有限状态机的Verilog描述

有限状态机的简介

(1) 基本概念

- 有限状态机用来实现一个数字电路设计的控制部分，
- 与CPU 的功能类似，综合了时序逻辑和组合逻辑电路。

(1) 有限状态机与CPU功能比

- 较控制功能的实现

  CPU通过操作指令和硬件操作单元。
  有限状态机通过状态转移。

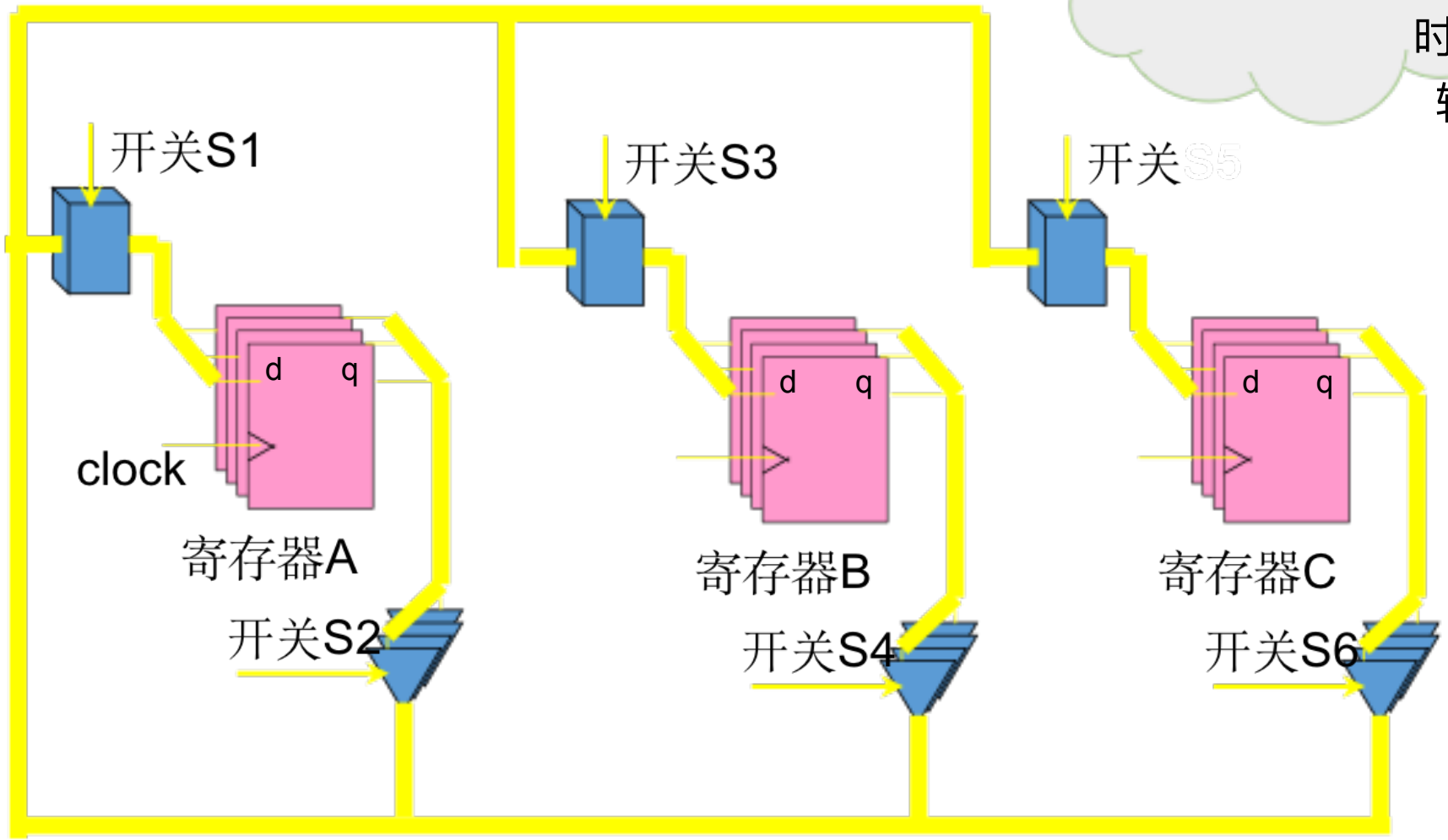
- 有限状态机适用于可编程逻辑器件。通过恰当的Verilog语言描述和EDA工具综合，可以生成性能优越的有限状态机，在执行时间、运行速度和占用资源等方面优于CPU实现的设计方案。

为什么要使用状态机

- 有限状态机克服了纯硬件数字系统顺序方式控制不灵活的缺点。
- 状态机的结构模式相对简单。
- 状态机容易构成性能良好的同步时序逻辑模块。
- 状态机的HDL表述丰富多样。
- 在高速运算和控制方面，状态机更有其巨大的优势。
- 就可靠性而言，状态机的优势也是十分明显的。

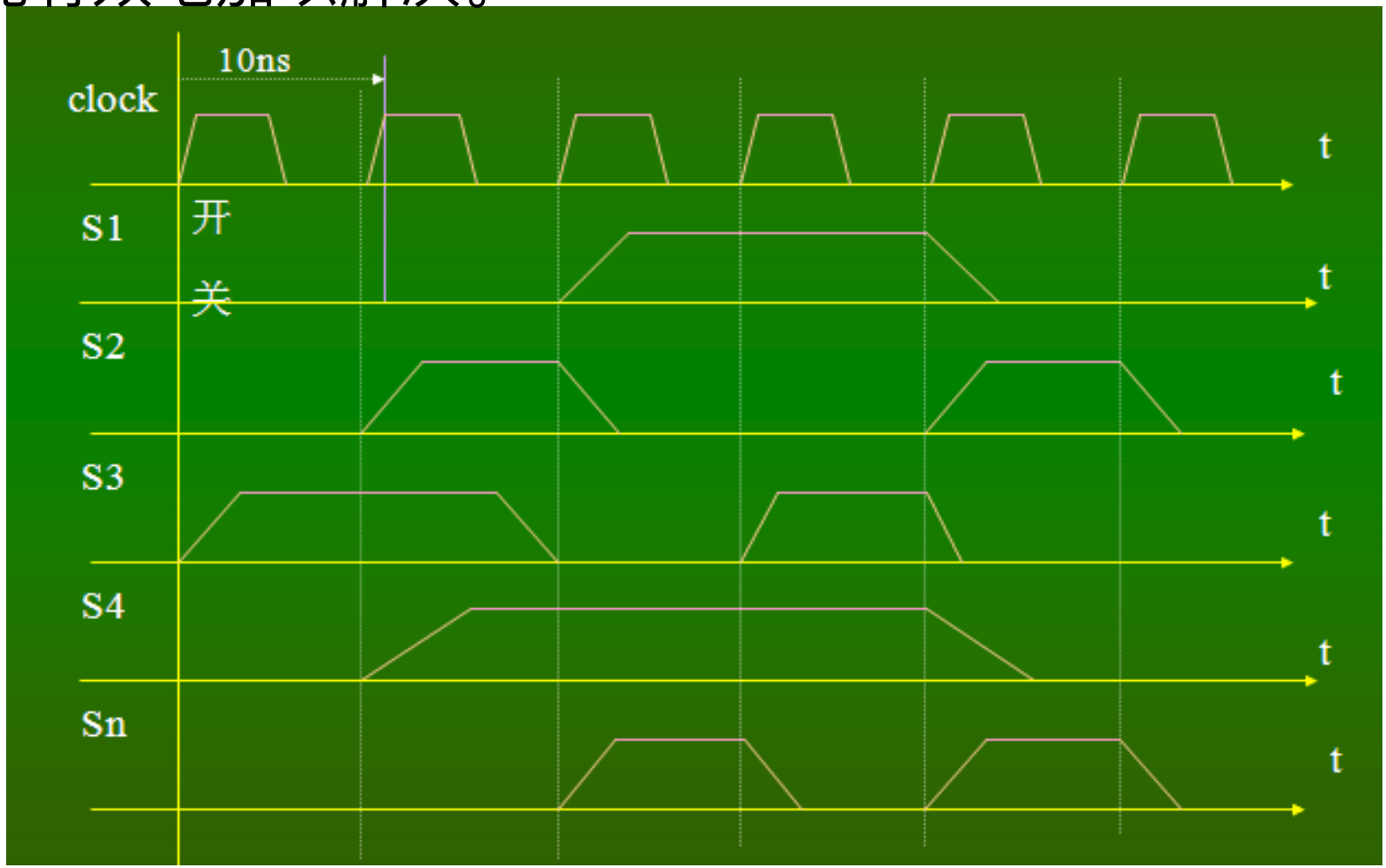
数据流动控制

生成与时钟
确配合的开
时序是计算
辑的核心



问题：如何准确实现数据在各个寄存器、逻辑电路中的流动控制？比如：如何实现寄存器*C*的值准确存入到寄存器A中？

- 如果能严格以时钟跳变沿为前提，按排好时序，来操作逻辑系统中每一个开关si，则系统中数据的流动和处理会按同一时钟节拍有序地进行，避免了冒险和竞争现象，时延问题就能有效地加以解决。



