
```

```
End process;
```

Driver
Tristate 1
BUSS

Driver
Tristate 2
BUSS

决断函数

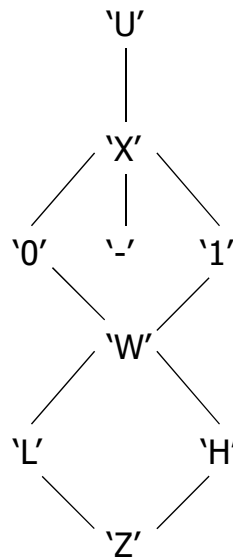
总线

VHDL 语法规则不区分大小写，但当把表示高阻态的'Z'值，赋给一个数据类型为STD_LOGIC 的变量或信号时，'Z'必须为大写。



STD_LOGIC的决断

- std_logic类型在std_logic_1164包中定义
- 架构中的最高值优先





初始值

- 信号和变量在仿真开始时初始化。
- 默认值是类型最左边的值。
- 注意：综合会忽略初始值。

```
type Opcode is (Add, Neg, Load, Store, Jmp, Halt);  
signal S: Opcode; —————→ 初始值为Add  
signal CLOCK, RESET: STD_LOGIC; —————→ 初始值为 'U'  
variable V1: STD_LOGIC_VECTOR(0 to 1); 初始值为 'UU'  
variable V2: STD_LOGIC_VECTOR(0 to 1) := "01";  
signal N: Opcode := Halt;  
constant size: INTEGER := 16;  
constant ZERO: STD_LOGIC_VECTOR := "0000";
```



隐含的运算关系

- 对于STD_LOGIC类型

```
'U' < 'X' < '0' < '1' < 'Z' < 'w' < 'L' < 'H' < '-'
```

- 对于std_logic_vector类型

```
'0' < "00" < "000" < "001" < "100" < "111" < "1111"
```



算术运算

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity ADDER is
    port ( A,B : in STD_LOGIC_VECTOR(7 downto 0);
          SUM : out STD_LOGIC_VECTOR(7 downto 0));
end entity;

architecture A1 of ADDER is
begin
    SUM <= A + B; ←
```

错误: “+” 不能用于
std_logic_vector类型



使用NUMERIC_STD

这里改变了端口的类型，变成了无符号数据显示方式。可以进行。

```
library IEEE;  
use IEEE.STD_LOGIC_1164.all;  
use IEEE.NUMERIC_STD.all;
```

IEEE Std 1076.3

```
entity ADDER is  
    port (      A,B : in UNSIGNED(7 downto 0);  
           SUM : out UNSIGNED(7 downto 0);  
end entity;
```

```
architecture A1 of ADDER is  
begin  
    SUM <= A + B;  
end architecture;
```

"+"运算符对UNSIGNED和SIGNED可用



如何对A和B相加，如果它们是td_logic_vector类型？



转换类型

```
Signal U: UNSIGNED(7 downto 0);  
Signal S: SIGNED (7 downto 0);  
Signal V: STD_LOGIC_VECTOR(7 downto 0);
```

接近的类型之间的转换

```
U.<= UNSIGNED(S);  
S <= SIGNED(U);  
U <= UNSIGNED(V);  
S <= SIGNED(V);  
V.<= STD_LOGIC_VECTOR(U);  
V <= STD_LOGIC_VECTOR(S);
```

类型名称用于类型转换



答案

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity ADDER is
    port ( A,B : in STD_LOGIC_VECTOR(7 downto 0);
          SUM : out STD_LOGIC_VECTOR(7 downto 0));
end entity;

architecture A1 of ADDER is
begin
    SUM <= STD_LOGIC_VECTOR(UNSIGNED(A) + UNSIGNED(B));
    -- ou SUM <= STD_LOGIC_VECTOR(SIGNED(A) + SIGNED(B));
end architecture;
```



转换函数

```
library IEEE;  
use IEEE.STD_LOGIC_1164.all;  
use IEEE.NUMERIC_STD.all;
```

```
Signal U: UNSIGNED(7 downto 0);  
Signal S: SIGNED (7 downto 0);  
Signal N: INTEGER;
```

转换操作

```
N <= TO_INTEGER(U);  
N <= TO_INTEGER(S);  
U <= TO_UNSIGNED(N, 8);  
S <= TO_SIGNED(N, 8);
```

指定向量长度

INTEGER 和STD_LOGIC_VECTOR之间的转换

```
library IEEE;  
use IEEE.STD_LOGIC_1164.all;  
use IEEE.NUMERIC_STD.all;
```

```
Signal V: STD_LOGIC_VECTOR(7 downto 0);  
Signal N: INTEGER;
```

转换函数

类型标识符转换

```
N <= TO_INTEGER(UNSIGNED(V));
```

```
V <= STD_LOGIC_VECTOR(TO_UNSIGNED(N, 8));
```

类型转换

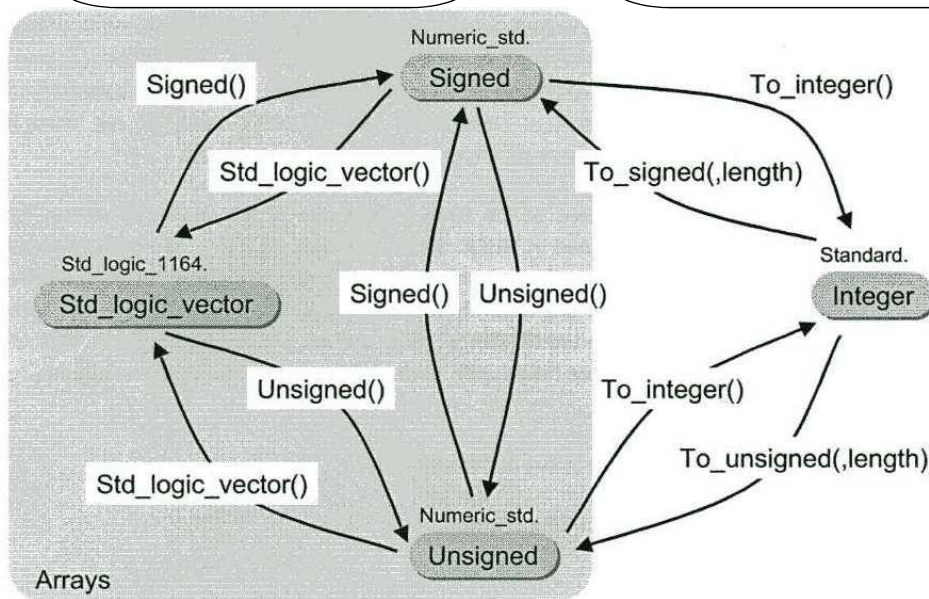
转换函数

```
V <= STD_LOGIC_VECTOR(SIGNED(V)+1);
```

图片概括

类型转换

转换函数





NUMERIC_STD总结

+ - * / rem mod
< <= > >= = /=

UNSIGNED x UNSIGNED
UNSIGNED x NATURAL
NATURAL x UNSIGNED
SIGNED x SIGNED
SIGNED x INTEGER
INTEGER x SIGNED

sll srl rol ror

UNSIGNED x UNSIGNED
SIGNED x INTEGER

not and or nand nor xor
xnor

UNSIGNED x UNSIGNED
SIGNED x INTEGER

TO_INTEGER	[UNSIGNED] return INTEGER
TO_INTEGER	[SIGNED] return INTEGER
TO_UNSIGNED	[NATURAL, NATURAL] return UNSIGNED
TO_SIGNED	[INTEGER, NATURAL] return SIGNED
RESIZE	[UNSIGNED, NATURAL] return UNSIGNED
RESIZE	[SIGNED, NATURAL] return SIGNED

Signature (VHDL1983)



练习 1 （ A、B、C加法）

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

Entity ADDER is
port( A      : in STD_LOGIC_VECTOR(7 downto 0);
      B      : in INTEGER;
      C      : in SIGNED(7 downto 0));
      SUM    : out STD_LOGIC_VECTOR(7 downto 0);
end entity;

Architecture BEHAVIOUR of ADDER is
Begin
SUM <= 
End architecture;
```



