Spring,2020

1090140071

FPGA设计及应用

Verilog HDL基础2：逻辑设计

大连理工大学 电信学部
夏书峰

Verilog HDL逻辑设计

- 语法及设计原则
  - 阻塞赋值与非阻塞赋值
  - always块语法原则
  - 赋值语法原则
- 组合逻辑电路设计
- 时序逻辑电路设计
- 时钟同步状态机设计

1. 深入理解阻塞赋值与非阻塞赋值

- 赋值时注意两个要点：

(1) 在描述组合逻辑的always块中用阻塞赋值,则综合成组合逻辑电路结构；

(2) 在描述时序逻辑的always块中用非阻塞赋值,则综合成时序逻辑电路结构

- 以上描述要求是为了使综合前和综合后的仿真结果一致。IEEE Verilog定义了有些语句有确定的执行时间,有些没有确定的执行时间,若有两条或更多的语句准备在同一时刻执行,但由于语句排列顺序不同,却产生了不同的输出结果,这是造成模块竞争和冒险的原因之一,为避免产生竞争,理解阻塞赋值和非阻塞赋值在执行时间上的差别至关重要。

1.1 阻塞赋值

- 阻塞赋值用等号“=”表示，赋值时先计算等号右侧表达式的值,赋值过程不允许任何别的语句干扰,直到赋值操作完成,才允许别的语句执行。所谓阻塞的概念是指在同一个always块中,其后面的赋值语句即使不设定延迟,从概念上也是在前一赋值语句结束后才开始赋值的。
- 一般可综合的阻塞赋值语句不能设定有延迟(即使是0延迟也不行),从理论上讲,后面的赋值语句只有概念上的先后而无实质上的延迟。
- 若一个过程块中阻塞赋值等号左边的变量恰好是另一个过程块等号右边的变量,两个过程又用同一时钟沿触发,则执行顺序无法确定,这时阻塞赋值操作就有问题,若阻塞赋值顺序安排不好就会出现竞争。

例1 采用阻塞赋值的反馈振荡器

```
model fbosc1(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
```

例1问题解释

- 按IEEE Verilog标准,上例中两个always块是并行执行的，与前后顺序无关,若前一个always块的复位信号rst先到0时刻,则y1和y2都会取0；而若后一个always块的rst信号先到0时刻,则y1和y2都会取1。这说明例1的Verilog模块是不稳定的，必定会产生竞争和冒险的情况。

1.2 非阻塞赋值

- 非阻塞赋值用符号“<=”表示, 赋值操作开始时刻先计算<=号右面表达式的值,赋值操作时刻结束时更新<=左侧的变量。在计算右面表达式和更新左侧变量期间, 允许其它Verilog语句同时进行操作。
- 非阻塞赋值语句可看做两个步骤的过程：

(1) 赋值时刻开始,计算非阻塞赋值右侧表达式;

(2) 赋值时刻结束,更新非阻塞赋值左侧变量。

- 非阻塞赋值只能用于对寄存器型变量进行赋值,因此只能用在initial块和always块等过程块中,而且非阻塞赋值不允许用于连续赋值。如例2所示：

例2 采用非阻塞赋值的反馈振荡器

```
model 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
```

例2问题解释

- 同样，按IEEE Verilog标准,上例中两个always块是并行执行的，与前后顺序无关。无论哪一个always块的复位信号rst先到0时刻，两个always块中的非阻塞赋值都在赋值开始时刻计算右侧表达式，而在结束时刻才更新左侧变量。
- 按以上分析，在rst信号到来后，在always块结束时y1取0和y2取1是确定的。从用户角度看，这两个非阻塞赋值正好是并行执行的。

1.3 语言指导原则—always块

(1) 每个always块只能有一个事件控制“@(事件表达式)”，而且要紧跟在always关键字后面；

(2) always块既可以表示时序逻辑或组合逻辑又可以同时表示电平敏感的透明锁存器和组合逻辑，但这种描述方法容易产生错误和多余的电平敏感透明锁存器，因此不推荐；

(3) 带有posedge或negedge关键字表示边沿触发的时序逻辑，没有这两个关键字的表示组合逻辑(和)或电平敏感的锁存器，时间表达式中若有多个沿或多个电平，必须用or连接；

(4) 每个表示时序的always块只能由一个时钟跳变沿触发，置位或复位最好也用该时钟跳变沿触发；

(5) always块中赋值的变量要定义成reg型或整型，默认整型32位，使用Verilog操作符可以对其进行二进制求补算术运算;

(6) always块中应避免组合反馈回路，每次执行always块时，在生成组合逻辑的always块中赋值的所有信号必须有明确的值；否则需要设计者加入电平敏感的锁存器来保持赋值前的最后一个值。如下例：

隐含的透明锁存器

```
input a,b,c,f;
reg d, e;
always @ (a or b or c)
   begin
      e = d & a & b;
      d = f | c;
   end
```

上例中，always块中程序用到的输入信号有a、b、c、d，但事件列表中只有a、b、c三个输入信号，故d变化时，e表达式并不能立刻被计算，要等到a或b或c变化时才能体现出来，也就是说实际上相当于存在一个电平敏感的透明锁存器在起作用，把d信号的变化锁存起来。

语言指导原则—赋值

(1) 对一个寄存器型(reg)和整型(integer)变量的给定位的赋值，只允许在一个always块里进行，若在另一个always块中也对其赋值，就是非法的。(always块之间是并列关系，若在多个块里对同一个reg或integer赋值，可以想到存在冲突的可能)

(2) 把某一信号赋值为'bx，综合器就把它解释成无关状态，此时综合器为其生成的硬件电路最简洁。

1.4 Verilog模块编程要点

(1) 时序电路建模时用非阻塞赋值

(2) (电平敏感)锁存器电路建模时用阻塞赋值

(3) 用always建模组合逻辑时用阻塞赋值

(4) 在同一always块里同时建立时序逻辑和组合逻辑电路时用非阻塞赋值

(5) 在always块里不要既用非阻塞赋值又用阻塞赋值

(6) 不要在一个以上的always块里为同一个变量赋值

(7) $strobe系统任务可用来显示用非阻塞赋值的变量的值(若用$display,其执行完赋值才发生)

(8) 赋值时不要使用延迟(包括#0)

1.5 综合的一般原则

(1) 建议综合之前一定要进行仿真，因为仿真会暴露逻辑错误。若不先进行仿真，没发现的逻辑错误会进入综合器，使综合的结果产生同样的逻辑错误；

(2) 每一次布局布线之后都要进行仿真，在器件编程或流片之前要做最后的仿真；

(3) 用Verilog HDL描述的异步状态机不能综合，因此应避免用综合器来设计异步状态机，可以改用电路图输入方法来设计；

(4) 若要为电平敏感的锁存器建模，使用连续赋值语句(assign)是最简单的方法。

1.6 同步设计的5条原则

在进行FPGA设计时，应当采用同步设计方法，以避免由于器件工艺改进或外界环境因素变化而造成异步逻辑工作不可靠。单时钟域的同步设计应遵循以下原则：

(1) 所有数据都要通过组合逻辑和触发器，这些触发器被同一个时钟信号同步；

(2) 延时总是由同步延时单元（触发器）控制，而不是由组合逻辑控制；

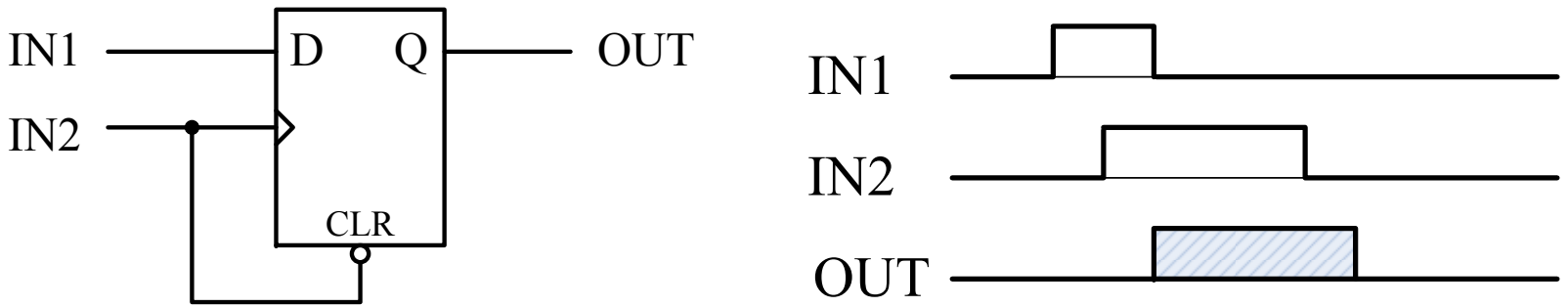
(3) 组合逻辑产生的信号不能在没有通过一个同步延时单元情况下反馈回同一个组合逻辑；

(4) 时钟信号不能被门控，必须直接到达延时单元的时钟输入端，而不经过任何组合逻辑；

(5) 数据信号必须只通向组合逻辑或延时单元的数据输入端（而不是时钟输入端）；

1.6.1 竞争条件

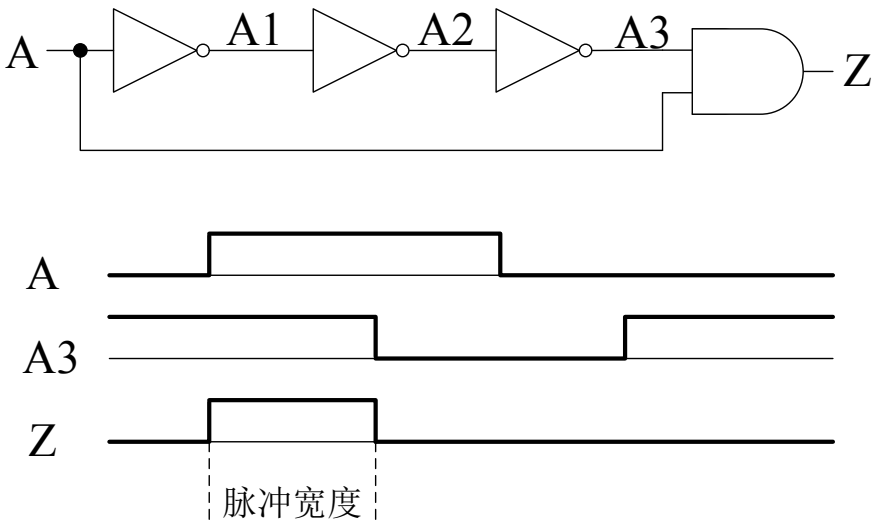
下图一个信号被同时接到时钟和异步清零输入，由于不清楚触发器内部“信号到时钟路径”和“信号到复位路径”的相对延时关系，因此不能确定到底时钟上升沿还是复位低电平先控制输出，结果产生一种竞争的情况：时钟上升沿先到达，而内部复位信号还未脱离有效电平，则Q输出低电平；复位信号先到达，内部复位解除，随后的上升沿使D的高电平输出到Q，最后输出将变为高电平。

