第三节 Verilog HDL的结构描述模块

从Verilog HDL的描述风格看，分为结构描述、数据流描述、行为描述以及混合描述。本节介绍逻辑电路的结构描述方式。

一.结构描述的概念

所谓结构描述就是通过调用逻辑元件、描述它们之间的连接来建立逻辑电路的Verilog HDL模型。

狭义理解：如何将传统意义上的“逻辑原理图”转换为 Verilog HDL的描述。

二.门级结构描述（门级建模）

门级结构描述就是利用Verilog HDL内置的基本门级元件以及它们之间的连接来构筑逻辑电路的模型。

“基本门级元件”是一种特殊的模块，由Verilog HDL语言本身提供，不需要用户定义。

三.Verilog HDL 内置基本门元件

⟹ 多输入门

	关键字
与门：	and
与非门：	nand
或门：	or
或非门：	nor
异或门：	xor
异或非门：	xnor

只有一个输出

多个输入

元件模型：

<门级元件名> (<输出>, <输入1>, <输入2> , …… , <输入n>)

多输出门

关键字

缓冲器：　　**buf**

非门：　　**not**

多个输出

只有一个输入

元件模型：

<门级元件名> (<输出1>, <输出2>, …… , <输出n>, <输入>)

三态门

关键字

高电平使能缓冲器:	**bufif1**
低电平使能缓冲器:	**bufif0**
高电平使能非门:	**notif1**
低电平使能非门:	**notif0**

元件模型

<元件名> (<数据输出>，<数据输入>，<控制输入>)

实现三态输出

四.Verilog HDL 内置基本门元件的调用

门级元件实例语句的格式：

<门级元件名>　<实例名>　(端口连接表)；

应按照各元件模型中输出、输入、控制的顺序描述信号的连接。

当对同一个基本门级元件进行多次调用时，可采用下面的元件实例语句格式：

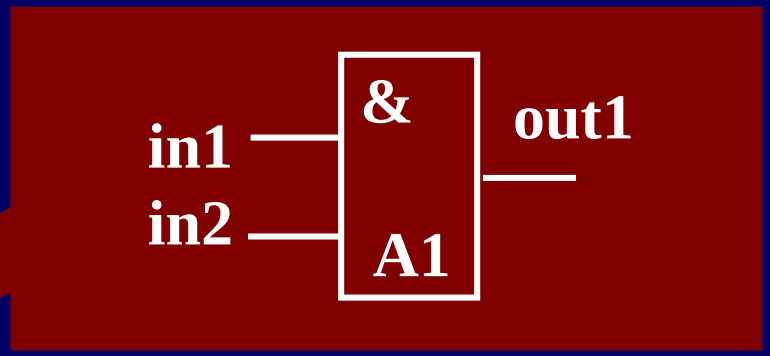
<门级元件名> <实例名1> (端口连接表1)，
<实例名2> (端口连接表2)，

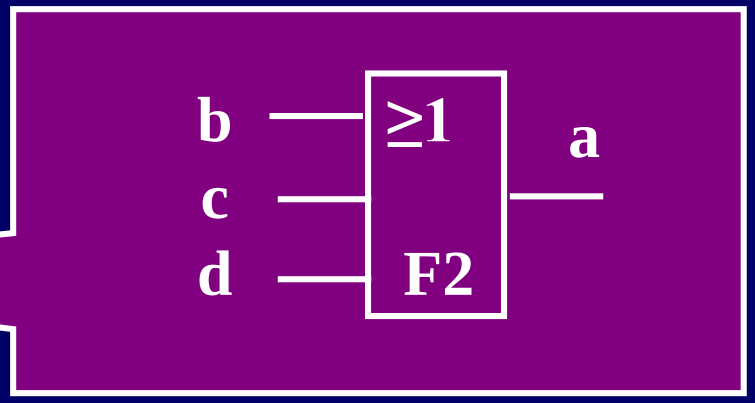
……

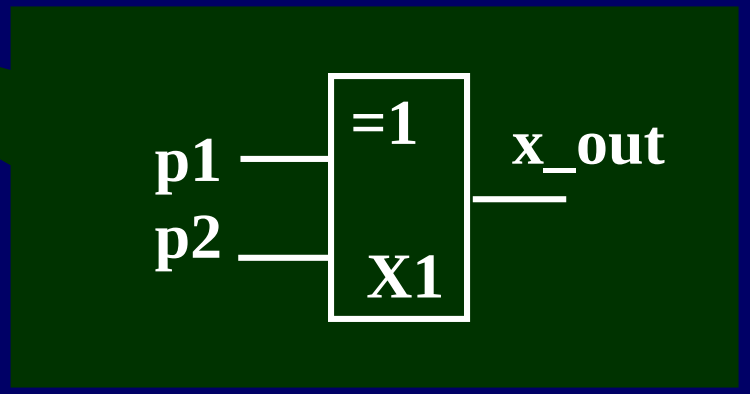
<实例名n> (端口连接表n)；

门级元件实例语句及其对应的逻辑示意图

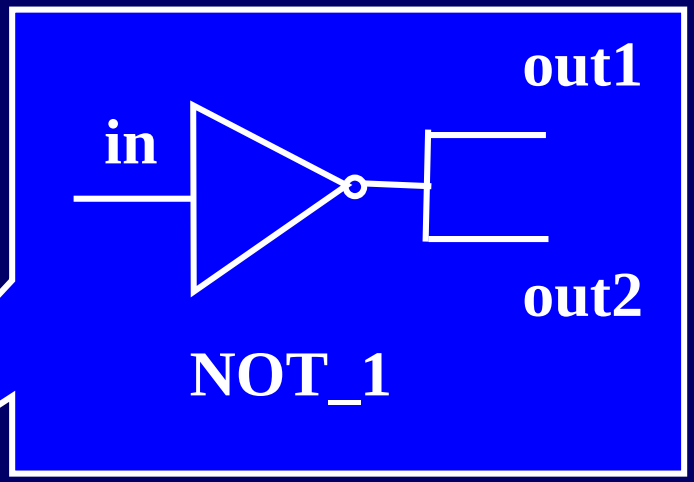
```
and  A1 (out1, in1, in2) ;

or    F2 (a, b, c, d) ;

xor  X1 (x_out, p1, p2) ;
```





