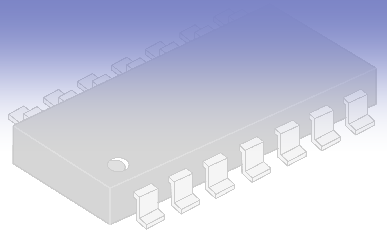
第 5 章

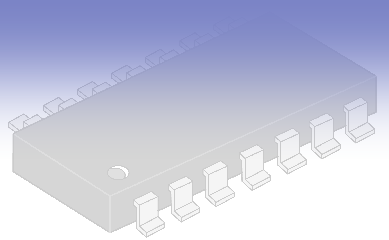
数据流建模

内容



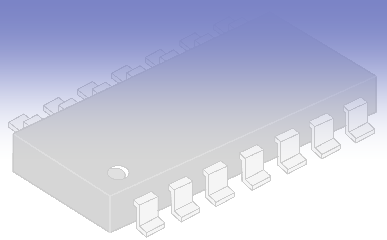
- 连续赋值语句
- 延时

引子



- 数据流（data-flow）描述
  - 采用比门级描述更高的抽象层次
  - 通过说明数据流程对模块进行描述
    - 输入 → 处理 → 输出
  - 描述**处理功能**的实现

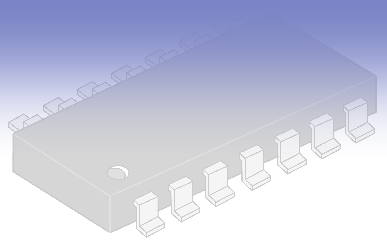
连续赋值语句



- 连续赋值语句将值赋给线网(wire)
  - 连续赋值语句不能对寄存器类型赋值
- 连续赋值语句使用关键字 assign
- 只要右边表达式中的操作数上有事件发生 —— 表达式立即被计算，新结果赋给左边的线网
- 语法

连续赋值 ::= **assign** [驱动强度] [延时] 线网赋值列表;
线网赋值列表 ::= 线网赋值 {, 线网赋值 }
线网赋值 ::= net_lvalue = expression

线网赋值列表

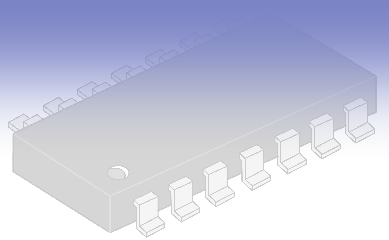


连续赋值 ::= **assign** [驱动强度] [延时] 线网赋值列表;
线网赋值列表 ::= 线网赋值 {, 线网赋值 }
线网赋值 ::= net_lvalue = expression

- assign mux = (s == 0) ? A : 'bz, // 逗号（,）结尾
- mux = (s == 1) ? B : 'bz, // 逗号（,）结尾
- mux = (s == 2) ? C : 'bz, // 逗号（,）结尾
- mux = (s == 3) ? D : 'bz; // 最后是分号（;）

- 相当于下面 4 个独立的连续赋值语句的简化形式：
  - assign mux = (s == 0) ? A : 'bz; //分号（;）结尾
  - assign mux = (s == 1) ? B : 'bz; //分号（;）结尾
  - assign mux = (s == 2) ? C : 'bz; //分号（;）结尾
  - assign mux = (s == 3) ? D : 'bz; //分号（;）结尾

目标类型

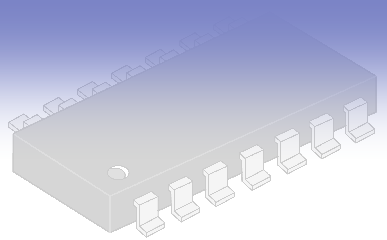


连续赋值的目标类型

- 线网标量 —— 默认位宽为 1 位的标量
- 线网向量 —— 指定位宽声明为向量
- 部分位选择
- 上述类型的任意的连接运算结果

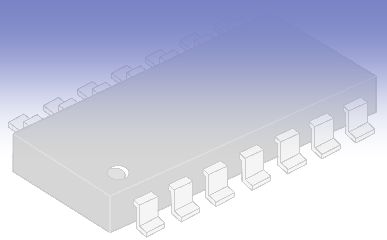
```
// 连续赋值，out 必须是线网类型变量，
// i1 和 i2 可以是 wire 类型， 也可以是 reg 类型
assign out = i1 & i2

// 对线网向量进行连续赋值
assign addr[15:0] = addr1_bits[15:0] ^& addr2_bits[15:0];

// 对一个1位线网标量和线网向量的连接（拼接）进行赋值
assign {c_out, sum[3:0]} = a[3:0] + b[3:0] c_in;
```

标量类型线网和向量类型线网



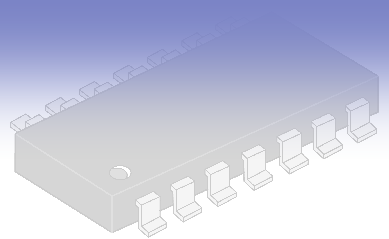
- 标量类型/向量类型线网/寄存器类型声明
- ***Vectored*** and ***scalared*** shall be optional advisory keywords to be used in **vector net** or **reg** declaration.
  - **标量型**线网向量
  - wire scalared [63:0] bus; // 可以位选择/部分位选择
    - **If the keyword scalared is used, bit-selects and part-selects of the object shall be permitted**
  - **向量型**线网向量
  - wire vectored [31:0] data; // 不许以位选择/部分位选择
    - **If the keyword vectored is used, bit-selects and part-selects may not be permitted**