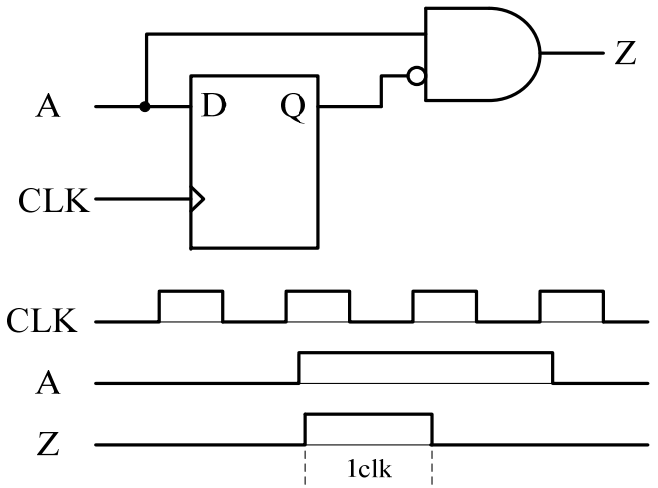


1.6.2 依赖于门的延时

下图是用于产生脉冲的电路。

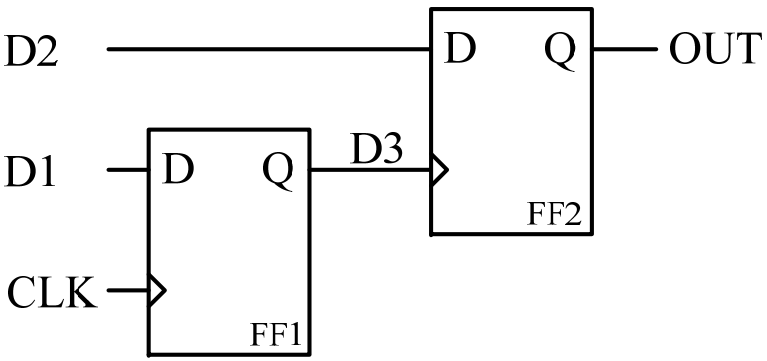
- 左图异步延时电路显然依靠各逻辑门的延时产生脉冲，当半导体工艺改变，门延时可能变短，导致输出脉冲变窄，甚至不能被后续电路识别。
- 右图是同步延时电路，脉冲长度仅取决于时钟周期，工艺变化不会对电路输出产生重大影响。

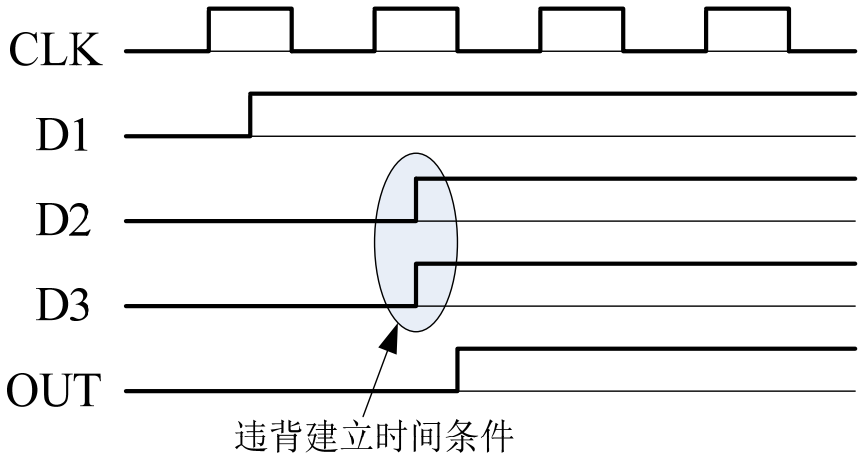




1.6.3 违背建立/保持时间条件

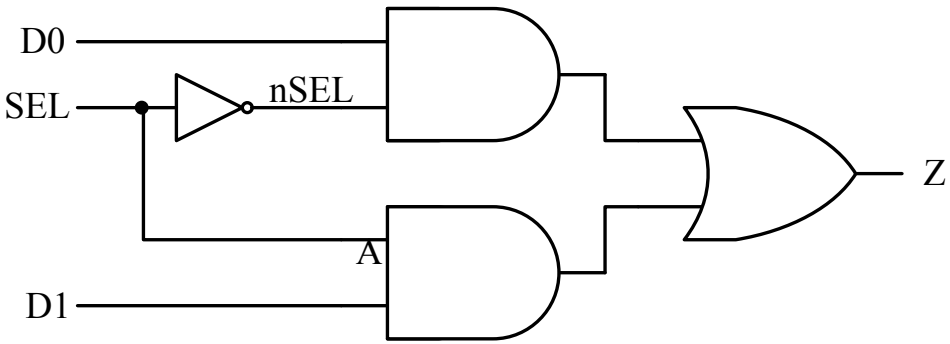
触发器往往要求信号在时钟有效沿之前建立，时钟有效沿之后保持若干时间。下图电路用D3作为触发器FF2的时钟输入，若D2输入与D3上升沿同时到来，则有可能违背FF2触发器的建立或保持时间条件，输出结果OUT可能是不确定的。利用带有使能端的触发器可以解决这个问题。

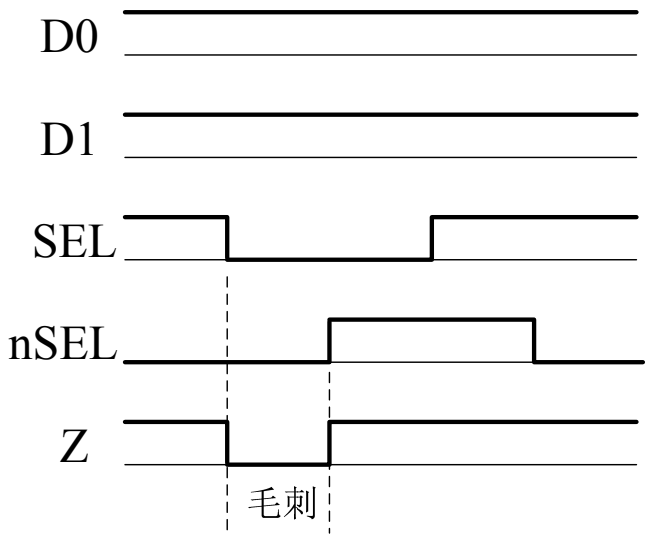




1.6.4 脉冲毛刺

组合逻辑电路中经常存在竞争-冒险问题，一个小延时就可能引起一个脉冲毛刺。例如下图的数据选择器，两个数据输入都是高电平，当SEL端从高到低再从低到高变化时，输出可能产生一个短暂的毛刺。这是因为由于其中反相器存在延时，会存在一段SEL和nSEL同时为低的短暂时间，形成输出端的一个毛刺。采用锁存器对输出进行一次同步，可以消除毛刺。



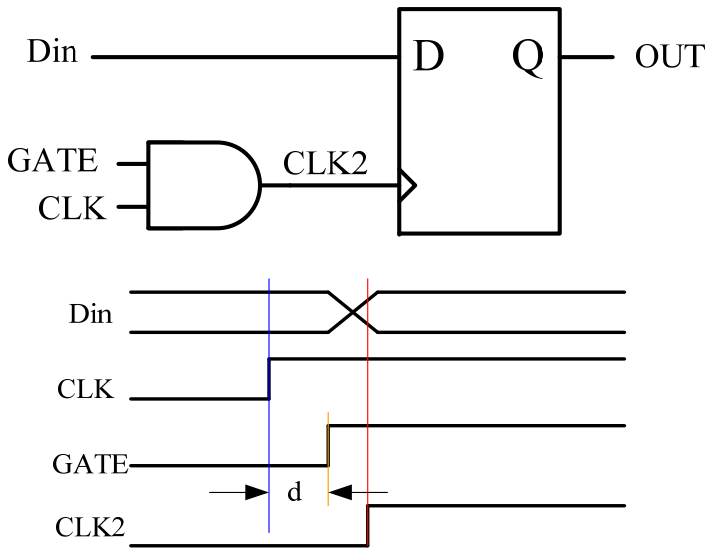


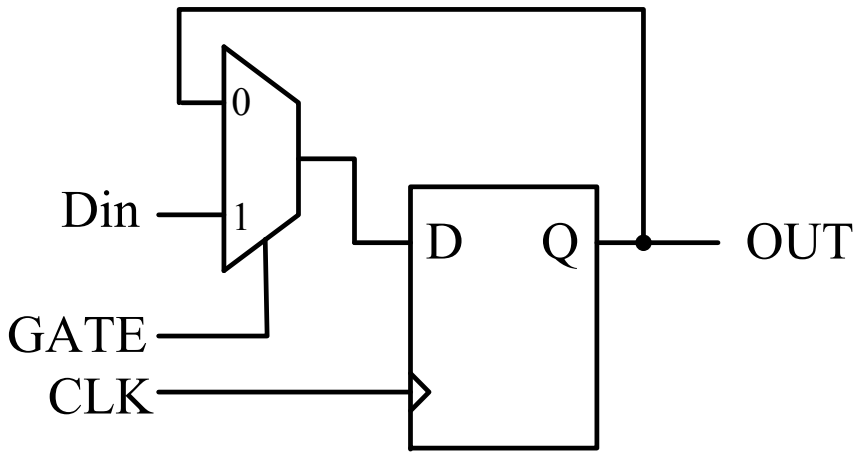
1.6.5 门控时钟

要实现的功能是若GATE有效，时钟CLK上升沿到来时新数据Din送到Q保持，否则旧数据Q保持原来的输出；

左图门控时钟违反同步设计规则(4)、(5)，在FPGA里容易出问题。门控信号GATE容易被延时，CLK信号有效沿到来时GATE无效；此后GATE有效到来时，即使该触发器有时钟上升沿产生，有效数据可能已经消失，D触发器的输出可能是无意义的，既没有保住新数据，也没有保持旧数据。