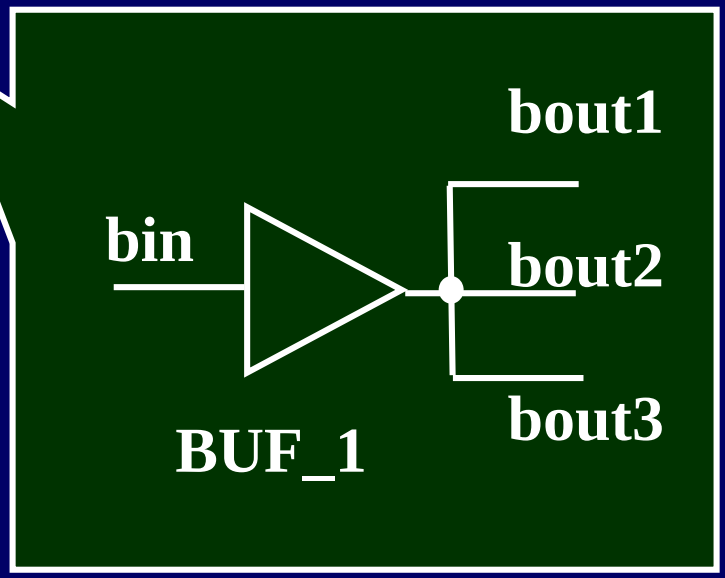


门级元件实例语句及其对应的逻辑示意图???



not NOT_1 (out1, out2, in) ;

buf BUF_1 (bout1, bout2, bout3, bin) ;



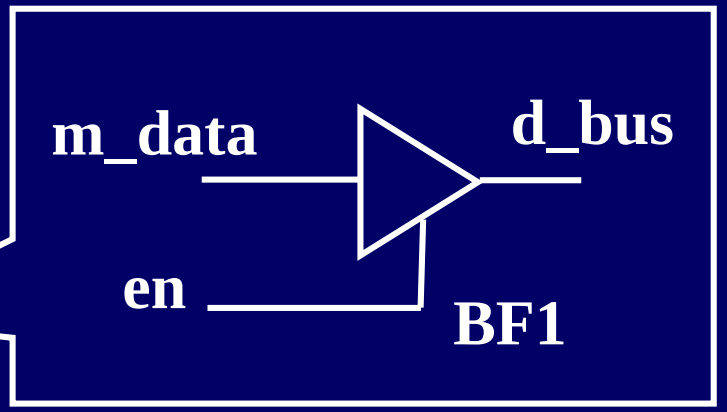
门级元件实例语句及其对应的逻辑示意图

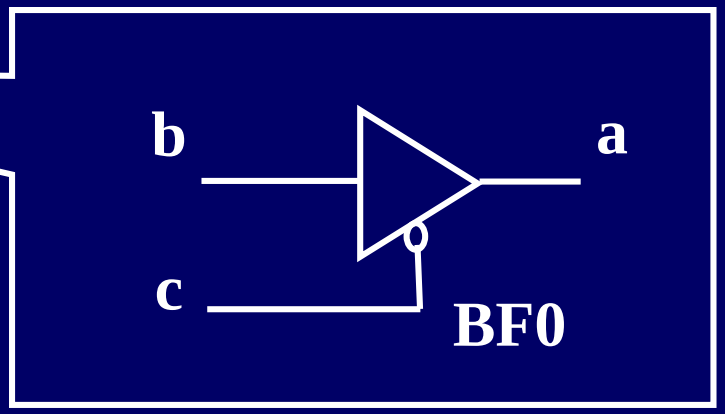
bufif1 BF1 (d_bus, m_data, en) ;

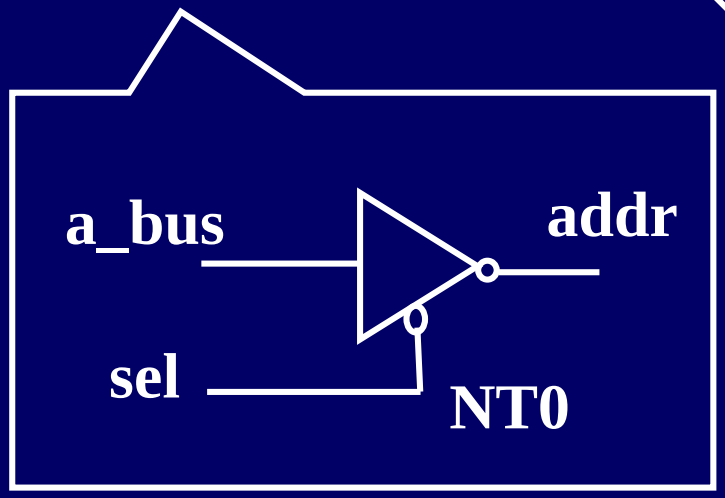
bufif0 BF0 (a, b, c) ;

