



北京航空航天大学
Beijing University of Aero. and Astro.
(BEIHANG University)



数字电路与系统 便携式实验

Verilog硬件描述语言简介 (中)

1



门级电路

- 逻辑电路是由许多逻辑门和开关所组成
- Verilog HDL 提供了一些描述门类型的关键字，可以用于门级结构建模。
- Verilog HDL 基本元件模型共有26种，其中14种为基本门级元件，12种为开关级元件，

2



门级电路（续）

■ 门级电路列表

<门类型><实例元件名>(<数据输出>, <控制输入>, <控制输入>)

| | |
|----------------------------|--|
| 多输入门 | and (与门) nand (与非门) or (或门) nor (或非门) xor (异或门) xnor (异或非门, 同或门) |
| 多输出门 | buf (缓冲器) not (非门) |
| 三态门 (如果不被使能, 则输出“z”) | bufif0 (低电平使能缓冲器) bufif1 (高电平使能缓冲器) notif0 (低电平使能非门) notif1 (高电平使能非门) |
| 上拉, 下拉电阻 | pullup (上拉电阻) pulldown (下拉电阻) |

3



门级电路（续）

■ 门级电路调用（实例化）

<门的类型> [**<驱动能力><延时>**]**<门实例1>**[, **<门实例2>**,
<门实例3>.....];

而每个门实例, 按照

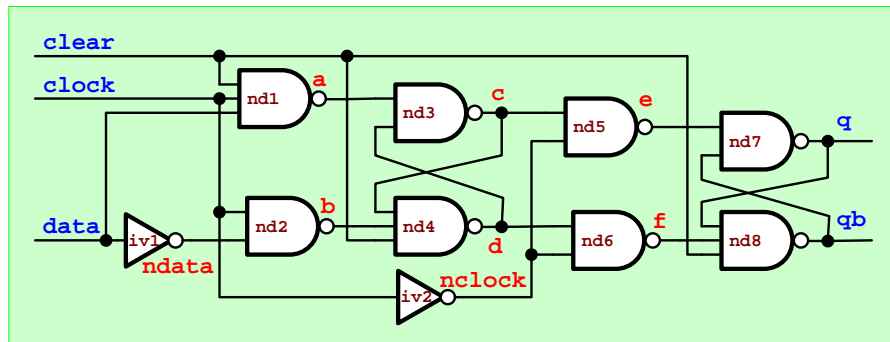
<实例元件名> (**<数据输出>**, **<控制输入>**, **<控制输入>**)

例: **nand** #10 **nd1**(**a**,**data**,**clock**,**clear**);

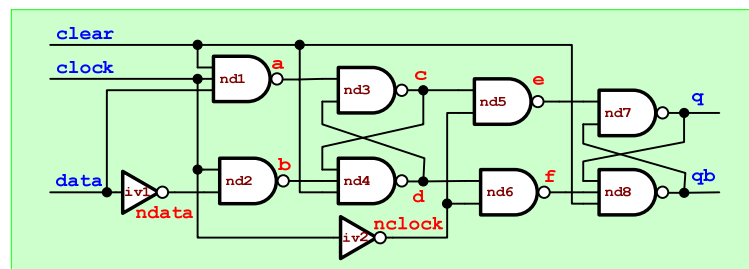
这个例子描述了一个名为**nd1**的与非门实例, 输入为**clock**,
data, **clear** 输出为**a**, 输出与输入的延时为10个单位时间。

4

■ 门级电路例子--D触发器



■ 门级电路例子--D触发器（续）



```
nand #10
    nd1(a,data,clear),
    nd2(b,ndata,clear),
    nd4(d,c,b,clear),
    nd5(e,c,nclock),
    nd6(f,d,nclock),
    nd8(qb,q,f,clear);

nand #9 nd3(c,a,d),
        nd7(q,e,qb);

not #10 iv1(ndata,data),
      iv2(nclock,clear);
```

3

但是，对于自行定义的模块，推荐的做法：

```
flop f0 (.data(d[0]), .clock(clk), .clear(clrb), .q(q[0]), ),
f1 (.data(d[1]), .clock(clk), .clear(clrb), .q(q[1]), ),
f2 (.data(d[2]), .clock(clk), .clear(clrb), .q(q[2]), ),
f3 (.data(d[3]), .clock(clk), .clear(clrb), .q(q[3]), );
```

门级电路

■ 门级电路例子—4位寄存器

```
`include "flop.v"
module hardreg
  (d,clk,clrb,q);
  input clk,clrb;
  input[3:0] d;
  output[3:0] q;
endmodule
```

注意

```
flop f0 (d[0],clk,clrb,q[0],),
f1 (d[1],clk,clrb,q[1],),
f2 (d[2],clk,clrb,q[2],),
f3 (d[3],clk,clrb,q[3],);
```