正确的禁止输出的方式不是将控制逻辑设计在时钟输入端，而是设计在逻辑输入端。实际该功能由一个使能触发器实现。





1.6.6 异步信号亚稳定性

- 异步信号被送入一个同步触发器时就可能产生亚稳定性问题。输入到芯片的信号来自于用户按下的一个按钮或者来自处理器的一个中断信号，或者来自某个时钟产生的信号，而该时钟属于其他时钟域。
- 若输入信号在时钟有效沿附近发生变化，而未能进入一个稳定电平，对于触发器来说，读到的电平不能确定，或者违背建立和保持时间，从而产生亚稳定性问题。
- 信号路径长度不同，则到达逻辑门输入端时间不同；电源路径压降使各单元电平阈值不同；半导体工艺的离散性使各电路的电平阈值不同；这些都可以引起亚稳定性问题。
- 工作频率提高，上述亚稳定性出现几率就会增加。
- 亚稳定性问题并没有可靠的解决方法，在设计中包含足够多的同步触发器，可以减小出现亚稳定性问题的概率。

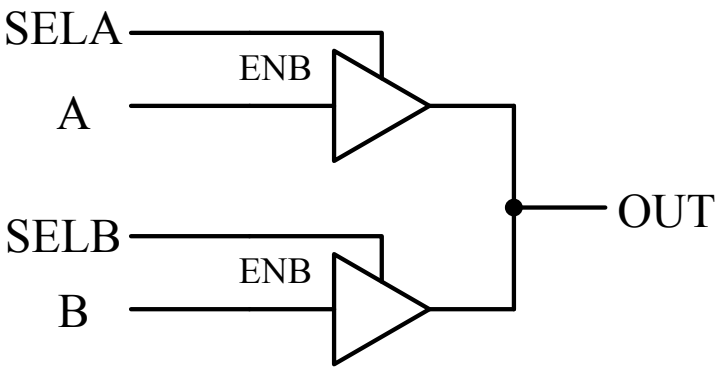
1.6.7 可以使用的异步逻辑

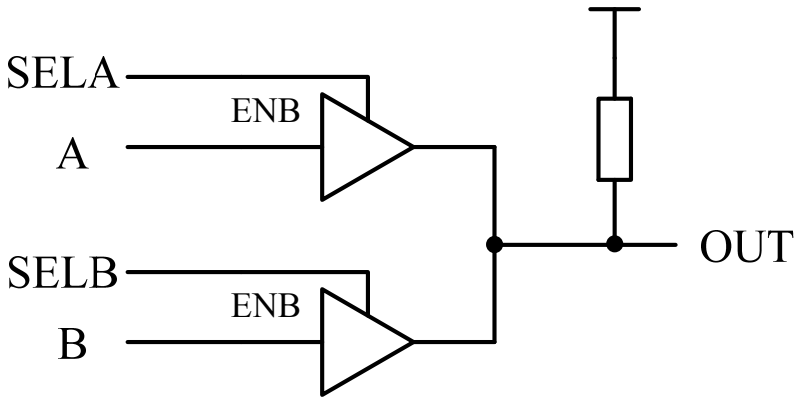
- 异步复位的使用规则
  - 只在芯片初始化时才使用，正常工作过程不该出现异步复位信号
  - 发出复位信号的持续时间至少为一个时钟周期
  - 复位后要确保芯片处于一个稳定状态，使输入变化前没有触发器会自己发生变化；或者说复位后，每个状态机都该处于空闲状态，以等待某个输入信号的变化；
  - 复位信号消失后，输入信号应稳定，至少保持一个时钟周期；
- 异步总线输入端上的异步锁存器
  - 异步总线接口信号：除非系统时钟频率比总线频率高很多，否则利用异步锁存器去捕获总线信号通常可以比用同步方式有更高效率，减少系统开销，减少定时问题的产生；
  - 例如8051外部总线上，锁存地址的“地址锁存使能”信号ALE；其他异步总线上锁存数据的“数据选通”信号DSTROBE等。
  - 异步锁存输入信号，保持稳定，然后将锁存的信号与内部时钟进行同步。

1.7 悬浮节点与总线争抢问题

当SELA和SELB都无效时，左图的OUT输出将进入悬浮状态，引起电源功耗增加或因信号上下跳动增大系统噪声。加上拉电阻可以解决这个问题

当SELA和SELB同时有效时，驱动器的输出可能会发生冲突，从而产生总线争抢问题，会增加系统电流，也可能损坏驱动器。采用互斥的电路可以避免总线争抢问题。





2 组合逻辑电路设计

- 组合逻辑电路

(1) 8位加法器
(2) 指令译码电路
(3) 比较器
(4) 3-8译码器
(5) 8-3编码器
(6) 多路器
(7) 奇偶校验生成
(8) 数字比较排序
(9) BCD-7段译码器
(A) 作业：闰年计算器

例2.1 带进位输入的 8位加法器（简单的算法描述）

```
module adder8(cout, sum, a, b, cin);
        input [7:0] a, b;
        input cin;
        output cout;
        output [7:0] sum;
        assign {cout, sum} = a + b + cin;
endmodule
```

8位的全加器，根据两个8bit的加数a、b和进位输入cin计算出和sum和进位输出cout。

assign是连续赋值语句，属于行为描述方式。

例2.2 指令译码电路示例

```
`define plus      3'd0   //加
`define minus     3'd1   //减
`define band      3'd2   //位与
`define bor       3'd3   //位或
`define cpl       3'd4   //取反
module alu(out, opcode, a, b);
    input [2:0] opcode;
    input [7:0] a, b;
    output [7:0] out;
    reg [7:0] out;
```

```
always @ (opcode or a or b)
    begin
        case (opcode)
            `plus :    out=a+b;
            `minus : out = a-b;
            `band :   out=a&b;
            `bor :     out = a|b;
            `cpl :     out = ~a;
            default : out = 8'hx;
        endcase
    end
endmodule
```

- 利用电平敏感的always块设计组合逻辑电路。

例2.3 比较器设计示例

```
module compare(equal, a, b);
    parameter size = 8;
    input [size-1:0]a, b;
    output equal;
    assign equal = (a == b) ? 1 : 0;
endmodule
```

利用连续赋值语句设计组合逻辑

例2.4 3-8译码器设计示例

```
module decorder(out, in);
    output [7:0] out;
    input [2:0] in;
    assign out = 1'b1 << in;
endmodule
```

利用移位操作实现3-8译码器的逻辑

A2	A1	A0	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
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

例2.5.1 8-3编码器设计示例1

```
module encoder1(none, out, in);
    input [7:0] in;
    output [2:0] out;
    output none;
    reg [2:0] out;
    reg none;
    always @(in)
        begin : local
            integer i;
            out = 0;
            none = 1;
    for (i=0; i<8; i = i +1)
        begin
            if ( in[i] )
                begin
                    out  = i;
                    none = 0;
                end
        end
    end
endmodule
```

优先编码器：输出的是in中为1的最高位的序号，若in无任何位为1，none输出1，out输出为0。

例2.5.2 8-3编码器设计示例2

用嵌套的条件运算符?:描述的编码逻辑关系。

4位的outvec是输出中间变量，最高位对应none，表示无1输入，低3位对应编码输出out2、out1、out0。

分析程序从assign最后一个表达式a ? 4'b0000 : 4'b1000看起。

```
module encoder2(none, out2, out1,
    out0, h, g, f, e, d, c, b, a);
    input a, b, c, d, e, f, g, h;
    output none, out2, out1, out0;
    wire [3:0] outvec;
    assign outvec = h ? 4'b0111 :
        g ? 4'b0110 : f  ? 4'b0101 :
        e ? 4'b0100 : d ? 4'b0011 :
        c ? 4'b0010 : b ? 4'b0001 :
        a ? 4'b0000 : 4'b1000;
    assign none = outvec[3];
    assign out2  = outvec[2];
    assign out1  = outvec[1];
    assign out0  = outvec[0];
endmodule
```

例2.5.3 8-3编码器设计示例3

用if…else语句逐位判断
注意其中独立的assign语句

```
module encoder3(none, out2, out1,
                out0, h, g, f, e, d, c, b, a);
   input  a, b, c, d, e, f, g, h;
   output none, out2, out1, out0;
   reg [3:0] outvec;
   assign {none, out2, out1, out0}
                = outvec;
   always @ (a or b or c or d or e
               or f or g or h)  begin
      if         (h) outvec = 4'b0111;
      else if (g) outvec = 4'b0110;
      else if (f)  outvec = 4'b0101;
      else if (e) outvec = 4'b0100;
      else if (d) outvec = 4'b0011;
      else if (c) outvec = 4'b0010;
      else if (b) outvec = 4'b0001;
      else if (a) outvec = 4'b0000;
      else          outvec = 4'b1000;
   end
endmodule
```

例2.6.1 多路器设计示例1

```
module mux1(out, a, b, sel);
    output out;
    input  a, b, sel;
    assign out = (sel) ? a : b;
endmodule
```

sel为1选择a，否则选择b

* 用连续赋值assign、case语句或if-else语句可以生成多路器电路，若条件语句(case或if-else)的分支条件是互斥的话，综合器可以自动生成并行的多路器。

例2.6.2 多路器设计示例2

```
module mux2(out, a, b, sel);
    output out;
    input  a, b, sel;
    reg out;
    always @ ( a or b or sel)          //电平触发
        begin
            case (sel)                 //sel决定输出a或b
                1'b0 : out = b;
                1'b1 : out = a;
                 default : out = 'bx;
            endcase
        end
endmodule
```

2.6.3 多路器设计示例3

```
module mux3(out, a, b, sel);
    output out;
    input  a, b, sel;
    reg out;
    always @ ( a or b or sel)          //电平触发
        begin
            if (sel)                   //sel决定输出a或b
                out = a;
            else
                out = b;
        end
endmodule
```