对一个线网标量和线网向量的拼接目标赋值



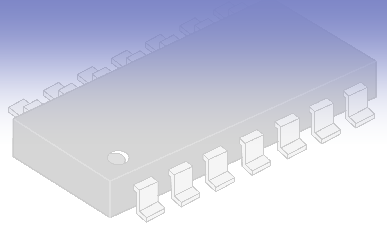
- wire　cout，cin;
- wire　[3 : 0]　sum，a，　b;
- assign　{cout, sum} = a + b + cin;
- 说明
  - 左边表达式是 5 位宽，cout 1 位，sum 4 位
  - a 和 b 是 4 位宽，相加的结果为 5 位
  - 将右边表达式计算结果的最右边 4 位赋给 sum，第 5 位（进位位）赋给 cout

连续赋值语句的执行



- 只要右边表达式中的操作数上有事件发生 —— 表达式立即被计算，新结果赋给左边的线网
- 事件 —— 操作数值的变化
- 例
  - wire [3 : 0] y, preset, clear;　　// 线网说明
  - assign　y = preset & clear;
    - 如果 preset 或 clear 有变化，计算右边的整个表达式
    - 如果结果变化，那么结果立即赋给线网 y

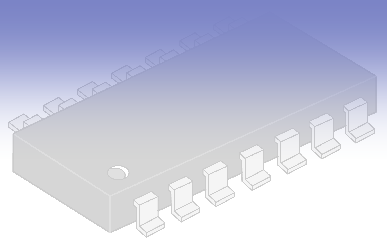
隐式连续赋值（1）



- 在线网声明的同时对其赋值

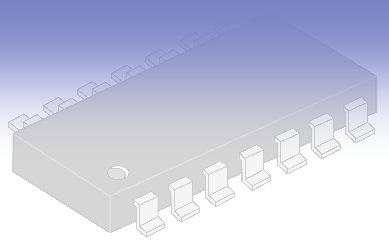
```
// 普通的连续赋值
wire out;
assign out = in1 & in2;

// 隐式连续赋值，实现与上面两条语句相同的功能
wire out = in1 & in2;
```

隐式连续赋值（2）



- 连续赋值可以作为线网说明本身的一部分
- 例
  - `wire [3 : 0] sum = 4'b0;`
  - `wire clear = 1'b1;`
  - `wire A_GT_B = A > B, B_GA_A = B > A;`
  - 这里
    - `wire clear = 1'b1;`
  - 等价于
    - `wire clear;`
    - `assign clear = 1'b1;`

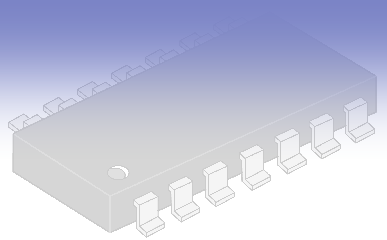
隐式线网声明



- 声明方法
  - 一个信号名被用在连续赋值语句的左边
- 例

```
// 连续赋值, out 是一个线网变量.
wire i1, i2;
assign out = i1 & i2;  // 虽然没有声明 out 为线网类型
```

延时

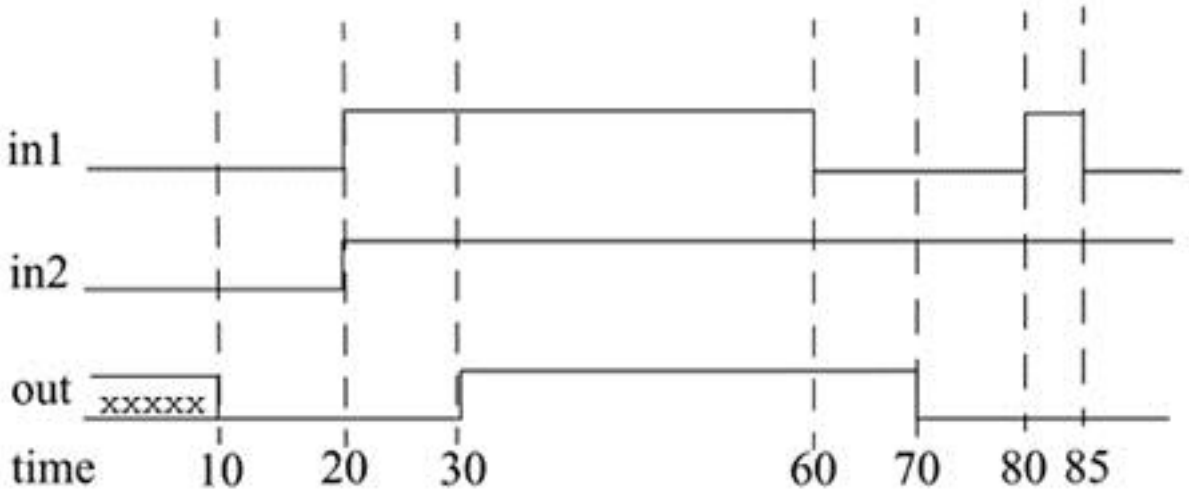


- 表达式右边的操作数发生变化传递到左边变量的时间间隔
- 三种延时
  - 普通赋值延时
  - 隐式赋值延时
  - 线网声明延时

普通赋值延时

- 在连续赋值语句中说明延时值
  - 延时值位于关键字 **assign** 的后面
- 例

```
assign #10 out = in1 & in2;
```