7

常用组合逻辑电路

■ 组合逻辑电路 — 功能上无记忆，结构上无反馈

- 电路任一时刻的输出状态只取决于该时刻各输入状态的组合，而与电路的原状态无关。
- 对于任何一个多输入多输出组合逻辑电路，可以用框图或者一组逻辑函数来表示。

$$y_1 = f_1(a_1, a_2, \dots, a_n)$$

$$y_2 = f_2(a_1, a_2, \dots, a_n)$$

$$\dots$$

$$y_m = f_m(a_1, a_2, \dots, a_n)$$

8

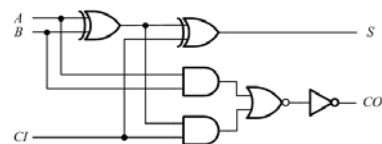


常用组合逻辑电路

■ 加法器

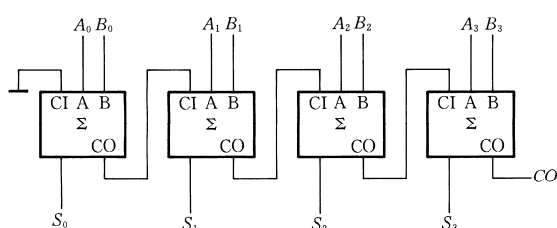
全加器的真值表

| 输入 | | | 输出 | |
|----|---|----|----|----|
| A | B | CI | S | CO |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |



$$S = A \oplus B \oplus CI$$

$$CO = AB + (A \oplus B) \cdot CI$$



➤ 多位加法器

9



常用组合逻辑电路

■ 加法器

```
module
add_4(A,B,CI,S,CO);
input[3:0]  A,B;
input      CI;
output[3:0] S;
output     CO;

assign      {CO,S}=A+B+CI;

endmodule
```

➤ 4位全加器

➤ 16位?



```
module
add_16(A,B,CI,S,CO);
input[15:0] A,B;
input      CI;
output[15:0] S;
output     CO;

assign      {CO,S}=A+B+CI;

endmodule
```

➤ 16位全加器

10



常用组合逻辑电路

■ 乘法器

$$\begin{array}{r}
 \text{被乘数:} \quad \quad \quad X_3 \quad X_2 \quad X_1 \quad X_0 \\
 \times \text{乘数:} \quad \quad \quad Y_3 \quad Y_2 \quad Y_1 \quad Y_0 \\
 \hline
 \quad \quad \quad \quad Y_0X_3 \quad Y_0X_2 \quad Y_0X_1 \quad Y_0X_0 \\
 \quad \quad Y_1X_3 \quad Y_1X_2 \quad Y_1X_1 \quad Y_1X_0 \\
 \quad Y_2X_3 \quad Y_2X_2 \quad Y_2X_1 \quad Y_2X_0 \\
 Y_3X_3 \quad Y_3X_2 \quad Y_3X_1 \quad Y_3X_0 \\
 \hline
 \text{乘积:} \quad Z_7 \quad Z_6 \quad Z_5 \quad Z_4 \quad Z_3 \quad Z_2 \quad Z_1 \quad Z_0
 \end{array}$$

```

module mult_4( X, Y, Product);
    input [3 : 0] X, Y;
    output [7 : 0] Product;
    assign      Product = X * Y;
endmodule

```

11




块语句

- 顺序块: begin end
 - 并行块: fork join
 - 过程块: initial、always
- initial只执行一次，它在仿真初始时刻（即0时刻）开始并发执行直到当前状态结束，在一个模块内每个initial块是独立并发执行。

```

■ module stimulus ; // 设模块的名字为stimulus
■     reg x, y, a, b, m;
■     initial
■     m = 1'b0;           // 单个过程声明语句，不需要在语句块中
■     initial begin
■         #5 a = 1'b1;      // 多个过程声明语句，需要在语句块中成组编写
■         #25 b = 1'b0;
■     end
■     initial
■         #50 $finish;     // 测试中的系统任务
■ endmodule

```



dig!C

块语句

- 顺序块: begin end


```
begin
  areg = breg;
  creg = areg;
end
```

creg=?

```
reg [2:0] A;
reg [3:0] B;
integer K, J;
initial
  begin
    K=0;
    A=0;
    K=K-1;
    J=K;
    A=A-1;
    B=A;
    J=J+1;
    B=B+1 ;
  end
```

A=?,B=?,J=?,K=?