- 如果我们能设计这样一个电路：

1）能记住自己目前所处的状态；

2）状态的变化只可能在同一个时钟的跳变沿时刻发生，而不可能发生在任意时刻；

3）在时钟跳变沿时刻，如输入条件满足，则进入下一状态，并记住自己目前所处的状态，否则仍保留原来的状态；

4）在进入不同的状态时刻，对系统的开关阵列做开启或关闭的操作。

- 有了以上电路，我们就不难设计出复杂的控制序列来操纵数字系统的控制开关阵列。能达到要求的电路就是时序和组合电路互相结合的产物：**同步有限状态机**和由状态和时钟共同控制的**开关逻辑阵列**。
- 只要掌握有限状态机的基本设计方法，加上对基本电路的掌握，再加上对数据处理的过程的细致了解，就可以规避由于逻辑器件和布线延迟产生的“竞争冒险”现象所造成的破坏，设计出符合要求的复杂数字逻辑系统。

- 有限状态机（Finite State Machine，FSM）

是由寄存器组和组合逻辑构成的时序电路，公共时钟信号。

状态的改变只可能发生在时钟的跳变沿上。

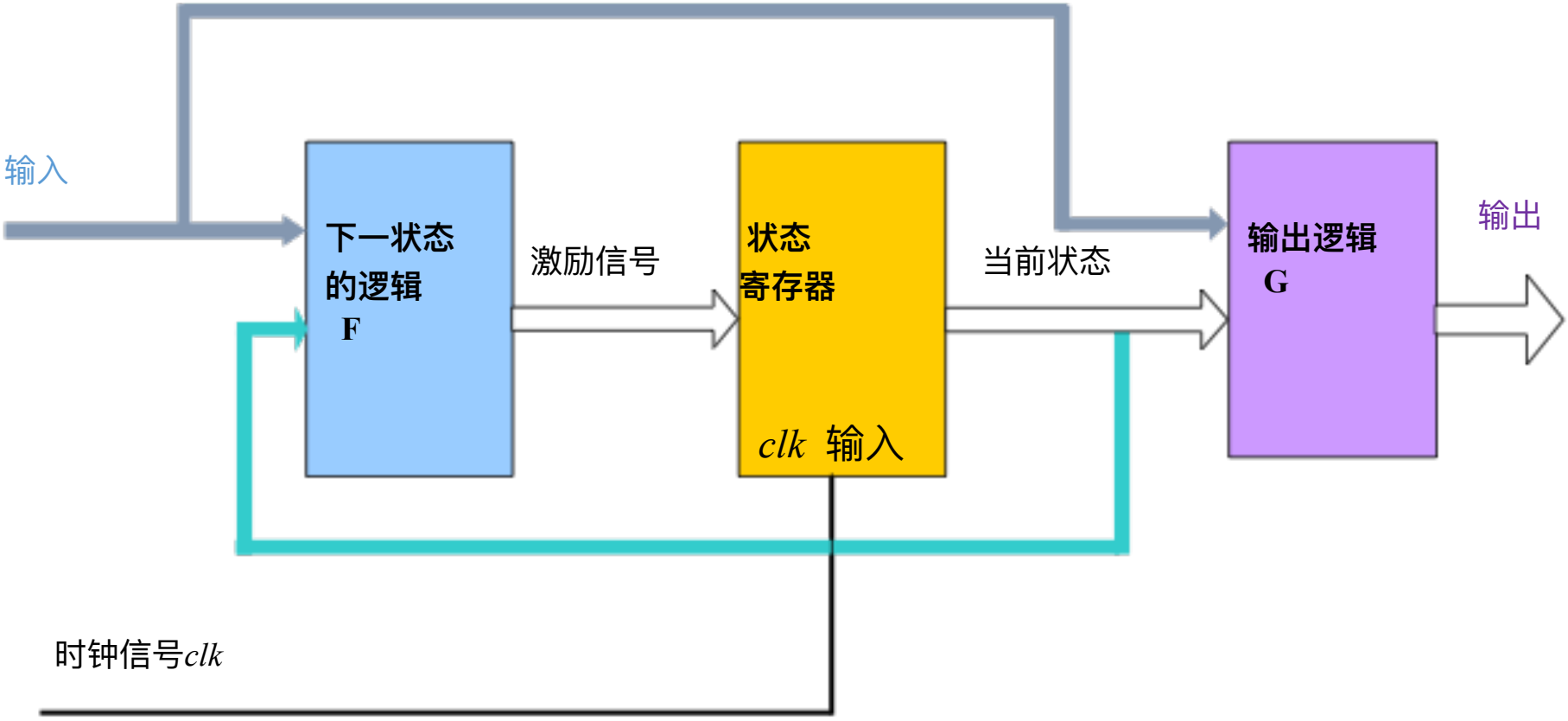
状态是否改变以及如何改变取决于当前状态与输入信号。

状态机可用于产生在时钟跳变沿开关的复杂的控制逻辑，是同步数字逻辑的控制核心。

Mealy状态机

