§5.5 Verilog语言程序基本结构

现代电子设计方法EDA技术

Electronic Design Automation

现代电路的设计方法：硬件设计+编程设计

掌握描述方式：硬件描述语言 （verilog）

熟悉设计工具：集成化开发系统

（ Altera（Intel）公司： QuartusII

Xilinx公司： Vivado

Lattice公司： ispDesignEXPERT ）

了解实现的载体：PLD （Programmable Logic Device）

学习方法：重视上机实践

硬件描述语言：描述数字系统的结构、行为、功能和接口

ABEL

AHDL

Verilog HDL

VHDL

} **IEEE标准**

Verilog

IEEE Std 1364-1987
IEEE Std 1364-1995
IEEE Std 1364-2000

Hardware Description Language (HDL)

设计开始之前都应当有一个硬件框图。

verilog 标识符

合法的标识符：

a，fft，and_4，max2uc ，buf$m

非法的标识符：

21A，max#2uc，a-b

- 第一个字符不能是数字；
- 区分大小写，不能和Verilog的保留字相同；
- 关键字必须小写；
- 存盘文件名应与设计的模块名相同。
- 各完整语句均以“;”结尾，注释语句（//，/* */），不参与编译。

verilog 逻辑值

Verilog 语言规定了信号的4种基本逻辑值：

0	逻辑0、逻辑非、低电平
1	逻辑1、逻辑真、高电平
x或X	不确定的逻辑状态
z或Z	高阻态

5.5.1 Verilog模块的基本结构

模块由两部分构成：

- ✓ 描述接口
- ✓ 描述功能

设计模块

```
module      模块名（端口列表）；
input       ;
output      ;      I/O端口模式
wire        ;
reg         ;      信号类型说明
assign      ;（连续赋值）
always @（过程赋值）
            ;（元件例化）
endmodule
```

功能描述

例：一位全加器

模块名

分号

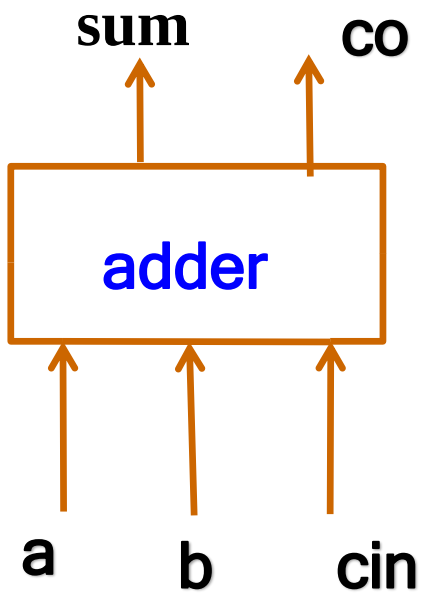
端口列表

端口说明

功能描述

无分号

```
module  adder ( a, b, cin , sum,  co ) ;
        input  a , b , cin ;
        output  sum ,  co ;

     assign {co,sum} = a+b+cin;

endmodule
```

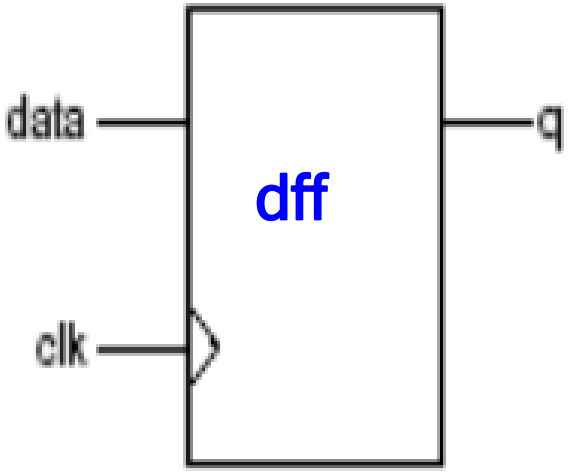


注意：

- 模块名必须与文件名相同。
- 模块名不能用中文。
- 数据类型默认wire 型 。

例：一位d寄存器

```
module dff ( clk, data , q ) ;
        input   clk, data;
        output  q;
        reg q ;

      always @ (posedge clk)
        begin
           q <= data ;
        end

endmodule
```



数据类型说明

功能描述

begin end 括起来语句块

5.5.1 Verilog的模块说明及数据类型

模块说明：描述系统的输入、输出端口。

- 模块说明：

```
module  模块名  （端口名）;
```

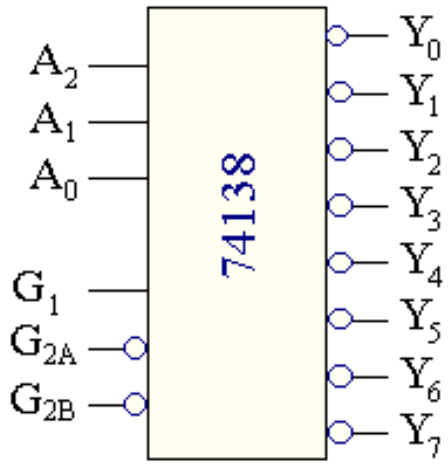
- 端口I/O说明：

```
端口模式   端口名;
```

方向定义(3种)	含义
input	输入 (只读）
output	输出 （在assign中只写）
inout	输入输出 （只能用assign）

```
module  adder ( a, b, cin , sum, co ) ;
        input [2:0]  a,b;     //逻辑矢量
        input   cin;
        output  [2:0]  sum;
        output co;
```

```
module dec38 ( a2,a1,a0,g1,g2a,g2b , y) ;
    input a2,a1,a0,g1,g2a,g2b ;
    output [7:0]  y ;
```



符号

数据类型类型说明:

信号类型　　信号名;

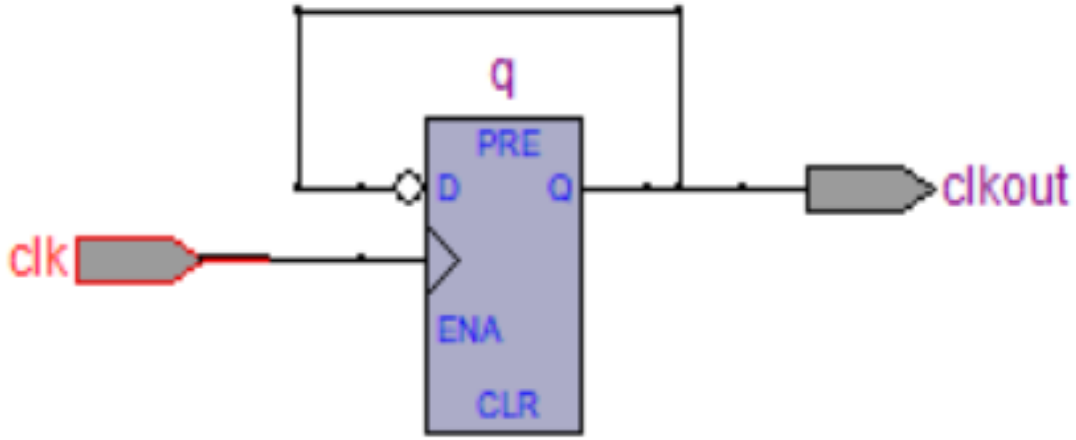
类型定义	含义
wire	线网型
reg	寄存型
parameters	运行时的常数

```
wire  a;
wire [15: 0]  busa;  //16位数据总线
reg  a;
reg [3: 0]  v;
reg [15:0] MEM [0:1023];
      // 1K 个 16 位的存储器
```