2.7 奇偶校验生成器设计示例

- 奇偶校验位有两种类型：偶校验位与奇校验位。
- 如果一组给定数据位中1 的个数是奇数，那么偶校验位就置为1，从而使全部9位数据中1 的个数是偶数。
- 如果一组给定数据位中1 的个数是奇数，那么奇校验位就置为0，从而使全部9位数据中1 的个数是奇数。

```
module parity(even_numbits, odd_numbits, input_bus);
    output even_numbits, odd_numbits;
    input  [7:0] input_bus;
    assign even_numbits = ^input_bus;
          //异或缩减(归约)运算,最后得到1个bit
    assign odd_numbits = ~ even_numbits;
          //取非运算，结果也等于~^input_bus
endmodule
```

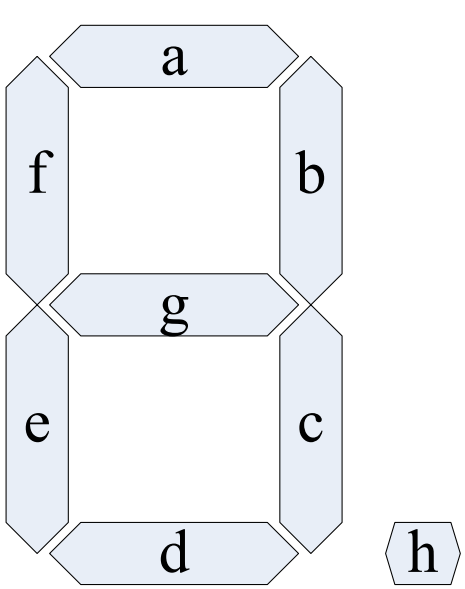
例2.8 数字比较排序组合逻辑设计示例

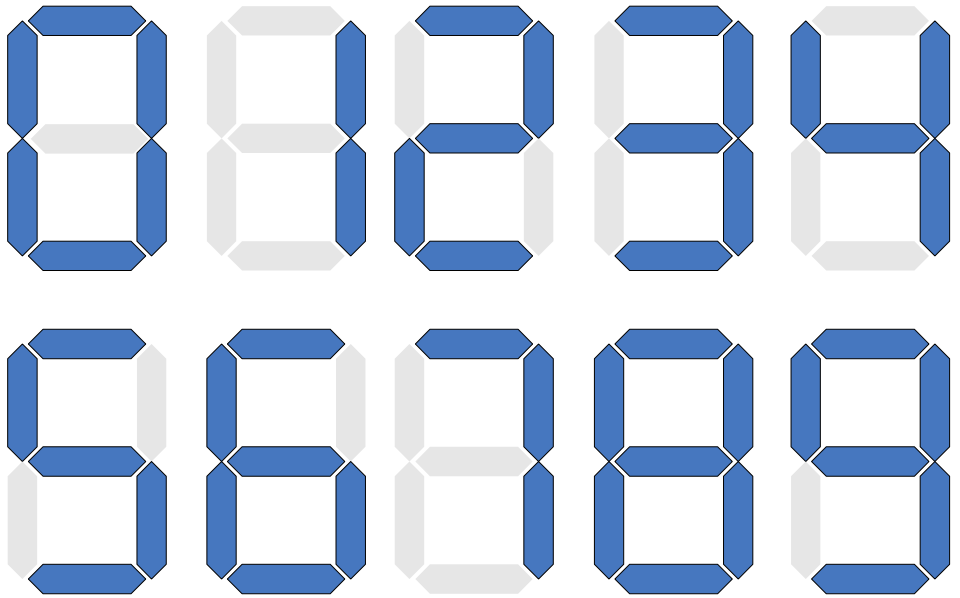
```
module sort4(ra, rb, rc, rd, a,  b, c, d);
    parameter size = 3;
    output [size : 0] ra, rb, rc, rd;
    input [size : 0]  a, b, c, d;
    reg [size : 0] ra, rb, rc, rd;
    always @ (a or b or c or d)
        begin : sort
            reg [size:0] ta, tb, tc, td;
            {ta,tb,tc,td} = {a,b,c,d};
            sort2 (ta, tb); sort2 (ta, tc);
            sort2 (ta, td);
            sort2 (tb, tc); sort2 (tb, td);
            sort2 (tc, td);
            {ra,rb,rc,rd} = {ta,tb,tc,td};
        end
```

```
    task sort2;
        inout [size:0] x, y;
        reg [size:0] tmp;
        if (x>y)
            begin
                tmp = x;
                x = y;
                y = tmp;
            end
    endtask
endmodule
```

4个数全部两两比较，较小的放在前面。

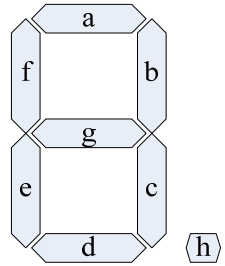
2.9 BCD－7段码 译码器设计示例





BCD－7段码真值表

	A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	1	0	1	1
A	1	0	1	0	0	0	0	0	0	0	0
b	1	0	1	1	0	0	0	0	0	0	0
C	1	1	0	0	0	0	0	0	0	0	0
d	1	1	0	1	0	0	0	0	0	0	0
E	1	1	1	0	0	0	0	0	0	0	0
F	1	1	1	1	0	0	0	0	0	0	0



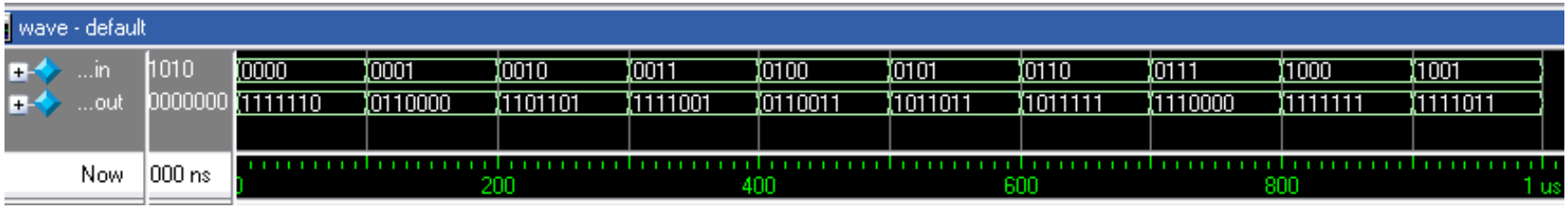
编写Verilog代码实现译码器

```
module bcd2_7seg(ABCD, abcdefg);
    input [3:0] ABCD;    output [6:0] abcdefg;    reg [6:0] abcdefg;
    always @ (ABCD)
    case (ABCD)
            4'b0000: abcdefg = 7'b1111110;
            4'b0001: abcdefg = 7'b0110000;
            4'b0010: abcdefg = 7'b1101101;
            4'b0011: abcdefg = 7'b1111001;
            4'b0100: abcdefg = 7'b0110011;
            4'b0101: abcdefg = 7'b1011011;
            4'b0110: abcdefg = 7'b1011111;
            4'b0111: abcdefg = 7'b1110000;
            4'b1000: abcdefg = 7'b1111111;
            4'b1001: abcdefg = 7'b1111011;
            default: abcdefg = 7'b0000000;
    endcase
endmodule
```

编写Verilog代码用于组合逻辑的行为测试

```
`include "bcd2-7seg.v"
module test7seg;
    reg [3:0] ABCDin;
    wire [6:0] a2g_out;
    initial begin
        ABCDin = 0;
        #100 ABCDin = 1;
        #100 ABCDin = 2;
        #100 ABCDin = 3;
        #100 ABCDin = 4;
        #100 ABCDin = 5;
        #100 ABCDin = 6;
        #100 ABCDin = 7;
        #100 ABCDin = 8;
        #100 ABCDin = 9;
        #100 ABCDin = 10;
        #100 ABCDin = 11;
        #100 ABCDin = 12;
        #100 ABCDin = 13;
        #100 ABCDin = 14;
        #100 ABCDin = 15;
        #100 $stop();
    end

    bcd2_7seg dut1(ABCDin, a2g_out);
endmodule
```

用ModelSim做行为仿真



用ISE做综合、布局、布线、下载验证

1. 在ISE里新建工程，把刚才在ModelSim里编写并仿真过的代码文件添加进工程，主模块属性是Syn/Imp+Sim，testbench属性是Sim Only；
2. 用PACE指定输入输出IO管脚约束，用S3E板右下角的4个拨动开关SW0～SW3做BCD码输入，用LD6～LD0这7个灯作为译码输出；
3. 综合、布局、布线；
4. 生成编程文件，下载到S3E板，拨动SWn开关验证逻辑。

2.10 闰年(leap year)计算器

1. 什么是闰年：1582年Pope Gregory提出闰年历，用于修正一年按365天计算的累计误差，故闰年年号>1582，此问题在430年后的今天可忽略；闰年计算规则：年号能被400整除的是闰年，年号能被4整除但不能被100整除的是闰年；
2. 如何给年号编码：用二进制表示还是用BCD码表示？
   16位二进制可表示到65535，16位BCD可表示到9999；
   虽然二进制在计算被4整除时方便，但不便于直接显示，因此适合采用BCD码。
3. 设计电路结构（被4、100、400整除的判断）；
4. 用原理图或VerilogHDL描述电路结构；
5. 仿真、验证、综合、实现。

3 时序逻辑电路设计

- 时序逻辑电路

(1) 触发器

(2) 电平敏感锁存器

(3) 带置位和复位端的电平敏感型锁存器

(4) 移位寄存器

(5) 8位计数器

(6) 思考: 如何描述一个74HC595 ？

例3.1 触发器设计示例—时序逻辑

电路设计常用上升沿触发的D触发器，其它类型触发器和其它触发类型很少使用

```
module dff ( q, data, clk);
    output q;
    input  data, clk;
    reg q;
    always @ (posedge clk)
        begin
            q <= data;
        end
endmodule
```

例3.2 电平敏感锁存器设计示例1、2

```
module latch1 ( q, data, le);
    output q;                    //data—数据输入
    input  data, le;             //q—锁存输出，le—锁存控制
    assign q = le ? data : q; //le为1，输出q随data变化，
endmodule          //le为0，q保持不变，和74373行为相同
```

```
module latch2 ( q, data, le);
    output q;      reg  q;
    input  data, le;
    always @(data or le)
        if (le)
            q = data;          //隐含了若le=0，q不变
endmodule
```