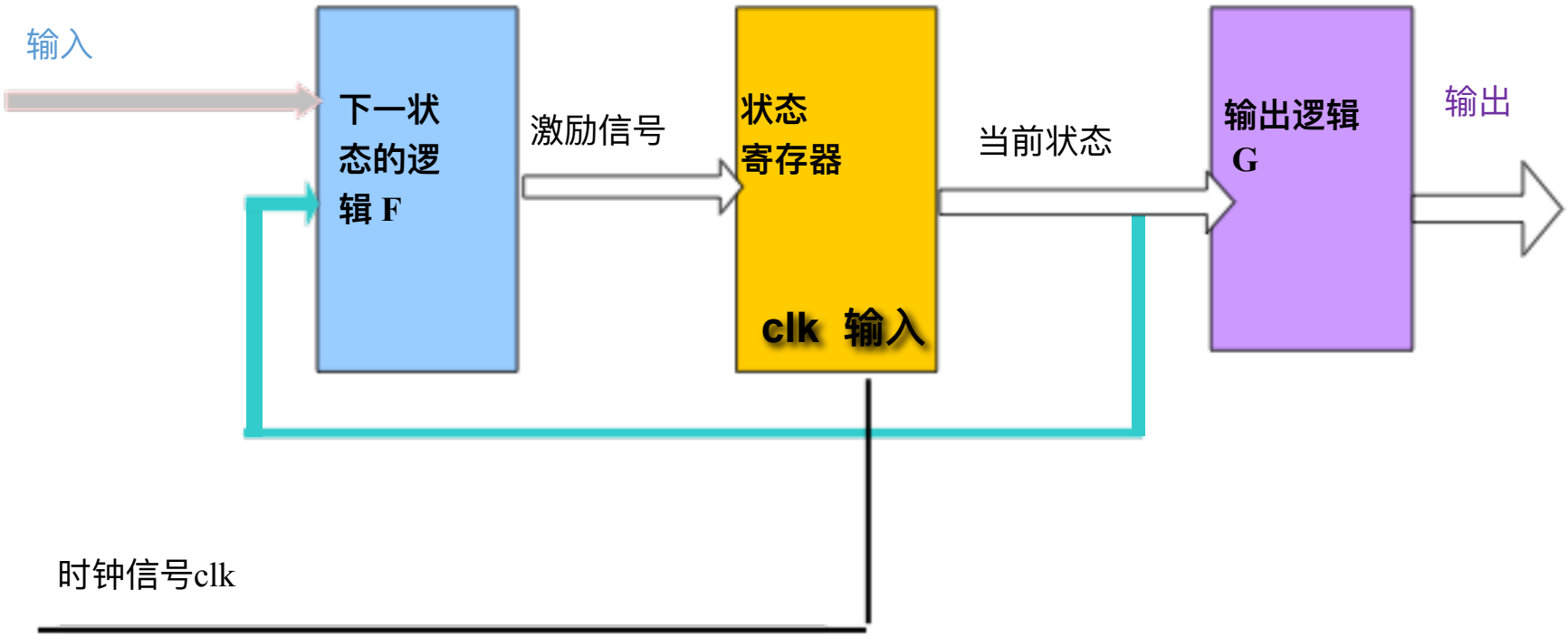
下一状态=F（当前状态，输入信号）；

输出信号=G（当前状态，输入信号）；



- Moore状态机

  - 下一状态=F（当前状态，输入信号）；
  - 输出信号=G（当前状态）；



状态机的表示方法1

- 方法一：状态转换表

输入	当前状态	下一状态	输出
0	000	001	0
1	000	000	0
…	…	…	…

状态机的表示方法2

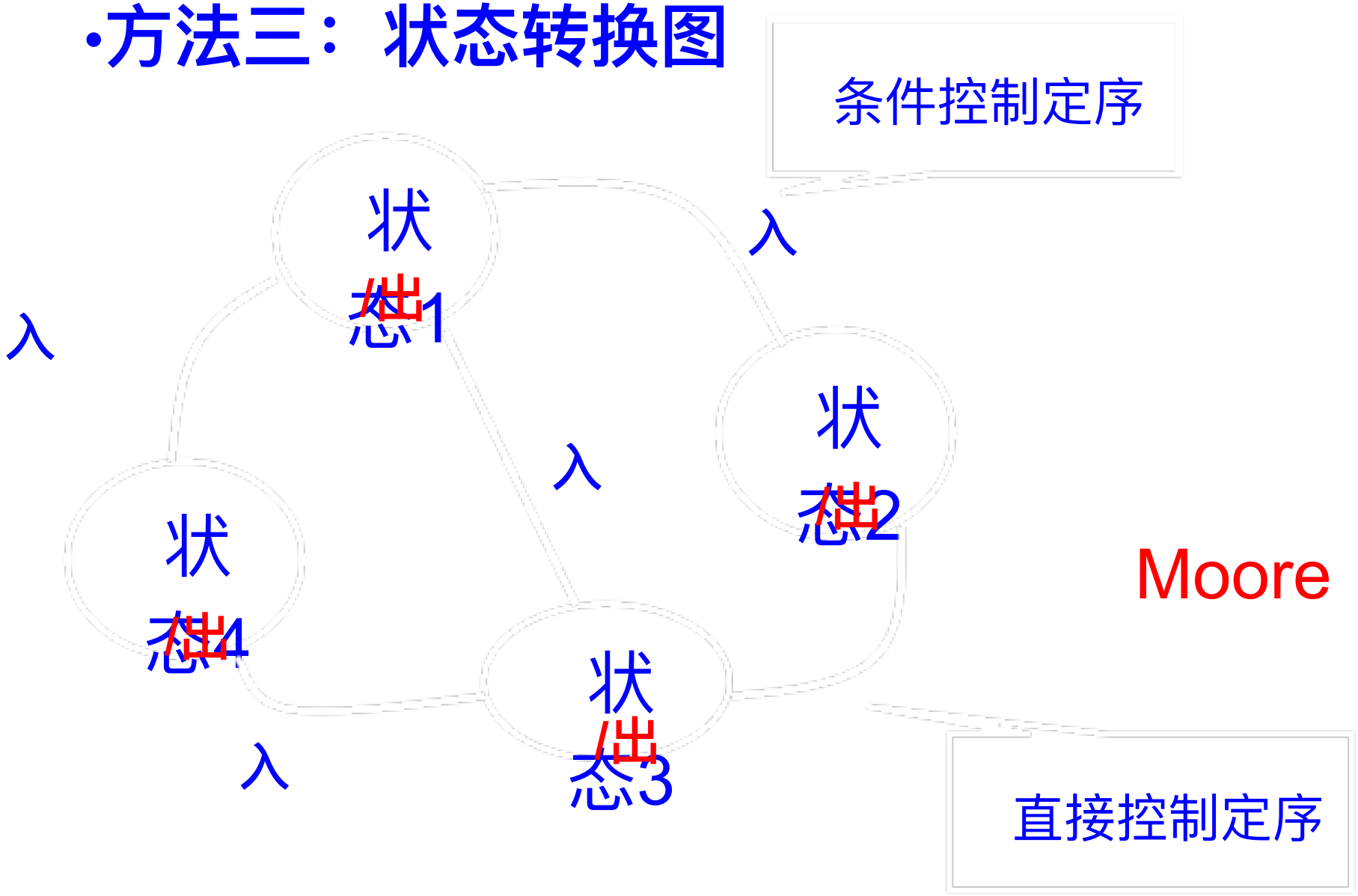
·方法二：算法流程图

方法与软件程序的流程图类似

状态转换表和算法流程图都不适合复杂系统的设计

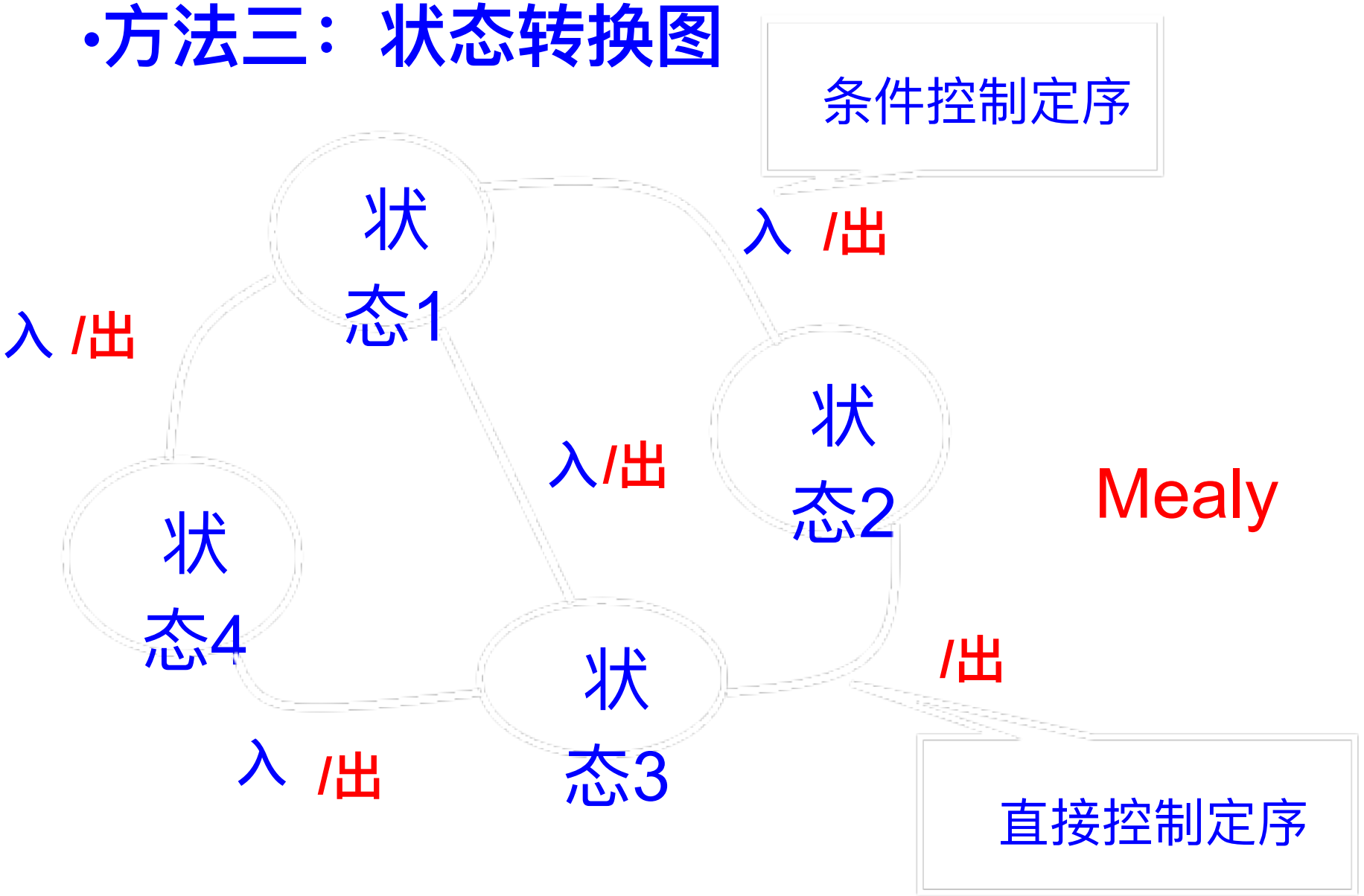
状态机的表示方法3

- 方法三：状态转换图



状态机的表示方法3

·方法三：状态转换图



有限状态机的Verilog描述

（1）用三个过程描述：即现态（CS）、次态（NS）、输出逻辑（OL）各用一个always过程描述。

（2）双过程描述（CS+NS、OL双过程描述）：使用两个always过程来描述有限状态机，一个过程描述现态和次态时序逻辑（CS+NS）；另一个过程描述输出逻辑（OL）。