- wire: 结构化器件的物理连线。端口如果没有声明，默认为wire型。
- reg：对存储单元或组合逻辑的描述，always 中被赋值的信号必须用reg 类型的变量。
- reg: 输入和双向端口不能声明为reg型。

```
module divf (  clk , clkout ) ;   //二分频
        input   clk ;
        output  clkout;
        reg q ;

       always @ (posedge clk)
        begin
            q <= ~q;
        end

       assign  clkout = q ;

endmodule
```

编译之后，点击
Tool>>Netlist Viewers>>RTL Viewers



■ 数据类型类型 parameter 说明:

- parameter：符号常量，其定义只在本模块内有效。
- parameter: 还可以在模块实例引用时，改变引用模块或实例中已定义的参数。

```
module mux (  i1, i2, sel , out ) ;
   parameter width=4  ;
   input sel ;
   input  [width-1:0] i1,i2;       //用参数来说明wire 的位宽
   output [width-1:0]  out ;

   assign   out = sel ? i1 : i2 ;
endmodule
```

- parameter: 可以用参数来表示存储器的大小。

```
parameter wordsize = 16;
parameter memsize = 1024;
reg  [wordsize-1:0]  MEM3 [0: memsize-1];
      // 1024个16位的MEM3存储器
```

数的表示：

<二级制位宽>’ <进制><数字>;

```
16                  // 32位十进制数


8’b0001_0000

8’d16

8’h10


32’bx                       //32位x

2’b?                // ?、z、Z都表示高阻
```

5.5.2 Verilog的 运算符

- 算术操作：+, -, *, /, %（取模）
- 关系操作：>, <, >=, <=, ==（相等）, !=（不等）
- 逻辑操作：&&, ||, !,
- 位操作：~（非）, &, |, ^（异或）, ~^（同或）
- 归约操作：&, ~&, |, ~|, ^, ~^
- 移位操作：<<, >>
- 条件操作：?:
- 连接和复制：{ , }

使用（ ）改变优先级

➢ 逻辑非！、按位反~

!的结果只有一位1，0或x。~的结果与操作数的位数相同。

```
rega = 4'b1011;
bit = ~rega;        // bit = 4’b0100
log = !rega;      // log = 0
```

➢ 逻辑运算、按位运算、归约运算。

逻辑操作、归约操作的结果只有一位。

```
a =  4'b0100;
b  = 4'b0111;;
a && b  ;          //  结果= 1’b1
a & b;       //  结果 = 4’b0100
&a   ;             // 结果= 1’b0，逐位与，判断是否全1
```

➢ 逐位缩减（一元归约）运算符。

单目运算符，对操作数的所有位逐位进行，结果只有一位。

```
& ( ~& )
// ~&与&的运算结果相反
|  ( ~| )
^        // xor
 ~^     //xnor
```

```
rega = 4'b0100;
 & rega ;          //  1'b0，判断是否全1
 | rega ;          //  1'b1，判断是否全0
^ rega ;           //  1'b1，判断1的奇偶
 ~& rega;          // ~(0&1&0&0) = 1'b1，判断是否有0
 ~| rega;          // ~(0|1|0|0) = 1'b0，判断是全0
```

➢ 移位运算符。

>> 右移 、<< 左移。

```
a  = 8'b11010001;
a  << 4 ;          // a = 8'b00010000

b  = 8'b11010001;
b  >>  4 ;         // b = 8'b00001101
```

- 位拼接运算符。

{ } ，将两个或多个信号的某些位拼接起来。不允许连接非定长常数。

```
wire [5:0]  a, b ;
wire [1:0] c ;
wire [5:0] d ;
assign a = 6'b101101 ;
assign b = 6'b111000 ;
assign c = 2'b11 ;
assign d ={a[5], b[2:0] , c} ;  // d=100011
```

×

```
assign b = { d ,1 }
//不允许拼接非定长常数
```

。

运算符说明：

- 自动调整位宽。运算表达式结果的长度由最长的操作数决定。
- 操作结果的长度：由赋值左端目标长度决定。

```
wire [3:0]  arc, bar, crt;
wire [5:0]  frx;
assign arc = bar + crt;      //  长度4位，加法的溢出部分被丢弃
assign frx = bar + crt;       //长度6位，任何溢出的位存储在frx [ 4 ]中

4'b0110 ^ 5'b10000   // 结果  5' b10110
```

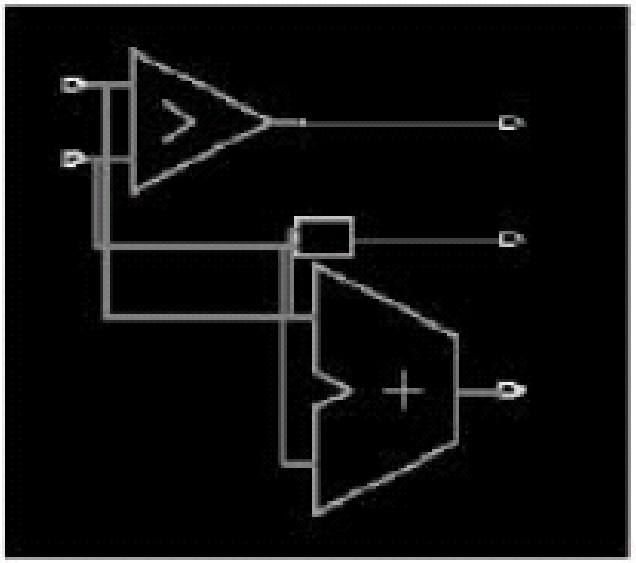
- 位操作、归约操作和逻辑操作

a = 1011
b = 0010

bit-wise	unary reduction	logical
a | b = 1011	| a = 1	a || b = 1
a & b = 0010	& a = 0	a && b = 1