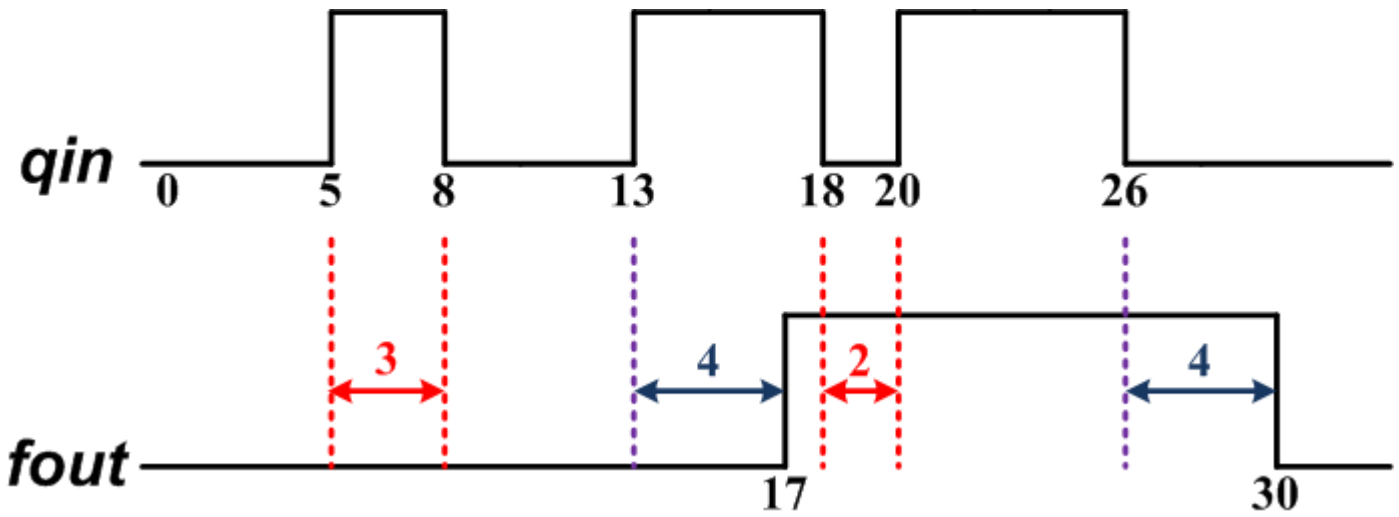


- 惯性延时——小于延时的脉冲将被取消
  - **信号保持的持续时间必须大于延时宽度**
- 赋值过程
  - 右边表达式中任意一个变量发生中变化，计算表达式的值
  - 计算表达式 i1 & i2 的新值赋给语句左值之前，需经过10个时间单位的延时
  - 如果在此期间，i1 及 i2 再发生变化，采用当前新值计算表达式

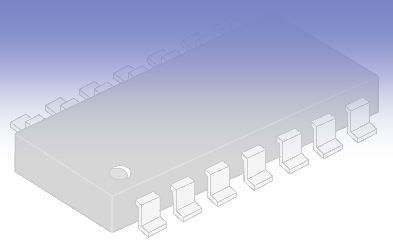
惯性延时

- 脉冲宽度小于赋值的延时的输入变化不会对输出产生影响
  - 赋值语句将右边表达式的中的变化传递的左边之前，其值必须在延时期间保持不变；
  - 信号保持的持续时间必须大于延时宽度
    - 若延时期间发生了变化，前面的值被消除，不能传递到左边。
    - 采用当前新值重新计算表达式
- 例、

```
assign #4 fout = qin;
```



隐式连续赋值延时

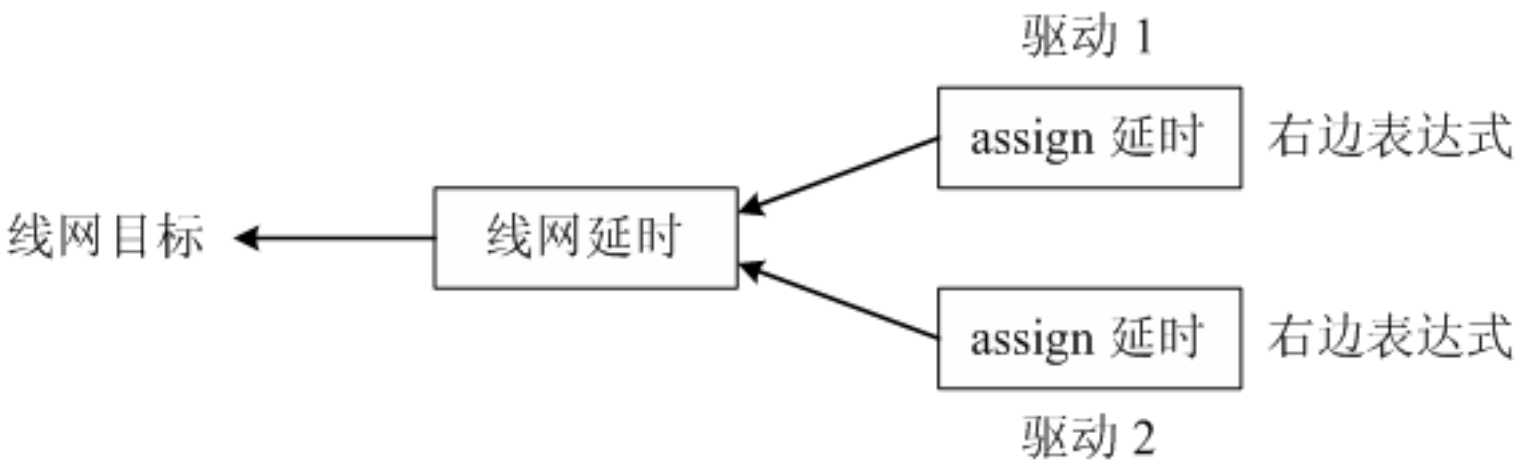


- 使用隐式连续赋值语句来说明对线网的赋值以及赋值延时
- 例

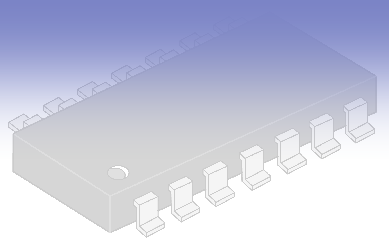
```
// 隐式连续赋值延时
wire #10 out = in1 & in2;

// 等效于
wire out;
assign #10 out = in1 & in2;
```