（3）双过程描述（CS、NS+OL双过程描述）：一个过程用来描述现态（CS）；另一个过程描述次态和输出逻辑（NS+OL）。

（4）单过程描述：在单过程描述方式中，将状态机的现态、次态和输出逻辑（CS+NS+OL）放在一个always过程中进行描

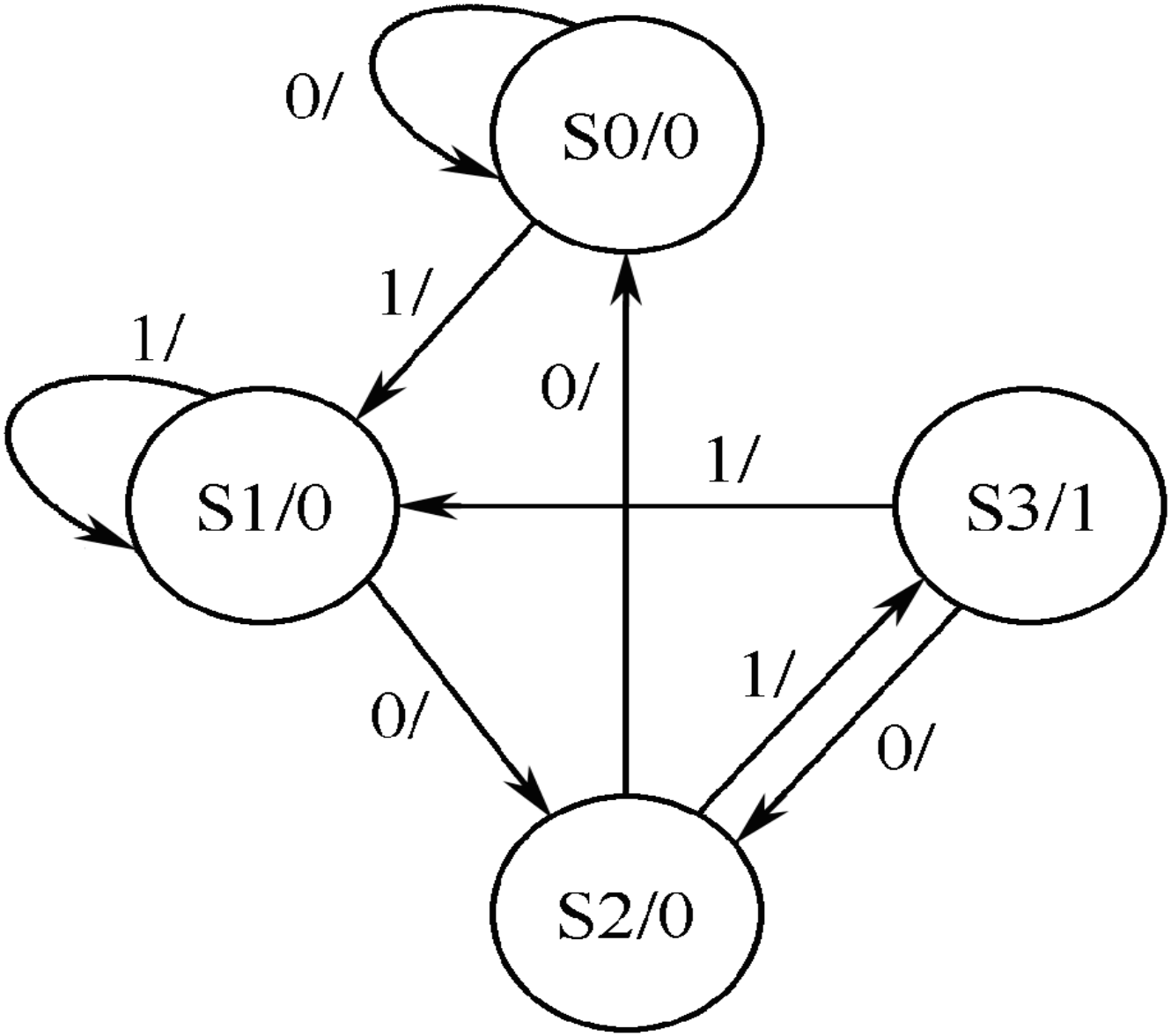
“101”序列检测器的Verilog描述（三个过程）

时钟周期： t0 t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12

w： 0 1 0 1 0 1 0 1 0 1 1 0 0

z： 0 0 0 1 0 1 0 0 0 1 0 0 0

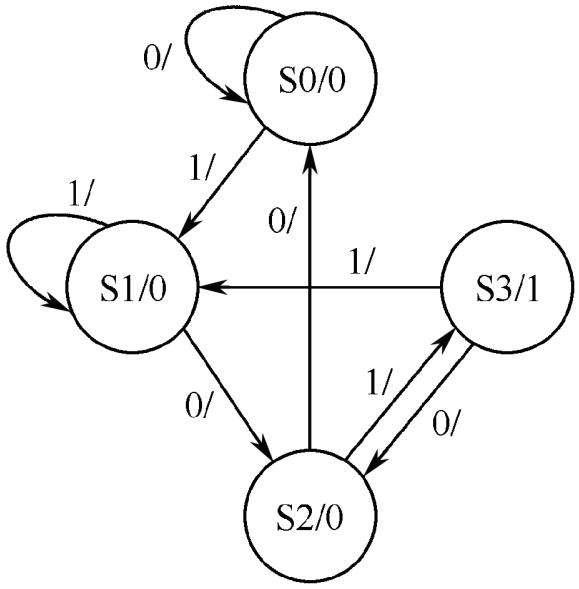
“101”序列检测器的Verilog描述（三个过程）



现 态	次 态		输出 Z
	W=0	W=1	
S0	S0	S1	0
S1	S2	S1	0
S2	S0	S3	1
S3	S2	S1	0

“101”序列检测器的Verilog描述（三个过程）

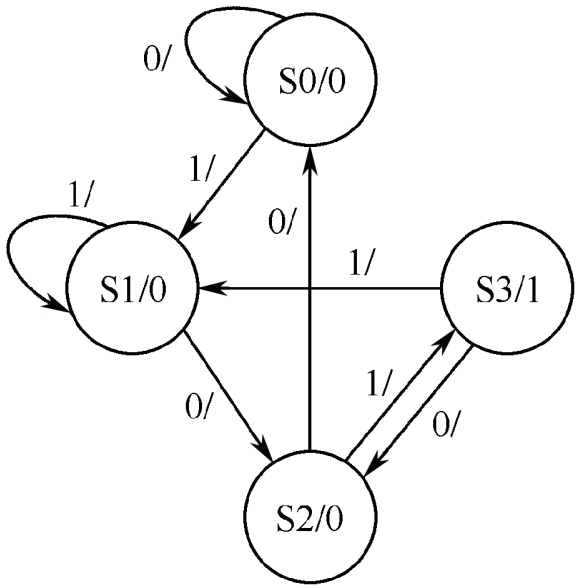
```
module fsm1_seq101(clk,clr,x,z);
input clk,clr,x;
output reg z;
reg[1:0] state,next_state;
parameter    S0=2'b00,S1=2'b01,S2=2'b11,S3=2'b10;
                /*状态编码，采用格雷（Gray）编码方式*/
always @(posedge clk or posedge clr) /*该过程定义当前状态*/
 begin if(clr) state<=S0;      //异步复位，s0为起始状态
       else  state<=next_state;
  end
always @(state or x) /*该过程定义次态*/
begin
case (state)
   S0:begin if(x) next_state=S1;
            else  next_state=S0; end
   S1:begin if(x) next_state=S1;
            else  next_state=S2; end
```



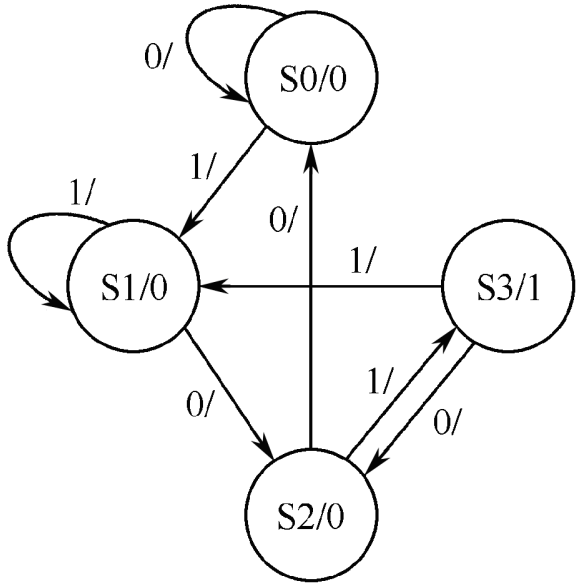
“101”序列检测器的Verilog描述（三个过程）

```
S2:begin
   if(x) next_state=S3;
   else next_state=S0;
  end
S3:begin          if(x)
next_state=S1;
   else    next_state=S2;
   end
default: next_state=S0;
   /*default语句*/
   endcase
end
```

```
always @(state)
/*该过程产生输出逻辑*/
begin  case(state)
       S3: z=1'b1;
     default:z=1'b0;
       endcase
end
endmodule
```