练习 2（8 选 1 数据选择器）

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity Mux8to1 is
port( Address :    in STD_LOGIC_VECTOR(2 downto 0);
      IP      :    in STD_LOGIC_VECTOR(7 downto 0);
      OP      :    out STD_LOGIC);
end entity;
architecture BEHAVIOUR of Mux8to1 is
begin

OP <= IP(  (Address));

end architecture ;
```

程序包STD_LOGIC_UNSIGNED/SIGNED

```
library IEEE;  
use IEEE.STD_LOGIC_1164.all;  
use IEEE.STD_LOGIC_UNSIGNED.all;  
-- use IEEE.STD_LOGIC_SIGNED.all;
```

不同时使用

+ -

STD_LOGIC_VECTOR
STD_ULOGIC
INTEGER

*

STD_LOGIC_VECTOR

< <= > >= = /=

STD_LOGIC_VECTOR
INTEGER

```
CONV_INTEGER[STD_LOGIC_VECTOR] return INTEGER
```



程序包STD_LOGIC_ARITH

```
library IEEE;  
use IEEE.STD_LOGIC_1164.all;  
use IEEE.STD_LOGIC_ARITH.all;
```

```
  +  -  
STD_ULOGIC  
UNSIGNED  
SIGNED  
INTEGER
```

```
  *  
UNSIGNED  
SIGNED
```

```
<  <=  >  >=  =  /=  
UNSIGNED  
SIGNED  
INTEGER
```

```
CONV_INTEGER[INTEGER/UNSIGNED/SIGNED/STD_ULOGIC] return INTEGER  
CONV_UNSIGNED[INTEGER/UNSIGNED/SIGNED/STD_ULOGIC, INTEGER] return UNSIGNED  
CONV_SIGNED[INTEGER/UNSIGNED/SIGNED/STD_ULOGIC, INTEGER] return SIGNED  
CONV_STD_LOGIC_VECTOR[INTEGER/UNSIGNED/SIGNED/STD_ULOGIC, INTEGER] return  
STD_LOGIC_VECTOR  
EXT[STD_LOGIC_VECTOR, INTEGER] return STD_LOGIC_VECTOR  
SXT[STD_LOGIC_VECTOR, INTEGER] return STD_LOGIC_VECTOR
```



练习

■ 练习 C7

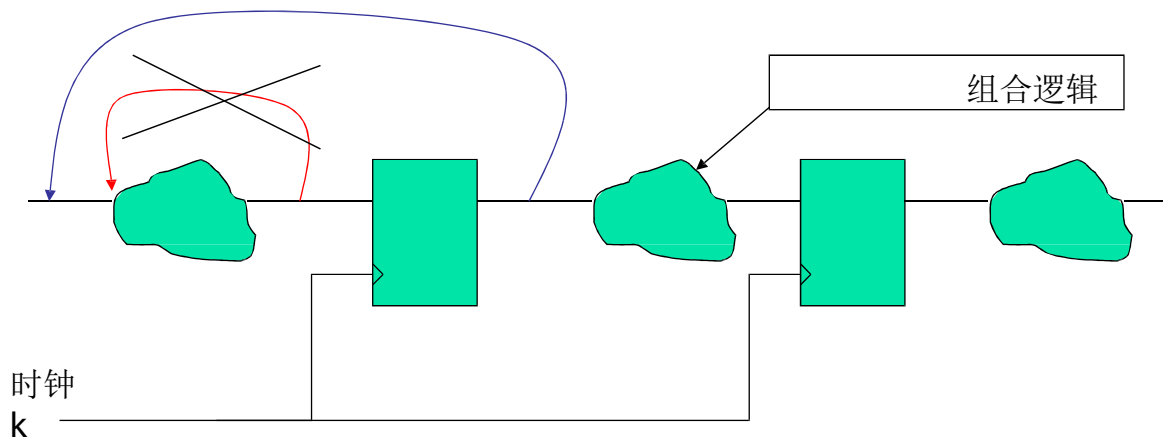


C8 –同步进程

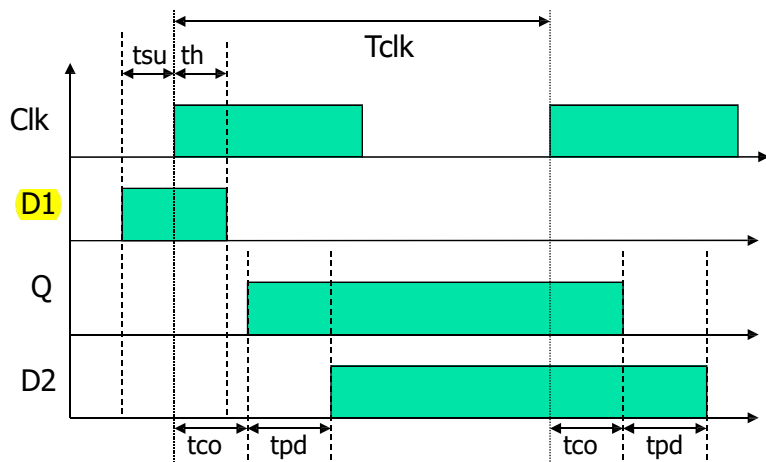
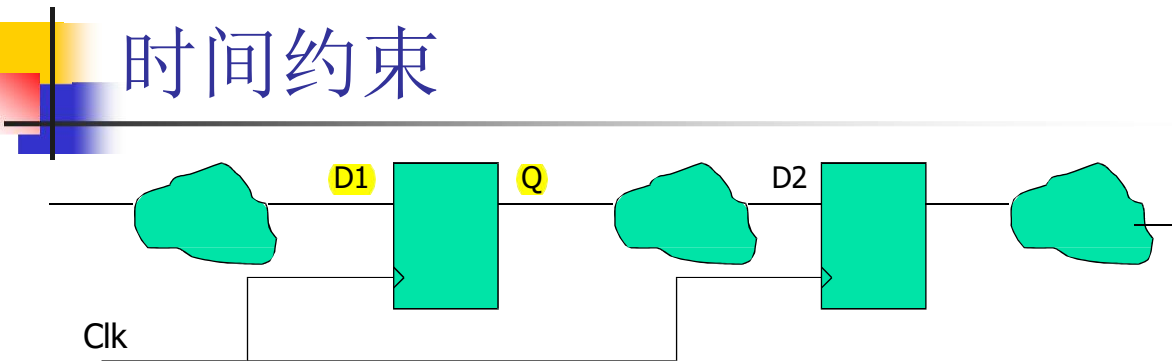
Yann DOUZE
VHDL

同步设计

- 所有寄存器都是在是外部的具有单个时钟的D触发器中。
- 组合逻辑是不能有环路的。



时间约束



Tclk: 时钟周期

tsu: 建立时间

th: 保持时间

tco: clock to output

tpd: propagation delay

$T_{clk} > t_{co} + t_{pd} + t_{su}$

$F_{max} = 1 / (t_{co} + t_{su} + t_{pd})$

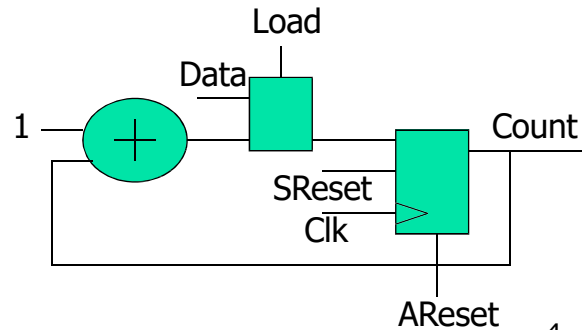
同步和异步动作

