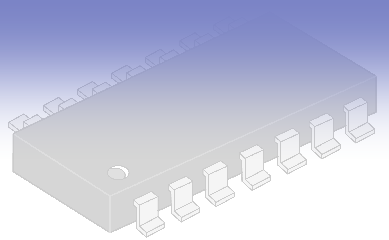
第 6 章

行为建模

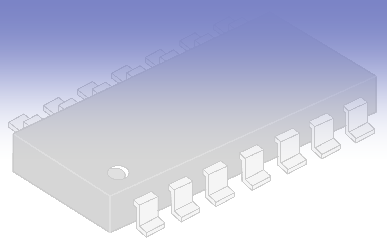
过程赋值语句



- 赋值对象
  - 寄存器类型
  - 存储器单元
  - 整型、实数、时间变量
- 过程赋值语句有两种类型
  - 阻塞赋值（Blocking assignment ）
  - 非阻塞赋值（Nonblocking Assignment）
- 过程赋值语句执行方式——分为两种类型
  - 顺序执行（Sequential）
  - 并发执行（Sequential）
- 语法

过程赋值 **::=** **<lvalue>** = [延时或事件控制] ***expression***

阻塞赋值（Blocking assignment）（1）



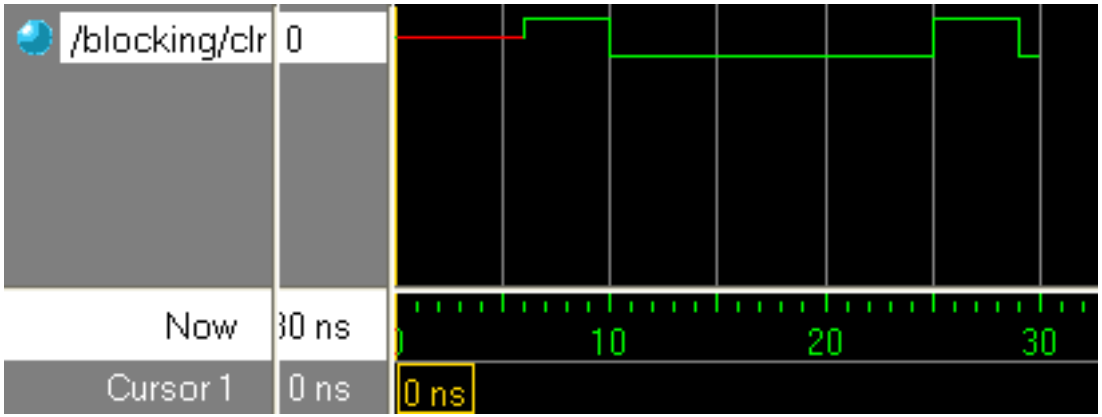
- 赋值符号：“=” —— Evaluated and assigned in a single step
- 在过程语句结构的顺序语句块中，阻塞赋值语句按顺序执行

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;

initial
begin
    x = 0; // 标量赋值
    y = 1;
    z = 1;
    count = 0; // 整型变量赋值
    reg_a = 16'b0; // 向量的初始化
    reg_b = reg_a;

    #15 reg_a[2] = 1'b1; // 带延时的位选择赋值
    #10 reg_b[15:13] = {x, y, z} // 拼接运算和赋值

    count = count + 1; // 整型变量赋值
end
```

阻塞赋值（Blocking assignment）（2）

- 阻塞性过程赋值
  - // 在过程语句块中
  - begin
  - …
    - 赋值语句执行后，立即改变被赋值对象的值
    - 下一条语句必须等待上面的阻塞赋值完成后才能执行
  - …
  - end // 等到阻塞赋值语句执行完成后，过程语句块才结束

```
1 `timescale 1ns / 1ns
2 module blocking;
3
4         reg clr;
5
6         initial
7         begin
8                   clr = #6 1'b1;
9                   clr = #4 1'b0;
10                  clr = #10 1'b0;
11                  clr = #5  1'b1;
12                  clr = #4  1'b0;
13        end
14 endmodule
```

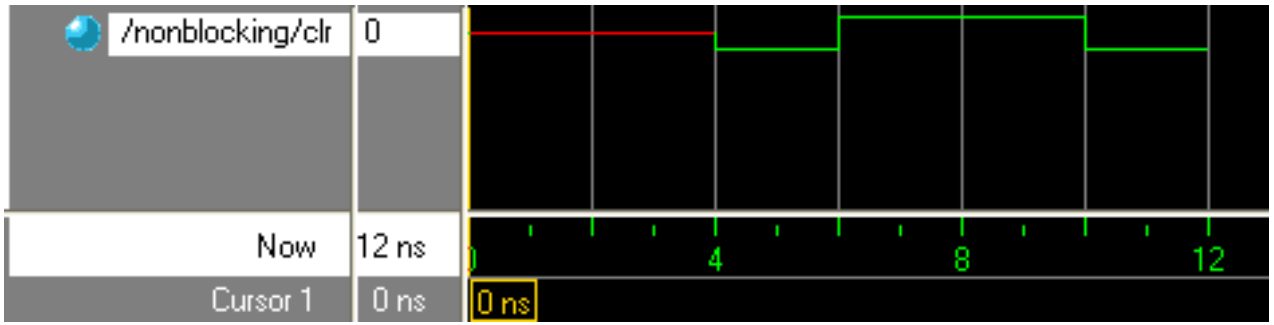


非阻塞赋值（Nonblocking Assignment）

- 赋值符号：“<=”
  - 符号与小于等于相同
  - 意义完全不同，用于过程中的赋值
- 非阻塞性过程赋值
  - // 在过程语句块中
  - begin
  - …
  - 非阻塞赋值语句执行后
    - 计算右侧表达式的值，并不立即赋值
    - 继续执行下一条语句
    - 在当前时间步的其它计算完成后，再对左边目标赋值
      - **同一时间步**
  - 非阻塞的含义
    - 执行下一条语句时，无需上一条赋值完成
    - 下一条语句的就可执行
  - …
  - end

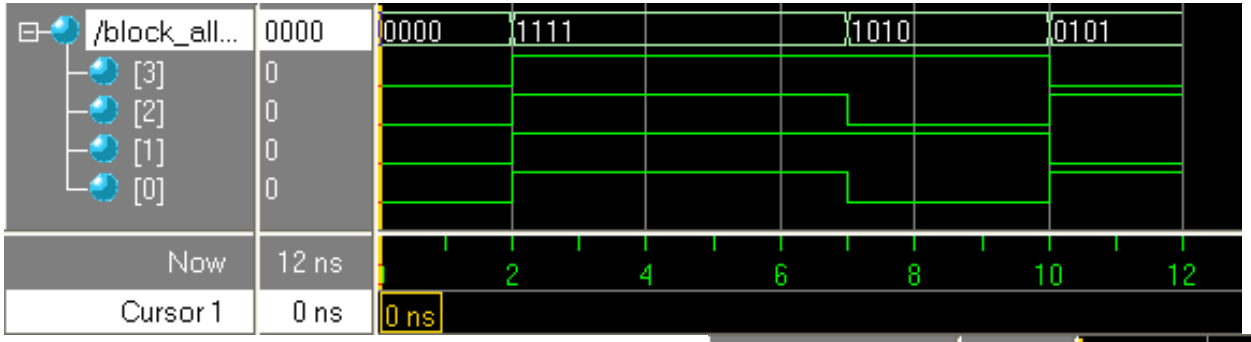
- 计算和赋值分成两步：
  - (1) 立刻完成右边表达式的计算
  - (2) 直到当前时间步的其它计算都完成后再对左边进行赋值