线网延时

- 延时也可以在线网说明中定义
  - 在线网类型的变量声明时，指定一个延时的值
  - **wire #5 rd;** —— 驱动源的值改变与线网 rd 本身之间的延时
  - 例
    - assign #2 rd = a & b;
    - 假设在仿真过程的第 10 个时刻，
    - a 上的事件导致右边表达式开始计算，
    - 经过 7 个时间单位延时，表达式结果赋值给左值
    - 由于定义了线网延时，**赋值发生在 17 (= 10 + 2 + 5)**
- 线网延时的过程



线网说明赋值延时



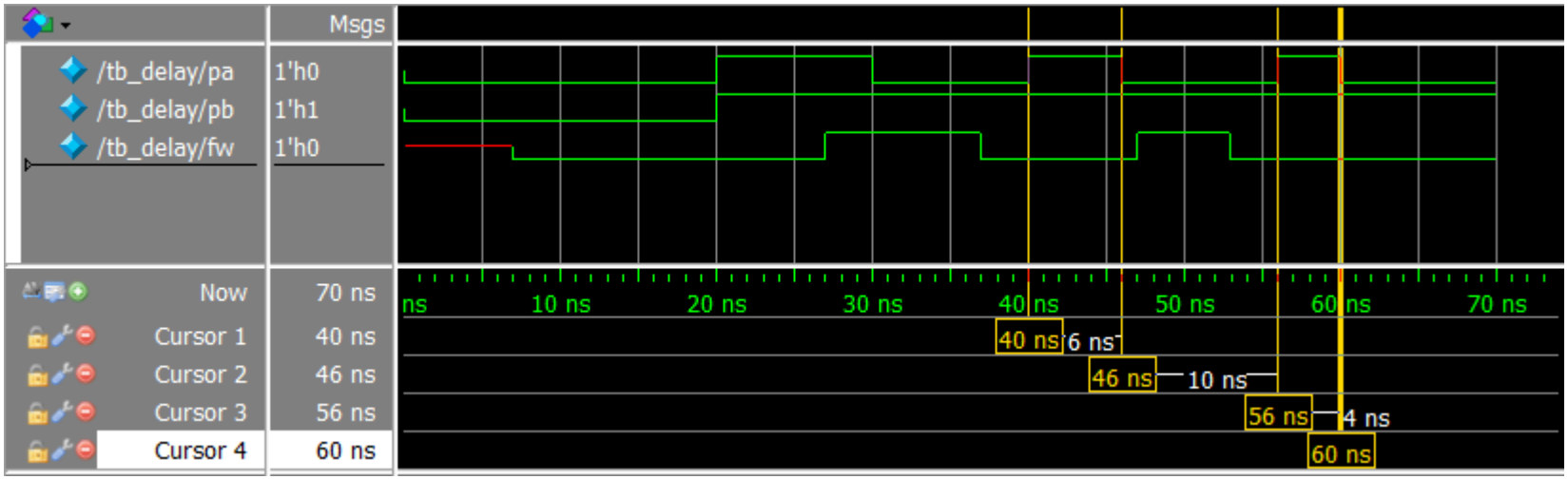
- 在线网说明赋值中的延时是赋值延时
  - `wire #2 a = b | c;  // 赋值延时`
- 例

```
// 线网延时
wire #10 out;
assign out = in1 & in2;

// 等效语句
wire out;
assign #10 out = in1 & in2;
```

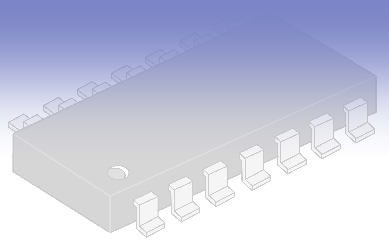
线网延时举例

- 既有线网延时又有赋值延时，信号保持的持续时间必须大于二个延时的最大值
  - max(线网延时, 赋值延时)
  - Net Declaration Delay

```
module wDelay( output f, input a, b);
    wire #5 temp;
    assign #2 temp = a & b;

    assign f = temp;
endmodule
```