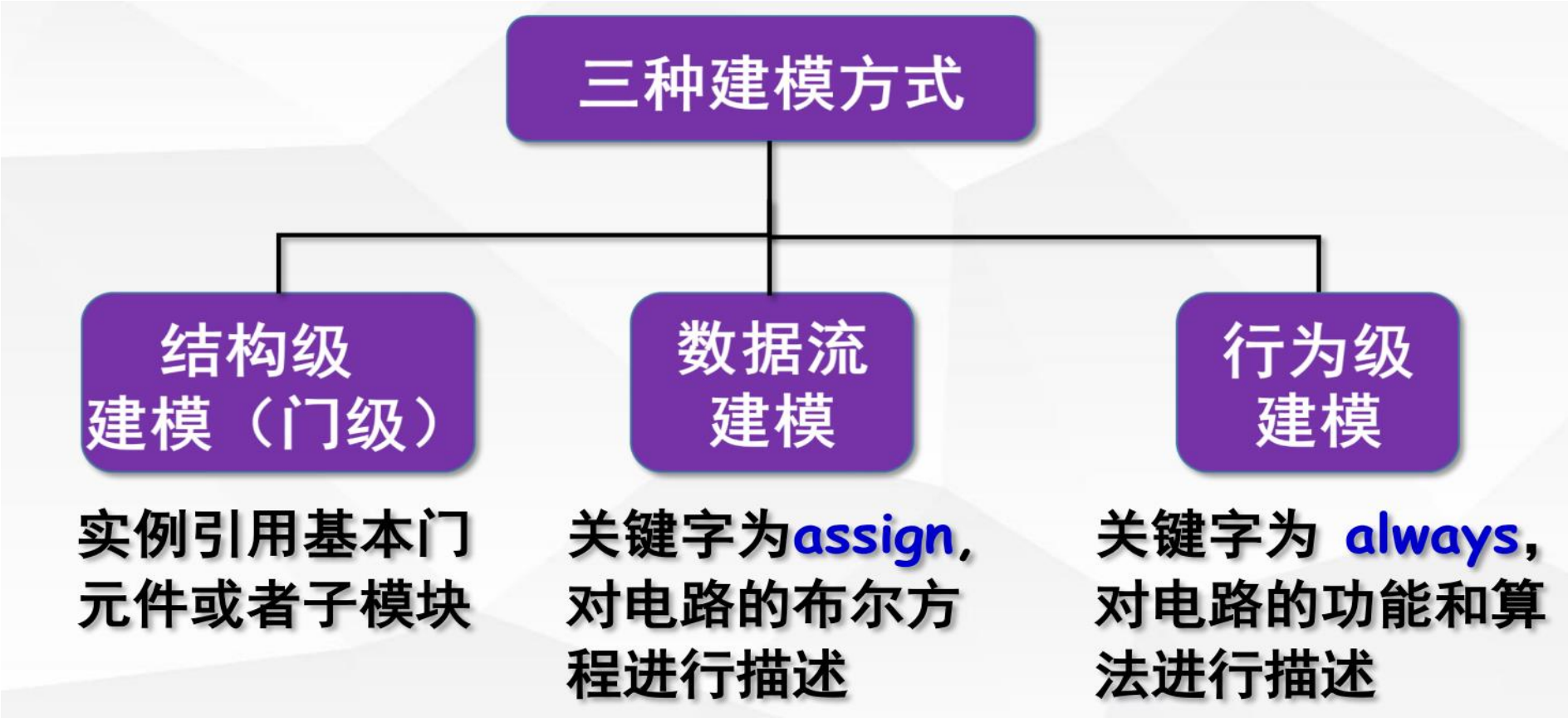
- 算术和关系操作

```
assign  c = a + b ;
assign  bigger = a > b ;
assign  eq = ( a == b ) ;
```



5.5.3 Verilog的功能描述语句

功能描述：assign，always，元件例化，是并发语句。

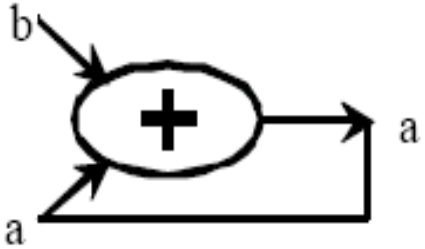


一、 连续赋值 assign 语句：

assign 赋值目标 = 表达式；

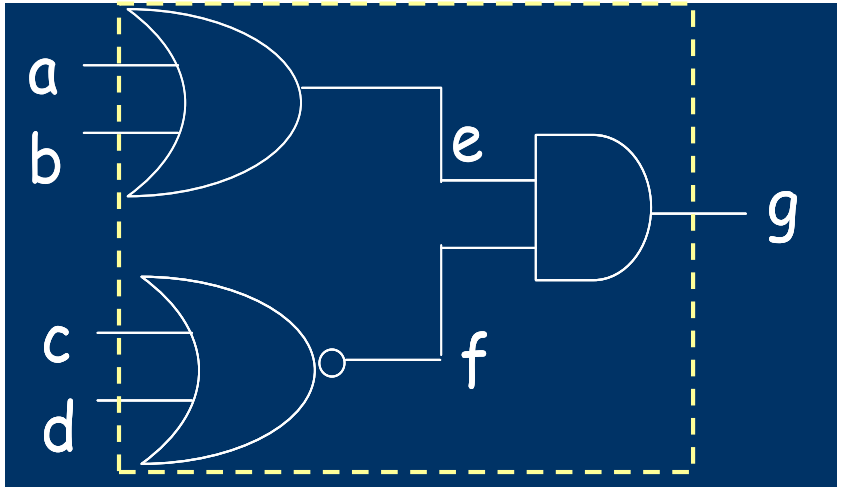
- 连续赋值运算符 = 。
- assign后面只能是一个表达式。
- 更新赋值，避免出现反馈。

assign a = b + a ; ×



- 在assign语句中输出引脚是只写，inout型引脚只能用在assing语句中。
- 赋值目标必须是wire型的。
- 多个 assign 之间是并行关系。
- assign 只能逐句使用，只能完成简单的组合逻辑。
- 不能对同一个信号进行assign 多次赋值（多源驱动） 。

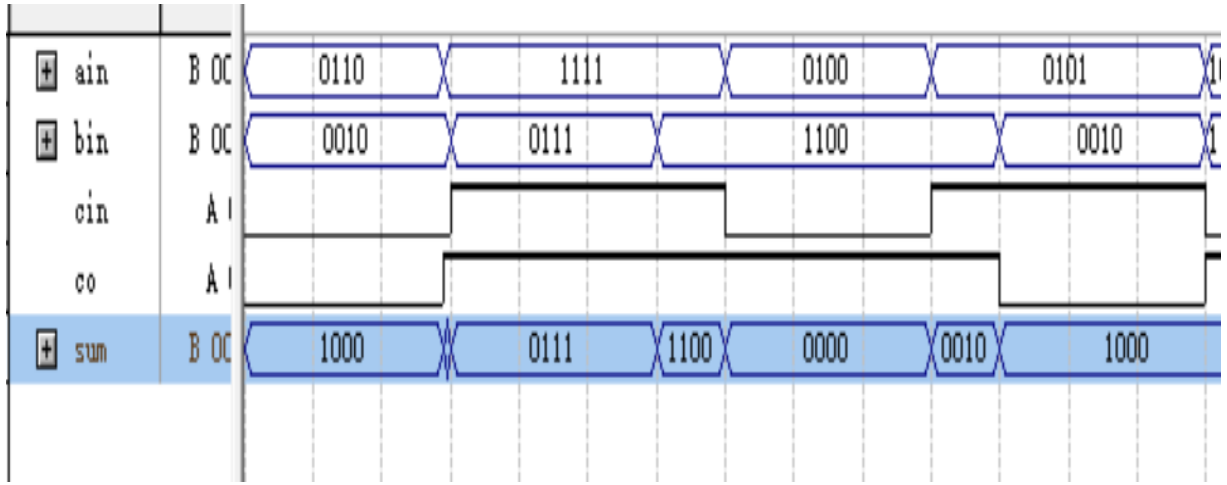
```
module exp1(  a ,b ,c ,d , g  ) ;

input  a , b ,c ,d ;
output   g ;
wire  e , f ;

assign  g = e & f  ;
assign  f = ~ (c | d)  ;
assign  e = a | b  ;

endmodule
```