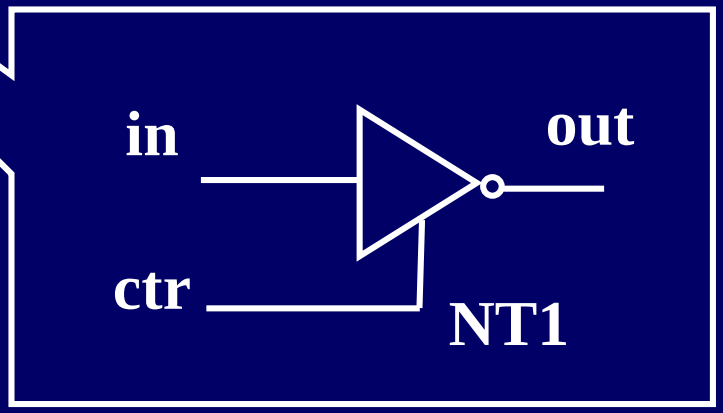
notif1 NT1 (out, in, ctr) ;

notif0 NT0 (addr, a_bus, sel) ;









五. Verilog HDL门级结构描述模块的设计模型

Verilog 结构描述（门级建模）模块基本结构

module 模块名 （端口列表）;

端口定义
input 输入端口
output 输出端口

数据类型说明
wire

门级建模描述
and u1 (输出，输入1，…输入n)
not u2 (输出1，…输出n，输入)
bufif1 u3 (输出，输入，控制)
…

endmodule

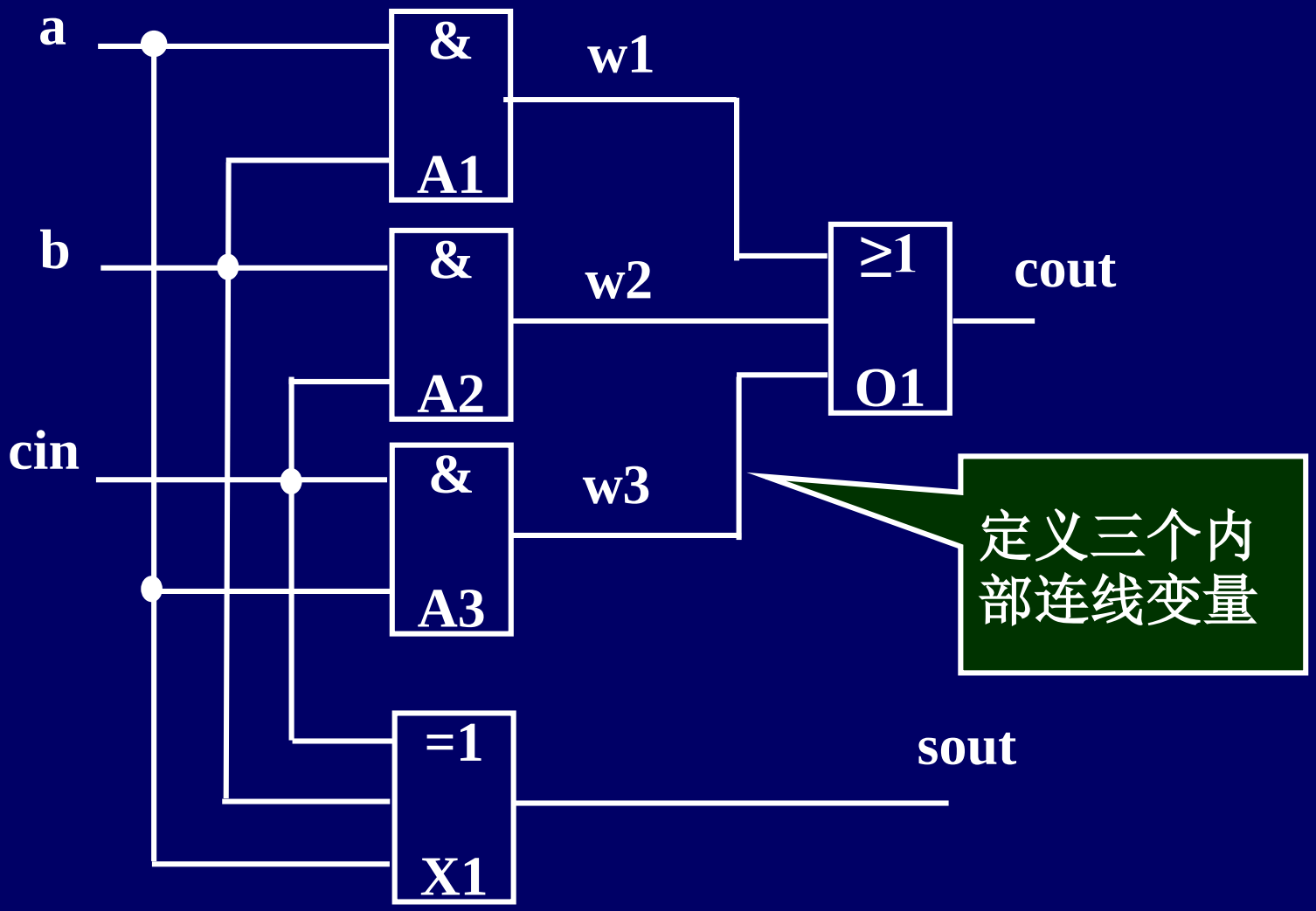
多输入门

多输出门

三态门

六. Verilog HDL门级建模举例

例：对下列逻辑电路进行Verilog HDL门级建模。



```
//图示逻辑电路的门级建模
 module full_adder (cout, sout, a, b, cin) ;
     output   cout, sout ;
     input    a, b, cin ;
     wire     w1, w2, w3 ;
//元件实例语句

     and  A1 ( w1, a, b ) ,
          A2 ( w2, b, cin ) ,
          A3 ( w3, a, cin ) ;
     or   O1 ( cout, w1, w2, w3 ) ;
     xor  X1 ( sout, a, b, cin ) ;
 endmodule
```