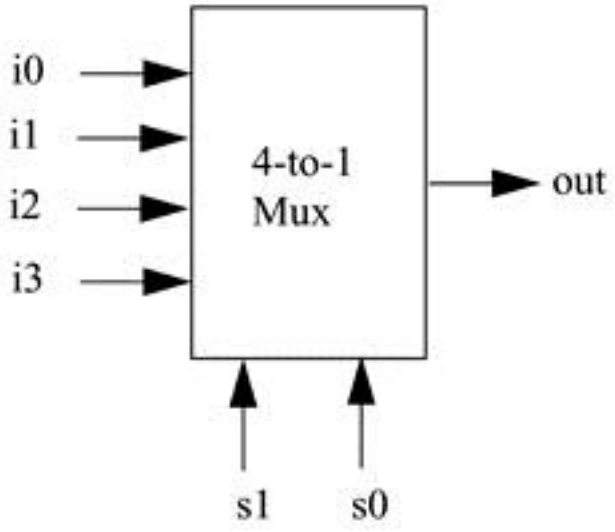
延时定义



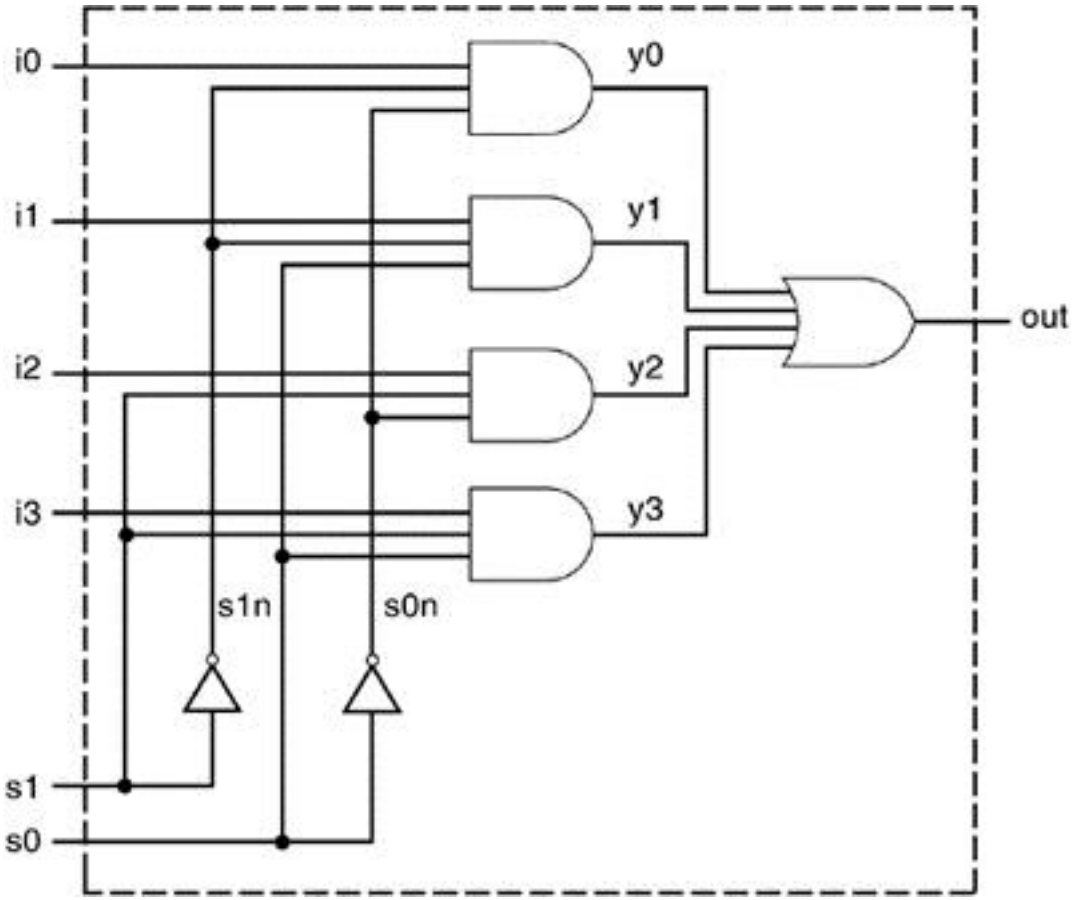
- 延时定义有三类
  - #value 或 #(value)
  - #(value, value)
  - #(value, value, value) —— #(上升延时, 下降延时, 关闭延时)
- 在连续赋值语句使用的语法
  - `assign #(rise, fall, turn-off) target = expression;`
  - 例
    - `assign #4  a = b | c;`
    - `assign #(4,8) a = q;`
    - `assign #(4,6,8)  a = &databus;`
    - `assign bus = men_addr[15:8];`