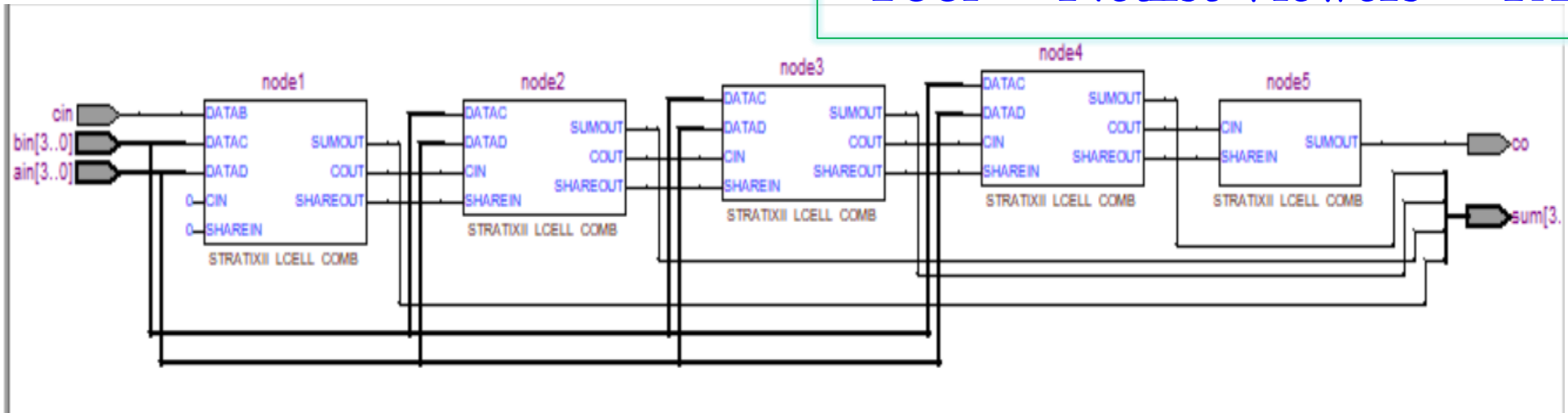
并行行为

实现 四位二进制全加器

```
module adde ( ain ,bin ,cin , sum, co  );

input [3:0]  ain , bin ;
input  cin ;
output [3:0]  sum ;
output co ;

assign {co , sum } =  ain+bin+cin ;

endmodule
```



编译之后，点击
Tool>>Netlist Viewers>>RTL Viewers



二、过程赋值always语句块：

（1）语法：

过程赋值语句 always 后面是语句块。可描述所有的功能。

赋值目标必须是 reg 型的。

```
always @(敏感信号表)
  begin
        =      ;
       <=      ;
     if 语句
     case 语句
  end
```

➢ 过程赋值运算符

- = ，阻塞赋值。立即更新数值。
- <= ，非阻塞赋值。延迟更新数。所有的赋值语句同时完成更新。
- 组合逻辑用 = 赋值。
- 时序逻辑用 <= 赋值 。

敏感信号表：

- ✓ 敏感信号的变化才能启动进程。
- ✓ 组合逻辑中，所有输入都作为敏感信号,否则仿真结果和综合结果会不一致。
- ✓ 边沿敏感、电平敏感。

```
always @ (敏感信号表)
  begin
      =      ;
     <=      ;
    if 语句
    case 语句
  end
```

上升沿触发：

```
always @(posedge 信号名)
```

下降沿触发：

```
always @(negedge 信号名)
```

电平触发：

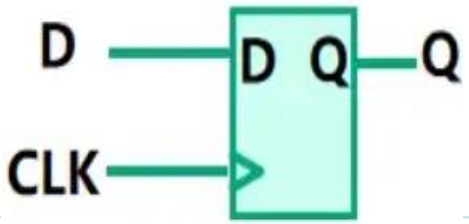
```
always @(信号名)
```

```
always @ ( reset or set or d)
always @ ( reset , set , d)
```

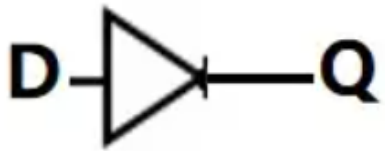
所有输入信号均触发：（组合逻辑）

```
always @( * )
```

```
always @ (posedge CLK)
  begin
     Q = D ;
  end
```




```
always @ (D)
  begin
     Q = D ;
  end
```



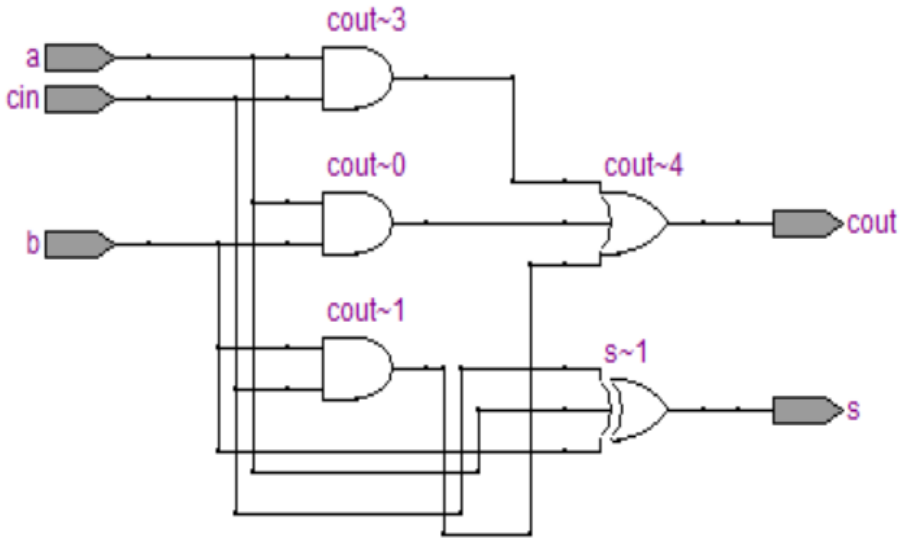
实现 一位二进制全加器。

```
module adder ( a , b, cin, s, cout );

input a , b, cin;
output s, cout;
 reg s , cout;

always @ ( *)
   begin
     s = a^b^cin ;
     cout = a&b | b&cin | a&cin :
   end

endmodule
```

reg 数据类型

组合逻辑通配符 *

顺序执行




```
module exp ( a , b, cin, s, cout );

 input a , b, cin;
 output s, cout;

asssign  cout = a&b | b&cin | a&cin ;
assign   s = a^b^cin ;

endmodule
```