```
1 `timescale 1ns / 1ns
2 module nonblocking;
3
4         reg clr;
5
6         initial
7         begin
8                 clr <= #6 1'b1;
9                 clr <= #4 1'b0;
10                clr <= #10 1'b0;
11        end
12 endmodule
```



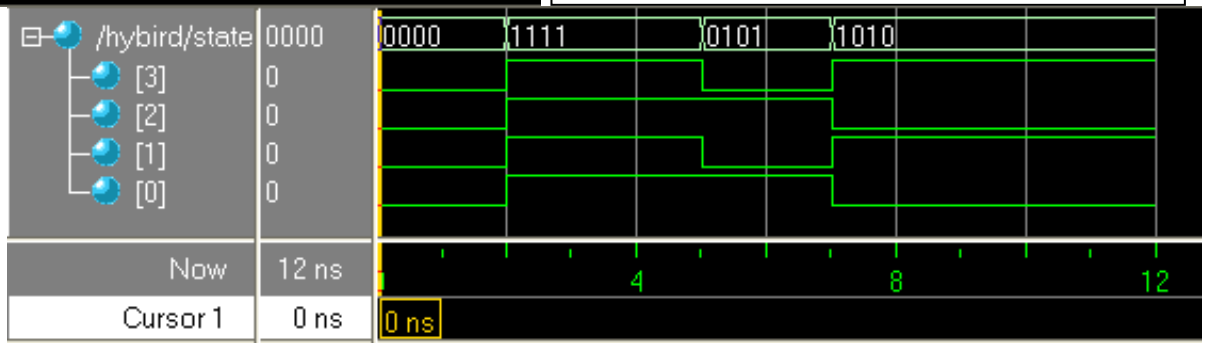
阻塞和非阻塞性赋值的混合及比较

```
1 `timescale 1ns / 1ns
2
3 module block_allot;
4     reg  [3 : 0] state;
5     initial
6     begin
7         state = 4'b0000;
8         #2 state = 4'b1111;
9         state = #5 4'b1010;
10        #3 state = 4'b0101;
11    end
12 endmodule
```




```
1 `timescale 1ns / 1ns
2
3 module hybird;
4     reg  [3 : 0] state;
5     initial
6     begin
7         state = 4'b0000;
8         #2 state = 4'b1111;
9         state <= #5 4'b1010;
10        #3 state = 4'b0101;
11    end
12 endmodule
```

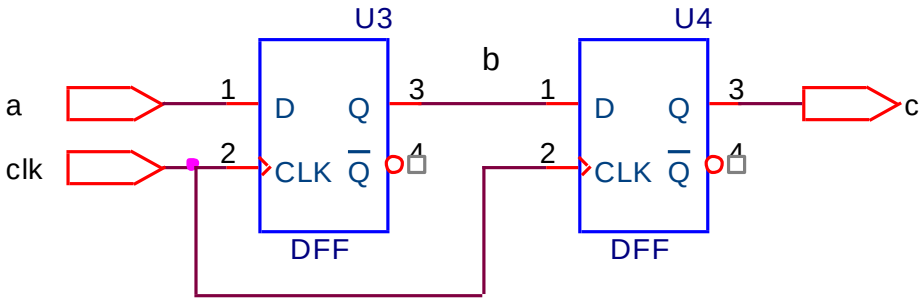
```
begin
    state = 4'b0000;
    #2 state = 4'b1111;
    temp <= 4'b1010;
    #3 state = 4'b0101;
    #2;
    state <= temp;
end
```

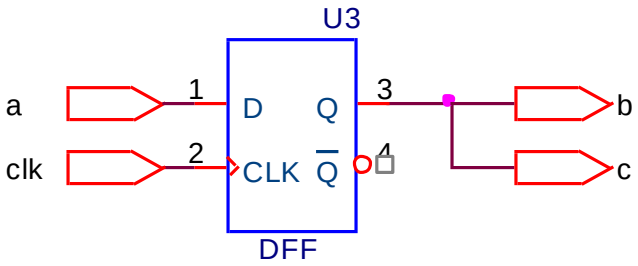


阻塞和非阻塞性赋值的比较

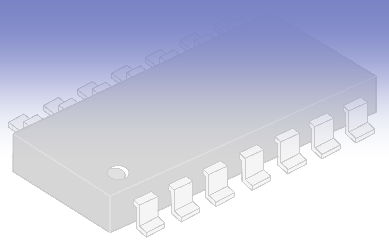
- 非阻塞性赋值
  - always @ (posedge clk)
  - begin
  - b <= a;
  - c <= b;
  - end

- 阻塞性赋值
  - always @ (posedge clk)
  - begin
  - b = a;
  - c = b;
  - end





顺序块非阻塞与并行块阻塞赋值的比较（1）



- 内嵌延时 —— 例1和例2具有同样的功能

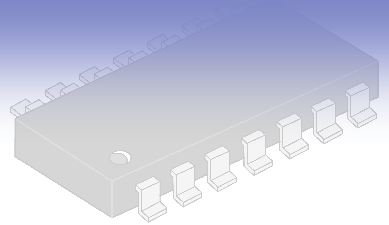
- 例1、顺序块中的非阻塞赋值
  - 可综合