第四节 Verilog HDL的数据流描述模块

一.数据流描述

根据信号（变量）之间的逻辑关系，采用持续赋值语句描述逻辑电路的方式，称为数据流描述。

狭义理解：将传统意义上的“逻辑表达式”，运用Verilog HDL中的运算符，改变成持续赋值语句（assign语句）中的表达式。

二.Verilog HDL数据流描述模块的设计模型

Verilog 数据流描述模块基本结构

module 模块名 （端口列表）;

端口定义
input 输入端口
outpu 输出端口

数据类型说明
wire

逻辑功能定义
assign <逻辑表达式1>;
……
assign <逻辑表达式n>;

endmodule

三.持续赋值语句（assign语句）

assign　连线型变量名 = 赋值表达式；

wire型变量没有数据保持能力，只有被连续驱动后，才能取得确定值。（而寄存器型变量只要在某时刻得到过一次过程赋值，就能一直保持该值，直到下一次过程赋值。）

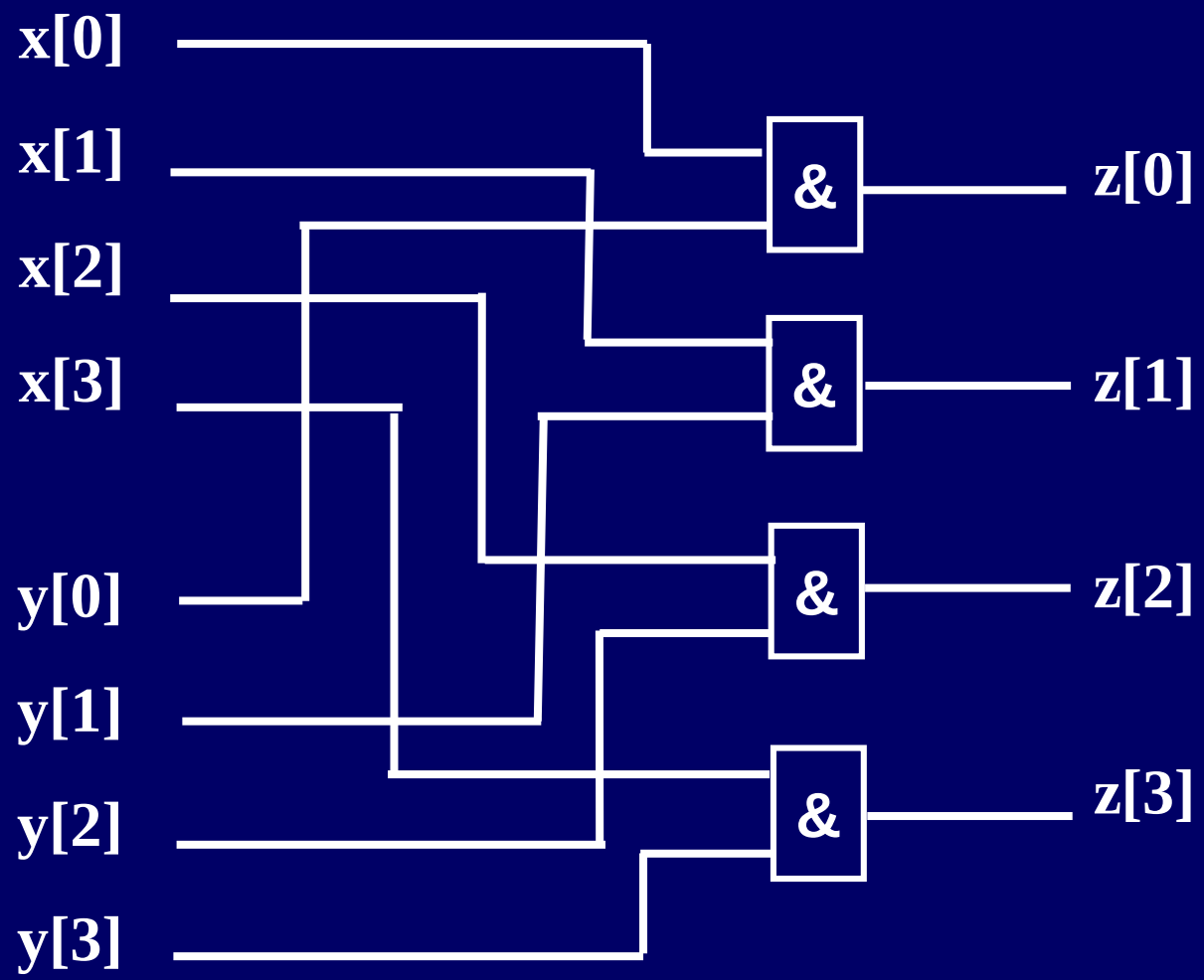
若一个连线型变量没有得到任何连续驱动，它的取值将是不定态“x”。

assign连续赋值语句就是实现对连线型变量进行连续驱动的一种方法。只要赋值表达式中任一操作数发生变化，立即对wire型变量进行更新操作，以保持对wire型变量的连续驱动。

```
//持续赋值语句应用举例
 module assignment ( z , x , y ) ;
      input  [3:0] x , y ;
     output [3:0] z ;
     wire    [3:0] z , x , y ;
     assign   z = x & y ;
 endmodule
```