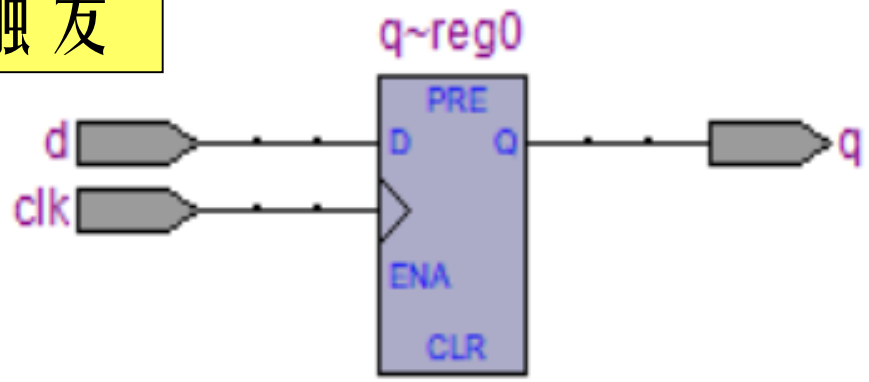
wire 数据类型

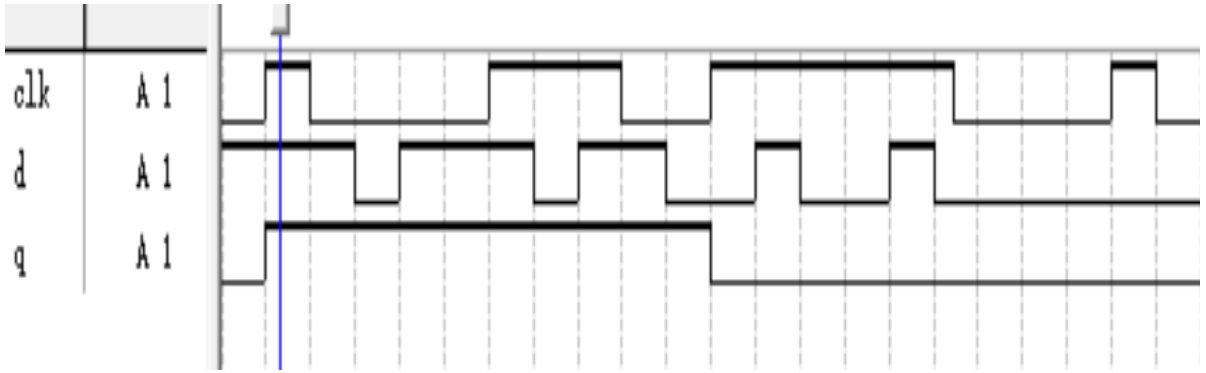
并发执行

```
module qff ( clk, d, q );

input clk, d;
output q ;
 reg q;

 always @ (  posedge clk )
    begin
          q <= d ;
     end
 endmodule
```

时钟上升沿触发

综合后生成寄存器（register）





（2）阻塞赋值与非阻塞赋值的比较

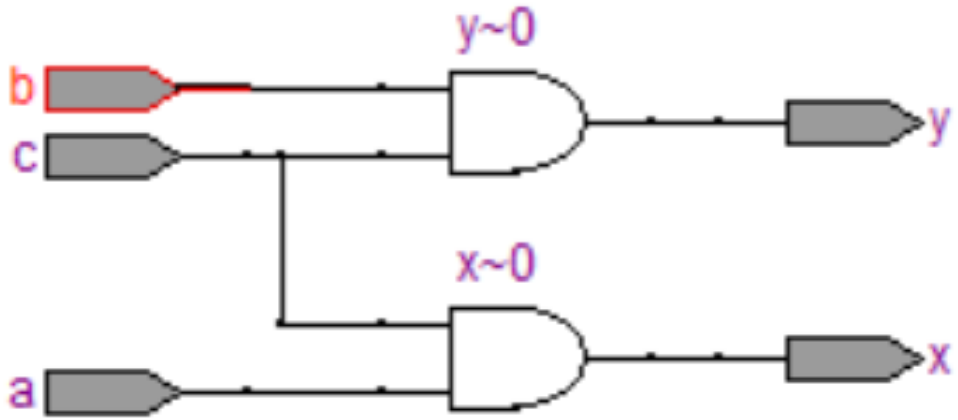
注意：

- ✓ 不要在组合逻辑中使用 <= 赋值；
- ✓ <=用于实现时序逻辑；
- ✓ 在同一个always块里不要混用两种赋值语句；

■ 阻塞赋值

在实现组合逻辑的always块里用阻塞赋值

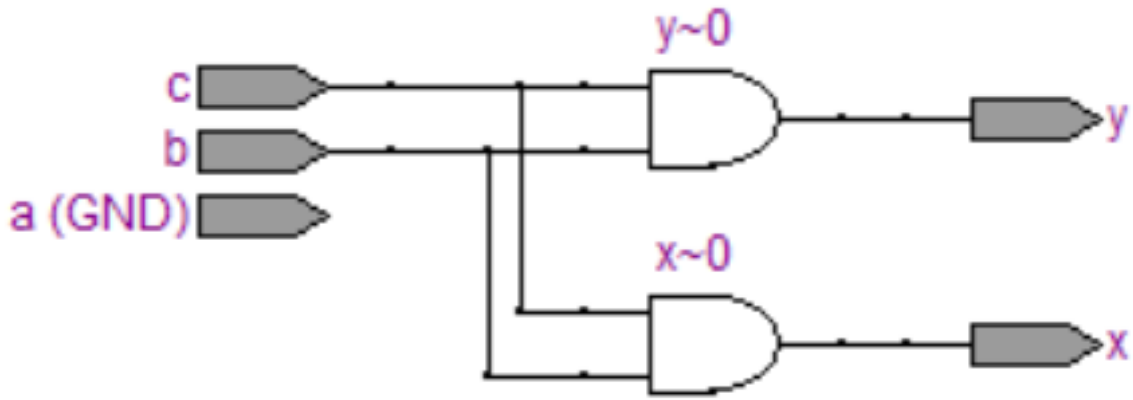
```
module expe ( a , b, c , x, y );

 input a , b, c;
 output y , x;
 reg d, x, y;

 always @ ( * )
    begin
      d = a;
      x = c & d ;
      d = b;
      y = c & d ;
    end

 endmodule
```



-- 结果: x = ca
y = cb

```
module expe ( a , b, c , x, y );

 input a , b, c;
 output y , x;
 reg d, x, y;

always @ ( * )
   begin
     d <= a;
     x <= c & d ;
     d <= b;
     y <= c & d ;
   end

endmodule
```

■ 非阻塞赋值



读出（begin中）

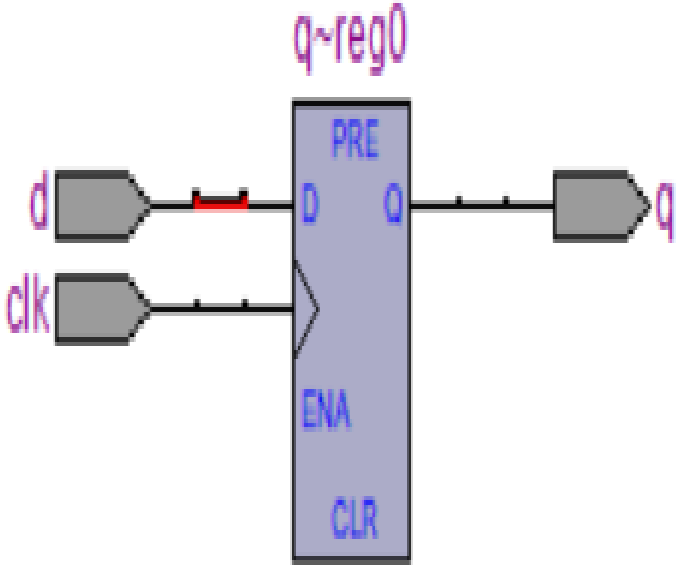
更新（end后）

不要在always中对同一信号多次非阻塞赋值

- - 结果: x = cb
  y = cb

```
module qff  ( clk, d , q );

 input clk, d;
 output q ;
 reg a, b, q;

 always @ ( posedge clk)
    begin
         a = d ;
         b = a ;
         q = b ;
     end
 endmodule
```