```
begin
    a <= 4'd1;
    a <= #10 4'd0;
    a <= #5 4'd4;
end
```

- 例2、并发块中的阻塞赋值
  - 综合结果不确定

```
fork
    a = 4'd1;
    a = #10 4'd0;
    a = #5 4'd4;
join
```

顺序块非阻塞与并行块阻塞赋值的比较（2）



- 内嵌延时——具有同样的功能

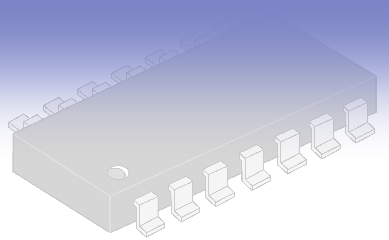
```
`timescale 1ns/1ns
module nonblocking;
    reg [3:0] a;

    initial
    begin
        a <= 4'd1;
        a <= #10 4'd0;
        a <= #5 4'd4;
    end

    initial
        $monitor("At time t=%4t, a=%4d", $time, a);
endmodule
```

```
`timescale 1ns/1ns
module blocking;
    reg [3:0] a;

    initial
    fork
        a = 4'd1;
        a = #10 4'd0;
        a = #5 4'd4;
    join

    initial
        $monitor("At time t=%4t, a=%4d", $time, a);
endmodule
```

```
# Loading work.nonblocking
VSIM 9> run
# At time t=    0, a=    1
# At time t=    5, a=    4
# At time t=   10, a=    0
```

```
# Loading work.blocking
VSIM 12> run
# At time t=    0, a=    1
# At time t=    5, a=    4
# At time t=   10, a=    0
```

顺序块非阻塞与并行块阻塞赋值的比较（3）



- 常规延时控制——不同的功能

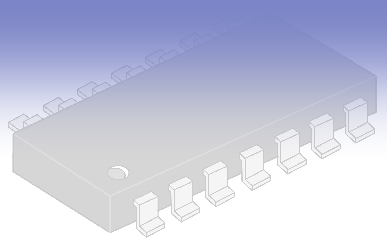
```
`timescale 1ns/1ns
module nonblocking_d;
    reg [3:0] a;
    initial
    begin
        a <= 4'd1;
        #10 a <= 4'd0;
        #5  a <= 4'd4;
    end
    initial
        $monitor("At time t=%4t, a=%4d", $time, a);
endmodule
```

```
# At time t=    0, a=    1
# At time t=   10, a=    0
# At time t=   15, a=    4
```