```
signal Count : unsigned(7 downto 0);
process (Clk, AReset)
begin
    if AReset = '1' then
        Count <= "00000000";
    elsif RISING_EDGE(Clk) then
        if SReset = '1' then
            Count <= "00000000";
        elsif Load = '1' then
            Count <= UNSIGNED(Data);
        else
            Count <= Count + '1';
        end if;
    end if;
end process;
```

异步Reset

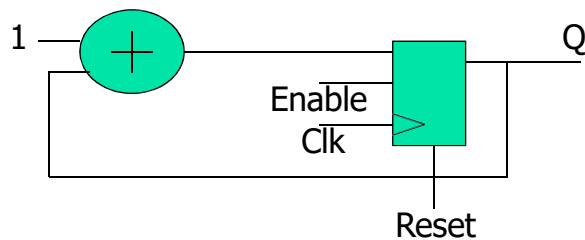
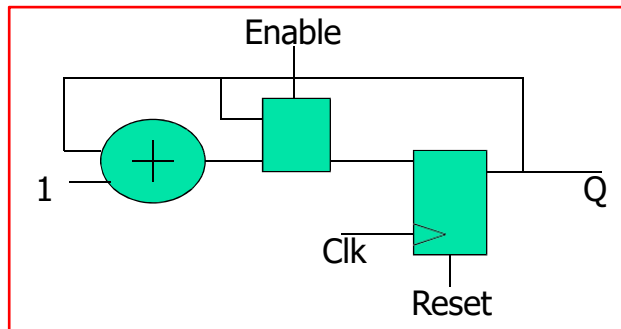
同步Reset

同步置数Load



时钟使能

```
process (Clk, Reset)
begin
    if Reset = '1' then
        Q <= "00000000";
    elsif RISING_EDGE(Clk) then
        if Enable = '1' then
            Q <= Q + 1;
        end if;
    end if;
end process;
```



Wait Until合法代码样式

--没有列表的进程

```
Process
```

```
begin
```

```
  wait until Clock = '1';
```

```
  if Reset = '1' then
```

```
    Q <= '0';
```

```
  else
```

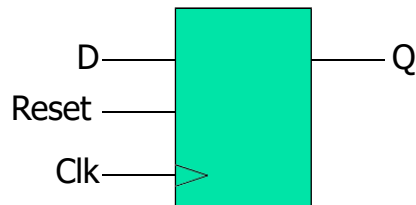
```
    Q <= D;
```

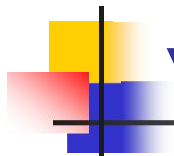
```
  end if;
```

```
end process;
```

边沿检测

同步复位





'EVENT合法代码样式

- S'EVENT 为真，当且仅当有一个事件发生在S上

```
process (Clk)
begin
    if Clk'EVENT and Clk = '1' then
        Q <= D;
    end if;
end process;
```

```
process
begin
    wait until Clk'EVENT and Clk = '1';
    ...
end process;
```




边沿检测

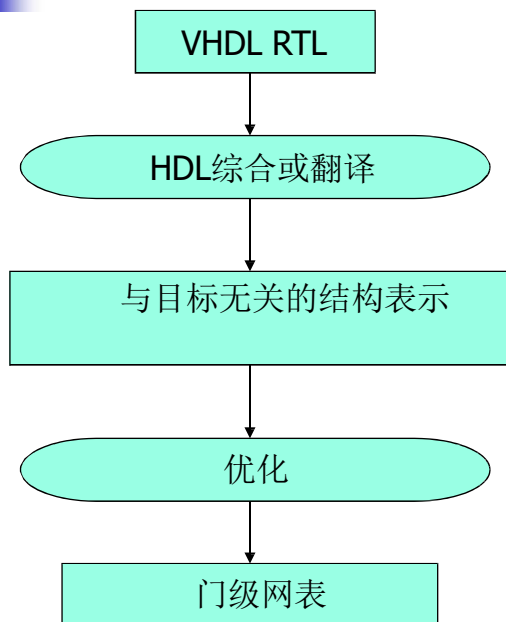
- 边缘检测操作

- (`Rising_edge (clk)` and `Clk'event and clk='1'`) 应该只用于检测时钟的边缘 (`clk`)。

- 每次我们进行边缘检测时，综合工具都知道它是一个时钟

- 时钟 = 电路中最快的信号。

综合工具如何工作



注意：可综合性。

时序行为如惯性或传输延迟的描述 都将被 VHDL 综合器忽略；

仿真模型可以描述一些无限制的 条件如无穷循环或无范围限制 的整型数，硬件却不可能提供 这些条件，在某些情况下，如 无穷循环或循环次数不确定的 情况下，综合工具会产生错误 并退出。

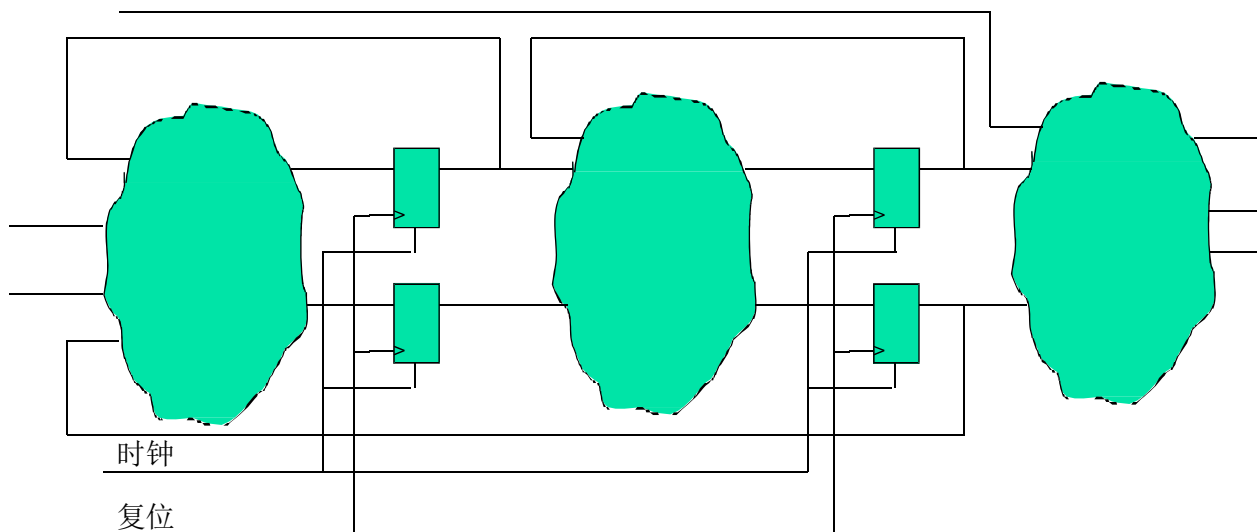
- 不是 优化

- 用VHDL代码推断的结构

- 优化组合逻辑和寄存器之间的时间约束

- 取决于目标

综合 RTL



- RTL综合不添加、删除或移动寄存器。
- RTL综合仅优化逻辑组合的。

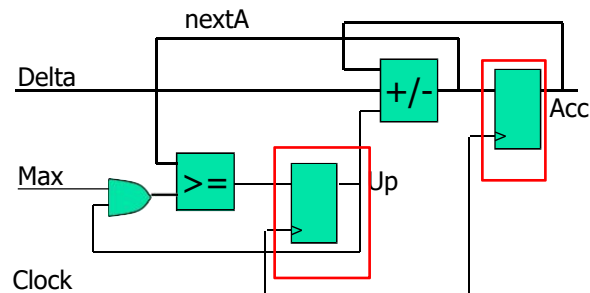


规则：寄存器综合

- RTL 综合工具按照某些基本规则，直接从 VHDL 代码推断寄存器：
 1. 仅在同步进程中被综合推断为寄存器。
 2. 信号：在同步进程中赋值的所有信号都综合成寄存器。
 3. 变量：在同步进程中赋值的变量可以综合成连线或寄存器。
 - 在读取之前赋新值的变量，综合成连线。（在被读取之前分配 => 连线）
 - 在赋值之前读取的变量，综合成寄存器。（分配前读取=>寄存器）

寄存器的综合