根据端口信号类型的隐含特性，此句可省。

这里，已不是传统意义上的单变量与运算，而是两个相同位宽向量的按位与运算。对应的逻辑原理图？



思考：若上述模块中的 **assign z = x & y**；改为 **assign z = x && y**； 将如何？

实际应用中，持续赋值语句的赋值目标可以是如下几种：

●变量（标量）

```
wire a , b ;
assign  a = b ;
```

●向量

```
wire [7:0] a , b ;
assign  a = b ;
```

●向量中某一位

```
wire [7:0] a , b ;
assign  a[3] = b[3] ;
```

●向量中某几位

```
wire [7:0] a , b ;
assign  a[3:2] = b[3:2] ;
```

●拼接

```
 wire a , b ;
 wire [2:1]  c ;
assign  {a , b} = c ;
```

四.数据流描述举例

例：请用Verilog HDL数据流描述方式描述 $F = AB + \overline{CD}$ 的逻辑功能。

```
module ff_1(A,B,C,D,F);
    input A,B,C,D;
    output F;
    wire w1,w2;
        assign w1=A&B;
      assign w2=~(C&D);
      assign F=w1|w2;
endmodule
```

第五节 Verilog HDL的行为描述模块

一.行为描述

行为描述关注逻辑电路输入、输出的因果关系（行为特性），即在何种输入条件下，产生何种输出（操作），并不关心电路的内部结构。EDA的综合工具能自动将行为描述转换成电路结构，形成网表文件。

二. Verilog HDL行为描述模块的设计模型

Verilog 行为描述模块基本结构

module 模块名 （端口列表）;

端口定义
input 输入端口
output 输出端口

数据类型说明
reg
parameter

逻辑功能定义
always @(敏感事件列表)
begin
阻塞、非阻塞、**if-else**、**case**、**for**等行为语句
end

endmodule

三.行为描述中的 always进程

应用模板

```
always @ ( <敏感信号表达式> )
    begin
        //过程赋值语句
        //if-else, case, casex, casez选择语句
        //for循环语句
    end
```

一般情况下，always进程带有触发条件，这些触发条件列在敏感信号表达式中，只有当触发条件满足时，begin-end块语句才被执行。

在一个Verilog HDL模块中可以有多个always进程，它们是并发执行的。

敏感信号表达式

又称敏感事件列表。当该表达式中任意一个信号（变量）的值改变时，就会引发块内语句的执行。因此，应将所有影响块内取值的信号（变量）列入。多个敏感信号用“or”连接。

例如：

@ (a)

@ (a or b)

这里a和b称为电平敏感型信号，代表的触发事件是，信号除了保持稳定状态以外的任意一种变化过程。

这种电平敏感型信号列表常用在组合逻辑的描述中，以体现输入随时影响输出的组合逻辑特性。

再例如：

@ (posedge clock)

@ (negedge clock)

@ (posedge clock or negedge reset)

这里的clock和reset信号称为边沿敏感型信号，posedge描述对信号的上升沿敏感；negedge描述对信号的下降沿敏感。显然，这种边沿敏感型信号列表适合描述同步时序电路，以体现同步时序电路的特点——在统一时钟作用下改变电路的状态。

posedge —— 代表的触发事件是， 信号发生了正跳变。

0→x ，0→z ，0→1 ， x→1 ， z→1

negedge ——代表的触发事件是， 信号发生了负跳变。

1→x ，1→z ，1→0 ， x→0 ， z→0

在每一个always过程语句中，最好只使用一种类型的敏感信号列表，不要混合使用。以避免使用不同的综合工具时发生错误。

四. 串行块

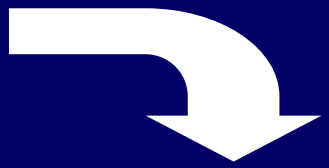
由关键字**begin-end**界定的一组语句。

```
begin
    语句1
    语句2
    ……
end
```

串行块的特点：

- 一般情况下，块内语句顺序执行，前面一条语句执行毕后，才开始执行下一条语句。
- 模块运行时，遇到串行块，块内第一条语句即开始执行，最后一条执行完毕，串行块结束。
- 整个串行块执行时间等于块内各条语句执行时间的总和。

```
module ff_1(A,B,C,D,F);
    input A,B,C,D;
    output F;
    wire w1,w2;
      assign w1=A&B;
      assign w2=~(C&D);
      assign F=w1|w2;
endmodule
```



行为描述

```
module ff_1(A,B,C,D,F);
    input A,B,C,D;
    output F;
    reg F, w1,w2;
      always @(A or B or C or D)
        begin
          w1=A&B;
          w2=~(C&D);
            F=w1|w2;
        end
endmodule
```