例、四选一多路选择器

- 模块框图、真值表和逻辑原理图

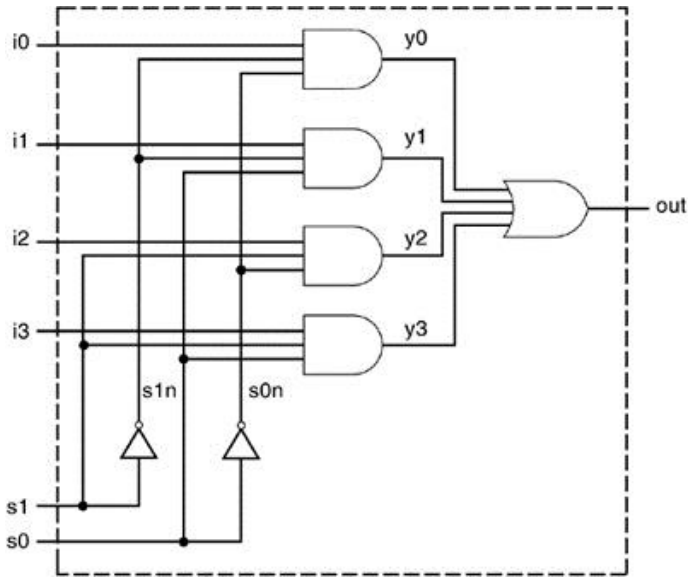


s1	s0	out
0	0	I0
0	1	I1
1	0	I2
1	1	I3



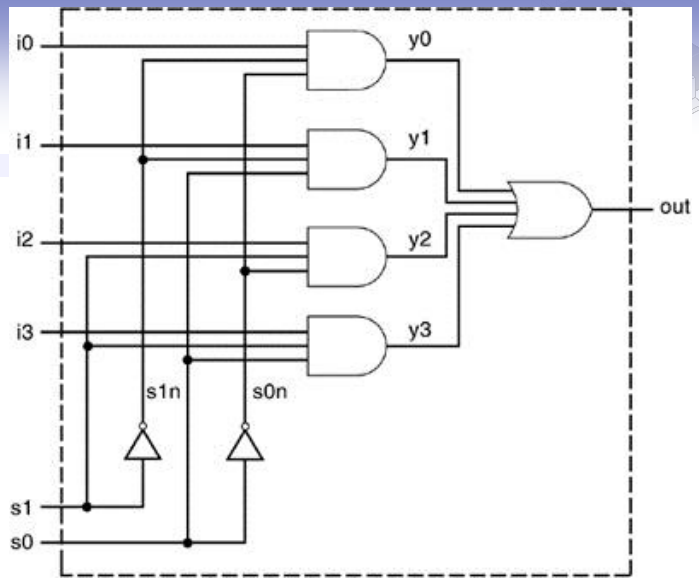
多路选择器描述（1）

- 门级描述
  - 使用门级原语




```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;

// Internal wire declarations
wire s1n, s0n;
wire y0, y1, y2, y3;

// Gate instantiations
// Create s1n and s0n signals.
not (s1n, s1);
not (s0n, s0);

// 3-input and gates instantiated
and (y0, i0, s1n, s0n);
and (y1, i1, s1n, s0);
and (y2, i2, s1, s0n);
and (y3, i3, s1, s0);

// 4-input or gate instantiated
or (out, y0, y1, y2, y3);

endmodule
```

多路选择器描述（2）



- 数据流描述
  - 使用逻辑方程

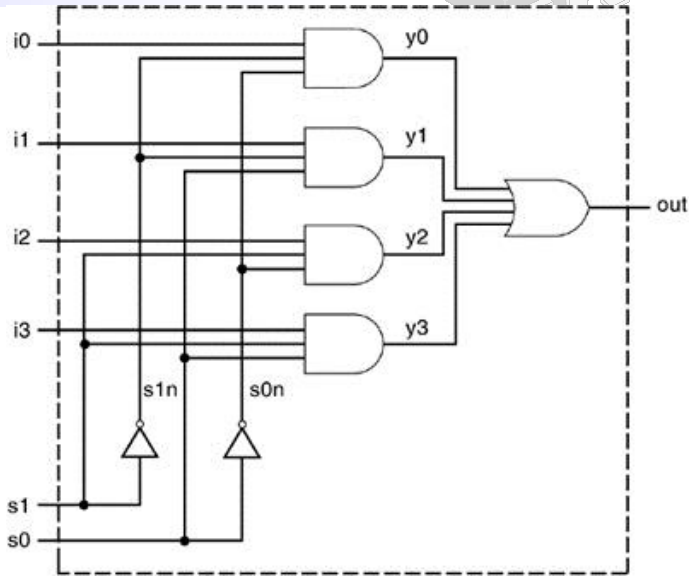
s1	s0	out
0	0	I0
0	1	I1
1	0	I2
1	1	I3

```
// 使用逻辑方程描述
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

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

// 产生输出的逻辑方程
assign out = (~s1 & ~s0 & i0)|
             (~s1 &  s0 & i1)|
             ( s1 & ~s0 & i2)|
             ( s1 &  s0 & i3) ;
endmodule
```

多路选择器描述（3）

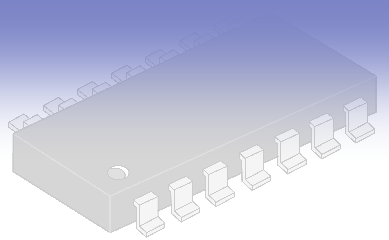
- 数据流描述
  - 使用条件操作符

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




```
module mux4x1 ( output out,
                input i0, i1, i2, i3,
                input s1, s0 );

    // 使用条件操作符
    assign out = s1 ? ( s0 ? i3 : i2) : (s0 ? i1 : i0);

endmodule
```

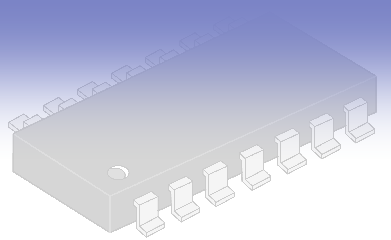
例、8 位数值比较器



- 数据流模型（Dataflow Modeling）
  - 使用关系运算符

```
module comparator #(parameter N=8)
     ( output aGTb, aEQb, aLTb,
       input [N-1:0] a, b );

    parameter EQ_DELAY = 5,
              LT_DELAY = 8,
              GT_DELAY = 8;

    assign #EQ_DELAY aEQb = a == b;
    assign #GT_DELAY aGTb = a > b;
    assign #LT_DELAY aLTb = a < b;

endmodule
```

例、4 位全加器

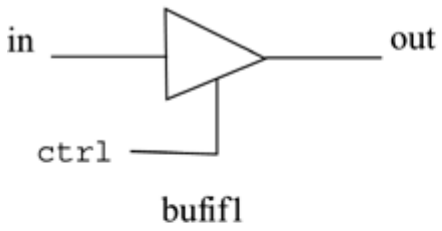


- 数据流模型（Dataflow Modeling）
  - 使用拼接运算符

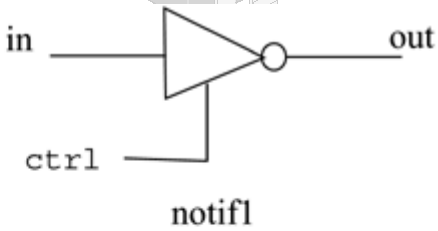
```
module fulladder #(parameter N=4)
     ( output [N-1:0] sum,
       output c_out,
       input  [N-1:0] a, b,
       input  c_in );

     // 定义全加器的功能
     assign {c_out, sum} = a + b + c_in;

endmodule
```