“101”序列检测器（单过程描述）




```
module fsm4_seq101(clk,clr,x,z);
input clk,clr,x; output reg z; reg[1:0] state;
parameter S0=2'b00,S1=2'b01,S2=2'b11,
        S3=2'b10; /*状态编码，采用格雷编码*/
always @(posedge clk or posedge clr)
begin if(clr) state<=S0; //异步复位，s0为起始状态
   else case(state)
        S0:begin if(x) begin state<=S1; z=1'b0;end
       else  begin state<=S0; z=1'b0;end
            end
        S1:begin if(x) begin state<=S1; z=1'b0;end
        else begin state<=S2; z=1'b0;end
            end
        S2:begin  if(x) begin state<=S3; z=1'b0;end
         else begin state<=S0; z=1'b0;end
           end
        S3:begin  if(x) begin state<=S1; z=1'b1;end
         else begin state<=S2; z=1'b1;end
            end
      default: begin  state<=S0; z=1'b0;end  /*default语句*/
   endcase
end
endmodule
```

常用的状 态编码方式

状态编码主要有二进制编码、Gray和一位独热编码等方式

◆二进制编码（Binary State Machine）

◆Gray（Sequential State Machine）

◆一位热码编码（One-Hot State Machine Encoding）

一位热码编码即采用n位（或n个触发器）来编码具有n个状态的状态机。比如对于state0、state1、state2、state3 四个状态可用码字1000、0100、0010、0001来代表。

状态编码的定义

在Verilog语言中，有两种方式可用于定义状态编码，分别用parameter和'define语句实现，比如要为state0、state1、state2、state3四个状态定义码字为：00、01、11、10，可采用下面两种方式。

方式1：用parameter参数定义

```
parameter
state1=2'b00, state2=2'b01;
state3=2'b11, state4=2'b10;
……
case(state)
state1:   …;       //调用
state2:   …;
……
```

方式2：用'define语句定义

```
'define state1  2'b00              //不要加分号“;”
'define state2  2'b01
'define state3  2'b11
'define state4  2'b10
   case(state)
'state1:    …;          //调用，不要漏掉符号“'”
'state2:    …;
……
```

要注意两种方式定义与调用时的区别，一般情况下，更倾向于采用方式1来定义状态编码。一般使用case、casez和casex语句来描述状态之间的转换，用case语句表述比用if-else语句更清晰明了。