```
signal Acc, Delta, Max: signed(11 downto 0);
process (Clock)
    variable Up : std_logic;
    variable nextA : signed(11 downto 0);
begin
    if RISING_EDGE(Clock) then
        if Up = '1' then
            nextA := Acc + Delta;
            if nextA >= Max then
                Up := '0';
            end if;
        else
            nextA := Acc - Delta;
            if nextA < 0 then
                Up := '1';
            end if;
        end if;
        Acc <= nextA;
    end if;
end process;
```

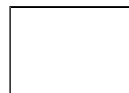




练习 1

```
Signal INPUT : std_logic_vector(7 downto 0)
Signal REG : std_logic;
AND-REG : process (CLOCK)
    variable V : STD_LOGIC;
begin
    if RISING_EDGE(CLOCK) then
        V := '1';
        for I in 0 to 7 loop
            V := V and INPUT(I);
        end loop;
        REG <= V;
    end if;
end process;
```

综合D触发器?

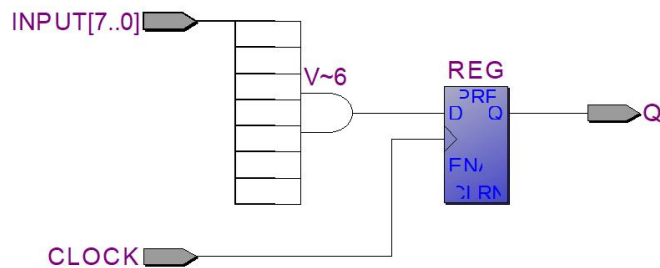


Flip-flops

全1检测器

练习 1

```
1  library IEEE;
2  USE IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.Numeric_std.all;
4
5  entity D is
6  port(
7      CLOCK: in std_logic;
8      INPUT:std_logic_vector(7 downto 0);
9      Q:out std_logic
10     );
11  end entity;
12
13  architecture D1 of D is
14  signal REG:std_logic;
15  begin
16  process(CLOCK)
17      variable V :STD_LOGIC;
18      begin
19      if RISING_EDGE(CLOCK) then
20          v := '1';
21          for I in 0 to 7 loop
22              v := v and INPUT(I);
23          end loop;
24          REG <= V;
25      end if;
26  end process;
27  Q <= REG;
28  end architecture;
```

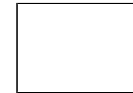




练习 2

```
Signal OUTPUT : std_logic;
COUNTER : process (CLOCK)
    variable COUNT : UNSIGNED(7 downto 0);
begin
    if RISING_EDGE(CLOCK) then
        if RESET = '1' then
            COUNT := "00000000";
        else
            COUNT := COUNT + 1;
        end if;
        OUTPUT <= COUNT(7);
    end if;
end process;
```

Combien de flip-flops ?



Flip-flops

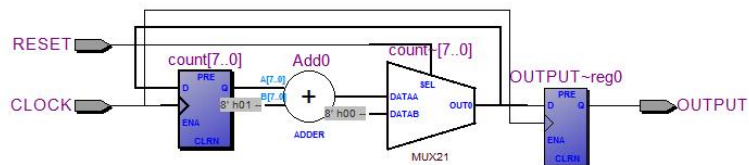
模几？

练习 2

```
library IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
use IEEE.Numeric_std.all;

entity D9 is
port(
    RESET,CLOCK: in std_logic;
    OUTPUT:out std_logic
);
end entity;

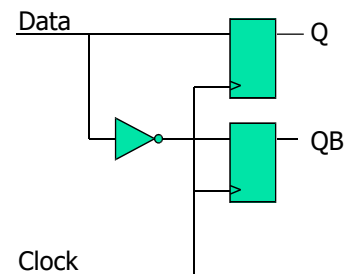
architecture D of D9 is
begin
    process(CLOCK,RESET)
        variable count :UNSIGNED(7 downto 0);
        begin
            if RISING_EDGE(CLOCK) then
                if RESET = '1' then
                    count := "00000000";
                else
                    count := count + 1;
                end if;
                OUTPUT <= count(7);
            end if;
        end process;
    end architecture;
```



综合不优化寄存器！

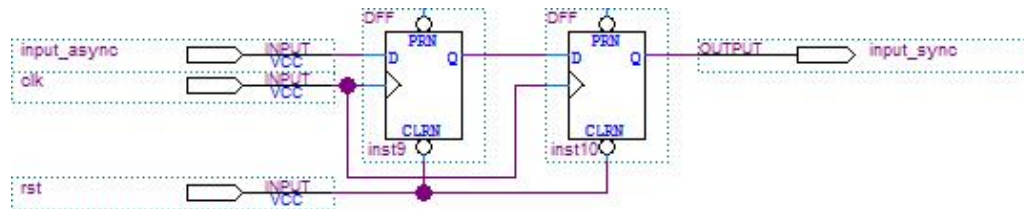
```
process (CLOCK)
begin
  if RISING_EDGE(CLOCK) then
    Q <= Data;
    QB <= not Data;
  end if;
end process;
```

靠人优化。



如何重写代码以确保只有一个D 触发器？

练习：同步输入



- 优点：操作安全
- 缺点：引入延迟 (pipeline)

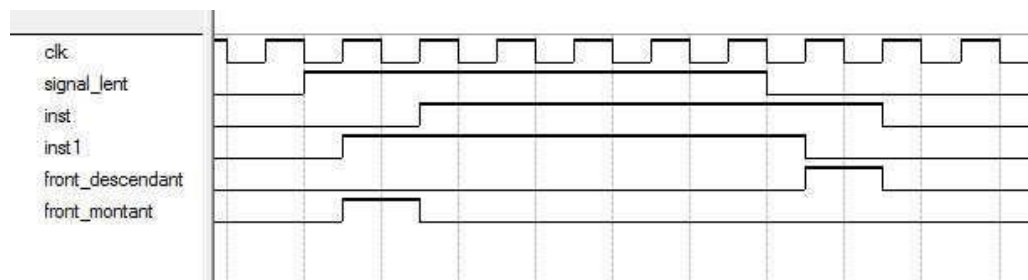
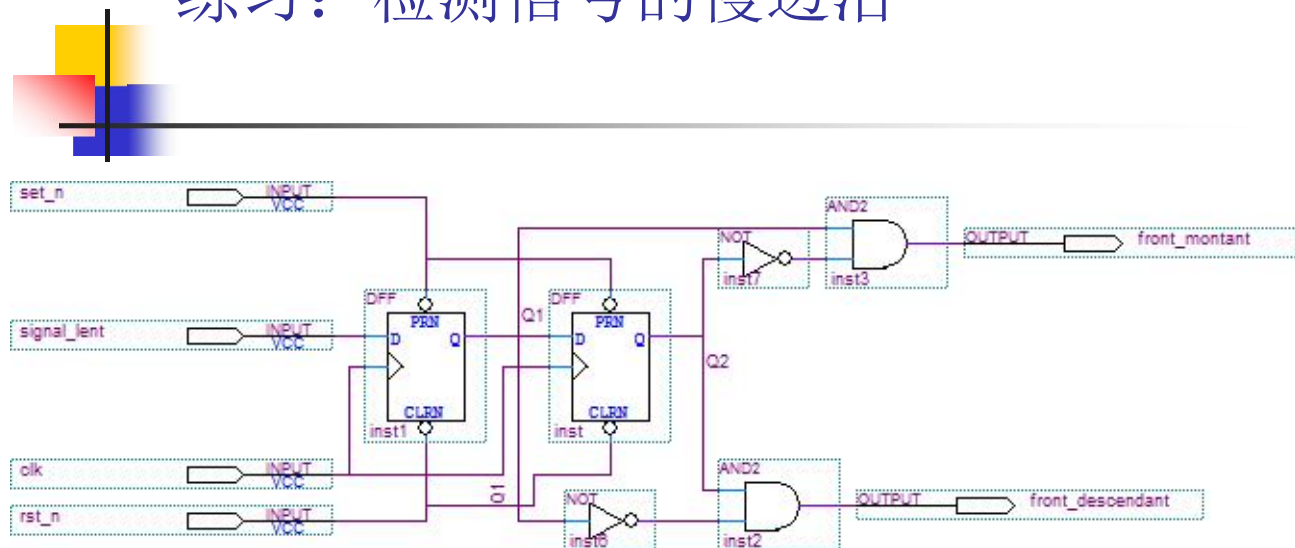


代码：同步输入