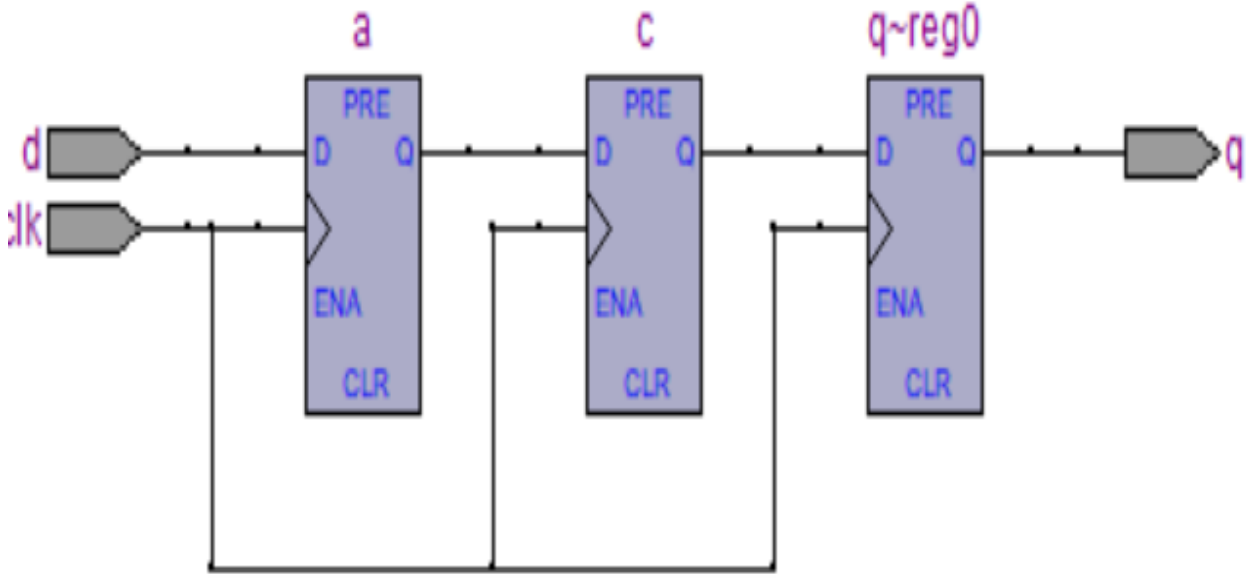
综合后生成一位寄存器



阻塞赋值

```
module qff  ( clk, d , q);

 input clk, d;
 output q ;
 reg a, c, q;

 always @ ( posedge clk)
    begin
         a <= d ;
         c <= a ;
         q <= c ;
     end
 endmodule
```

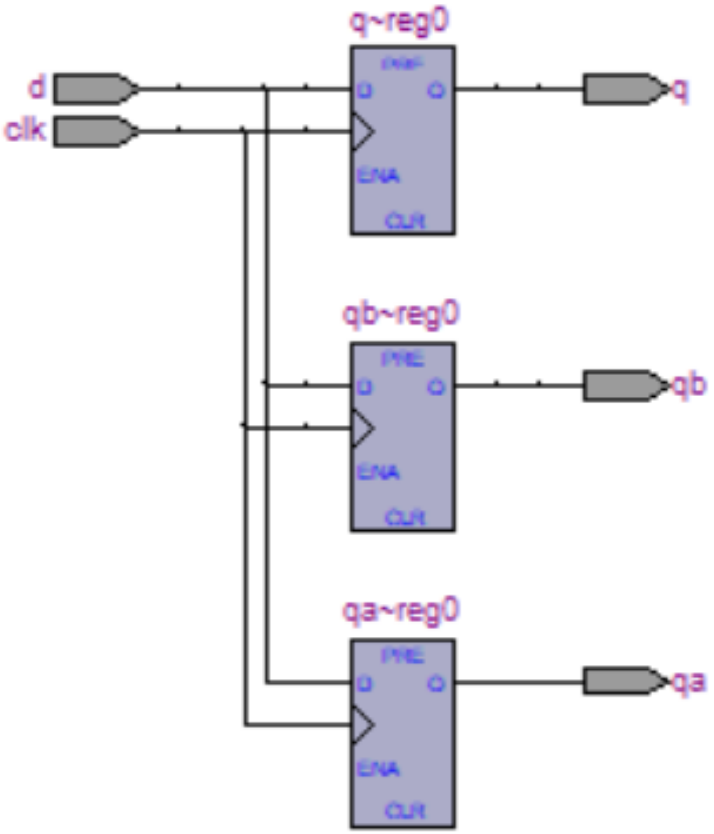
综合后生成移位寄存器



非阻塞赋值

```
module qff  ( clk , d ,q ,  qa , qb );

 input clk, d;
 output qa ,qb , q  ;
 reg qa, qb, q;

 always @ ( posedge clk)
    begin
         qa = d ;
         qb = qa ;
         q = qb ;
     end
endmodule
```

综合后生成3个寄存器并联



阻塞赋值，

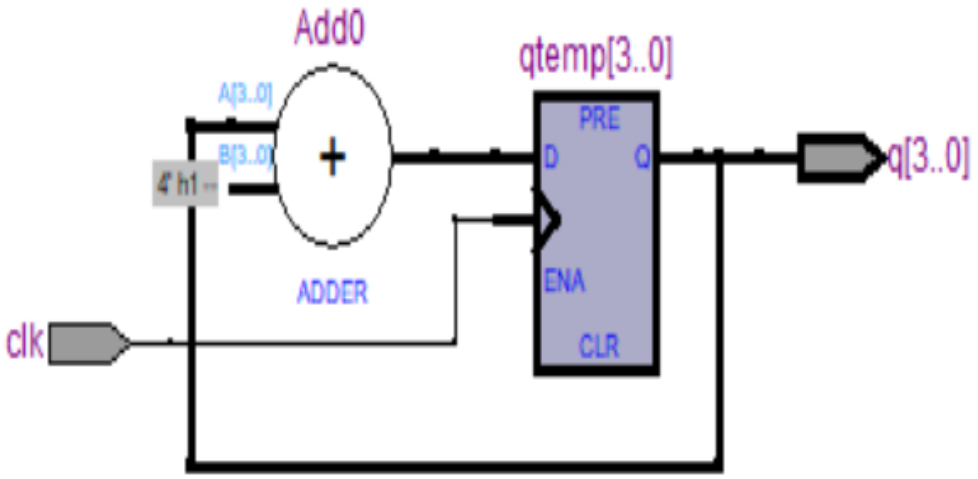
输出引脚，

存储器不要使用阻塞赋值

(3) assign 和 always 并发

注意:

- ✓ 不要在一个aways 中同时使用 = ， <= 赋值；
- ✓ 不要对一个信号多次驱动;
- ✓ 信号可以在多个always 、assign 间传递;
- ✓ 不能在多个always，与assign中对同一个信号多次赋值




```
module  count4 ( clk , q );

    input  clk;
    output [3:0]  q;
    reg [3:0]   qtemp ;

     always @ (posedge clk)
       begin
          qtemp <= qtemp+1;
       end

    assign q = qtemp ;

endmodule
```