串行块只应用在always进程中：
多条语句；
顺序执行。

五. 过程赋值语句

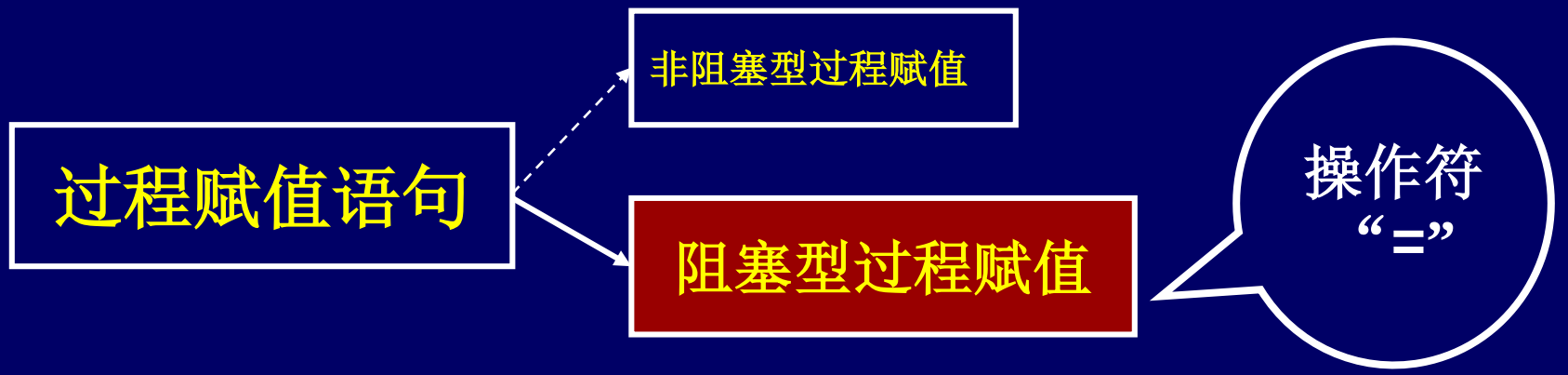
过程赋值语句必须放在always进程中，分为阻塞型和非阻塞型，其基本格式为：

〈被赋值变量〉 〈赋值操作符〉 〈赋值表达式〉

- reg 或 integer 类型变量；
- 寄存器向量的某一位或某几位；
- 用拼接符{ }拼接起来的寄存器。

任意合法表达式

```
//过程赋值语句的目标变量形式
……
 reg a ;
 reg [7:0] b ;
 integer  i ;
  always @ (敏感事件列表)
       begin
              a = 0 ;
              i  =356 ;
              b[2] = 1′b1 ;
              b[3:0] = 4′b1111 ;
              {a,b} = 9′b101110110 ;
       end
……
```



在前面讨论中，用到的赋值语句都是阻塞型过程赋值语句

- 串行块（**begin-end** ）内各条阻塞型过程赋值语句按顺序依次执行。下一条语句的执行被阻塞，等本条语句的赋值操作完成后，才开始执行。

- 阻塞型过程赋值语句的执行过程：先计算“赋值表达式”的值，然后立即赋值给“=”左边的“被赋值变量”。

特点：

- 在**begin-end**串行块语句中，各条非阻塞过程赋值语句对应的“赋值表达式”同时开始计算。
- 在过程块结束时，才将结果赋值给各个 “被赋值变量”。
- 可理解为先同时采样，最后一起赋值。

```
……
    begin
        A <= B ;  //S1
        B <= A ;  //S2
    end
……
```

这里，S1、 S2语句均为非阻塞赋值，立即开始计算B 和 A值（上次的值）。在过程块结束时，进行赋值操作，将计算得到的B,A的值赋给变量A,B。（实现A,B交换）

如果不能很好地理解阻塞赋值与非阻塞赋值的区别，往往给设计带来麻烦，特别是在可综合逻辑模块中，不易把握reg型变量的赋值过程。建议同学在编写模块时，只采用一种过程赋值方式，并且最好不要将输出再次作为输入使用。

为了更好地理解阻塞赋值与非阻塞赋值的区别，我们观察下面的示例。

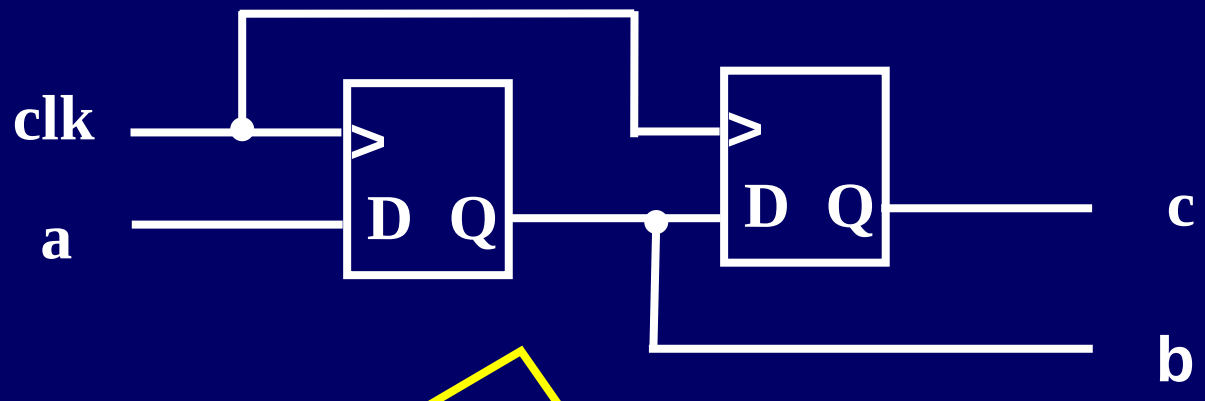
```
//例1：非阻塞赋值
module  n_block(c,b,a,clk);
  output  c, b ;
  input   clk, a ;
  reg     b, c;
  always @(posedge clk)
    begin
      b<=a;
      c<=b;
    end
endmodule
```

结果：b更新为a的值，c为上个时钟周期b的值。

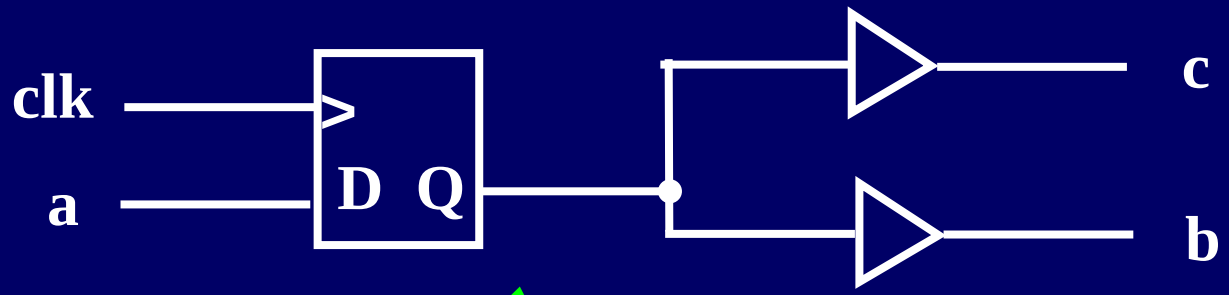
```
//例2：阻塞赋值
module  block(c,b,a,clk);
  output  c, b ;
  input   clk, a ;
  reg     b, c ;
  always @(posedge clk)
    begin
      b=a;
      c=b;
    end
endmodule
```