13



dig!C

块语句

- 顺序块: begin end

```
begin
  areg = breg;
  creg = areg;
end
```

creg=breg

```
reg [2:0] A;
reg [3:0] B;
integer K,J;
initial
  begin
    K=0;
    A=0;
    K=K-1;
    J=K;
    A=A-1; //虽然reg无符号
    B=A;
    J=J+1;
    B=B+1 ;
  end
```

A=7,B=8, J=0,K=-1

14



块语句

- 顺序块: begin end
- 并行块: fork join

```
fork
  #50 r='h35;
  #100 r='hE2;
  #150 r='h00;
  #200 r='hF7;
  #250 r='h00;
join
```



```
fork
  #250 r='h00;
  #200 r='hF7;
  #50 r='h35;
  #100 r='hE2;
  #150 r='h00;
join
```

15



条件语句-if

```
If (a>b)
  out1 = int1;
```

```
If (a>b)
  out1 = int1;
else
  out1 = int2;
```

```
if(a>b)
  out1 = int1;
else if ( a>c )
  out1=int2;
else
  out1=int3;
```

还可以采用**嵌套**，并加**块语句**组合成更加复杂的逻辑描述。
在嵌套if序列中，**else**和最近的if相关，为提高可读性，习惯上使用**begin...end**块语句指定其作用域。

可综合问题：

- 条件赋值操作中通常被综合为多路器 (MUX) ；
- always 语句中，被综合为**多路器、锁存器**等

综合中的问题：

- 如果省略else分支，可能会生成不期望的锁存器 (latch)

16



条件语句-case

```
reg[15:0] rega;
reg[9:0] result;
case(rega)
  16'd0:result=10'b0111111111;
  16'd1:result=10'b1011111111;
  16'd2:result=10'b1101111111;
  16'd3:result=10'b1110111111;
  .....
endcase
```

也叫多路分支语句

可综合问题：

- case赋值操作通常被综合为多路器；
- always语句中，被综合为多路器、透明锁存器、循环移位寄存器等

综合中的问题：

- case中的default分支虽然可以缺少，但一般不要缺少，否则会产生不期望的锁存器（因为不总是能够产生新值）

17



条件语句-常见错误

```
always@(al or d)
begin
  if(al) q<=d;
end
```




综合后生成锁存器

```
always@(al or d)
begin
  if(al) q<=d;
  else q<=0;
end
```




综合成逻辑组合电路

18


 **条件语句-常见“错误”**

```
always@(sel[1:0] or a or b)
case(sel[1:0])
2'b00:q<=a;
2'b11:q<=b;
endcase
```




综合后生成锁存器

```
always@(sel[1:0] or a or b)
case(sel[1:0])
2'b00:q<=a;
2'b11:q<=b;
default: q<='b0;
endcase
```



综合成逻辑组合电路

19

 **循环语句**

- forever：连续执行
- repeat：执行n遍
- while：条件执行
- for：条件执行

20



循环语句-forever

- 通常用于产生连续信号，作为仿真测试信号，一般不可综合。

```
forever  
#10 Clock = !Clock;
```

21



循环语句-repeat

- 表示重复性操作，部分综合工具可综合。

```
initial  
begin  
  Clock=0;  
  repeat (MaxClockCycles)  
  begin  
    #10 Clock=1;  
    #10 Clock=0;  
  end  
end
```

22



循环语句-while

- 只要控制表达式为真循环语句就一直执行。当循环块有事件控制（如：@(posedge clock)）时才可综合。

```
begin: countIs
    reg[7:0] tempreg;
    count=0;
    tempreg = rega;
    while ( tempreg )
        begin
            if ( tempreg[0] ) count=count+1;
            tempreg = tempreg >> 1;
        end
    end
end
```

23



循环语句-for

- 一般循环语句，允许一条或更多的语句重复执行。如果循环的边界是**固定**的，综合时循环语句被认为是重复的硬件结构。

```
V=0 ;
for( i=0; i<4; i=i+1 )
    begin
        F[i] = A[i] & B[3-i]; //四个独立的与门
        V = V ^ A[i];        //四个独立的异或门
    end
```

24



always过程块

- 不断重复执行的结构化语句，符合标准化结构编写的代码是可综合的

```
always @( 时序控制 ) <语句>
```

如果没有时序控制，则always语句将会生成一个死锁。如：

```
always areg=~areg;
```

但如果加上时序控制，就会变成很有用的信号发生器：

```
always #half_period areg=~areg;
```

可以用边沿触发或电平敏感触发等多种时序控制方式。

25



■ 非阻塞赋值与阻塞赋值的区别

1. 非阻塞赋值方式

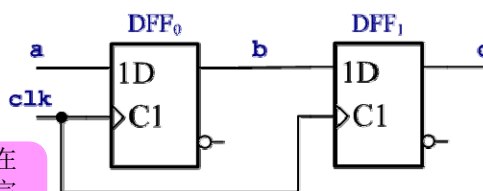
```
always @(posedge clk)
```

```
begin
```

```
    b <= a ;
```

```
    c <= b;
```

```
end
```