```
Entity resynchro is
port(
    clk, rst, input_async:      in      std_logic;
    input_sync:                  out      std_logic);
end entity;
architecture RTL of resynchro is
Signal Q : std_logic;
Begin
    Process (clk,rst)
    begin
        if (rst='1') then
            Q <= '0'; input_sync <= '0';
        elsif rising_edge(clk) then
            Q<= input_async;
            input_sync <= Q;
        end if;
    end process;
end process;
```

两个寄存器

练习：检测信号的慢边沿





慢边缘检测： 编码

Entity detect_fronts is

port(

clk, rst, signal_lent : in std_logic;

fm, fd: out std_logic);

end entity;

architecture RTL of detect_fronts is

Signal Q1,Q2 : std_logic;

begin

PROCESS (clk , rst)

BEGIN

if rst ='1' then

Q1 <= '0'; Q2 <= '0';

elsif rising_edge (clk) then

Q1 <= signal_lent;

Q2 <= Q1;

end if;

end process;

Fm <= Q1 and (not Q2);

Fd <= Q2 and (not Q1);

end architecture;

fm上升沿检测； fd下降沿检测；

使用示例：事件计数器

```
entity cnt_evt is
port(clk, rst, sig_lent : std_logic;
      cnt : std_logic_vector(7 downto 0));

end entity;
architecture RTL of cnt_evt is
    signal Q : unsigned (7 downto 0);
    signal fm : std_logic;

Begin
U1 : entity work.detect_front port map (
    clk => clk, rst => rst, signal_lent=>sig_lent, fm => fm);
Process (clk,rst)
begin
    if (rst='1') then
        Q <= (others => '0');
    elsif rising_edge(clk) then
        if (fm='1') then
            Q<= Q + '1';
        end if;
    end if;
end process;
cnt <= std_logic_vector(Q);
End architecture;
```

ZR

2020-02-27 19:13:56

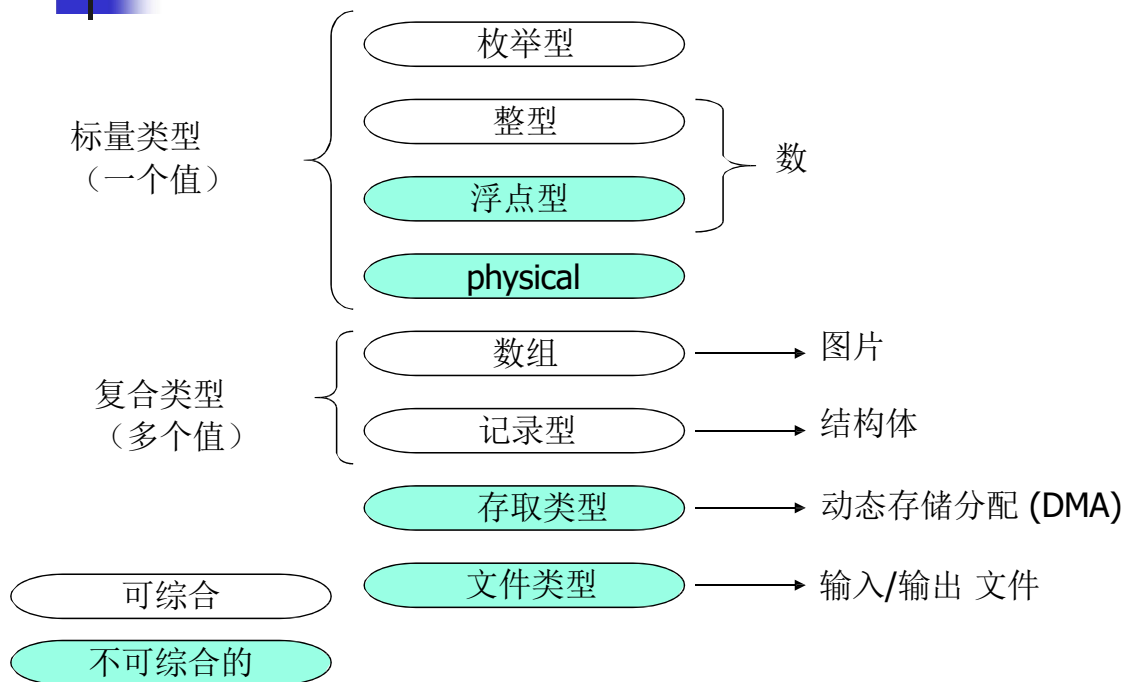
对fm进行计数。



C9 –子类型 （亚型）

Yann DOUZE
VHDL

VHDL 中的数据类型

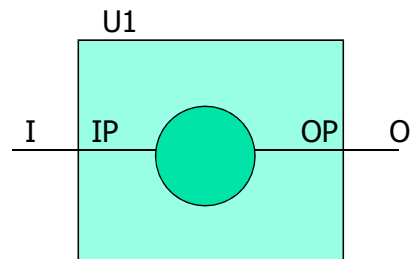


程序包里的数据类型

```
package MY_TYPES is
  type CODE is (A, B, C, D, E);
end package MY_TYPES;
```

在单独的文件中

```
use WORK.MY_TYPES.all;
entity FILTER is
  port(IP: in CODE;
        OP : out CODE);
end entity FILTER;
```



```
use WORK.MY_TYPES.all;
...
signal I, O : CODE;
...
U1: entity work.FILTER port map (IP => I, OP => O);
```



整数类型的子类型

```
type INTEGER is range -2**31+1 to 2**31-1;  
subtype NATURAL is INTEGER range 0 to 2**31-1;  
subtype POSITIVE is INTEGER range 1 to 2**31-1;
```

定义在程序包STD.STANDARD

```
subtype SHORT is INTEGER range -128 to +127;  
subtype LONG is INTEGER range -2**15 to 2**15-1;  
signal S: SHORT;  
signal L: LONG;
```

用户定义的子类型

```
variable I: INTEGER range 0 to 255;
```

匿名亚型

```
I := -1;  
S <= L;
```

这些赋值是否错误?



整数子类型的综合

```
subtype Byte is INTEGER range 0 to 255;  
signal B: Byte;
```

8 bits, 无符号

```
subtype Int8 is INTEGER range -128 to +127;  
signal I: Int8;
```

8 bits, 有符号补码