有限状态机设计要点

1. 起始状态的选择：

起始状态是指电路复位后所处的状态，选择一个合理的起始状态将使整个系统简洁、高效。多数EDA软件会自动为基于状态机的设计选择一个最佳的起始状态。

2. 有限状态机的同步复位

3. 有限状态机的异步复位

多余状态的处理

一般有如下两种处理多余状态的方法：

（1）在case语句中用default分支决定如果进入无效状态所采取的措施；

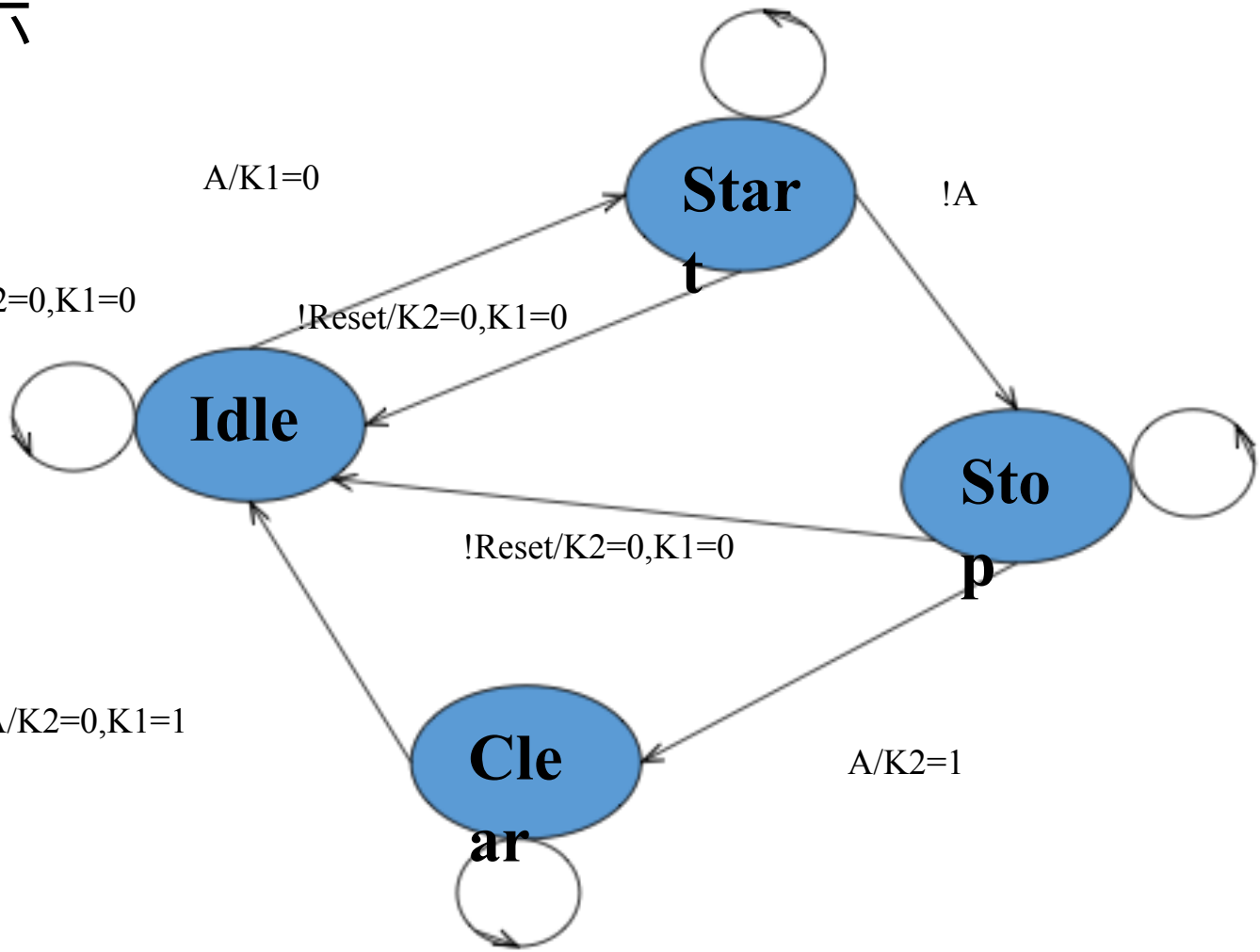
（2）编写必要的Verilog源代码明确定义进入无效状态所采取的行为。

简单的有限状态机设计

- 状态转移图表示

图形表示：

状态转移条件开关（操作）

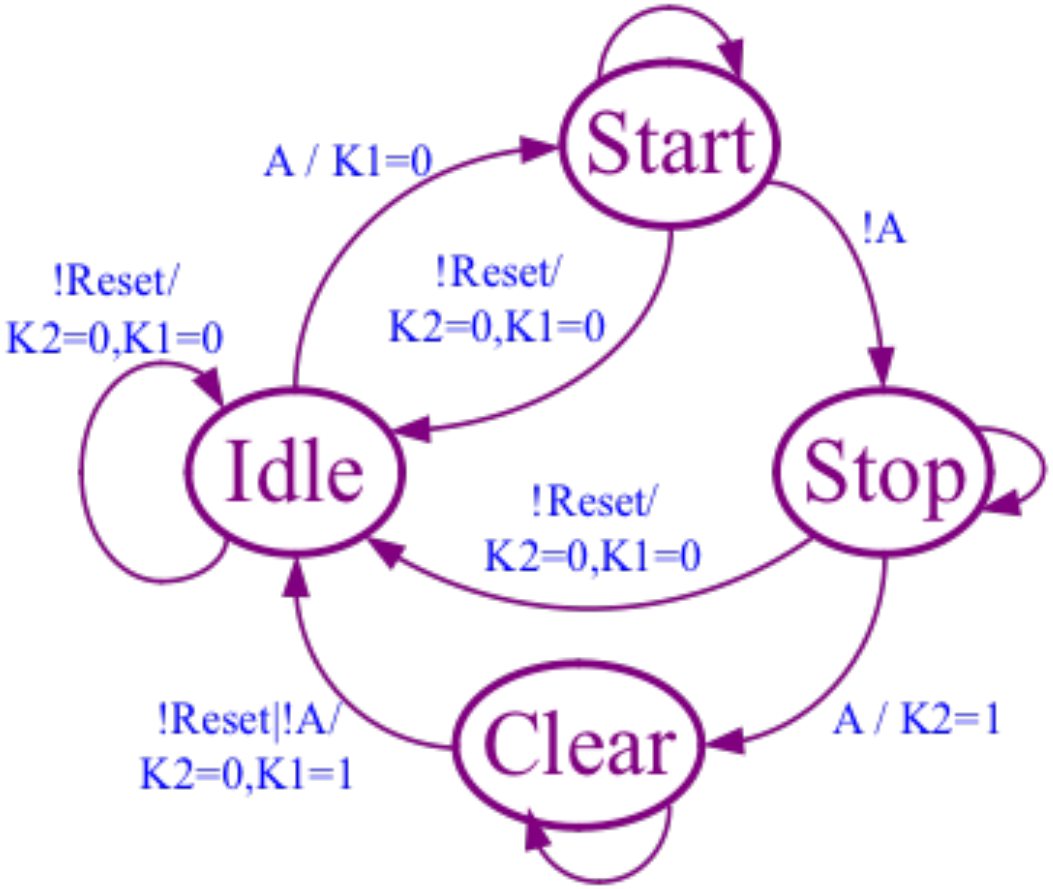


有限状态机的Verilog描述

1.定义模块名和输入输出端口
2.定义输入、输出变量或寄存器
3.定义时钟和复位信号
4.定义**状态变量**和**状态寄存器**
5.用时钟沿触发的always块表示状态转移过程
6.在复位信号有效时给状态寄存器赋初始值
7.描述状态的**转换过程**
8.验证状态转移的正确性，必须完整和全面

状态转移图

模块定义




```
module  fsm (Clock, Reset, A, K2, K1);
input Clock, Reset, A; //定义时钟、复位和输入信号
output K2, K1;         //定义输出控制信号的端口
reg K2, K1;            //定义输出控制信号的寄存器
reg [1:0] state ;      //定义状态寄存器
```

有限状态机的Verilog描述：　（方法一）

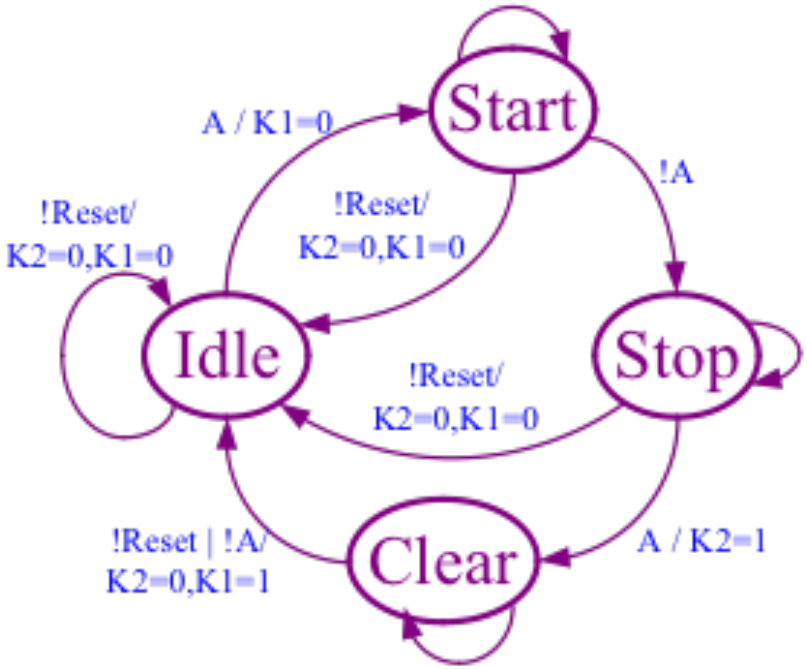
```
module  fsm (Clock, Reset, A, K2, K1);
input Clock, Reset, A; //定义时钟、复位和输入信号
output K2, K1; //定义输出控制信号的端口
reg K2, K1;    //定义输出控制信号的寄存器
reg [1:0] state ;  //定义状态寄存器

parameter Idle=2'b00,Start=2'b01,
          Stop=2'b11,Clear=2'b10;//定义状态变量参数值
always @(posedge Clock)
   if (!Reset)
     begin      //定义复位后的初始状态和输出值
       state <= Idle;  K2<=0; K1<=0;
     end
```

采用Gray编码

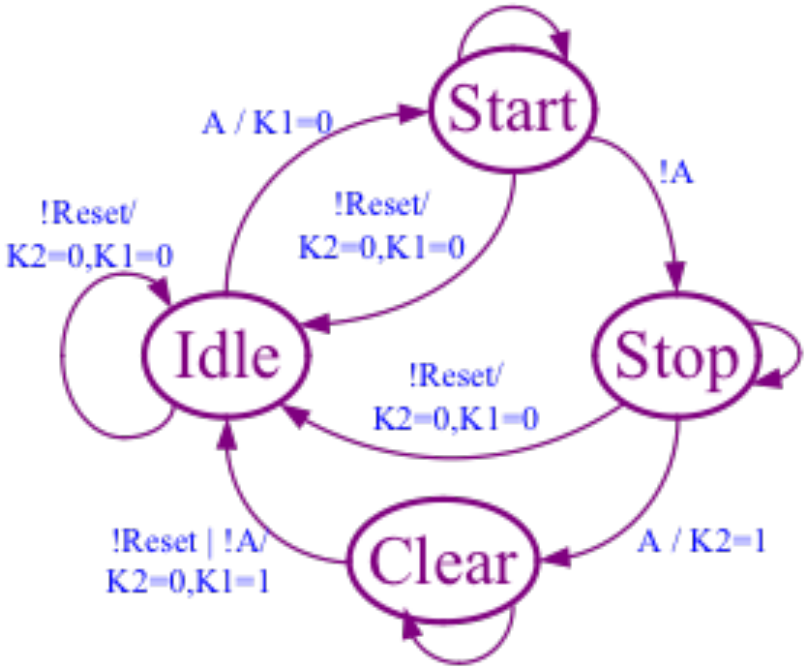
有限状态机的Verilog描述：方法一(续)

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



有限状态机的Verilog描述：方法一(续)

```
Stop:  begin   //符合条件进入新状态，否则留在原状态
         if(A)  begin
            state<= Clear;
             K2 <= 1;
            end
        else   state <= Stop;
      end
Clear: begin
         if(!A | !Reset)  begin
             state <= Idle;
              K2<=0;  K1<=1;
                end
           else state <= Clear;
         end
  endcase
endmodule
```



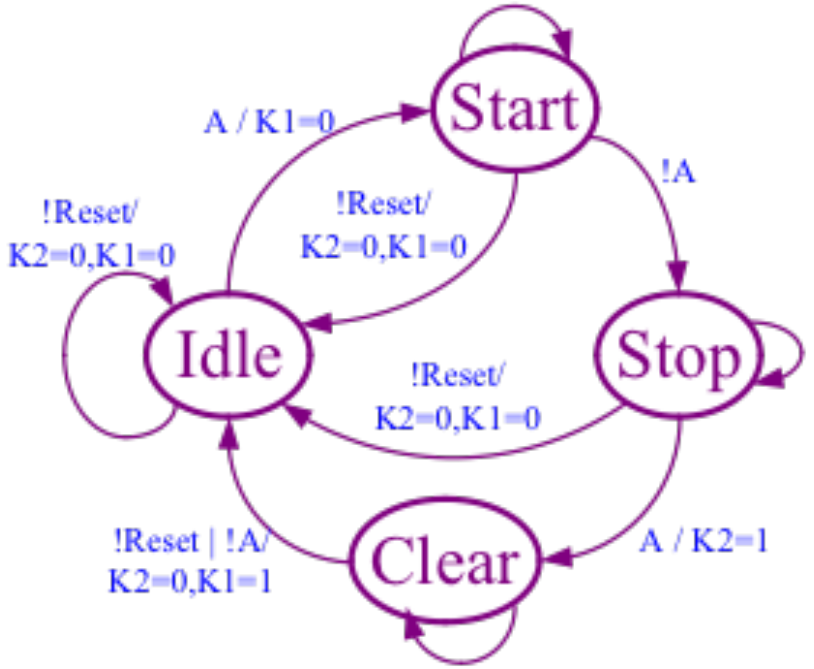
- 有限状态机的Verilog描述：方法二

```
module  fsm (Clock, Reset, A,  K2, K1);
input Clock, Reset, A;    //定义时钟、复位和输入信号
output K2, K1;        //定义输出控制信号的端口
reg K2, K1;        //定义输出控制信号的寄存器
reg [3:0] state ;     //定义状态寄存器 NOTE!
parameter    Idle    =  4'b1000,
             Start    =  4'b0100,
             Stop    =  4'b0010,
             Clear    =  4'b0001;
```