非阻塞赋值在块结束时才完成赋值操作！

注：c的值比b的值落后一个时钟周期！

26



■ 非阻塞赋值与阻塞赋值的区别

2. 阻塞赋值方式

```
always @(posedge clk)
```

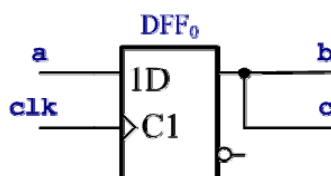
```
begin
```

```
    b = a ;
```

```
    c = b;
```

```
end
```

阻塞赋值在**该语句**结束时就完成赋值操作！



注：在一个块语句中，如果有多条阻塞赋值语句，在前面的赋值语句没有完成之前，后面的语句就不能被执行，就像被阻塞了一样，因此称为**阻塞赋值方式**。
这里c的值与b的值一样！

27



非阻塞赋值与阻塞赋值方式的主要区别

➤ 非阻塞 (non-blocking) 赋值方式 ($b \leftarrow a$):

- b的值被赋成新值a的操作，并不是立刻完成的，而是在块结束时才完成；
- 块内的多条赋值语句在块结束时同时赋值；
- 可参考对应的同步时序逻辑电路。

➤ 阻塞 (blocking) 赋值方式 ($b = a$):

- b的值立刻被赋成新值a；
- 完成该赋值语句后才能执行下一句的操作；
- 在边沿触发时序控制的情况下，可能会由于疏忽，使**综合结果未知**。（可用但慎用）

28



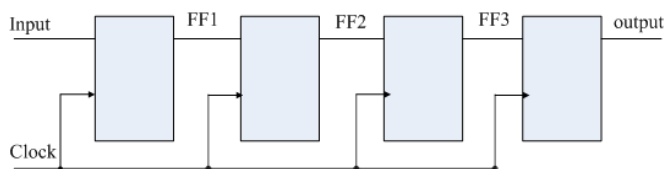
3.5 always语句

- 例子-请排定下列语句的次序

```
Output<=FF3;
reg FF1,FF2,FF3;
FF2<=FF1;
always@(posedge Clock)
end
FF3<=FF2;
FF1<=Input;
begin
```

?

思考：为
什么大
写“O”



29

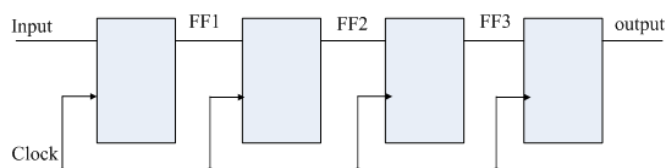


3.5 always语句

- 例子-请排定下列语句的次序

```
Output<=FF3;
reg FF1,FF2,FF3;
FF2<=FF1;
always@(posedge Clock)
end
FF3<=FF2;
FF1<=Input;
begin
```

```
reg FF1,FF2,FF3;
always @(posedge Clock)
begin
    Output<=FF3;
    FF3<=FF2;
    FF2<=FF1;
    FF1<=Input;
end
```



30



阻塞与非阻塞

■ 阻塞 V.S. 非阻塞

■ 阻塞赋值 “=”

```
module fboscl(y1,y2,clk,rst);
  output y1,y2;
  input clk,rst;
  reg y1,y2;

  always @ (posedge clk or
            posedge rst)
    if ( rst) y1 = 0;
    else y1 = y2;

  always @ (posedge clk or
            posedge rst)
    if ( rst ) y2 = 1;
    else y2 = y1;
endmodule
```

■ 假设复位信号已从1到0，考虑第一个clk上升沿到达时 y1 和 y2 的取值情况。

■ 情况一：考虑第一个always语句 clk 早到几个 ps

```
y1 = 1;
y2 = 1;
```

■ 情况二：考虑第二个always语句 clk 早到几个 ps

```
y1 = 0;
y2 = 0;
```

y1 和 y2 的取值情况互相矛盾，不稳定，存在竞争现象。

31



阻塞与非阻塞

■ 阻塞 V.S. 非阻塞

■ 阻塞赋值 “=”

```
module fboscl(y1,y2,clk,rst);
  output y1,y2;
  input clk,rst;
  reg y1,y2;

  always@ (posedge clk or
           posedge rst)
    if(rst) y1 = 0;
    else y1 = y2;

  always@ (posedge clk or
           posedge rst)
    if(rst) y2 = 1;
    else y2 = y1;
endmodule
```