结果：b 、c都更新为a的值。

这两个程序进行逻辑综合后的结果如下：



例1 非阻塞赋值综合结果



例2 阻塞赋值综合结果

六. 条件语句

高级程序语句 → if-else 条件分支语句

高级程序语句 → case 分支控制语句

6.1 if-else 条件分支语句

两路分支选择控制

格式1　　if (<条件表达式>)　语句或语句块;

格式2　　if (<条件表达式>) 语句或语句块1;
　　else　　语句或语句块2;

多路分支选择控制

格式3　　if　　(<条件表达式1>) 语句或语句块1;
　　else if (<条件表达式2>) 语句或语句块2;
　　… …
　　else if (<条件表达式n>) 语句或语句块n;
　　else　　语句或语句块n+1;

为了清晰表达 if 和 else 的匹配关系，建议最好用begin-end 将“指定语句”括起来。

```
//if-else条件分支语句应用举例
 module   sel-from-three (q,sela,selb,a,b,c) ;
       input sela,selb,a,b,c ;
       output q ;
       reg q ;
       always @ (sela or selb or a or b or c)
          begin
            if    (sela)        q=a ;
            else  if  (selb) q=b ;
                  else        q=c ;
         end
 endmodule
```

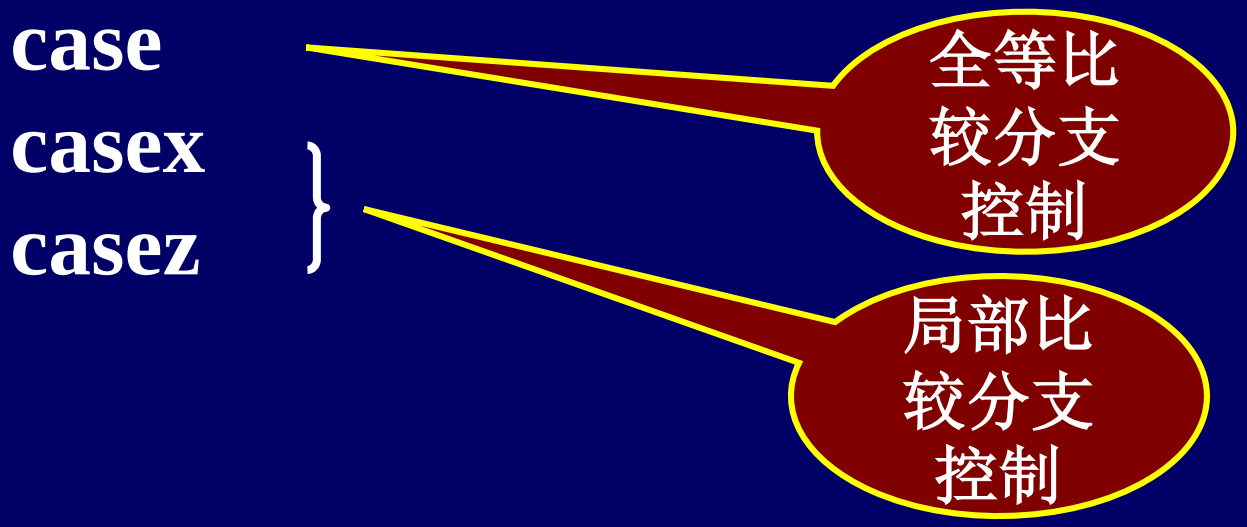
sela	selb	语句
0	0	q=c
0	1	q=b
1	0	q=a
1	1	q=a

注意隐含的优先级关系。排在前面的分支项指定的操作具有较高优先级。例：11时，执行q=a，不是q=b。

6.2 case 分支控制语句

相对if-else语句只有两个分支而言，case语句是一种多分支语句。所以，常用来描述译码器、多路数据选择器、微处理器的指令译码和有限状态机。

case 分支控制语句有三种形式:



“全等比较分支控制” case 语句的格式：

```
case (<控制表达式>)
    <分支项表达式1> : 语句块1 ;
    <分支项表达式2> : 语句块2 ;
          … …
    <分支项表达式n> : 语句块n ;
    default :          语句块n+1 ;
endcase
```

对程序流向进行控制的信号（变量）

控制信号（变量）的具体状态组合取值

受控的分支操作，可单句，也可多句。

未列入分支控制的状态组合的统称。
(其余状态时)

未列入分支控制的状态组合下应进行的操作

与真值表存在某种对应关系

按位全等比较case语句示例

```
…… ……
case (op_code)
     2′b00 :  out = a  | b ;
     2′b01 :  out = a & b ;
     2′b10 :  out = ~ (a & b) ;
     2′b11 :  out = a ^ b ;
     default :  out = 0 ;
endcase
…… ……
```

case语句在执行时，控制表达式和分支项表达式之间进行的是按位全等比较，只有对应每一位都相等，才认为控制表达式和分支项表达式是相等的。显然，这种比较包含了信号的 0 、 1 、 x 、 z 四种状态。