采用独热（one-hot）编码

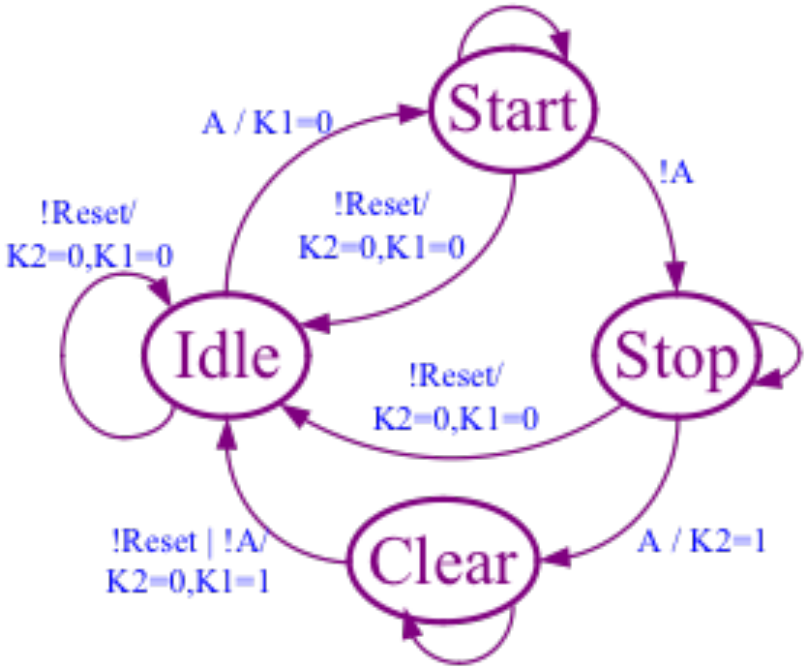
有限状态机的Verilog描述：方法二(续)

```
always @(posedge clock)
  if (!Reset)
    begin
      state <= Idle;
        K2<=0; K1<=0;
    end
  else
   case (state)
     Idle:
          if (A)  begin
                  state <= Start;
                      K1 <= 0;
                  end
        else   state <= Idle;
```



有限状态机的Verilog描述：方法二（续）

```
Start:
     if (!A)  state <= Stop;
       else  state <= Start;
Stop:
     if (A) begin
            state <= Clear;
              K2 <= 1;
            end
     else  state <= Stop;
Clear:
     if (!A | !Reset)  begin
        state <= Idle;
           K2 <=0;  K1<=1;
        end
       else   state <= Clear;
default:  state <= Idle; //采用独热编码后产生了多余状态，有些状态不可达，增
   加 default项，确保最后回到Idle状态。
endcase
endmodule
```



注：采用独热码可以使case电路控制更加简洁，从而提高电路的速度和可靠性。

- 有限状态机的Verilog描述：方法三

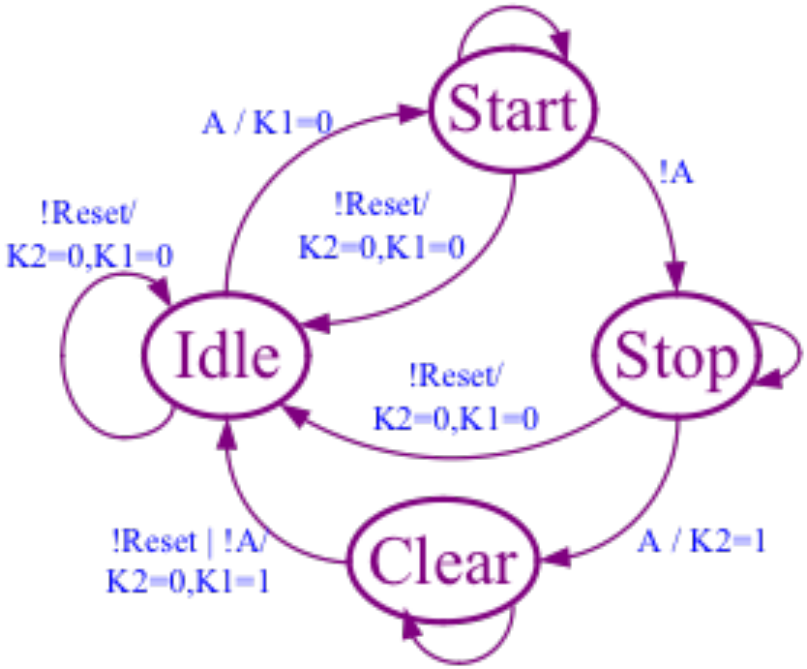
对于较复杂
的状态机设计

思路：把状态的变化与输出开关的控制分成两部分来考虑；为调试方便，常把每个开关写成独立的always组合块。

优点：在调试多输出状态机时，这样做比较容易发现问题和改正模块编写中出现的问题。

有限状态机的Verilog描述：方法三

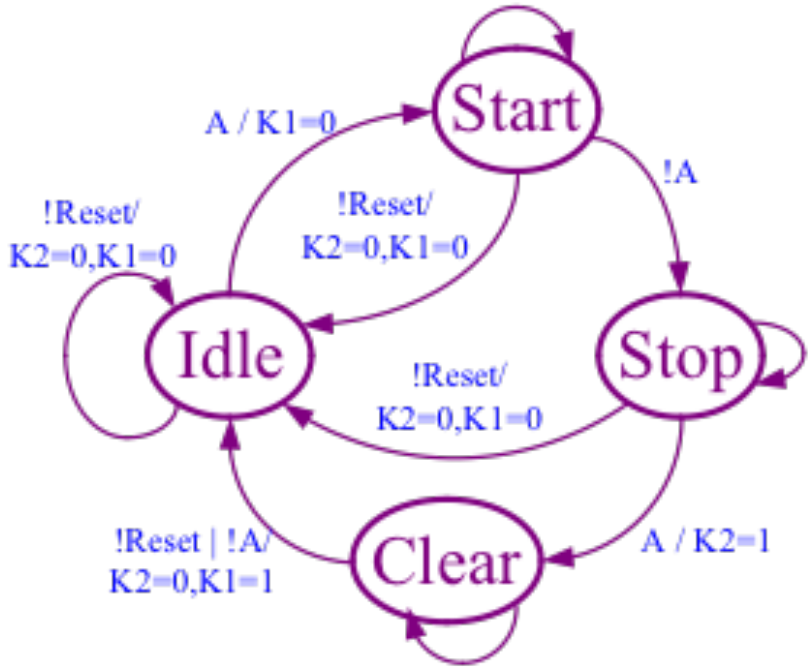
```
module  fsm (Clock, Reset, A,  K2, K1);
input Clock, Reset, A;
output K2, K1;
reg K2, K1;
reg [1:0] state, nextstate ;
parameter
Idle     =  2'b00,
Start    =  2'b01,
Stop     =  2'b10,
Clear    =  2'b11;
```




```
//-------- 每一个时钟沿产生一次可能的状态变化----------
always @(posedge Clock)
 begin   if (!Reset)
                state <= Idle;
         else   state <= nextstate;
 end
```

有限状态机的Verilog描述：方法三（续）

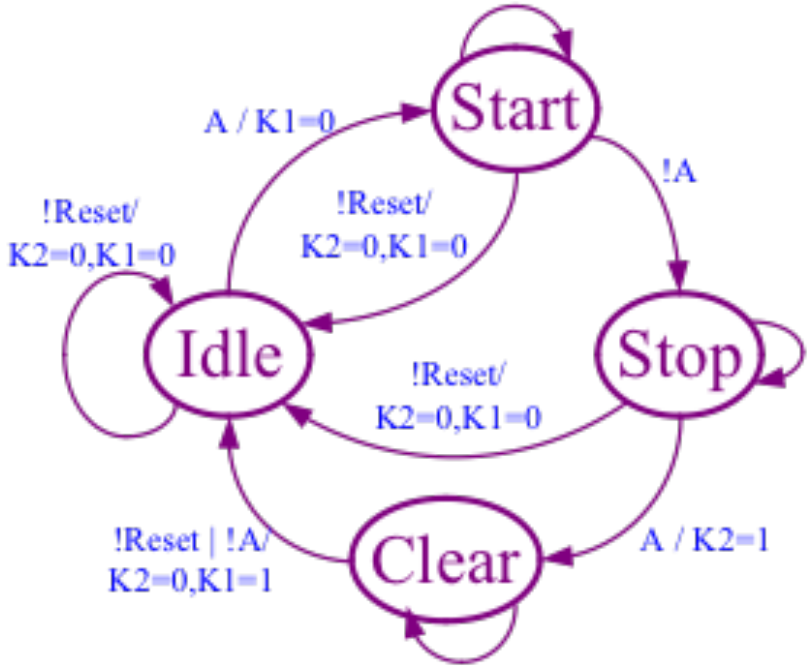
```
//------ 产生下一状态的组合逻辑
always @(state or A)
 case (state)
   Idle:
     if (A) nextstate = Start;
     else   nextstate = Idle;
      Start:
     if (!A)  nextstate = Stop;          else
nextstate = Start;
      Stop:
     if (A)  nextstate = Clear;          else
 nextstate = Stop;
      Clear:
     if (!A !reset)
              nextstate = Idle;
     else   nextstate = Clear;
   default: nextstate = Idle;
 endcase
```