32



阻塞与非阻塞

■ 阻塞 V.S. 非阻塞

■ 非阻塞赋值 “<=”

```
module fbosc2(y1,y2,clk,rst);
    output y1, y2;
    input clk, rst;
    reg y1, y2;

    always @ (posedge clk or
              posedge rst)
        if(rst) y1 <= 0;
        else y1 <= y2;

    always @ (posedge clk or
              posedge rst)
        if(rst) y2 <= 1;
        else y2 <= y1;
endmodule
```

■ 假设复位信号已从1到0，考虑第一个clk上升沿到达时y1和y2的取值情况。

■ 情况一：考虑第一个always语句clk早到几个ps

```
y1 = 1;
y2 = 0;
```

■ 情况二：考虑第二个always语句clk早到几个ps

```
y1 = 1;
y2 = 0;
```

y1和y2的取值情况一致，并在后续clk时钟上升沿触发下发生反转。

33



阻塞与非阻塞

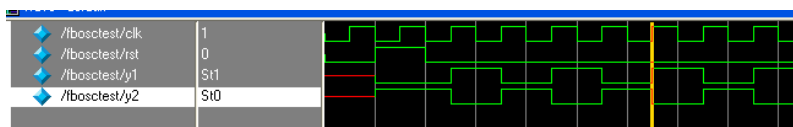
■ 阻塞 V.S. 非阻塞

■ 非阻塞赋值 “<=”

```
module fbosc2(y1,y2,clk,rst);
    output y1,y2;
    input clk,rst;
    reg y1,y2;

    always@ (posedge clk or
             posedge rst)
        if(rst) y1 <= 0;
        else y1 <= y2;

    always@ (posedge clk or
             posedge rst)
        if(rst) y2 <= 1;
        else y2 <= y1;
endmodule
```



34



阻塞与非阻塞

■ always块的纯组合逻辑

```
module ao1(y,a,b,c,d);
  output y;
  input a,b,c,d;
  reg y,temp1,temp2;
```

```
  always@ (a or b or c or d)
  begin
    temp1<=a&b;
    temp2<=c &d;
    y<=temp1 | temp2;
  end
endmodule
```

- y反映的是刚进入always块时temp1和temp2的值，而不是在always块中经过计算后得到的值。

再换一种方法→

```
module ao2(y,a,b,c,d);
  output y;
  input a,b,c,d;
  reg y,temp1,temp2;
```

```
  always@ (a or b or c or d or temp1 or temp2)
  begin
    temp1<=a&b;
    temp2<=c &d;
    y<=temp1 | temp2;
  end
endmodule
```

- y输出值正确，但在always块中有多次参数传递，降低了仿真器的性能。

35



阻塞与非阻塞

■ always块的纯组合逻辑

```
module ao1(y,a,b,c,d);
  output y;
  input a,b,c,d;
  reg y,temp1,temp2;
```

再换一种方法→

```
  always @ (a or b or c)
  begin
    temp1 = a & b;
    temp2 = c & d;
    y = temp1 | temp2;
  end
endmodule
```

- y输出值正确，并提高了仿真效率（其实综合的结果也更合理）。
- 所以说：
该用阻塞的就不能用非阻塞

36

阻塞与非阻塞

- always自触发例子

```

module osc1(clk);
    output clk;
    reg clk;

    initial #10 clk=0;
    always@ (clk) #10 clk=~clk;
endmodule

module osc2(clk);
    output clk;
    reg clk;

    initial #10 clk=0;
    always@ (clk) #10 clk<=~clk;
endmodule

```

换一种方法→

该例子中能形成clk时钟吗？

注意区别→

```

always
begin
    #10 clk=~clk;
end

```

注意：非阻塞方式能否自触发，取决于仿真调度引擎。不推荐这种方式产生时钟信号。

37

阻塞与非阻塞

- 移位寄存器
- 观察那种方法能实现图示电路？

```

module pipeb1(q3,d,clk);
    output[7:0] q3;
    input[7:0] d;
    input clk;
    reg [7:0] q3,q2,q1;

    always @ (posedge clk)
    begin
        q1 = d;
        q2 = q1;
        q3 = q2;
    end
endmodule

module pipeb2(q3,d,clk);
    output[7:0] q3;
    input[7:0] d;
    input clk;
    reg [7:0] q3,q2,q1;

    always @ (posedge clk)
    begin
        q3 = q2;
        q2 = q1;
        q1 = d;
    end
endmodule

```

换一种方法→

可行但不推荐

38

阻塞与非阻塞

- 移位寄存器
- 观察那种方法能实现图示电路?