```
`timescale 1ns/1ns
module blocking_d;
    reg [3:0] a;
    initial
    fork
        a = 4'd1;
        #10 a = 4'd0;
        #5  a = 4'd4;
    join
    initial
        $monitor("At time t=%4t, a=%4d", $time, a);
endmodule
```

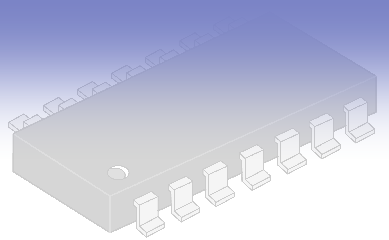
```
# At time t=    0, a=    1
# At time t=    5, a=    4
# At time t=   10, a=    0
```

过程赋值与连续赋值的比较



- 过程赋值
  - 在 initial 和 always 语句内使用
  - 赋值符号
    - 阻塞 “=”
    - 非阻塞“<=”
    - 无关键字：assign
  - 驱动寄存器
  - 根据上下文环境确定执行顺序
- 连续赋值
  - 在一个模块中出现
  - 赋值符号
    - “=”
  - 使用关键字：“assign”
  - 在右端表达式中操作数的值发生变化时执行
  - 驱动线网变量
  - 与 initial、always、原语实例、UDP实例、模块实例并行执行

条件语句



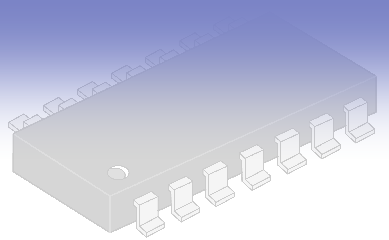
- 语法
  - if (条件1)
  - 　过程语句 1;

  - if (条件1)
  - 　过程语句 1;
  - else
  - 　过程语句 2;

  - if (条件1)
  - 　过程语句 1;
  - else if (条件2)
  - 　过程语句 2;
  - else if (条件3)
  - 　过程语句 3;

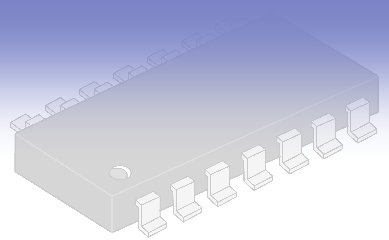
- 条件
  - 逻辑表达式
    - 必须使用括号括起来
  - 如果逻辑表达式的值为 0，x，z，则按“假”处理
  - 如果有多条过程语句，需要使用 begin…end 形成语句块
  - if 语句可以嵌套

if 语句举例



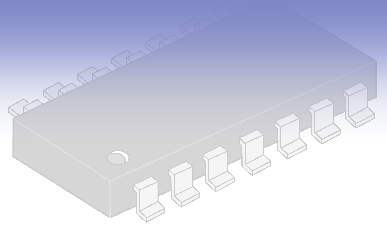
- 例1
  - if (a == 5)
  - b = 15;
  - else
  - b = 25;
- 例2
  - if (a) // 如果 a 非零
  - b = 4;
  - else if (d) // 否则如果 d 非零
  - b = 5;
  - else
  - b = 1;
- 例3
  - if (counter > 10)
  - ; // 空语句
  - else
  - begin
  - counter = counter + 1;
  - data_out = counter;
  - end

多路分支语句



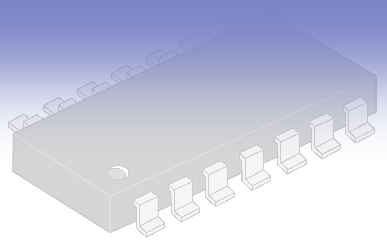
- 多路条件分支语句
- 语法
  - case (条件表达式) 或 casez (条件表达式) 或 casex(条件表达式)
  - <case 分支项>
    - 分支项 [，分支项]：过程语句；
    - 分支项：过程语句；
    - …
    - 分支项：过程语句；
    - default：过程语句
  - endcase
  - 说明
    - 对条件表达式求值
    - 依次对各分支项进行比较，如相等，就执行表达式后面的语句
    - 如分支项中没有与条件表达式的值相等的，就执行default 后面的语句
    - default项是可选的，一个 case 语句只允许有一个 default 项
    - 在 case 语句中，**x 和 z 作为值 x 和 z 解释**
    - 在 casez 语句中，表达式中的 z 是无关位 —— **忽略，不进行比较**
    - 在 casex 语句中，表达式中的 **x 和 z** 都是无关位

case 语句举例（1）



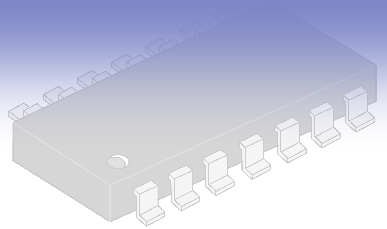
- 例1
  - reg [1:0] address;
  - **case** (address)
  - 2'b00 : statement1;
  - 2'b01, 2'b10 : statement2;
  - **default** : statement3;
  - **endcase**
- 例2
  - **reg** a;
  - **casez** (a)
  - 1'b0 : statement1;
  - 1'b1 : statement2;
  - 1'bx : statement3;
  - 1'bz : statement4;
  - **endcase**
  - 什么时候执行语句 4 ？

case 语句举例（2）



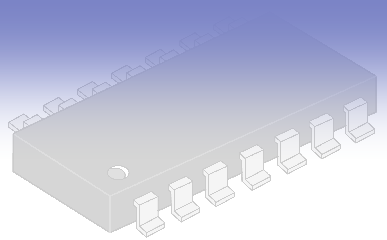
- 例3
  - reg a;
  - **case** (a)
  - 1'b0 : statement1;
  - 1'b1 : statement2;
  - 1'bx : statement3;
  - 1'bz : statement4;
  - **endcase**
  - 若 a 为 1‘b0，则执行语句1
  - 若 a 被赋值为 z，应当执行哪条语句？
- 例4
  - reg [4:0] a;
  - **case** (3’b101 << 2)
  - 3’b100 : a = 3’d1;
  - 4’b1000 : a = 4’d4;
  - 5’b1_0100 : a = 5’d16;
  - default: a = 5’dx;
  - **endcase**
  - 结果 a = ？
    - 先扩展位宽，再进行计算，然后进行比较

循环语句和禁止语句



- 禁止语句
  - disable
- 4类循环语句
  - ① forever
  - ② repeat
  - ③ while
  - ④ for

禁止语句



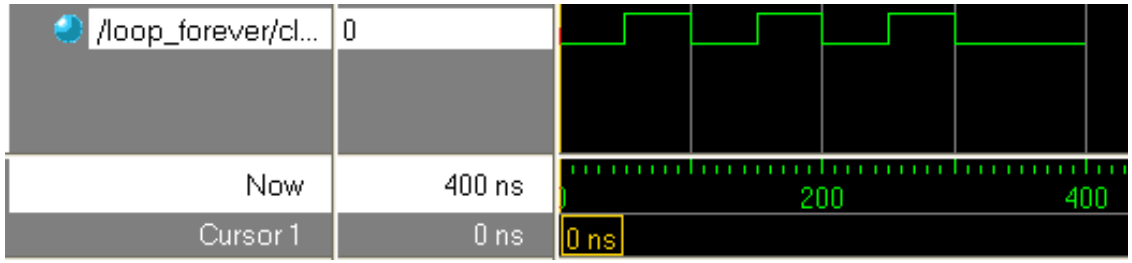
- 语法
  - disable　任务标识符；
  - disable　语句块标识符；
  - 禁止语句是过程性语句，只能出现在 always 和 initial 的语句块中
  - 使用禁止语句，可以终止任务和语句块
  - 可用于对硬件中断、全局复位的建模
  - 例、
    - begin: BLOCK0
    - 　　语句1；
    - 　　语句2；
    - 　　disable BLOCK0;
    - 　　语句3；
    - 　　语句4；
    - end
    - 　　语句5；
  - 语句3和4从未得到执行，执行禁止语句后，开始执行语句5

forever 循环语句

- 语法
  - forever
  - *procedual_statement*