使用条件操作符表示三态门

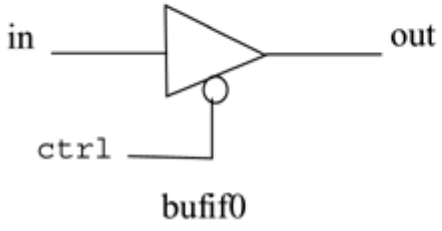
- 原语：
  - **bufif1，bufif0，notif1， notif0**
- 功能
  - 只有在控制信号有效时才能传递数据（信号）
  - 如果控制信号无效，则输出为高阻抗 **z**
- 连续赋值语句
  - 条件表达式



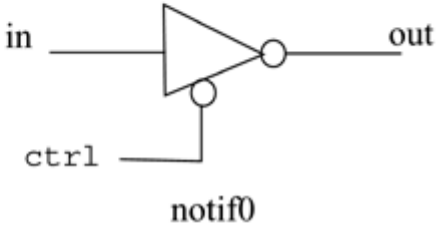
bufif1



notif1



bufif0



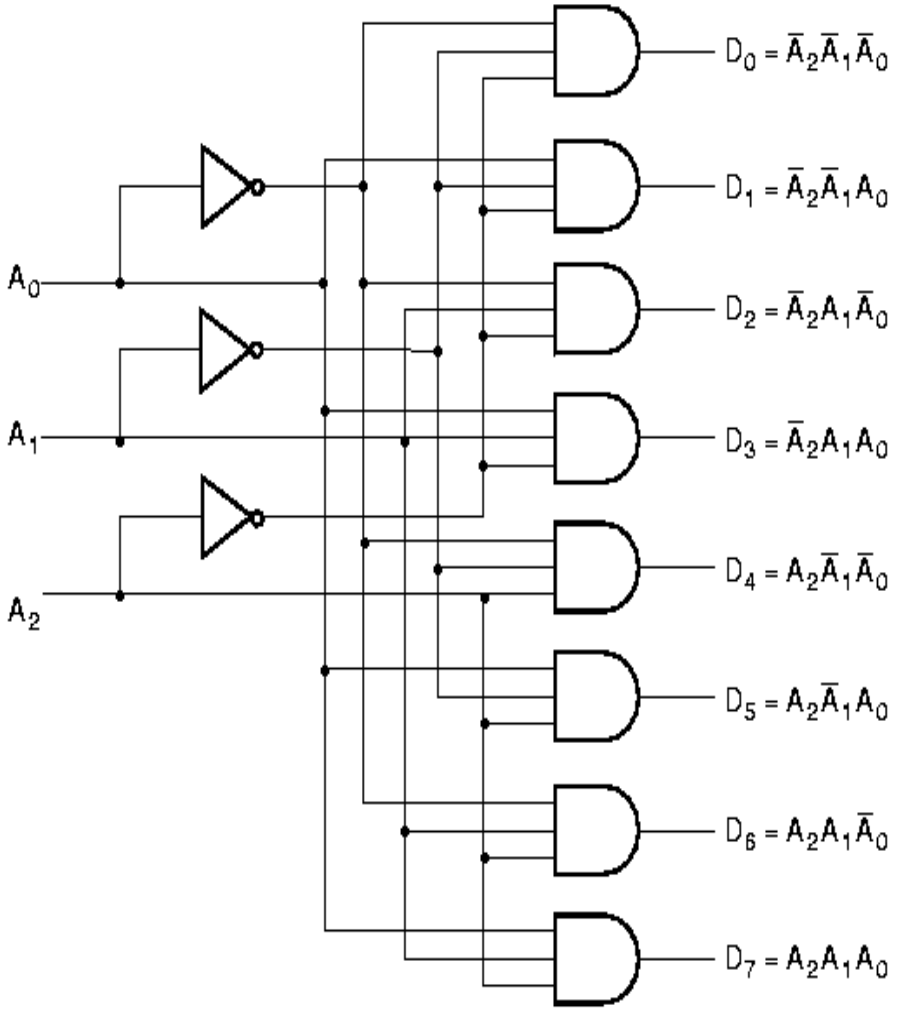
notif0

```
assign y0 =  (ctrl) ?  in : 1'bz; // bufif1
assign y1 =  (ctrl) ? ~in : 1'bz; // notif1
assign y2 = (!ctrl) ?  in : 1'bz; // bufif0
assign y3 = (!ctrl) ? ~in : 1'bz; // notif0
```