4位二进制计数器

(4) 转向控制语句

通过条件控制决定是否执行一条或几条语句，或重新执行一条或几条语句，仿真时顺序进行。

主要有:

if 语句、case 语句、for 语句

if 、case、for 语句必需在 always 块中。

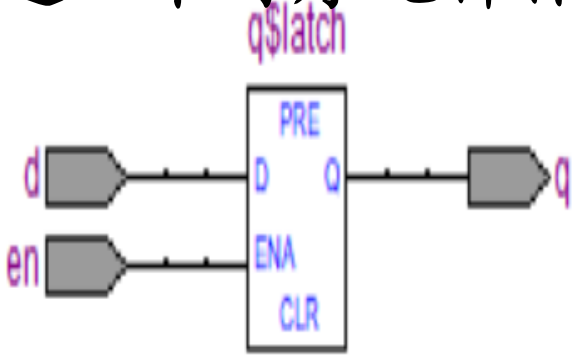
1) if 语句

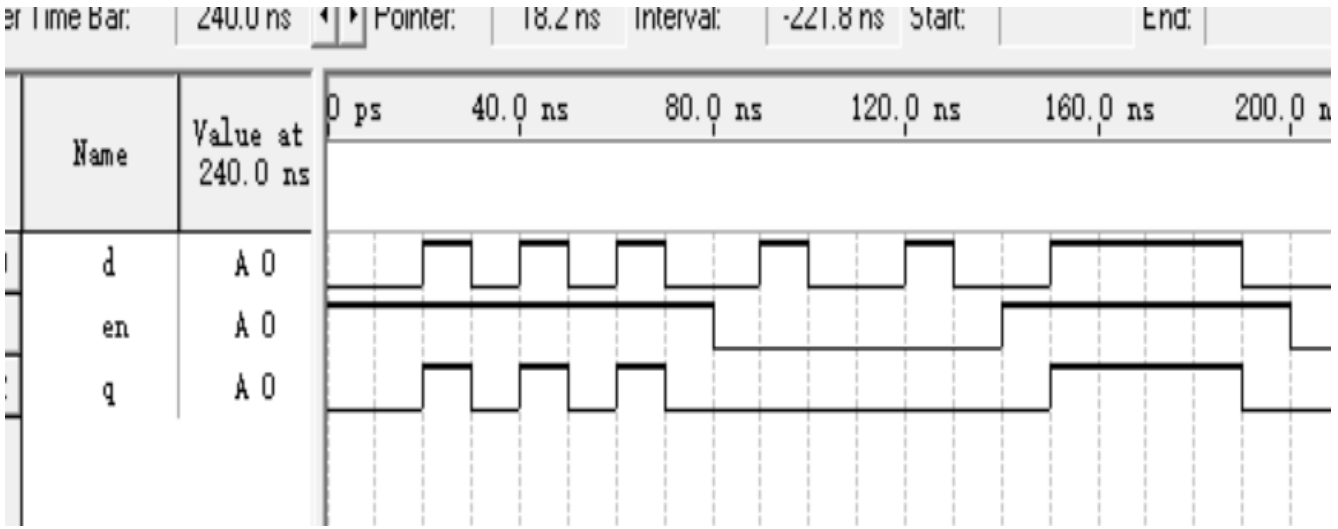
• if 语句的门闩控制（不完全if ）

对于不完全的if语句，verilog综合器将引进一个时序元件保持当前状态值。用于锁存器。不建议用不完全的if。

```
module latch ( en,  d , q );

input en, d;
output q ;
reg q;

always @ ( en, d )
   begin
     if ( en )
        q <= d ;    // else  q <= q ;
    end
endmodule
```

电平触发



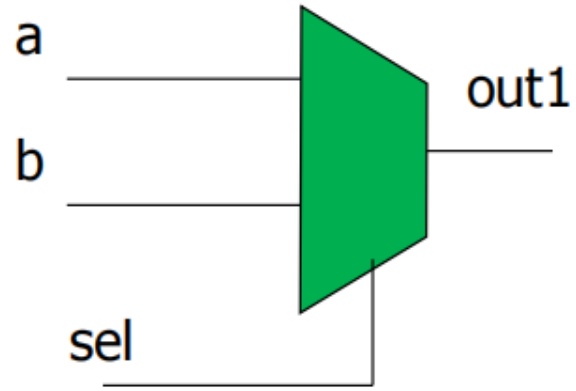


2) if else 语句

- if 语句中最先出现的条件优先级最高（自上而下优先）。
- 可以有多个else if，但必须有一个else（组合逻辑，所有分支条件都要对变量赋值）。
- 良好的代码风格，建议写成完整的 if · · · · · else 结构

```
if （条件表达式1)
    begin
        语句块1;
    end
else if (条件表达式2 )
    begin
        语句块2;
    end
else if  (条件表达式3 )
    begin
        语句块3;
    end
else
    begin
        语句块n;
    end
```

例：2选1选择器。




```
module mux2_1  (  a, b, sel , out1 );

input a , b, sel;
output out1;
reg  out1;

always @ ( * )
  begin
    if ( sel )
      out1 = a ;
    else
      out1 = b ;
  end

endmodule
```

```
module mux2_1  ( a, b, sel , out1 );

  input a , b, sel;
  output out1;

assign  out1 = sel ? a , b ;

endmodule
```

3) case 语句

根据某个表达式的值来选择执行体。 无优先级。

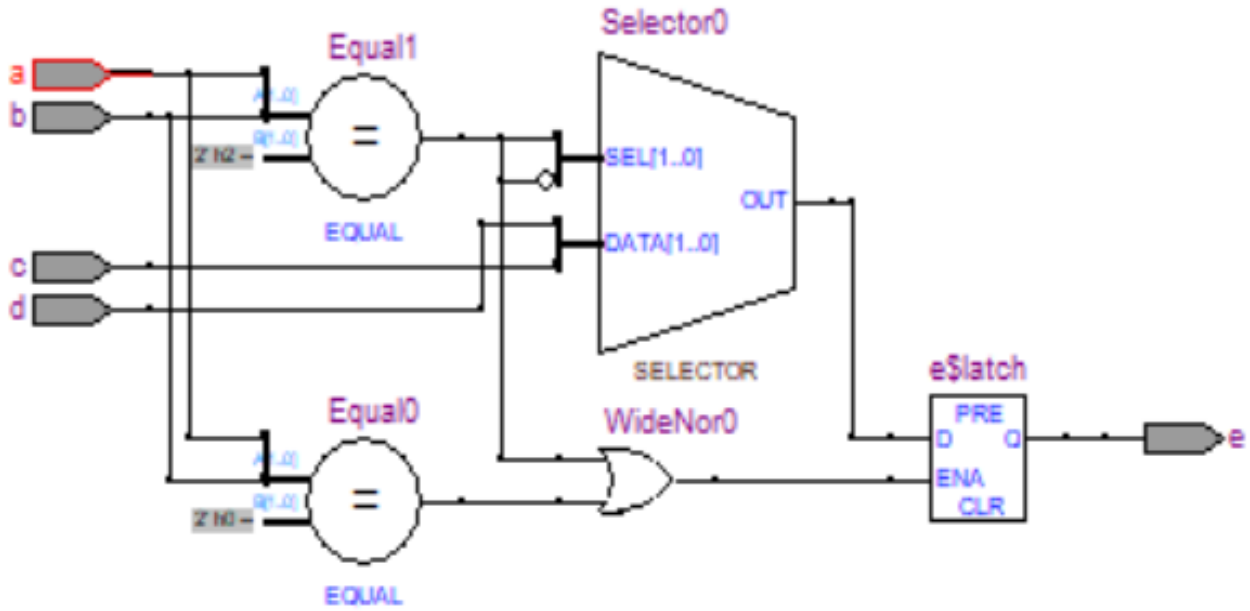
```
case (选择表达式)
    值1:
        begin
         语句块1;
        end
    值2:
        begin
         语句块2;
        end
 . . . . . .
   default:
        begin
         语句块n+1;
        end
  endcase
```