```
subtype Silly is INTEGER range 1000 to 1001;  
signal S: Silly;
```

10 bits, 无符号

```
signal J: INTEGER;
```

32 bits, 补码, 注意!

算术运算符

指数 $A^B = A^{**}B$

取余 A/B

取模 $A \bmod B$

绝对值 $|A|$

不同的

与信号赋值相同

+
-
*
/
**
rem
mod
abs
=
/=
<
<=
>
>=

可以综合

取决于综合工具

常数或幂2 示例: $A/4$ 或 $2^{**}N$

取决于综合工具

好的 综合



整数值的表示

整数

0 99 1e6 1E6 1000000 1_000_000

_ 忽略

以二进制、八进制和十六进制写入STD_LOGIC_VECTOR

B"0000_1111" O"017" X"0F"

VHDL 1993



数组子类型（数组）

自定义了一个类型的数据类型

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package BusType is
    subtype DataBus is STD_LOGIC_VECTOR(7 downto 0);
end package Bustype;
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
Use WORK.BusType.all;

Entity CNTL is
    port( CLK    : in      STD_LOGIC;
          D      : in      DataBus;
          Sin    : in      STD_LOGIC_VECTOR(3 downto 0);
          Q      : out     DataBus;
          Sout   : out     STD_LOGIC_VECTOR(1 downto 0));
End entity CNTL;
```



数组类型（数组）

不受约束的数组类型

```
type STD_LOGIC_VECTOR is array (NATURAL range <>) of STD_LOGIC;
```

索引类型

元素类型

```
subtype Byte is STD_LOGIC_VECTOR(7 downto 0);
```

```
type RAM1Kx8 is array (0 to 1023) of Byte;  
variable RAM: RAM1Kx8;
```

受约束的数组类型

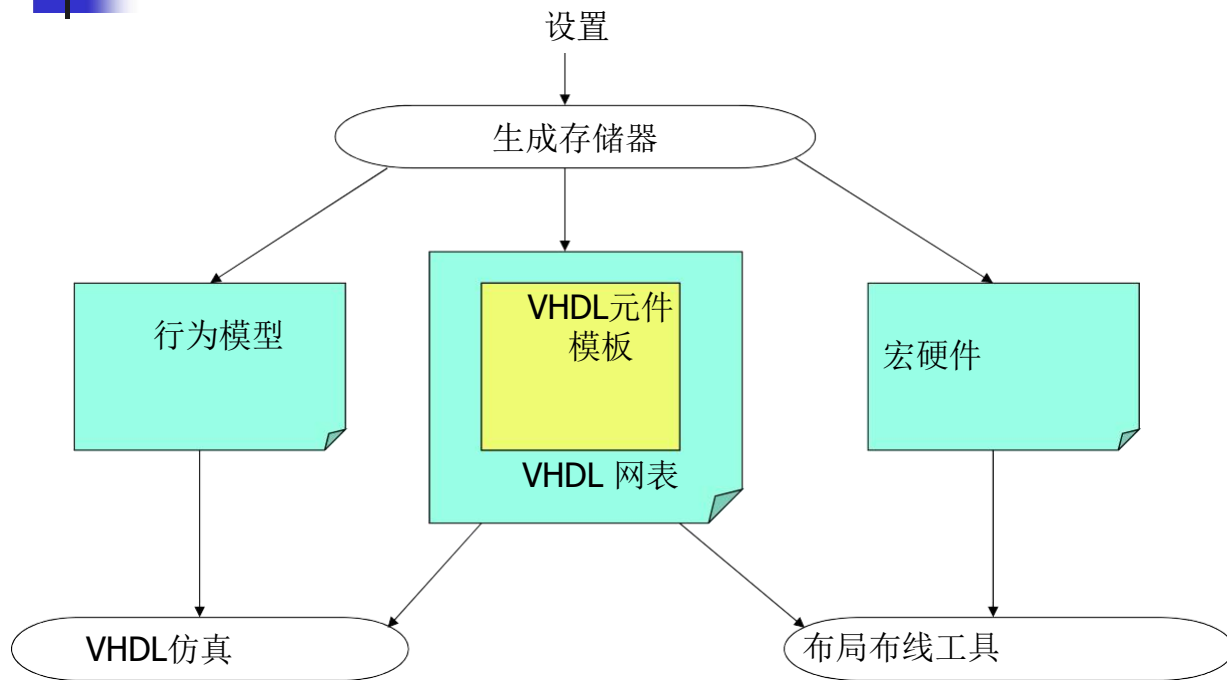


存储器建模

```
entity DualPortRam is
    port ( Clock, Wr, Rd : in      Std_logic;
           AddrWr, AddrRd : in     Std_logic_vector(3 downto 0);
           DataWr         : in     Std_logic_vector(7 downto 0);
           DataRd         : out     Std_logic_vector(7 downto 0));
end entity;

architecture modele of DualPortRam is
    type RamType is array (0 to 15) of Std_logic_vector(7 downto 0);
    signal RAM : RamType;
begin
    process (Clock)
    begin
        if RISING_EDGE(Clock) then
            if Wr = '1' then
                Ram(To_integer(Unsigned(AddrWr))) <= DataWr;
            end if;
            if Rd = '1' then
                DataRd <= Ram(To_integer(Unsigned(AddrRd)));
            end if;
        end if;
    end process;
end architecture;
```

存储器实例化





属性函数 'RANGE

```
Signal A: STD_LOGIC_VECTOR(...);
```

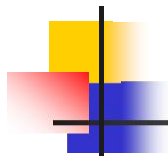
```
process(A)
  variable V: STD_LOGIC;
begin
  V := '0';

  for I in 0 to 7 loop
    V := V xor A(I);
  end loop;

  for I in A'RANGE loop
    V := V xor A(I);
  end loop;
  ...
end process
```

不是 通用的

可通用



类型和数组属性

```
signal A: STD_LOGIC_VECTOR(7 downto 0);  
subtype SHORT is INTEGER range 0 to 15;  
type MODE is (W, X, Y, Z);
```

- 数组的属性（尽可能使用）

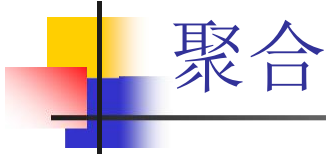
```
A'LOW    = 0  
A'HIGH   = 7  
A'LEFT   = 7  
A'RIGHT  = 0
```

```
A'RANGE = 7 downto 0  
A'REVERSE_RANGE = 0 to 7  
A'LENGHT = 8
```

- 类型的属性（在综合中要避免）

```
SHORT'LOW    = 0  
SHORT'HIGH   = 15  
SHORT'LEFT   = 0  
SHORT'RIGHT  = 15
```

```
MODE'LOW    = W  
MODE'HIGH   = Z  
MODE'LEFT   = W  
MODE'RIGHT  = Z
```



聚合

```
Type BCD6 is array (5 downto 0) of STD_LOGIC_VECTOR(3 downto 0);  
Variable V: BCD6 := ("1001", "1000", "0111", "0110", "0101", "0100");
```

```
V := ("1001", "1000", others => "0000");  
V := (3 => "0110", 1 => "1001", others => "0000");  
V := (others => "0000");
```

```
Variable A: STD_LOGIC_VECTOR(3 downto 0);
```

```
A := (others => '1');
```



模糊的类型

```
port (A,B: in STD_LOGIC);
```

```
process(A, B)
begin
  case A & B is
  when "00" => ...
```

不合法!