再换一种方法→

```

module pipeb3(q3,d,clk);
  output[7:0] q3;
  input[7:0] d;
  input clk;
  reg [7:0] q3,q2,q1;

  always @ (posedge clk) q1=d;
  always @ (posedge clk) q2=q1;
  always @ (posedge clk) q3=q2;
endmodule

```

```

module pipeb4(q3,d,clk);
  output[7:0] q3;
  input[7:0] d;
  input clk;
  reg [7:0] q3,q2,q1;

  always @ (posedge clk) q3=q2;
  always @ (posedge clk) q2=q1;
  always @ (posedge clk) q1=d;
endmodule

```

再换一种方法→

39

阻塞与非阻塞

- 移位寄存器
- 观察那种方法能实现图示电路?

再换一种方法→

```

module pipen1(q3,d,clk);
  output[7:0] q3;
  input[7:0] d;
  input clk;
  reg [7:0] q3,q2,q1;

  always@ (posedge clk)
  begin
    q1 <= d;
    q2 <= q1;
    q3 <= q2;
  end
endmodule

```

```

module pipen2(q3,d,clk);
  output[7:0] q3;
  input[7:0] d;
  input clk;
  reg [7:0] q3,q2,q1;

  always@ (posedge clk)
  begin
    q3 <= q2;
    q2 <= q1;
    q1 <= d;
  end
endmodule

```

再换一种方法→

40

阻塞与非阻塞

- 移位寄存器
- 观察那种方法能实现图示电路?

再换一种方法→

再换一种方法→

```

module pipen3(q3,d,clk);
    output[7:0] q3;
    input[7:0] d;
    input clk;
    reg [7:0] q3,q2,q1;

    always @ (posedge clk) q1<=d;
    always @ (posedge clk) q2<=q1;
    always @ (posedge clk) q3<=q2;

endmodule

```

```

module pipen4(q3,d,clk);
    output[7:0] q3;
    input[7:0] d;
    input clk;
    reg [7:0] q3,q2,q1;

    always @ (posedge clk) q3<=q2;
    always @ (posedge clk) q2<=q1;
    always @ (posedge clk) q1<=d;

endmodule

```

41

阻塞与非阻塞

- 八大原则
 - 时序电路建模时，用非阻塞赋值；
 - 锁存器电路建模时，用非阻塞赋值；（逻辑上是安全的→）
 - 用 always 块 建立组合逻辑模型时，用阻塞赋值；
 - 在同一个always块中建立 时序 和 组合逻辑电路 时，用非阻塞赋值；（但是，逻辑上不一定正确）
 - 在同一个always块中，建议不要既用非阻塞赋值又用阻塞赋值；
 - 不要在一个以上的 always 块中为同一个变量赋值；
 - 用 \$strobe 系统任务来显示用非阻塞赋值的变量值；
 - 在赋值时不要用#0延迟。

42



■ register型变量与 net型变量的进一步思考：

- ❖ register型变量需要被明确地赋值，并且在被重新赋值前一直保持原值。
- ❖ register型变量必须通过**过程赋值语句**赋值！不能通过assign语句赋值！
- ❖ 在过程块内被赋值的每个信号必须定义成reg型！

43



计数器设计

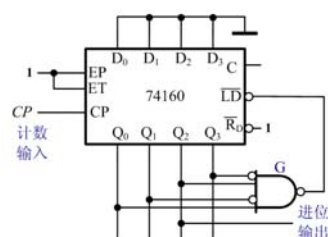
- 计数器verilog HDL实现
对时序个数进行统计，产生13进制计数器

```

module CNT13 (CLR, CLK, Q
  input CLR, CLK;
  output [3:0] Q;
  reg [3:0] Q;

  always @( posedge CLK or negedge CLR )
  if ( !CLR ) Q <= 0;
  else if ( Q == 12 )
    Q <= 0;
  else
    Q <= Q + 1;
endmodule

```



- 中型数字电路逻辑设计
74160→六进制

44

计数器设计

- 计数器verilog实现
 - 对时序个数进行统计，产生13进制计数器

如何实现减法计数器？

基于此能否实现更加复杂的分频电路？



45

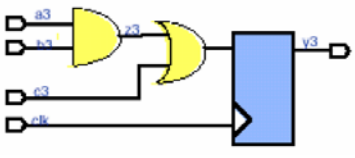
阻塞与非阻塞

- 几个例子

```

reg z3, y3;
always @(posedge clk)
begin
    z3 = a3 & b3;
    y3 = z3 | c3;
end

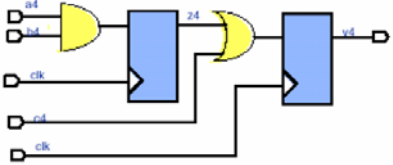
```



```

reg z4, y4;
always @(posedge clk)
begin
    z4 <= a4 & b4;
    y4 <= z4 | c4;
end