例3.3 带置位和复位的电平敏感锁存器

```
module latch3 ( q, data, le, set, reset);
    output q;
    input  data, le, set, reset;
    assign q = reset ? 0 : (set ? 1 : (le ? data : q) );
    //表达式隐含了reset > set > le的优先级关系
endmodule
```

always语句声明电平敏感型锁存器，综合时有的综合器会产生警告信息，告知产生了一个电平敏感型锁存器；因为设计的就是电平敏感型锁存器，可以不管这个警告信息。

例3.4 移位寄存器设计示例—时序逻辑

```
module shifter ( din, clk, clr, dout);
    output [7:0] dout;
    input  din, clk, clr;
    reg [7:0] dout;
    always @ (posedge clk)
        begin
            if (clr)
                dout <= 8'b0;                  //清零
            else
                begin
                    dout <= dout << 1;         //左移
                    dout[0] <= din;            //移入新数据
                end
        end
endmodule
```

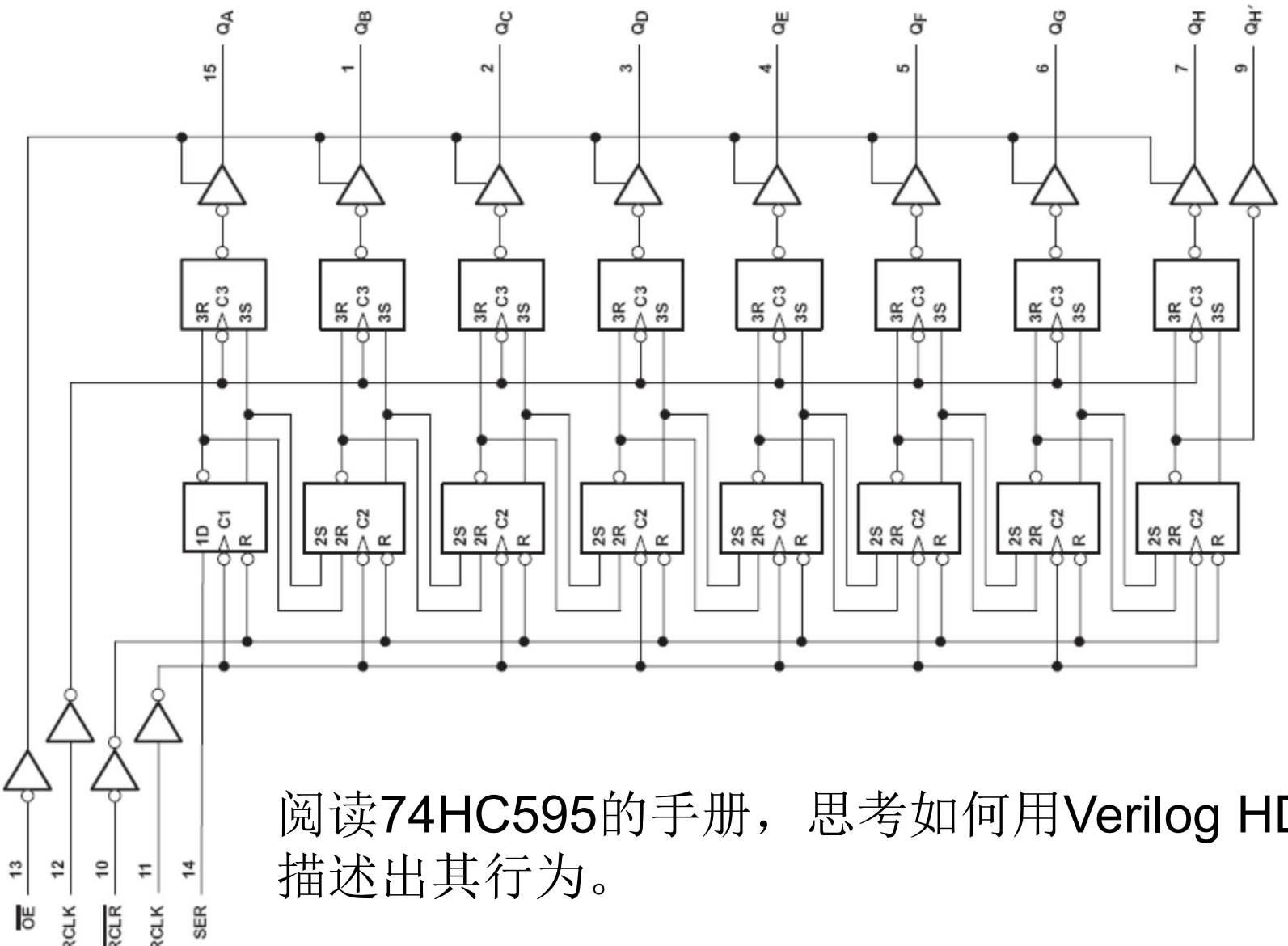
例3.5.1 8位计数器1—时序逻辑

```
module counter1 ( out, cout, data, load, cin, clk);
    output [7:0] out;              //计数输出
    output cout;                   //进位输出
    input [7:0] data;              //预置数输入
    input  load, cin, clk;
    reg [7:0] out;
    always @ (posedge clk)
        begin
            if (load)
                out <= data;       //预置数
            else
                out <= out + cin; //若cin=1计数+1
        end
    assign cout = &out & cin //&out是缩减(规约)运算
    //当out[7:0]各位都为1且cin也为1时就发生进位
endmodule
```

例3.5.2 8位计数器2—时序逻辑

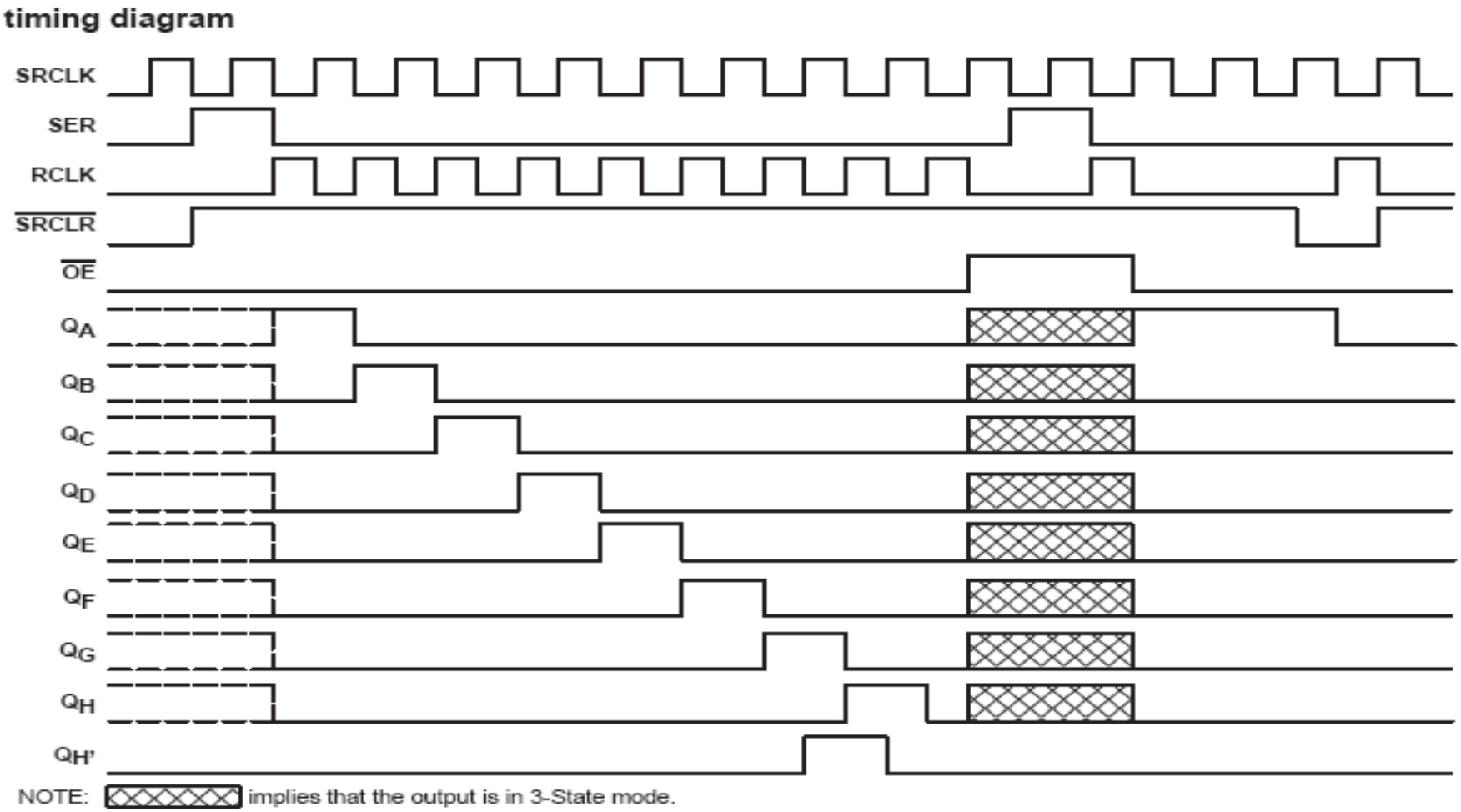
```
module counter2 (out, cout, data, load, cin, clk);
    output [7:0] out;              output cout;
    input [7:0] data;              input  load, cin, clk;
    reg [7:0] out, preout;         reg cout;
    always @ (posedge clk)
        begin
            out <= preout;          //preout用于暂存计数状态
        end
    // 上例中clk到来时应发生进位，若恰好load有效，则进位丢失
    //为提高性能,不希望加载影响进位,下面计算计数器下一个状态
    always @ (out or data or load or cin)
        begin     //这里是组合逻辑,cin的毛刺可能造成误进位？
            {cout, preout}  = out + cin;
            if (load)   preout = data;
        end
endmodule
```

例3.6 如何描述74595的行为

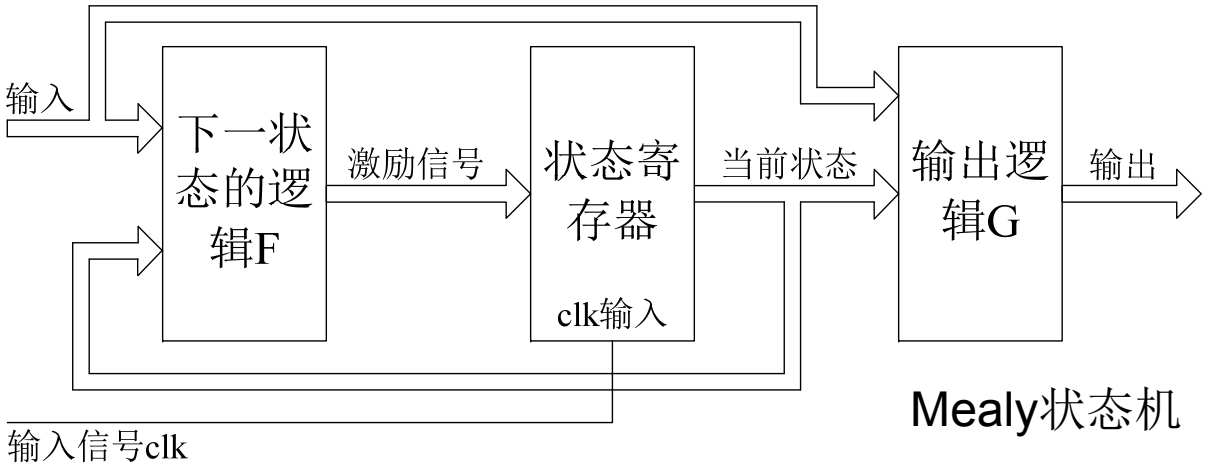


阅读74HC595的手册，思考如何用Verilog HDL描述出其行为。

74HC595时序图


timing diagram
SRCLK
SER
RCLK
SRCLR
OE
QA
QB
QC
QD
QE
QF
QG
QH
QH'
NOTE: implies that the output is in 3-State mode.