难以判定数据类型

```
library IEEE;
use IEEE.NUMERIC_STD.all
```

```
variable N: INTEGER;
```

```
N := TO_INTEGER("1111");
```

不合法



合格的表达式

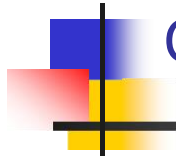
```
process(A, B)
  subtype T is STD_LOGIC_VECTOR(0 to 1);
begin
  case T' (A & B) is
    when "00" => ...
```

```
N := TO_INTEGER(UNSIGNED' ("1111"));
```

N = 15

```
N := TO_INTEGER(SIGNED' ("1111"));
```

N = -1



C10 -Generic, generate

Yann DOUZE

端口映射

-- 元件实体 COUNTER

```
entity COUNTER is
  port ( CLK, RST: in      Std_logic;
         UpDn      : in      Std_logic := '0';
         Q          : out     Std_logic_vector(2 downto 0));
```

默认取值

end entity;

--结构体 STRUCT 元件 BLOK 实例 COUNTER

```
architecture STRUCT of BLOK is
begin
```

-- 位置关联

```
G1 : entity work.COUNTER port map (Clk32MHz, RST, open, Count);
```

-- 端口名关联

```
G2 : entity work.COUNTER port map( RST => RST,
                                   CLK => Clk32MHz,
                                   Q(2) => Q1MHz,
                                   Q(1) => Q2MHz,
                                   Q(0) => Q4MHz);
```

不连接的

end architecture;

VHDL 93

2



8位计数器

```
entity COUNTER8BIT is
    port ( CLK, RST: in      Std_logic;
           Q       : out     Std_logic_vector( 7 downto 0));
end entity;
architecture RTL of COUNTER8BIT is
    signal CNT: unsigned( 7 downto 0);
begin
    process (CLK,RST)
    begin
        if RST = '1' then
            CNT <= "00000000";
        elsif rising_edge(CLK) then
            CNT <= CNT + '1';
        end if;
    end process;
    Q <= std_logic_vector(CNT);
end architecture;
```



通用计数器

```
entity COUNTER is
  generic(N : integer:=8);
  port ( CLK, RST :in      Std_logic;
         Q         : out   Std_logic_vector(N-1 downto 0));
end entity;
architecture RTL of COUNTER is
  signal CNT: unsigned(N-1 downto 0);
begin
  process (CLK,RST)
  begin
    if RST = '1' then
      CNT <= (others => '0');
    elsif rising_edge(CLK) then
      CNT <= CNT + '1';
    end if;
  end process;
  Q <= std_logic_vector(CNT);
end architecture;
```



实例化通用组件

```
-- 元件实体COUNTER
entity COUNTER is
    generic(N : integer:=8);
    port (CLK, RST : in  Std_logic;
          Q       : out  Std_logic_vector(N-1 downto 0));
end entity;

-- Utilisé dans l'architecture STRUCT d'un composant BLOK
architecture STRUCT of BLOK is
    signal Count4: std_logic_vector(3 downto 0);
    signal Count6: std_logic_vector(5 downto 0);
begin
    -- association par position
    U1: entity work.COUNTER generic map(4)
        port map(CLK , RST, Count4);
    -- association par nom
    U2: entity work.COUNTER generic map (N => 6)
        port map (CLK => CLK, RST => RST, Q => Count6);
end architecture;
```



通用延迟

```
-- Entité du composant NAND2
entity NAND2 is
    generic (TPLH,TPHL: TIME := 0 NS);
    port ( A, B: in      Std_logic;
           F   : out     Std_logic);
end entity;

-- Architecture STRUCT du composant BLOK
architecture STRUCT of BLOK is
    signal N1,N2,N3,N4,N5,N6,N7,N8,N9 : Std_logic;
begin
    G1: entity work.NAND2 generic map (1.9 NS, 2.8 NS)
        port map (N1, N2, N3);
    G2: entity work.NAND2 generic map (TPLH => 2 NS, TPHL => 3 NS)
        port map (A => N4, B => N5, F => N6);
    G3: entity work.NAND2 port map (A => N7, B => N8, F => N9);
end architecture;
```

generate操作说明

```
architecture A1 of BLOK is
begin
    G1: for I in SOME_RANGE generate
        -- Instanciation de composant ou process
    end generate;

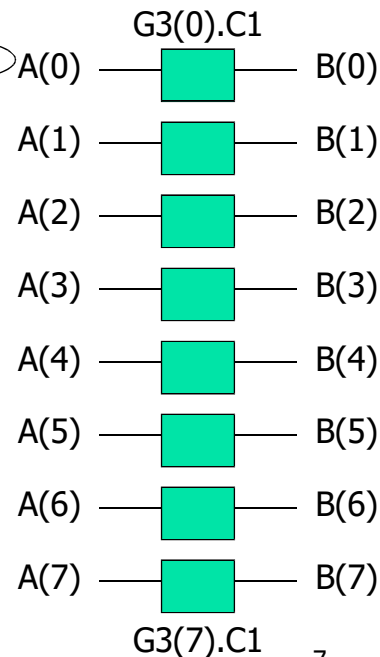
    G2: if CONDITION generate
        -- Instanciation de composant ou process
    end generate;

--Exemple :
G3: for I in 0 to 7 generate
    C1: entity work.COMP port map (D=>A(I), Q=>B(I));
end generate;

end architecture;
```

常规结构

可选结构





通用结构加法器

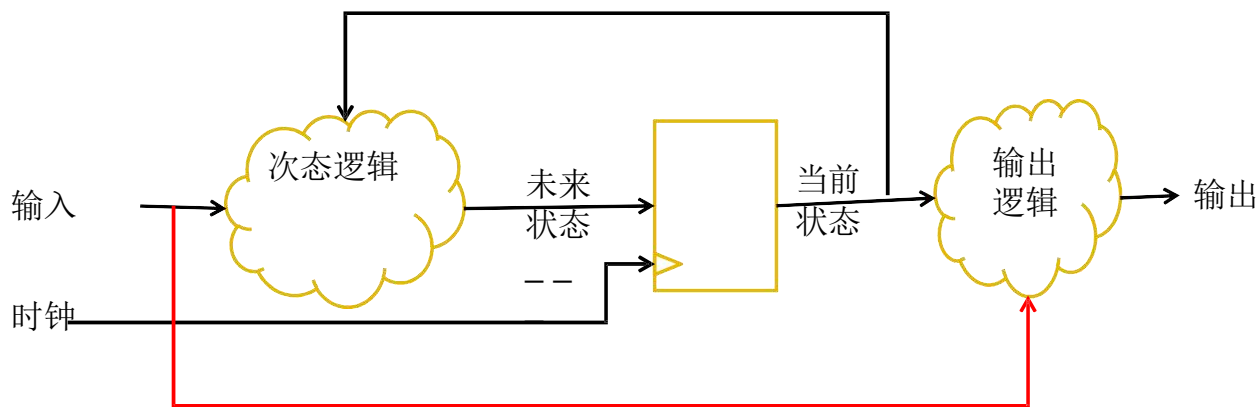
```
entity ADDN is
    generic(N: positive :=4);
    port( Cin : in std_logic;
          A,B: in std_logic_vector(N-1 downto 0);
          Cout : out std_logic;
          SUM: out std_logic_vector(N-1 downto 0));
end entity;
architecture STRUCT of ADDN is
    signal C : std_logic_vector(N downto 0);
begin
    C(0) <= Cin;
    L1: for I in A'reverse_range generate
        U1 : entity work.ADDC1 port
            map( Cin => C(I), A => A(I), B =>
                B(I), SUM => SUM(I), Cout =>
                C(I+1));
        end generate;
    Cout <= C(N);
end architecture;
```