```



46

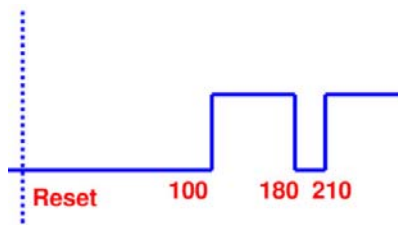
阻塞与非阻塞

■ 几个例子

```

initial
begin
    Reset = 0;
    #100 Reset = 1;
    #80 Reset = 0;
    #30 Reset = 1;
end

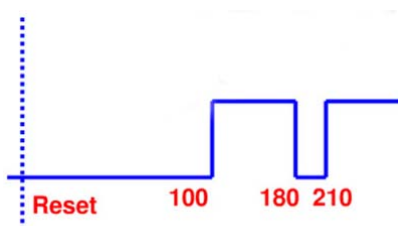
```



```

initial
begin
    Reset <= 0;
    #100 Reset <= 1;
    #180 Reset <= 0;
    #210 Reset <= 1;
end

```



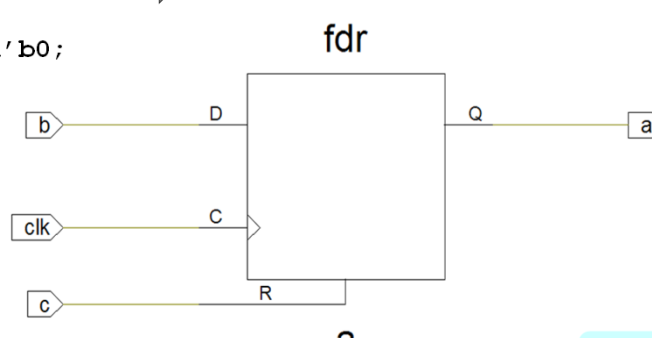
阻塞与非阻塞

■ 几个例子

```

reg a;
always @(posedge clk)
begin
    a <= b;
    if(c)
        a <= 1'b0;
end

```



48

阻塞与非阻塞

■ 几个例子

```

reg f,g;
always @(a or b)
  case (a)
    1'b0: f = b;
    1'b1: g = b;
  end

```

49

阻塞与非阻塞

■ 几个例子

```

reg B,C,D;
always @(posedge clk)
begin
  C = B;
  D <= C;
  B = A;
end

```



仿真与测试

■ 仿真（Simulation）

- 仿真过程是正确实现设计的关键环节，用来验证设计者的设计思想是否正确，及在设计实现过程中各种分布参数引入后，其设计的功能是否依然正确无误。
- 仿真分为功能仿真（前仿真）和时序仿真（后仿真）。
 - 功能仿真是在设计输入后进行
 - 时序仿真是在逻辑综合后或布局布线后进行

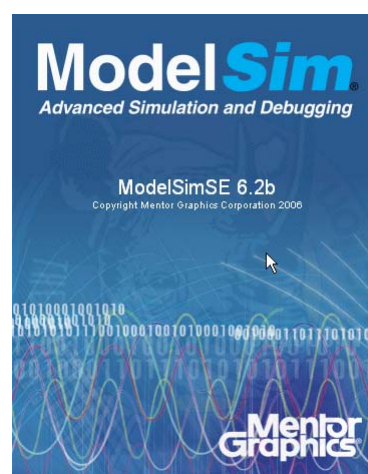
51



仿真工具——ModelSim

■ ModelSim 总体概览

- ModelSim 仿真工具是工业上最流行、最通用的仿真器之一
- 可支持 Verilog、VHDL 或是 VHDL/ Verilog 混合输入的仿真，它的 OEM 版本允许 Verilog 仿真或 VHDL 仿真。



52



仿真工具——ModelSim



Modelsim 用户界面

53



仿真工具——ModelSim

■ ModelSim 的仿真实现方式

- 交互式的命令行（Cmd）的方式 —— 惟一的界面是控制台的命令行，没有用户界面。
- 用户界面UI的方式——可以接受菜单输入和命令行输入的仿真方式。
- 批处理模式——从 DOS 或 UNIX 命令行运行批处理文件的仿真方式。

54



仿真工具——ModelSim

■ ModelSim 基本仿真步骤

- 建立库。
- 映射库到物理层目录。
- 编译源代码 —— 所有的 HDL 代码必须被编译； Verilog 和 VHDL 必须有不同的编译器支持。
- 启动仿真器， 执行仿真。 也可以从其他软件上直接调用， 启动内嵌的仿真器执行仿真。