```
1 `timescale 10ns / 1ns
2
3 module loop_forever;
4
5     parameter half_cycle = 5;
6     parameter stop_time  = 35;
7
8     reg clock = 0;
9
10    initial
11        begin  : clock_loop    // clock_loop is named block statement
12
13            forever
14                begin
15                    #half_cycle clock = 1;
16                    #half_cycle clock = 0;
17                end
18        end
19
20    initial
21        #stop_time disable clock_loop;
22
23 endmodule
```

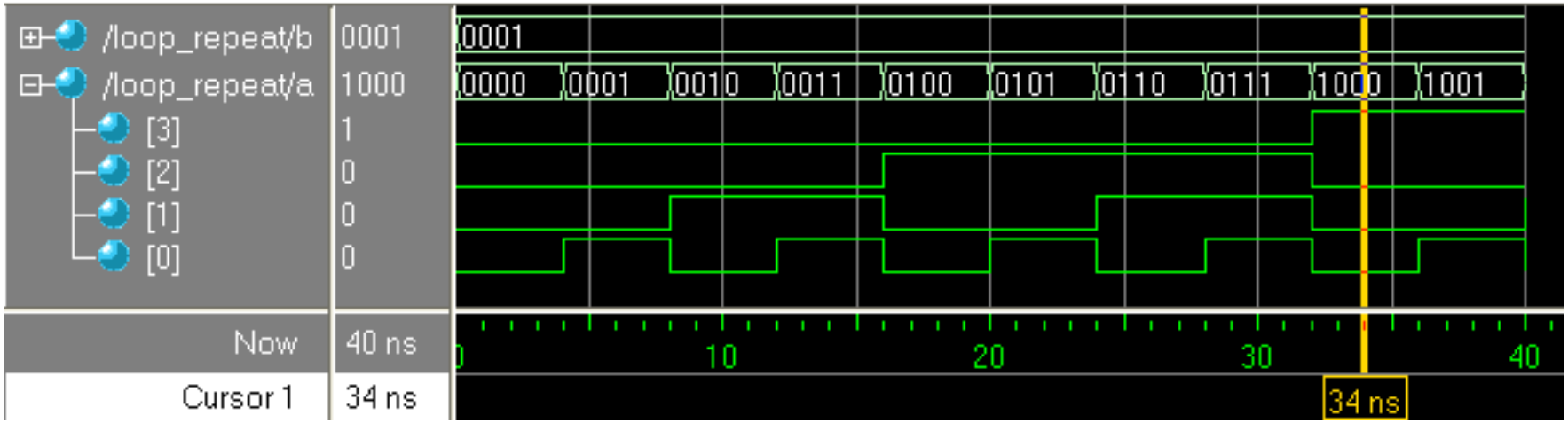
- 使用方法
  - 永远循环下去，不断地执行循环中的过程语句
  - 使用禁止语句跳出循环



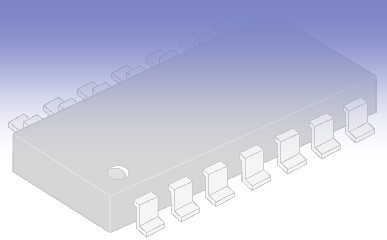
repeat 语句

- 语法
  - repeat (循环计数表达式)
  - 过程语句;
- 执行指定循环次数的过程语句
- 如果计数表达式的值不确定，即为 x 和 z，则循环次数按 0 处理

```
1
2 `timescale 1ns / 1ns
3
4 module loop_repeat;
5
6     reg [3 : 0] a, b;
7
8     initial begin
9
10        a = 4'b0000;
11        b = 4'b0001;
12
13        repeat (10) a = #4 a + b;
14
15        #5 b = 4'b0000;
16
17    end
18
19 endmodule
```



while 循环语句

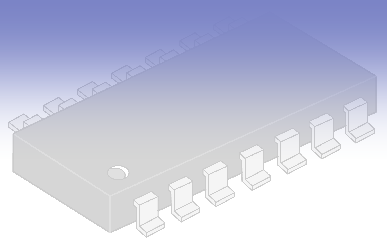


- 语法
  - while (条件表达式)
  - 过程语句;
- 执行过程语句直到条件表达式为假
- 如果条件表达式的值为 x 和 z，则按 0（假）处理

- 例

```
module test
    parameter MSB = 8;
    reg [MSB-1:0] Vector;
    integer t;
    initial
    begin
        t = 0;
        while (t < MSB);
        begin
        // Initializes vector elelments
            Vector[t] = 1'b0;
            t = t + 1;
        end
    end
endmodule
```

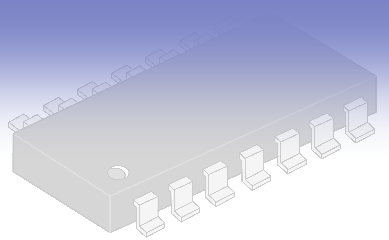
for 循环语句



- 语法
  - for (初始赋值; 条件表达式; 步长赋值)
  - 过程语句;
- 说明
  - 初始赋值
    - 指定循环变量的初始值
  - 条件表达式
    - 指定循环结束的条件，只有条件为真，循环语句就继续执行
  - 步长
    - 增加/减少循环变量的计数
- 例