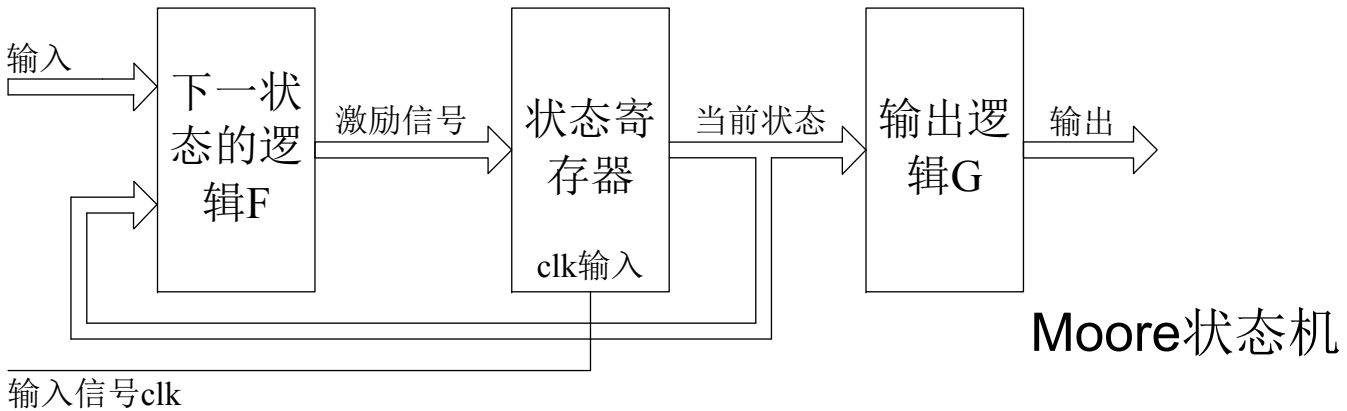
4. 时钟同步状态机



上图是数字电路设计中常用的时钟同步状态机结构，其中状态寄存器由一组触发器构成,用来记忆状态机当前所处的状态,若状态寄存器由n个寄存器构成,最多可记忆2^n个状态.所有触发器时钟端都连接在一个共同的时钟信号上,状态改变只可能发生在时钟跳变沿时刻,故称同步状态机。在可编程逻辑器件上用综合工具生成的状态机电路结构大多使用正跳变沿触发的D触发器。

状态机的类型

- 若时序逻辑的输出不仅取决于当前状态还取决于输入，如上页图所示，称为Mealy状态机。
- 有些时序逻辑电路的输出只取决于当前状态，这样的电路称为Moore状态机，结构如下图所示。
- 两种状态机大部分结构都相同，实际设计工作中的状态机的输出逻辑或多或少与输入相关，使得大部分状态机都属于Mealy状态机。



Moore状态机

- 设计高速电路时常需要使状态机的输出与时钟几乎完全同步，一种方法是将状态变量通过wire直接用作输出；此外还可以采用流水线化(pipelined)的输出。

有限状态机FSM设计的一般步骤

(1) 逻辑抽象得出状态转换图：确定输入输出变量及电路的状态数；定义输入输出逻辑状态含义并将电路状态顺序编号；画出状态转换表/图。

(2) 状态化简：在相同的输入下转换到相同的状态并得到相同的输出这样两个状态称为等价状态,化简为了合并等价状态从而得到最简单的状态转换图以节省存储电路;

(3) 状态分配：又称状态编码,编码选择得当,电路可简单,实际设计时需要综合考虑电路复杂度与电路性能两个因素;

(4) 选定触发器类型并求出状态方程、驱动方程和输出方程;

(5) 按方程得到逻辑图。

用Verilog HDL描述有限状态机可充分发挥硬件描述语言的抽象建模能力，使用always语句和case(if)等语句即可方便实现，具体的逻辑化简、逻辑电路和触发器映射均可由计算机自动完成，上述步骤的(2)、(4)、(5)不需要人工干预，工作得到简化，效率得到很大的提高。

状态机的Verilog语言描述

- 状态机的描述可以分为三个部分：
  - 现在状态逻辑
  - 下一个状态逻辑
  - 输出逻辑
- 以上每个部分分别用一个always块描述，可以将其中两个always块组合到一起
- “下一个状态逻辑”用case描述最方便

异步复位(Asynchronous Reset)风格1

```
reg [3:0] CurrentState, NextState;
always @ (posedge clock or posedge reset)
begin
    if (reset)
        CurrentState <= ST0;
    else
        CurrentState <= NextState;
end
//ST0是预定义的表示状态的常数
```

异步复位(Asynchronous Reset)风格2

```
always @ (posedge clock or negedge reset)
begin
    if (!reset)
        State <= ST0;       //Idle state
    else
        case (state)
            ST0: if (a)  state <= ST0;
                    else  state <= ST1;
            ST1: state <= ST2;
            ST2: state <= ST0;
        endcase
end
```

同步复位(Synchronous Reset)风格

```
reg [3:0] CurrentState;
always @ (posedge clock)
begin
    if (!reset)
        CurrentState <= ST0;      //Idle state
    else
        CurrentState <= ST1;
end
```

状态编码类型

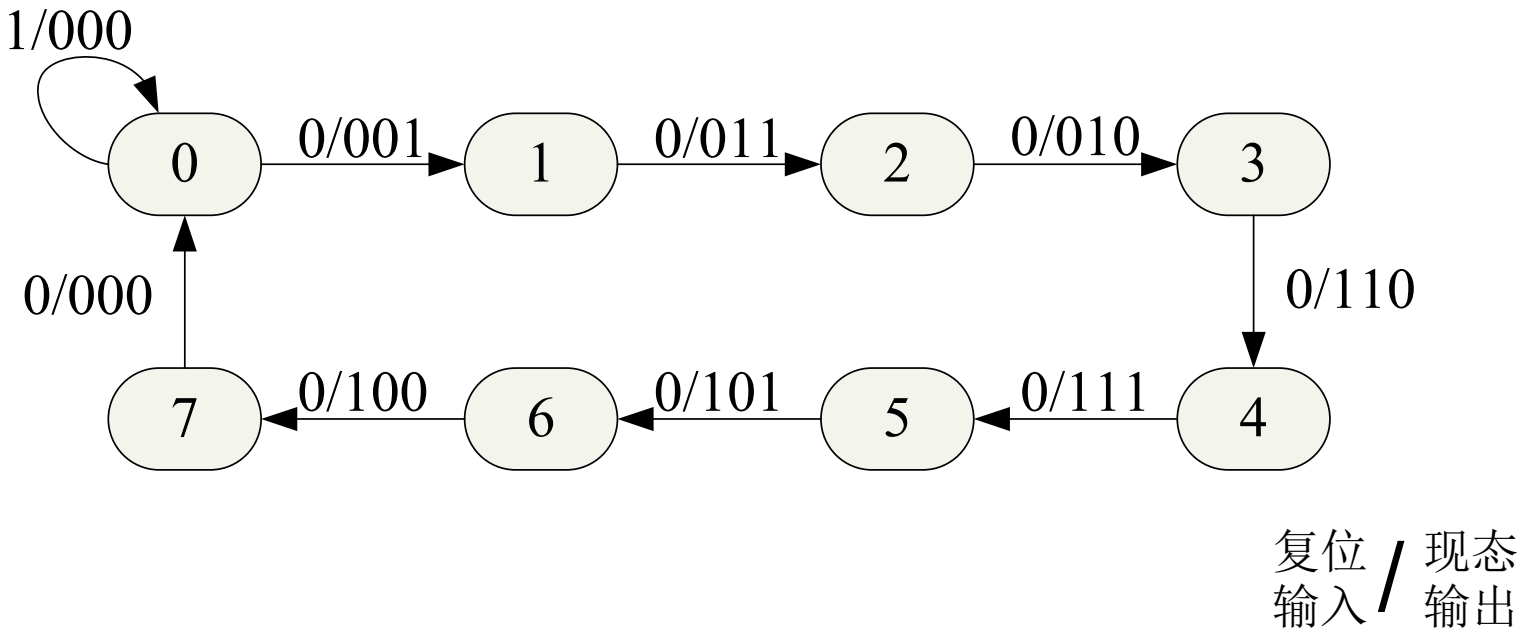
No.	Binary	Gray	One-hot	Johnson
0	000	000	001	000
1	001	001	010	001
2	010	011	100	011
3	011	010		111
4	100	110		
5	101	111		
6	110	101		
7	111	100		

此外还有随机编码Random、Two-hot编码等

编码类型

- Binary: 普通二进制编码，基于状态枚举的定义，例如需要每个状态有不同的二进制编码值，只须改变枚举值的顺序;
- Onehot: 1位热码状态机编码，即采用n位(或n个)触发器来对具有n个状态的状态机编码。虽多用了触发器，但可以有效的节省和简化组合电路。对于寄存器数量多而门逻辑相对缺乏的FPGA器件，采用1位热码编码可以有效的提高电路的速度和可靠性，也有利于提高器件资源的利用率。每一个状态只有一个位被设为高电平;
- Twohot: 2位热码状态机编码，每一个状态对应2个位，2位翻转进行一次状态转换，常用于设计大型FSM(有限状态机);
- Random: 随机编码，状态表示不可被预知，也没有方法让综合工具生成它。用来察看电路的大小或运行情况是否严重依赖状态编码 。
- Gray: 格林编码，这种编码相邻数值只有一个位发生变化，可用于高速计数器等高速电路;
- Auto: 自动，基于位宽来变化编码方式，枚举类型多用Onehot，枚举类型少用Binary。