C11 MAE / FSM

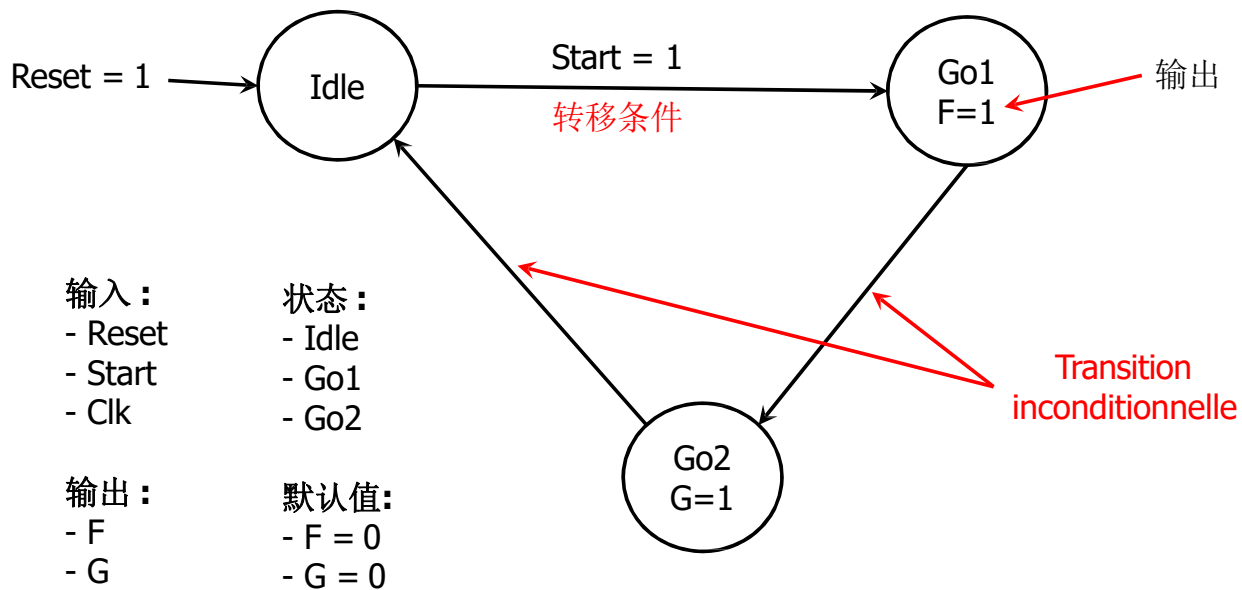
Yann DOUZE

摩尔,米里状态机



- 摩尔状态机 _ 输出仅取决于状态。
- 米里状态机 _ 输出取决于当前状态和输入。

状态图





VHDL描述：摩尔状态机（要避免，效率低）

```
architecture FSM of EXAMPLE is
    type StateType is (Idle, Go1, Go2);
    Signal State : StateType;
begin
    process(Clk, Reset)
        if Reset = '1' then
            State <= Idle;
        elsif Rising_edge(Clk) then
            case State is
                when Idle =>
                    if Start = '1' then
                        State <= Go1;
                    end if;
                when Go1 =>
                    State <= Go2;
                when Go2 =>
                    State <= Idle;
            end case;
        end if;
    end process;
```

```
output : process(State)
begin
    F <= '0';
    G <= '0';
    if State = Go1 then
        F <= '1';
    elsif State = Go2 then
        G <= '1';
    end if;
end process;
end architecture;
```

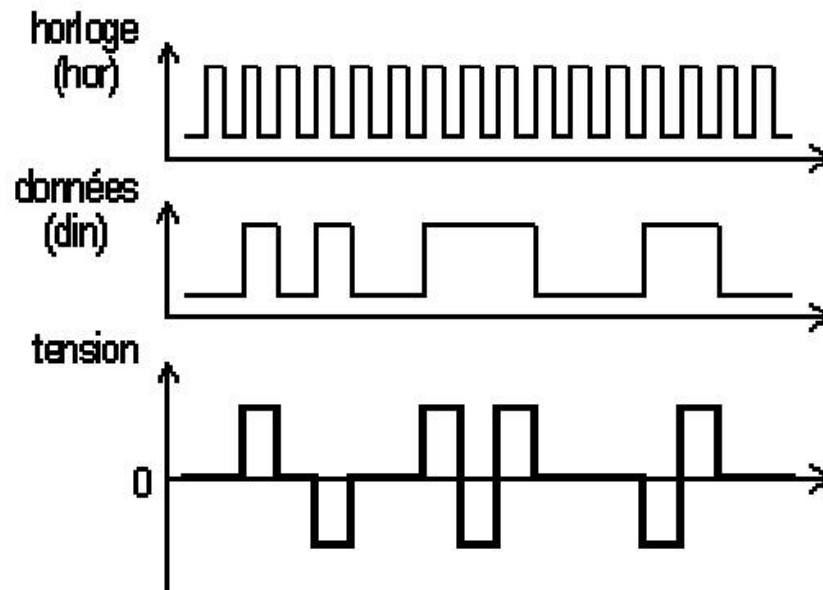


描述VHDL : Mealy machine

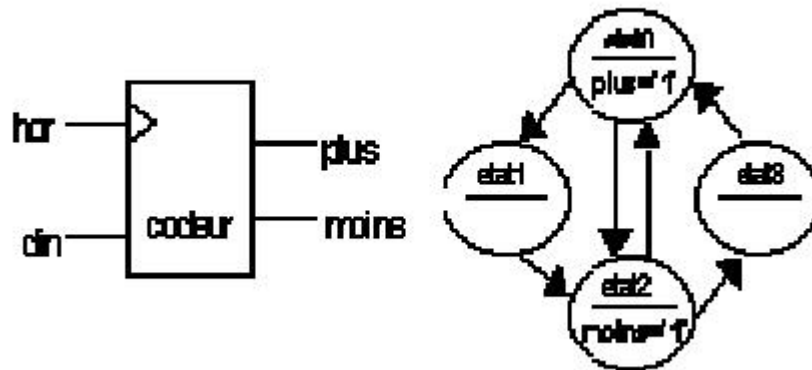
```
architecture FSM of EXAMPLE is
    type StateType is (Idle, Go1, Go2);
    Signal State : StateType;
begin
    process(Clk, Reset)
    begin
        if Reset = '1' then
            State <= Idle;
            G <= '0';
            F <= '0';
        elsif Rising_edge(Clk) then
            case State is
            when Idle =>
                if Start = '1' then
                    State <= Go1;
                    F <= '1';
                end if;
```

```
            when Go1 =>
                State <= Go2;
                G <= '1';
                F <= '0';
            when Go2 =>
                State <= Idle;
                G <= '0';
            end case;
        end if;
    end process;
end architecture;
```

AMI编码器（备用标记 反转）



AMI编码器状态图





编码器VHDL代码 朋友

```
architecture MAE of AMI is
    type StateType is (E0, E1, E2, E3);
    Signal State : StateType;
begin
    process(Clk, Reset)
    begin
        if Reset = '1' then
            State <= E3;
            plus <= '0';
            moins <= '0';
        elsif Rising_edge(Clk) then
            case State is
                when E0 => if Din = '0' then
                    State <= E1;
                    plus <= '0';
                elsif Din = '1' then
                    State <= E2;
                    moins <= '1';
                    plus <= '0';
                end if;
            end case;
        end if;
    end process;
```

```
        when E1 => if Din = '1' then
            State <= E2;
            moins <= '1';
            plus <= '0';
        end if;
        when E2 => if Din = '1' then
            State <= E0;
            plus <= '1';
            moins <= '0';
        elsif Din = '0' then
            State <= E3;
            plus <= '0';
            moins <= '0';
        end if;
        when E3 => if Din = '1' then
            State <= E0;
            plus <= '1';
            moins <= '0';
        end if;
    end case;
end if;
```

```
end process;
end architecture;
```