```
initial begin
    for (index=0; index < 10; index = index + 2)
        mem[index] = index;
end
```

模块的结构



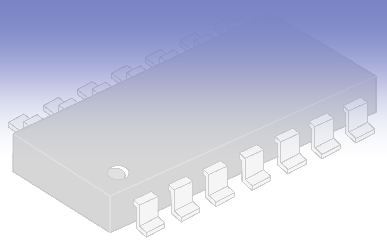
```
module fname( port_list );

    // 门原语
    nand ( f1, a, b );

    // 连续赋值语句
    assign f2 = c ^ d;

    // 模块实例引用
    mux2x1 u0( y, a, b, sel);


    initial begin

        .......
    end


    always @(*) begin

        .......
    end

endmodule
```

- 问题
  - 怎样使用条件、多项选择以及循环语句控制门原语、连续赋值、模块实例调用

```
if   语句
case   语句
for   while   repeat
forever
```

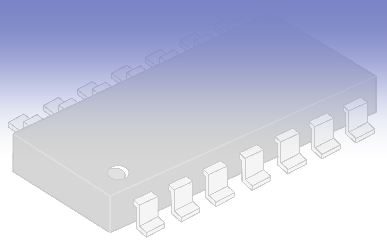
```
if   语句
case   语句
for   while   repeat
forever
```

生成块



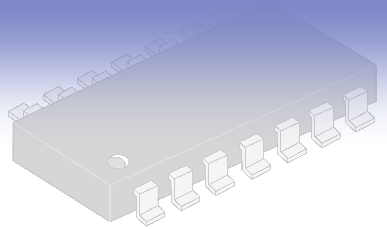
- 语法
  - 使用关键字 **generate—endgenerate** 对，指定生成范围
  - 生成语句可以简化代码编写
  - 生成重复操作
- 生成实例
  - 具有唯一的表示符
  - 可以用层次命名规则引用
- 可以生成的实例种类
  - 模块实例引用
  - 用户定义原语
  - 门级原语
  - 连续赋值语句
  - initial 和 always 块

在生成域中的数据类型



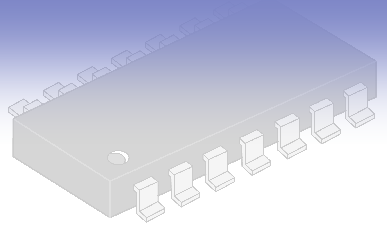
- 可以声明的数据类型
  - wire 和 reg
    - integer、real（实数型）、time 和 realtime（实数时间型）
  - event（事件）
  - 具有唯一的标识符
  - 可以使用层次引用
- 可以声明任务和函数
  - 具有唯一的标识符
  - 可以使用层次引用
- 不允许出现的声明
  - 参数、局部参数
  - 输入、输出和输入/输出声明

三种生成语句



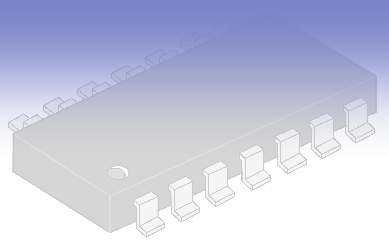
- 循环生成
- 条件生成
- case 生成

循环生成语句



- 在循环生成语句中可以多次引用
  - 变量声明
  - 模块实例
  - 用户定义原语、门级原语
  - 连续赋值语句
  - initial 和 always 块

对两个N位总线变量进行按位异或



```
module bitwise_xor (out, i0, i1);

parameter N = 32; // 32-bit bus by default

output [N-1:0] out;
input [N-1:0] i0, i1;

// 声明用于循环生成块的循环控制变量，只用于生成快
genvar j;

// 生成按位异或
generate for (j=0; j<N; j=j+1)
begin : xor_loop
    xor g1 (out[j], i0[j], i1[j]);
end
endgenerate // 生成块结束

// 另一种设计方式，使用 always 过程语句块
// reg [N-1:0] out;
// generate for (j=0; j<N; j=j+1) begin: bit
//     Always @(*) out[j] = i0[j] ^ i1[j];
// end
// endgenerate

endmodule
```

条件生成

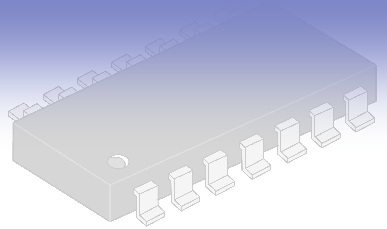
```
module half_adder(output cout, sum,
                  input a, b);
    assign {co, sum} = a + b;
endmodule
```

```
module full_adder(output cout, sum,
                  input  a, b, cin);
    assign {co, sum} = a + b + ci;
endmodule
```

- 例、根据条件循环生成

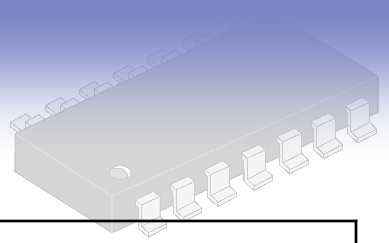
```
module adder #(parameter SIZE=4) ( output cout,
                                   output [SIZE-1:0] sum,
                                   input [SIZE-1:0] a, b );
    wire [SIZE-1:0] carry;
    genvar gk;