例4.1：3bit格雷码计数器状态机描述



- 上图所示的状态转移图表示了一个有8个状态的有限状态机（FSM：Finite State Machine），有2位输入信号：时钟信号、复位信号；有3位输出信号：根据状态机的当前状态决定对应的格雷码输出。

“gray.v”状态机描述

```
module gray(clk, rst, out);
input clk, rst;    output [2:0] out;
reg [2:0] out;
reg [2:0] state, next;
parameter st0=0, st1=1, st2=2,
st3=3, st4=4, st5=5, st6=6, st7=7;

always @(state) //Next state logic
case (state)
    st0: next = st1;
    st1: next = st2;
    st2: next = st3;
    st3: next = st4;
    st4: next = st5;
    st5: next = st6;
    st6: next = st7;
    st7: next = st0;
endcase
```

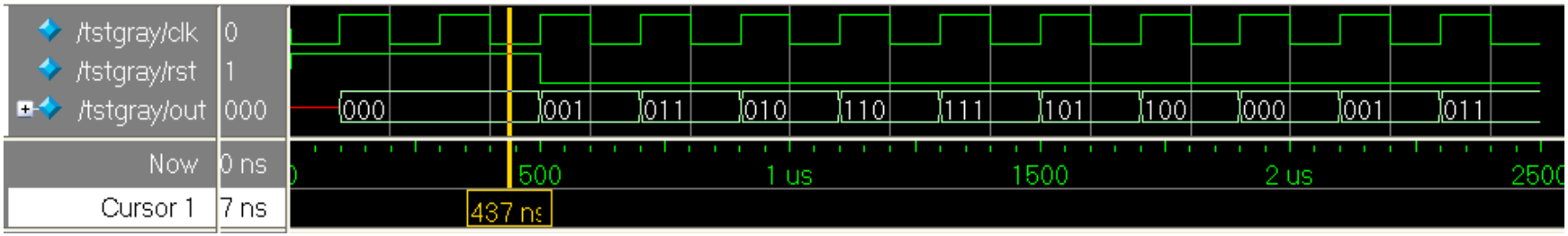
```
//Current state logic
always @(posedge clk)
    if (rst)  state <= st0;
    else state <= next;

always @(state)  //Output logic
case (state)
    st0: out = 3’b000;
    st1: out = 3’b001;
    st2: out = 3’b011;
    st3: out = 3’b010;
    st4: out = 3’b110;
    st5: out = 3’b111;
    st6: out = 3’b101;
    st7: out = 3’b100;
endcase
endmodule
```

“tstgray.v”测试向量描述

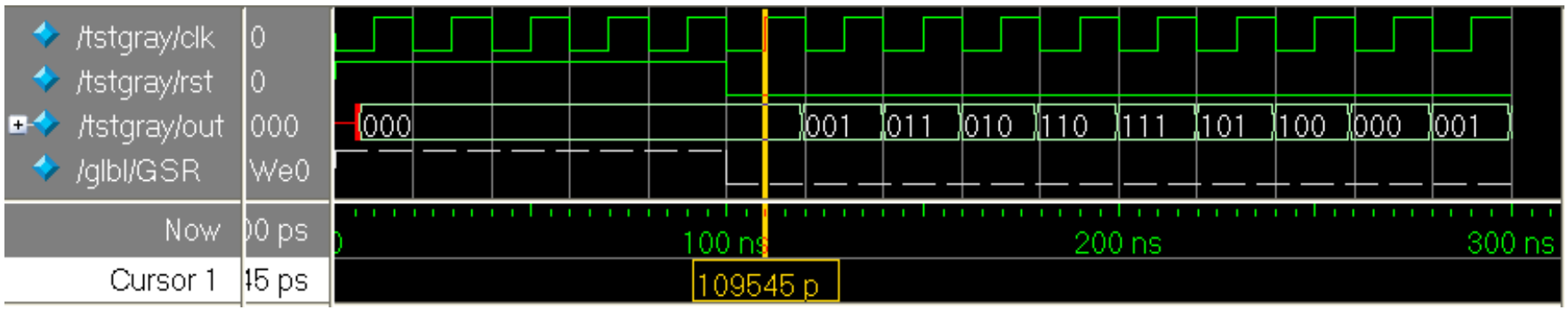
```
`timescale 1ns / 1ps    //
module tstgray;
reg clk, rst;      wire [2:0] out;

initial  begin
              clk = 0;
              rst  = 1;
   #500     rst = 0;
   #2000   $stop();
end

always
     #100   clk = ~clk;

gray  u1(.clk(clk),  .rst(rst),  .out(out) );

endmodule
```

仿真结果

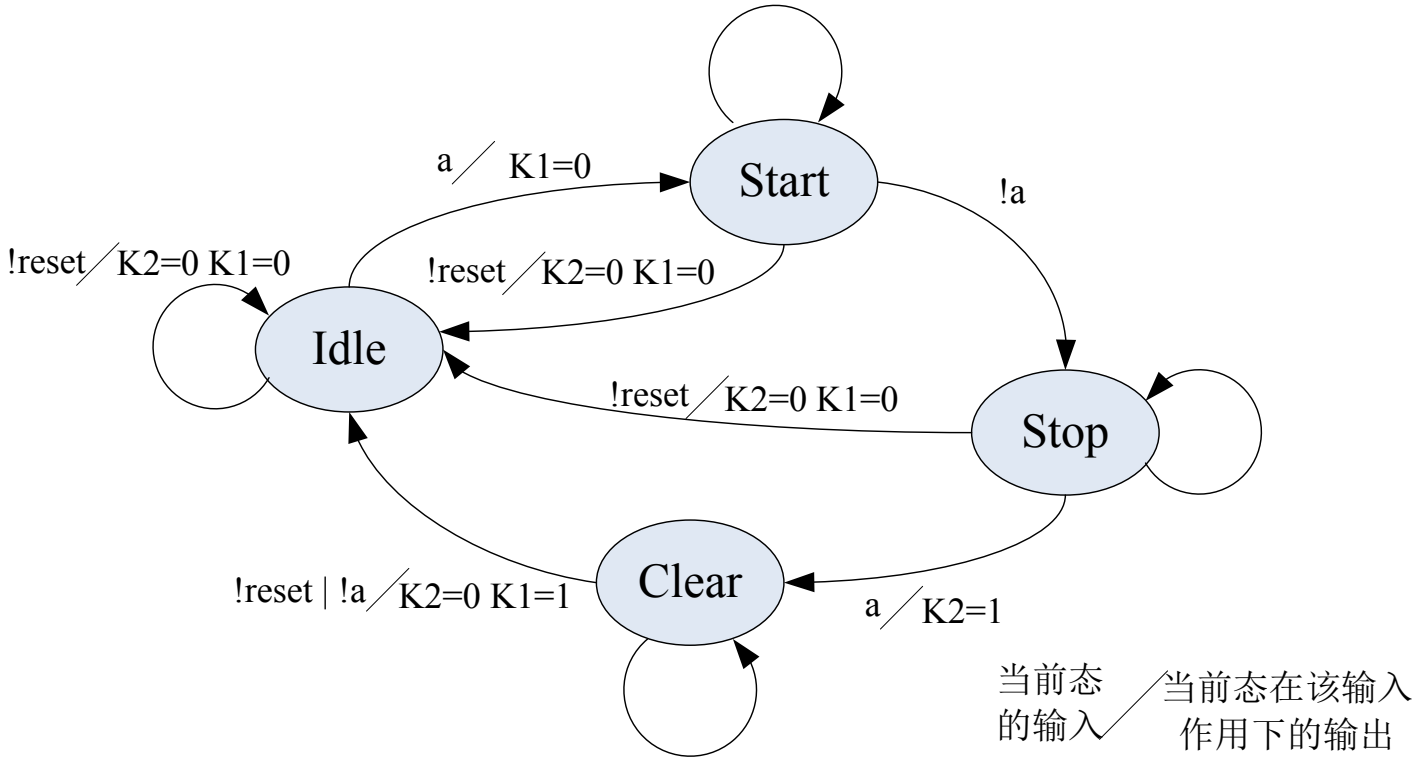
设计好的模块用ModelSim做行为仿真结果：



ISE+ModelSim布线后仿真结果（clk=50MHz, delay=9.4ns）：

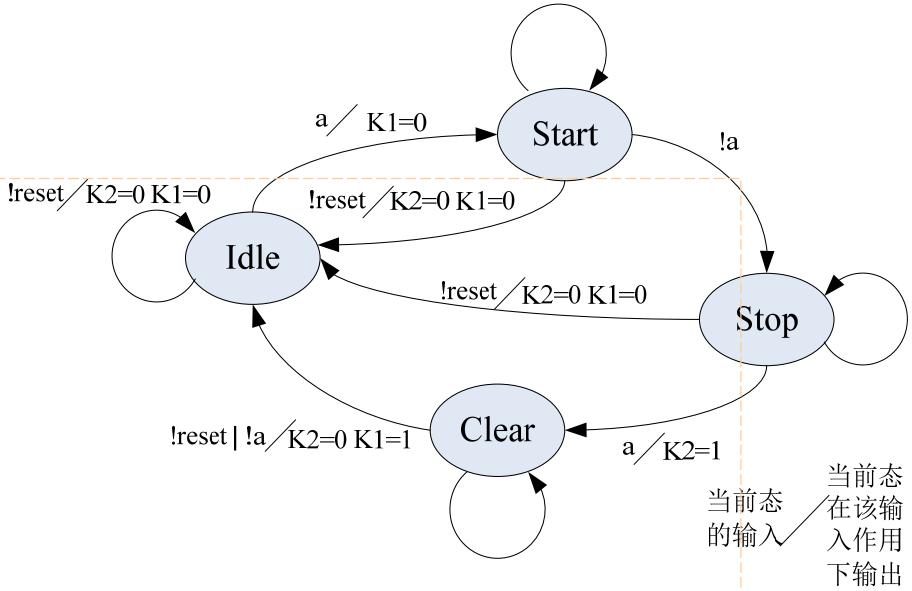


例4.2 现态逻辑、次态逻辑、输出逻辑组合到一起



上图所示的状态转移图表示了一个4状态的有限状态机(FSM:Finite State Machine)，其同步时钟是clock，输入信号是a和reset，输出信号是K1和K2。状态转移只能在同步时钟clock的上升沿时发生，往哪个状态转移则取决于目前所在的状态和输入信号reset和a。