根据按位全等比较的特点，要求case语句中的控制表达式和分支项表达式必须具有相同的位宽。当各个分支项表达式以常数形式给出时，必须明确标明位宽，否则编译器默认为与机器字长相同的位宽（例如32位）。

使用if-else 条件分支语句和case 分支控制语句时应注意：

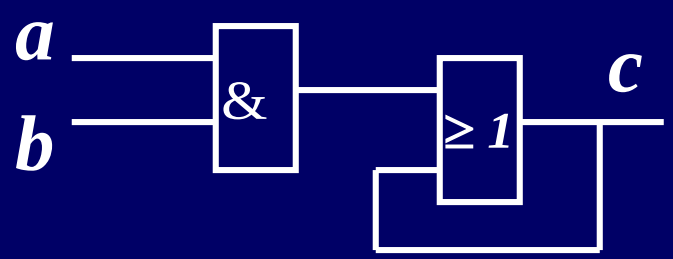
1. 应注意列出所有条件分支，否则，编译器认为条件不满足时，会引入触发器保持原值。这一点可用于时序电路的设计，例如，计数器，条件满足加1，否则保持不变；但在组合电路设计时，应避免这种隐含触发器的存在。

2. 当不能列出所有条件分支时，可在 if 语句的最后加上else ,在case语句的最后加上default。

//隐含触发器举例

```
… …
always @ (a  or  b)
   begin
      if ((a==1)&&(b==1))   c=1 ;
   end
… …
```

原意：2 输入与门。因无else处理，编译器认为存在“c=c”,保持不变。



在利用if-else语句描述组合电路时，应避免此类错误的发生。

```
//描述组合逻辑时，避免隐含触发器举例
 … …
 always @ (a  or  b)
    begin
       if ((a==1)&&(b==1))   c=1 ;

       else  c = 0 ;

    end
… …
```

这条else处理，避免了a=1, b=1 , 则c=1 后，c一直保持为 1 的隐含触发器情况。

七. 循环语句

for （< 语句1 > ；< 条件表达式 > ； < 语句2 >）

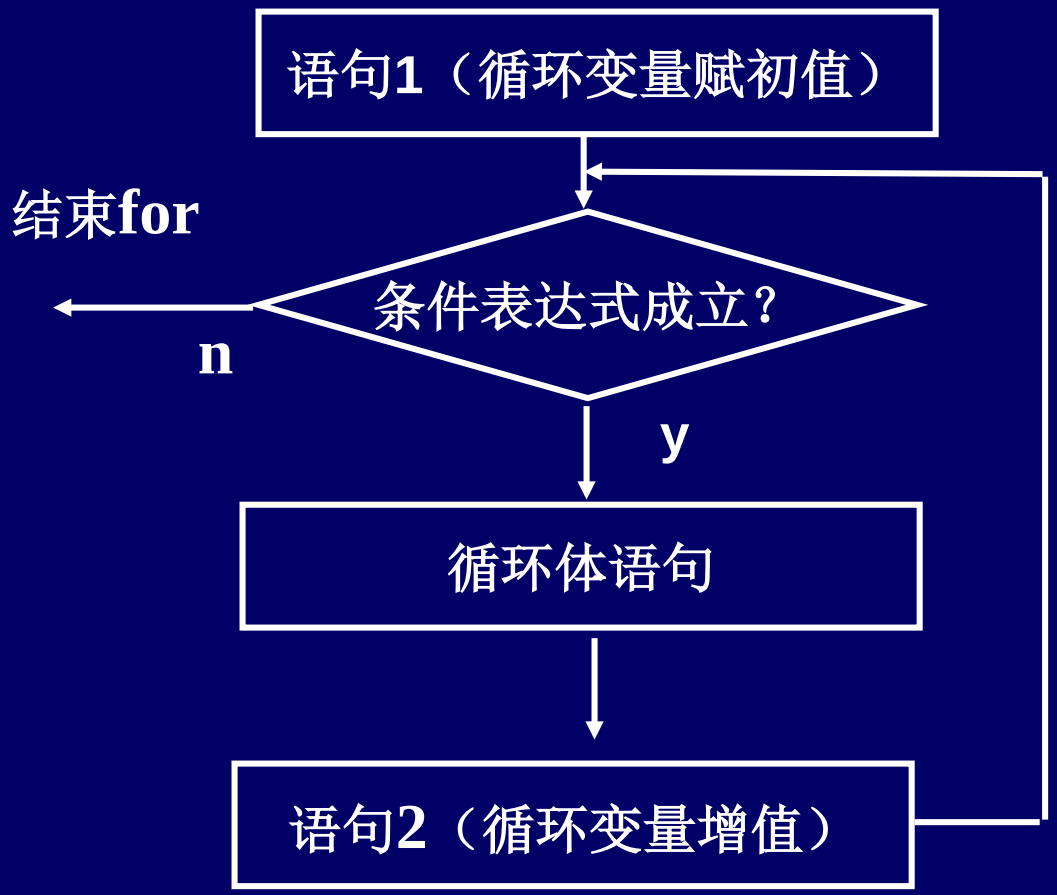
循环体中的语句或语句块；

“语句1”和“语句2”是过程赋值语句，对循环变量进行赋初值操作和增值操作。

“条件表达式”描述进行循环的条件，常为逻辑表达式。执行循环体语句之前，都要对其是否成立进行判断。

关于 for 循环语句格式的记忆性描述：

for （循环变量初值；循环结束条件；循环变量增值） 执行语句；



关于for循环语句的应用，将在后续课程中涉及。

作业7：

3.9

3.10

3.11

3.12（2）

3.14