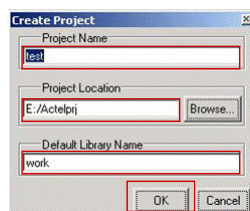
55



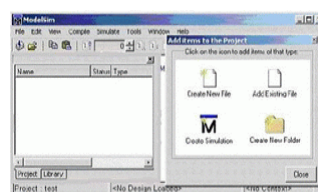
ModelSim 仿真过程

■ 创建一个项目

- 启动 ModelSim
- 在主窗口选择 File → New → Project → Create a Project 打开项目对话框。



项目设立对话框



项目选择页面

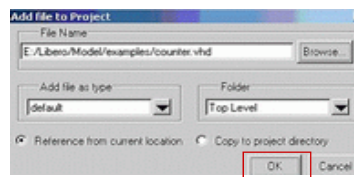
56



ModelSim 仿真过程

■ 创建一个项目（续）

- 添加包含设计内容的源文件到项目中，在 Add items to the Project 对话框中点击 Add Existing File



选择文件至当前项目栏

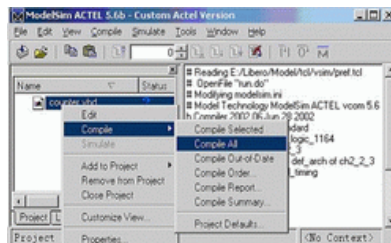
57



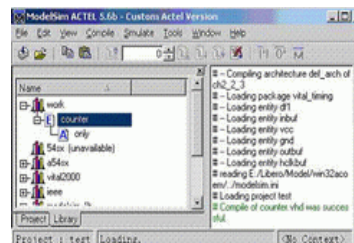
ModelSim 仿真过程

■ 创建一个项目（续）

- 在工具栏点击编译按键或在项目页面点击鼠标右键并选择 Compile → Compile All



选择编译文件



目标文件的显示

- 加入的文件被编译后，点击 Library 标签，并且通过点击 “+” 图标展开 work 库，将会看到被编译的设计例举单元

58

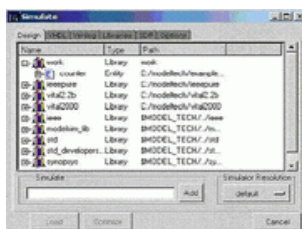


ModelSim 仿真过程

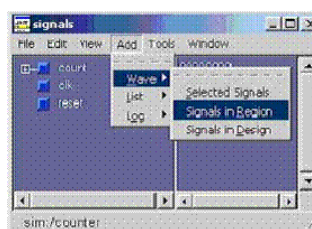
■ 仿真过程

➢ 设计文件的装载

- 通过选择Simulate→Simulate来装载设计单元，出现仿真对话框，点击“work”下面的“+”扩展符号，可看到counter设计元目录
- 从主窗口菜单中选择View → All Window来打开所有窗口
- 在信号窗口菜单中通过选择 Add → Wave → Signals in Region 来加载顶层信号到波形窗口中。



选择 Simulate 装载设计单元



信号窗口

59



ModelSim 仿真过程

■ 运行仿真

➢ 激励信号的加载主要有两种方式

- 用 force 命令的人机交互式
- 建立测试平台程序的方式。

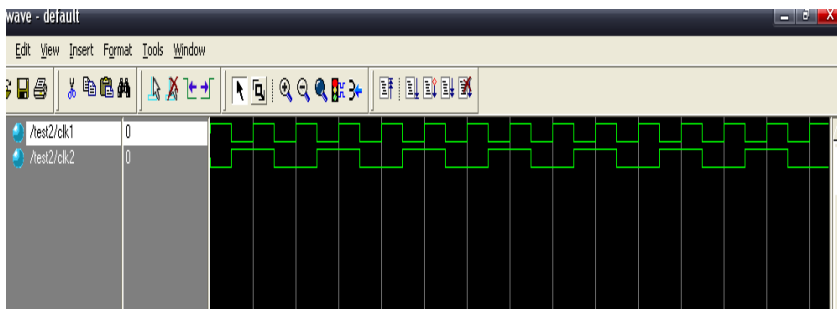
■ 仿真结果的调试

- 声明调试方式
- 在波形窗口中组合信号
- 创建并浏览数据表（datasets）

60

测试代码

- 用ModelSim编译仿真后的波形如下



如果输入信号的值有规律的变化，例如按相同的延迟重复出现 n 次，那么就可以通过repeat循环语句来实现。

61

ModelSim仿真器的使用技巧

- 应用
 - run -all 与 \$stop
 - *.do文件（极大降低工作量）
 - 存储为VCD格式的文件
 - VCD文件到WLF文件的转换

62

 dig!C

课程信息



- 教师：何锋
 - 新主楼 F-712
 - e-Mail: fenghe@buaa.edu.cn
(可预约答疑)
- 课程资料
 - 如果有解压密码，约定就是**buaa123456**