    generate
        for ( gk=0; gk<SIZE; gk=gk+1 )
        begin: gen_adder
            if ( gk==0 ) // 最低位无进位参与计算
                half_adder m_ha(.cout(carry[gk]), .sum(sum[gk]),
                                .a(a[gk]), .b(b[gk]));
            else
                full_adder m_fa(.cout(carry[gk]), .sum(sum[gk]),
                                .a(a[gk]), .b(b[gk]),
                                .cin[carry[gk-1]));
        end
    endgenerate
endmodule
```

case生成语句



- 在生成语句中有条件的引用
  - 模块实例
  - 用户定义原语、门级原语
  - 连续赋值语句
  - initial 和 always 块

case 生成语句举例



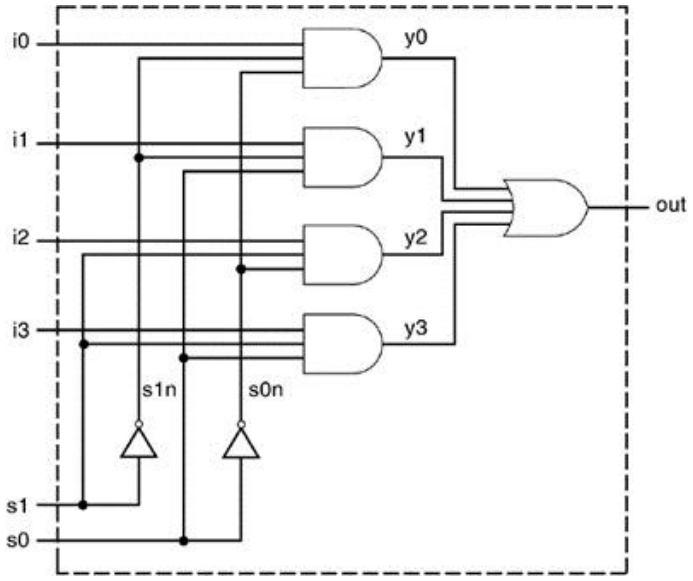
```
// 本模块产生 N-bit 加法器
module adder(co, sum, a0, a1, ci);
// 参数声明，此参数可以重载
parameter N = 4; // 缺省为 4-bit 总线
// 端口声明
output [N-1:0] sum;
output co;
input [N-1:0] a0, a1;
input ci;
// 根据总线位宽，调用（实例引用）相应的加法器
generate
case (N)
  // 当 N = 1 or 2 时，选择 1 位或 2 位 加法器
  1: adder_1bit adder1(c0, sum, a0, a1, ci); //1-bit implementation
  2: adder_2bit adder2(c0, sum, a0, a1, ci); //2-bit implementation
  // 缺省为 N-bit carry look ahead adder
  default: adder_cla #(N) adder3(c0, sum, a0, a1, ci);
endcase
endgenerate
endmodule
```

行为级建模举例

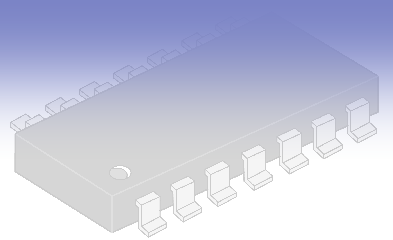
- 例、四选一多路选择器

```
// 4-to-1 multiplexer.
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// I/O 端口声明
output reg out;
input i0, i1, i2, i3;
input s1, s0;

// always @( s1 or s0 or i0 or i1 or i2 or i3 )
always @(*)
begin
    case ({s1, s0})
        2'b00: out = i0;
        2'b01: out = i1;
        2'b10: out = i2;
        2'b11: out = i3;
        default: out = 1'bx;
    endcase
end

endmodule
```

s1	s0	out
0	0	I0
0	1	I1
1	0	I2
1	1	I3

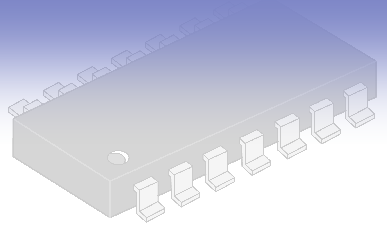


例、4 位计数器



```
// 4-bit Binary  counter
module counter(Q , clock, clear);

// I/O ports
output reg [3:0] Q;
input clock, clear;

// always @( posedge clear or negedge clock)
always @( posedge clear or negedge clock)
begin
  if (clear)
     Q <= 4'd0;  // 时序逻辑电路，使用非阻塞赋值
  else
     Q <= Q + 1;
end

endmodule
```

例、编码转换（1）



- 二进制编码转换成格雷码
  - 转换公式

$$Binary(b_{N-1}b_{N-2}b_{N-3}\cdots b_1b_0) \Rightarrow Gray(g_{N-1}g_{N-2}g_{N-3}\cdots g_1g_0)$$

$$g_{N-1} = b_{N-1}$$

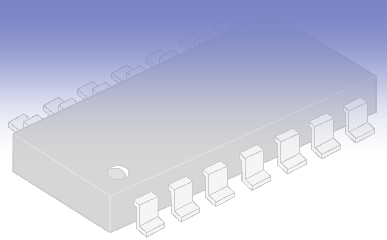
$$g_{N-2} = b_{N-1} \oplus b_{N-2}$$

$$\cdots\cdots$$

$$g_0 = b_1 \oplus b_0$$

  - 例： Binary = 11001010
  - Gray = 10101111

二进制编码转换成格雷码



$$Binary(b_{N-1}b_{N-2}b_{N-3}\cdots b_1b_0) \Rightarrow Gray(g_{N-1}g_{N-2}g_{N-3}\cdots g_1g_0)$$

$$g_{N-1} = b_{N-1}$$

$$g_{N-2} = b_{N-1} \oplus b_{N-2}$$

……

$$g_0 = b_1 \oplus b_0$$

设计一、

```
module bin2gray1 #(parameter N=8)
    ( output reg [N-1:0] gray_val, input [N-1:0] bin_val );
    always @(*) gray_val = { bin_val[N-1], bin_val[N-1:1]^bin_val[N-2:0]};
endmodule
```

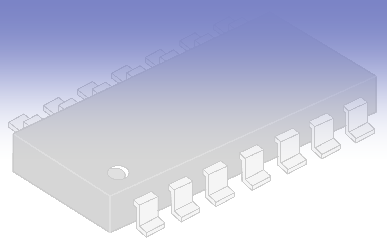
设计二、