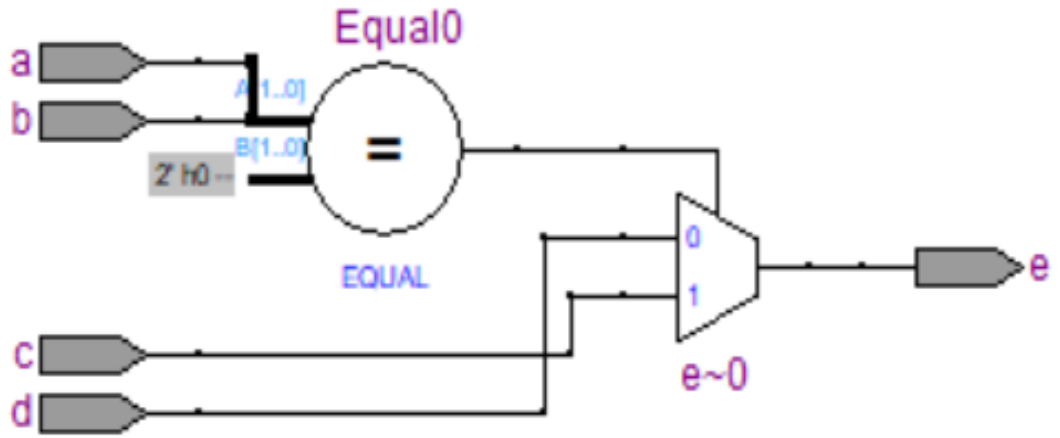
- 选择表达式必须使用（ ）。
- 分支条件须在表达式范围内，且不能重合。
- 执行时必须选中且只能选中一个分支。
- 组合逻辑，每个条件都有赋值，描述真值表。
- 所有值必须列举穷尽，组合、时序都写

default。

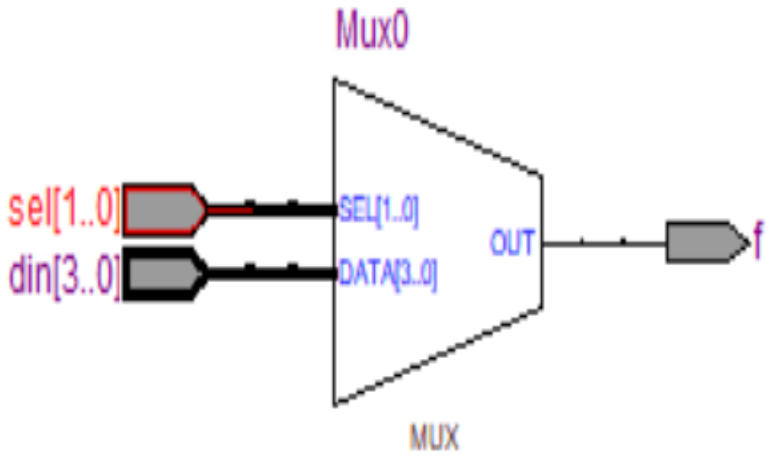
```
module mux2 ( a , b , c , d , e ) ;

input a , b , c ,d  ;
output e;
reg e ;

always @ ( * )
  begin
     case ( {a,b} )
        2'b00 : e = c ;
        2'b10 : e = d ;
     endcase
  end

endmodule
```

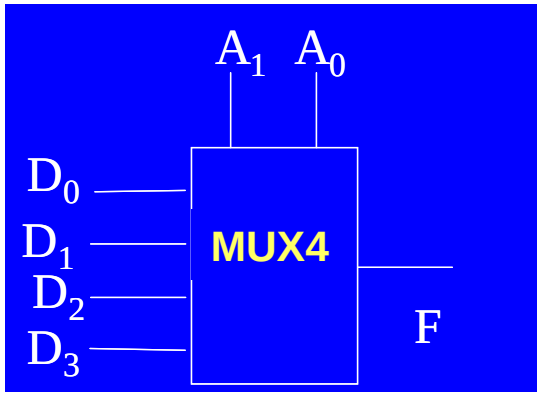
default: e = d ;





例　4选1多路选择器描述方式





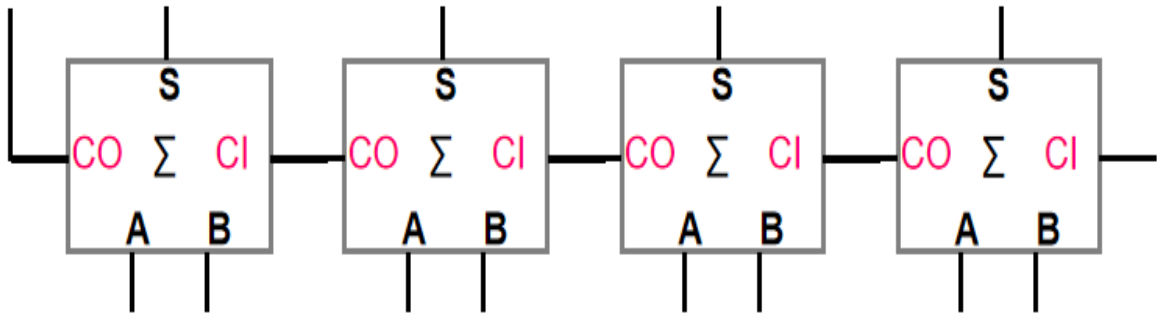

```
module mux4 ( sel , din , f) ;

 input [1:0] sel;
 input [3:0] din ;
 output f ;
 reg f;

  always @ ( * )
    begin
       case (sel)
          2'b00:    f = din[0] ;
          2'b01:    f = din[1] ;
          2'b10:    f = din[2] ;
          default:  f = din[3] ;
       endcase
    end

endmodule
```

4) for 语句 (不提倡使用）

```
for (表达式1，表达式2，表达式3)
   begin
        语句块;
            …   ..   ..    …
   end
```

- 表达式1：设置循环变量初值，必须定义一个 i 循环变量;
- 表达式2：循环执行判断条件;
- 表达式3：更新循环变量，只能用类似 i=i+1 或 i=i-1;
- 循环次数不定的循环，无法被综合 ;
- 注意：初值、步长、循环条件中不能有变量 。

例 n位串行加法器




```
module adde (  ain , bin , cin , sum , co  ) ;

parameter n=4 ;
input [n-1:0] ain , bin  ;
input cin ;
output [n-1:0] sum ;
output co ;
reg [n-1:0] sum ;
reg co ;
reg [n:0] ci ;
integer k ;  // 32根线

always @ ( * )
  begin
     ci[0] = cin ;
     for ( k=0 ; k<n ; k=k+1 )
        begin
           sum[k] = ain[k] ^ bin[k] ^ ci[k] ;
           ci[k+1] = ain[k] & bin[k] | (ain[k]^bin[k])&ci[k] ;
        end
      co = ci[n] ;
  end

endmodule
```