有限状态机Verilog模型示例

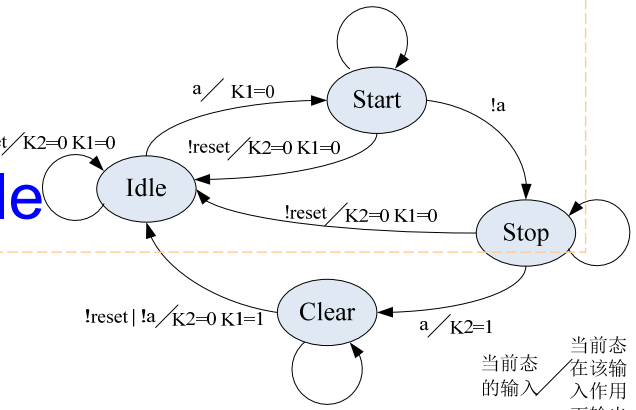



```
module fsm(clk, reset, a,K2,K1);
    input clk, reset, a;
    output K2, K1;
    reg K2, K1;
    reg [1:0] state;
    parameter Idle = 2'b00, Start = 2'b01, Stop = 2'b10,
         Clear = 2'b11;       //4个状态的状态编码
    always @(posedge clk)
        if (!reset)
            begin
                state <= Idle;  K2<=0;  K1<=0;
            end
        else
```

(续前页例子)

```
case (state)
Idle: begin
    if (a) begin
        state <= Start;   K1 <= 0;
    end
    else   state <= Idle;
end

Stop: begin
    if (a) begin
        state <= Clear; K2 <=1;
    end
    else   state <= Stop;
end
```

```
    Start: begin
        if (!a)  state <= Stop;
        else     state <= Start;
    end

    Clear: begin
        if (!a) begin
            state <= Idle;
            K2 <= 0;  K1 <= 1;
        end
        else     state <= Clear;
    end
endcase
endmodule
```



- 可以采用不同的状态编码实现不同的设计要求。

例4.3 简单的状态机设计—有限序列检测器

序列检测器是时序逻辑电路设计的经典范例，就是将一个指定序列从串行数字码流中识别出来，广泛用于异步串行通讯的帧同步等应用，例如以太网MAC层帧头识别、USB接口帧头识别等等。

本例检测器的要求是：复位后，如果在输入序列里发现010序列，就输出一个时钟长度的1，否则输出0；如果发现100序列，那么就停止检测，直到再次复位。

- 设X为数字码流输入，Z为检出标记输出，Z高电平表示“发现指定序列”，低电平表示“未发现指定序列”。

- 为了便于理解要求，给出两段码流的例子：

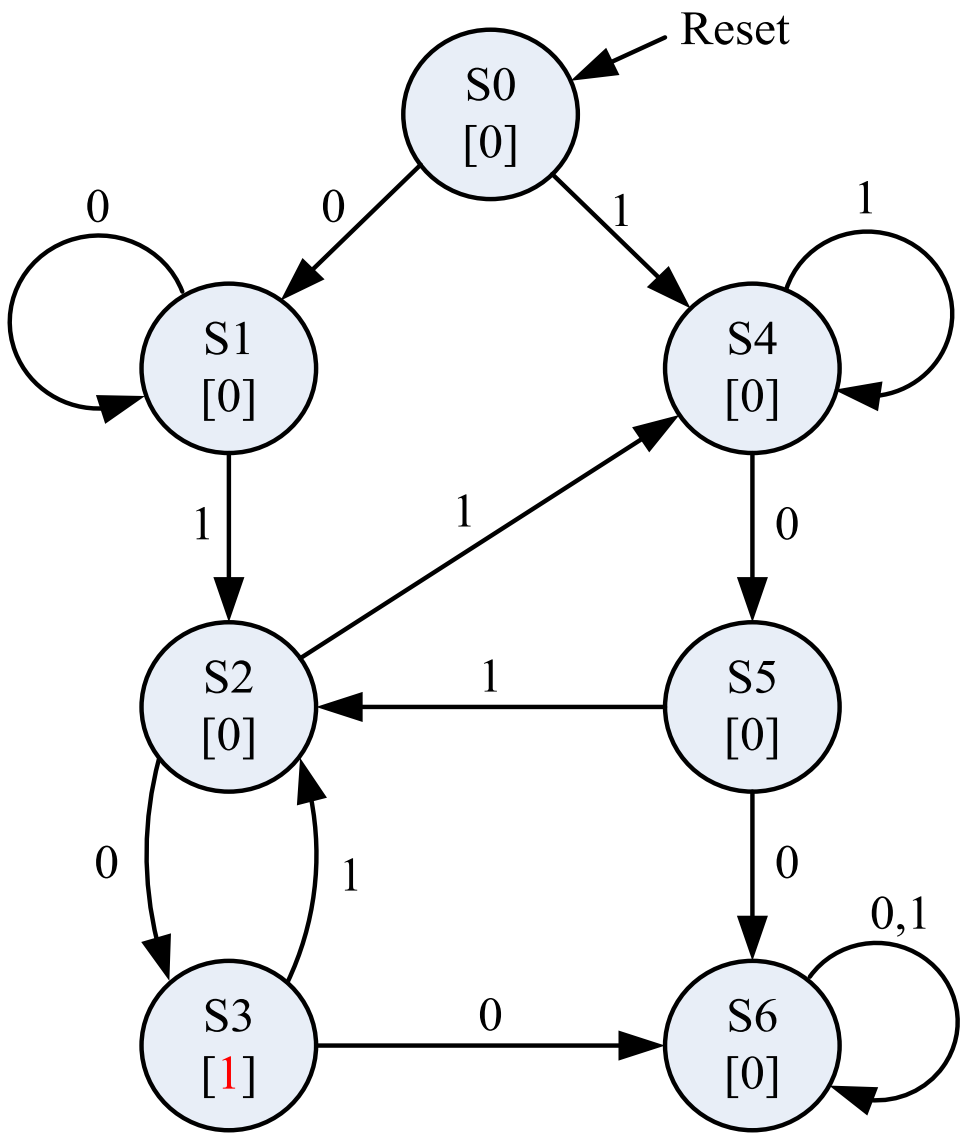
时钟	1	2	3	4	5	6	7	8	9	10	11	12
X	0	0	1	0	1	0	1	0	0	1	0	…
Z	0	0	0	1	0	1	0	1	0	0	0	…

时钟	1	2	3	4	5	6	7	8	9	10	11	12
X	1	1	0	1	1	0	1	0	0	1	0	…
Z	0	0	0	0	0	0	0	1	0	0	0	…

- 注意第一个例子，时钟4、6、8处的检出是重叠的。

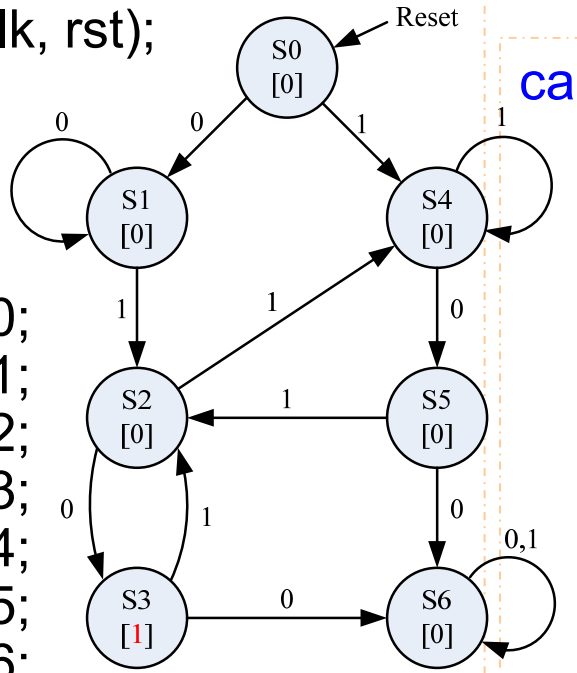
状态图

- 根据功能描述分析，考虑重叠状态，可逐步绘制出右面的状态转换图；
- 状态名S0～S6
- S0是复位后初始状态
- S6是停止检测状态
- 中括号内数字是当前状态Z的输出值。



Verilog程序

```
module seqdet (x, z, clk, rst);
    input  x, clk, rst;
    output  z;
    reg [2:0] state;
     //状态编码定义
    parameter S0 = 3'd0;
    parameter S1 = 3'd1;
    parameter S2 = 3'd2;
    parameter S3 = 3'd3;
    parameter S4 = 3'd4;
    parameter S5 = 3'd5;
    parameter S6 = 3'd6;

    assign z = (state == S3);
    //状态为S3时应输出z=1

    always @ (posedge clk)
          if (rst)
                state <= S0;
           else
```




```
casex (state)
    S0: if ( x==1 )  state <= S4;
          else  state <= S1;
    S1: if ( x==1 )  state <= S2;
          else  state <= S1;
    S2: if ( x ==1 )  state <= S4;
          else  state <= S3;
    S3: if ( x ==1 )  state <= S2;
          else  state <= S6;
    S4: if ( x ==1 )  state <= S4;
          else  state <= S5;
    S5: if ( x ==1 )  state <= S2;
          else  state <= S6;
    S6: state <= S6;

    default : state <= S0;
endcase
endmodule
```

```
`timescale 1ns/1ns                                    //测试文件名tst.v
module t;
    reg clk, rst;     reg [11:0] data;      wire z, x;
    assign x = data[11];
    initial  begin
        clk =1;
        rst =1;
        #10  rst =0;
        data = 12'b0001_0101_0010;
    //  data = 12'b0110_1101_0010;
    end
    always #10 clk = ~ clk;
    always @(posedge clk)
          data = {data[10:0], data[11]};
    seqdet m(.x(x), .z(z), .clk(clk), .rst(rst));
endmodule
```

用ModelSim软件仿真得到的信号波形如下：