```
module bin2gray2 #(parameter N=8) ( output [N-1:0] gray_val,
                                    input [N-1:0] bin_val );
    genvar k;
    generate for (k=0; k<N; k=k+1 ) begin:loop
        assign gray_val[k] = ( k==N-1) ? bin_val[k] : bin_val[k]^bin_val[k+1];
    end
    endgenerate
endmodule
```

设计三、

```
module bin2gray3 #(parameter N=8) ( output reg  [N-1:0] gray_val,
                                    input [N-1:0] bin_val );
    genvar k;
    generate for (k=0; k<N; k=k+1 ) begin:loop
        always @(*) gray_val[k] = ( k==N-1) ? bin_val[k] : bin_val[k]^bin_val[k+1];
    end
    endgenerate
endmodule
```

二进制编码转换成格雷码



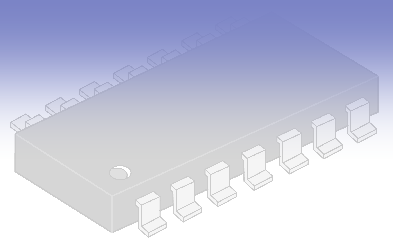
仿真测试

设计一、

```
module bin2gray1 #(parameter N=8)
    ( output reg [N-1:0] gray_val, input [N-1:0] bin_val );
    always @(*) gray_val = { bin_val[N-1], bin_val[N-1:1]^bin_val[N-2:0]};
endmodule
```

```
`timescale 1ns/1ns
module bin2gray1_tb;
    parameter N = 8, NLOOP = 2**N;

    reg [N-1:0] p_bin;
    wire [N-1:0] p_gray;

    bin2gray1 #(N) b2g1( .gray_val(p_gray), .bin_val(p_bin) );

    integer k;
    initial begin
        p_bin = 8'b1;
        for ( k=1; k<NLOOP-1; k=k+1 ) #5 p_bin = p_bin + 1'b1;
    end

    initial
        $monitor("At time t=%4t, k=%4d, bin=%8b, ----> gray=%8b", $time, k, p_bin, p_gray );

    initial #1290 $stop;
endmodule
```

仿真测试结果

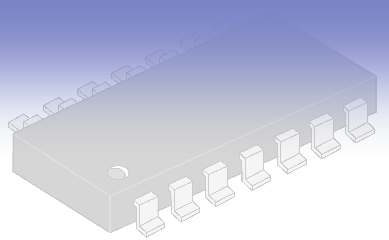


设计一、

```
module bin2gray1 #(parameter N=8)
    ( output reg [N-1:0] gray_val, input [N-1:0] bin_val );
    always @(*) gray_val = { bin_val[N-1], bin_val[N-1:1]^bin_val[N-2:0]};
endmodule
```

```
# At time t=1195, k= 240, bin=11110000, ----> gray=10001000
# At time t=1200, k= 241, bin=11110001, ----> gray=10001001
# At time t=1205, k= 242, bin=11110010, ----> gray=10001011
# At time t=1210, k= 243, bin=11110011, ----> gray=10001010
# At time t=1215, k= 244, bin=11110100, ----> gray=10001110
# At time t=1220, k= 245, bin=11110101, ----> gray=10001111
# At time t=1225, k= 246, bin=11110110, ----> gray=10001101
# At time t=1230, k= 247, bin=11110111, ----> gray=10001100
# At time t=1235, k= 248, bin=11111000, ----> gray=10000100
# At time t=1240, k= 249, bin=11111001, ----> gray=10000101
# At time t=1245, k= 250, bin=11111010, ----> gray=10000111
# At time t=1250, k= 251, bin=11111011, ----> gray=10000110
# At time t=1255, k= 252, bin=11111100, ----> gray=10000010
# At time t=1260, k= 253, bin=11111101, ----> gray=10000011
# At time t=1265, k= 254, bin=11111110, ----> gray=10000001
# At time t=1270, k= 255, bin=11111111, ----> gray=10000000
```

例、编码转换（2）



- 格雷码转换成二进制编码
  - 转换公式

$$Gray(g_{N-1}g_{N-2}g_{N-3}\cdots g_1g_0) \Rightarrow Binary(b_{N-1}b_{N-2}b_{N-3}\cdots b_1b_0)$$

$$b_{N-1} = g_{N-1}$$

$$b_{N-2} = b_{N-1} \oplus g_{N-2}$$

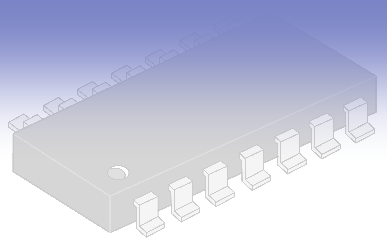
$$\cdots\cdots$$

$$b_0 = b_1 \oplus g_0$$

  - 例： Gray = 10101111 ➔ Binary = 11001010
  - Verilog 模块

```
// Convert Gray code into binary code
module gray2bin1 #(parameter N=8)
    ( output reg [N-1:0] bin, input [N-1:0] gray );

    genvar k;
    generate for (k = 0; k < N; k = k + 1) begin:LOOP
        always @(*) bin[k] = (k==N-1) ? gray[N-1] : ^gray[N-1:k];
    end endgenerate
endmodule
```

小结



- initial 语句块
  - initial块从仿真0时刻开始执行，在整个仿真过程中只执行一次
  - initial 语句块用于描述测试模块
- always 语句块
  - 从仿真0时刻开始执行，无限循环下去
  - always 语句块用于设计时序逻辑电路
- 语句块
  - 顺序语句块（begin … end）
  - 并行语句块（fork … join）
- 过程赋值只能对 reg 类型变量进行赋值
  - 阻塞赋值
  - 非阻塞赋值
- 延时
  - 常规延时、内嵌延时、0延时