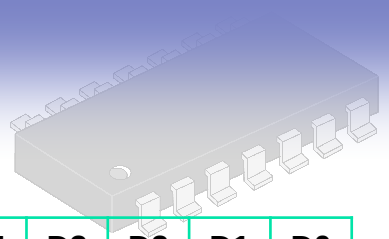
使用条件操作符描述译码器（1）

- 3-8译码器（3-to-8 Decoder）
  - 真值表和原理图

A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0



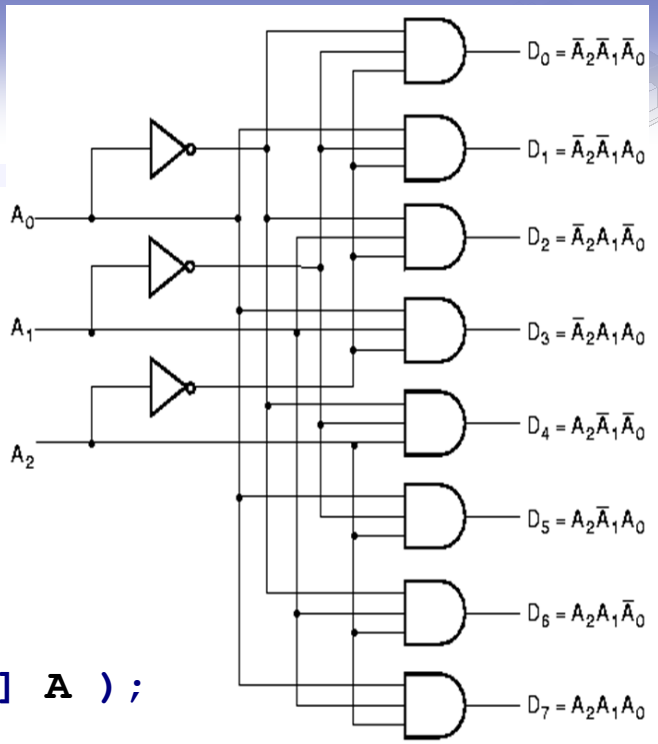
使用条件操作符描述译码器（2）



A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

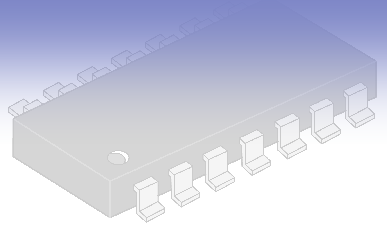
```
Module decoder3x8 ( output [7:0] fout, input [2:0] din );
   assign fout = (din == 3'b000 ) ? 8'b0000_0001 :
                 (din == 3'b001 ) ? 8'b0000_0010 :
                 (din == 3'b010 ) ? 8'b0000_0100 :
                 (din == 3'b011 ) ? 8'b0000_1000 :
                 (din == 3'b100 ) ? 8'b0001_0000 :
                 (din == 3'b101 ) ? 8'b0010_0000 :
                 (din == 3'b110 ) ? 8'b0100_0000 :
                 (din == 3'b111 ) ? 8'b1000_0000 : 8'h00;
endmodule
```

使用门原语描述译码器

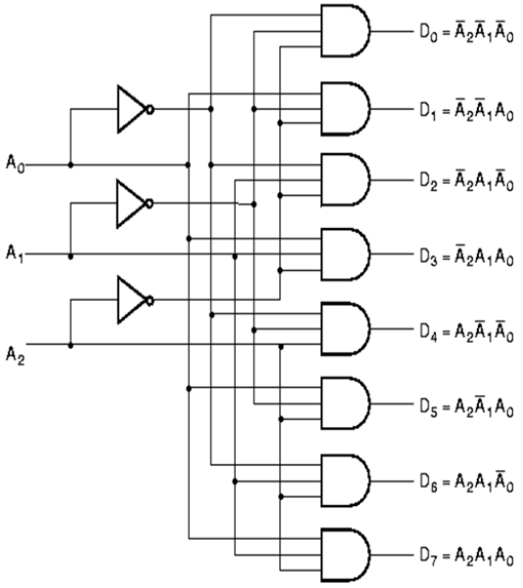



```
module decoder3x8( output [7:0] D, input [2:0] A );
    wire [2:0] Abar;

    not u1 [0:2] ( Abar, A);

    and u20(D[0], Abar[2], Abar[1], Abar[0]),
        u21(D[1], Abar[2], Abar[1],    A[0]),
        u22(D[2], Abar[2], A[1],    Abar[0]),
        u23(D[3], Abar[2], A[1],       A[0]),
        u24(D[4], A[2],    Abar[1], Abar[0]),
        u25(D[5], A[2],    Abar[1],    A[0]),
        u26(D[6], A[2],    A[1],    Abar[0]),
        u27(D[7], A[2],    A[1],       A[0]);
endmodule
```

连续赋值语句与门原语比较



A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0




```
Module decoder3x8 ( output [7:0] fout, input [2:0] din );
   assign fout = (din == 3'b000 ) ? 8'b0000_0001 :
                 (din == 3'b001 ) ? 8'b0000_0010 :
                 (din == 3'b010 ) ? 8'b0000_0100 :
                 (din == 3'b011 ) ? 8'b0000_1000 :
                 (din == 3'b100 ) ? 8'b0001_0000 :
                 (din == 3'b101 ) ? 8'b0010_0000 :
                 (din == 3'b110 ) ? 8'b0100_0000 :
                 (din == 3'b111 ) ? 8'b1000_0000 : 8'h00;
endmodule
```

```
module decoder3x8( output [7:0] D, input [2:0] A );
    wire [2:0] Abar;

    not u1 [0:2] ( Abar, A);

    and u20(D[0], Abar[2], Abar[1], Abar[0]),
        u21(D[1], Abar[2], Abar[1],    A[0]),
        u22(D[2], Abar[2], A[1],    Abar[0]),
        u23(D[3], Abar[2], A[1],       A[0]),
        u24(D[4], A[2],    Abar[1], Abar[0]),
        u25(D[5], A[2],    Abar[1],    A[0]),
        u26(D[6], A[2],    A[1],    Abar[0]),
        u27(D[7], A[2],    A[1],       A[0]);
endmodule
```