注意：这是组合逻辑，阻塞赋值

- 有限状态机的Verilog描述：方法三（续）

```
//---- 产生输出K1的组合逻辑
always @(state or Reset or A)
 if (!Reset) K1=0;
   else
     if (state == Clear && !A)
      //从Clear转向 Idle
         K1=1;
        else   K1= 0;
//--- 产生输出K2的组合逻辑
always @(state or Reset or A )
  if (!Reset) K2 = 0;
   else
     if (state == Stop && A)  // 从Stop转向 Clear
         K2 = 1;
     else K2 = 0;
```



- 小结：

上面例子是同一个状态机的四种不同的Verilog HDL模型，它们都是可综合的，在设计复杂程度不同的状态机时有它们各自的优势。如用不同的综合器对这四个例子进行综合，综合出的逻辑电路可能会有些不同，但逻辑功能是相同的。

状态机的设计举例

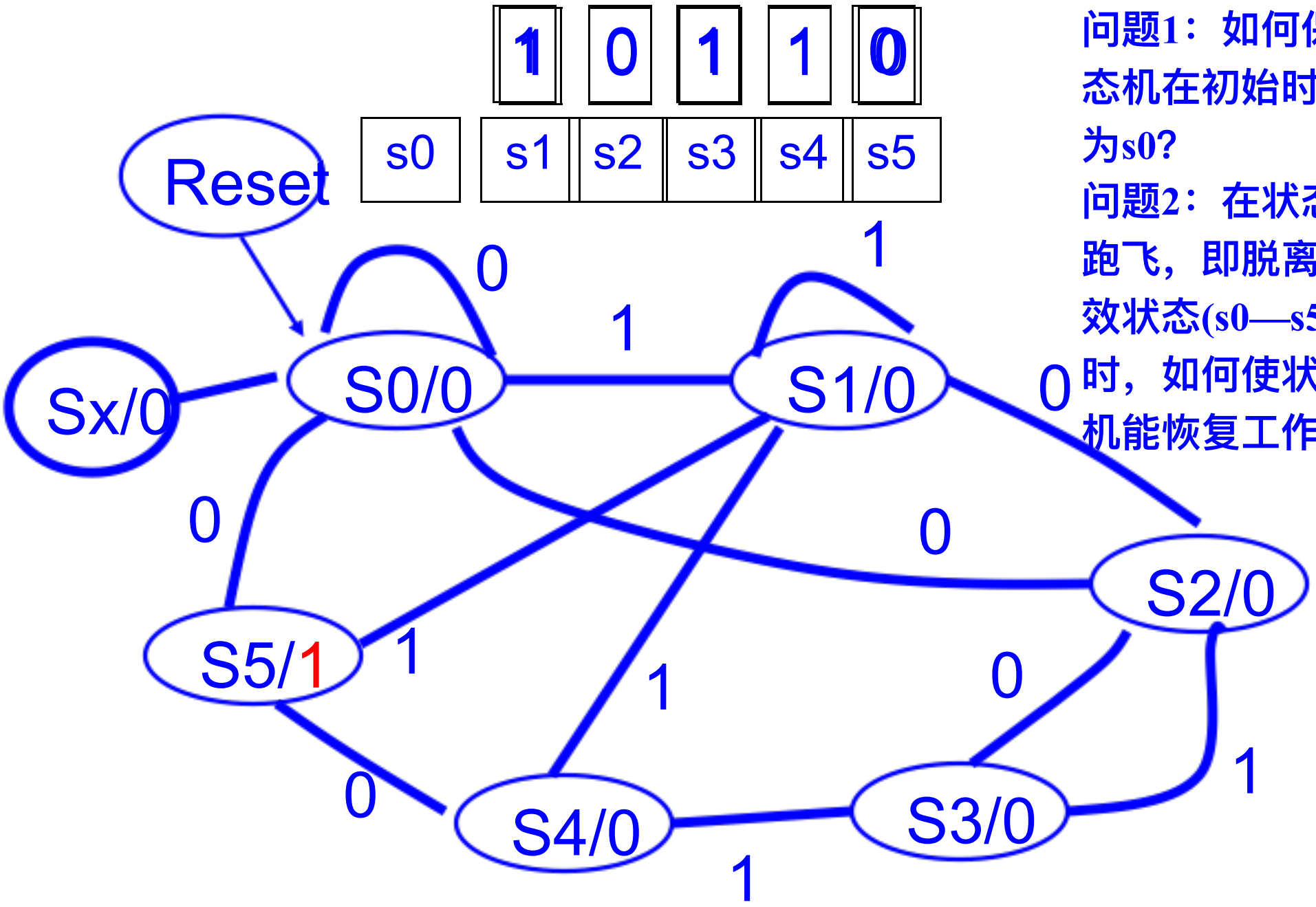
例：设计一个二进制序列检测器，当检测到10110序列时，就输出1(一个时钟周期的脉冲)。其他情况下输出0。

规定检测到一次之后，检测器复位到最初始的状态，重新从头检测。如下所示：

输入：01101101101100

输出：00000001000001

状态转换图设计（Moore）

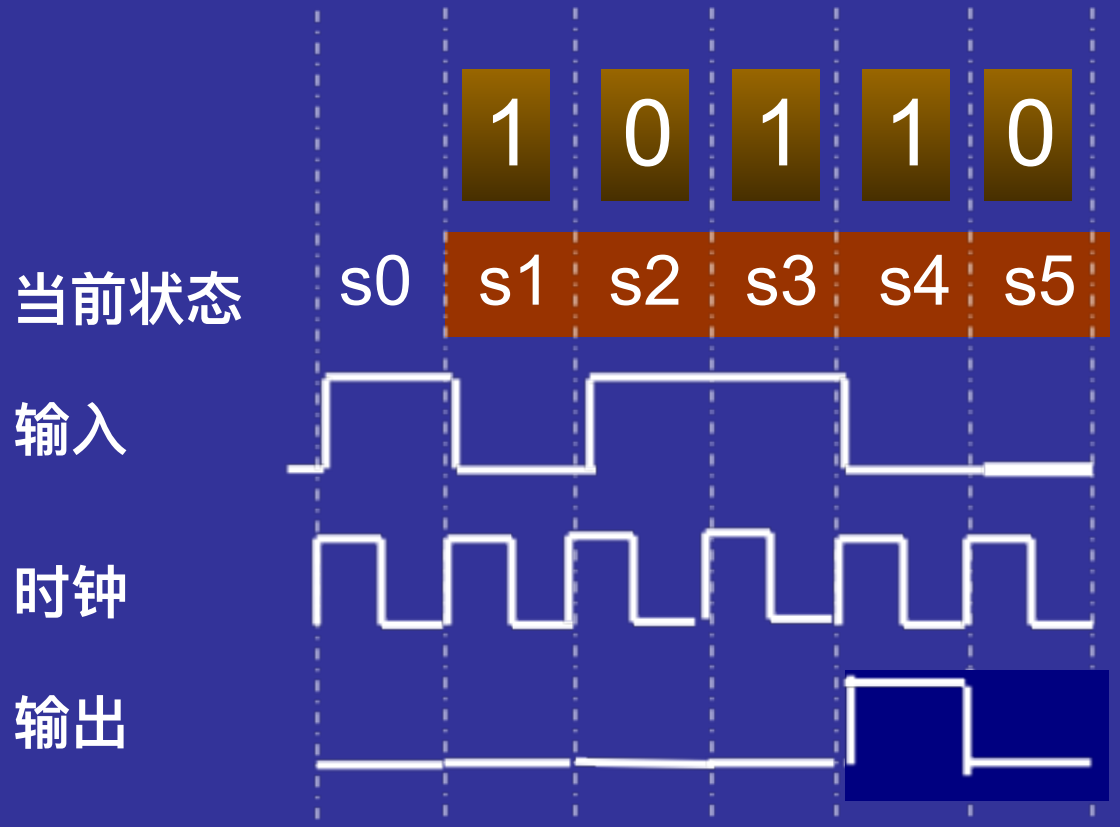


问题1：如何保证状态机在初始时状态为s0?

问题2：在状态机跑飞，即脱离有效状态(s0—s5)时，如何使状态机能恢复工作

波形

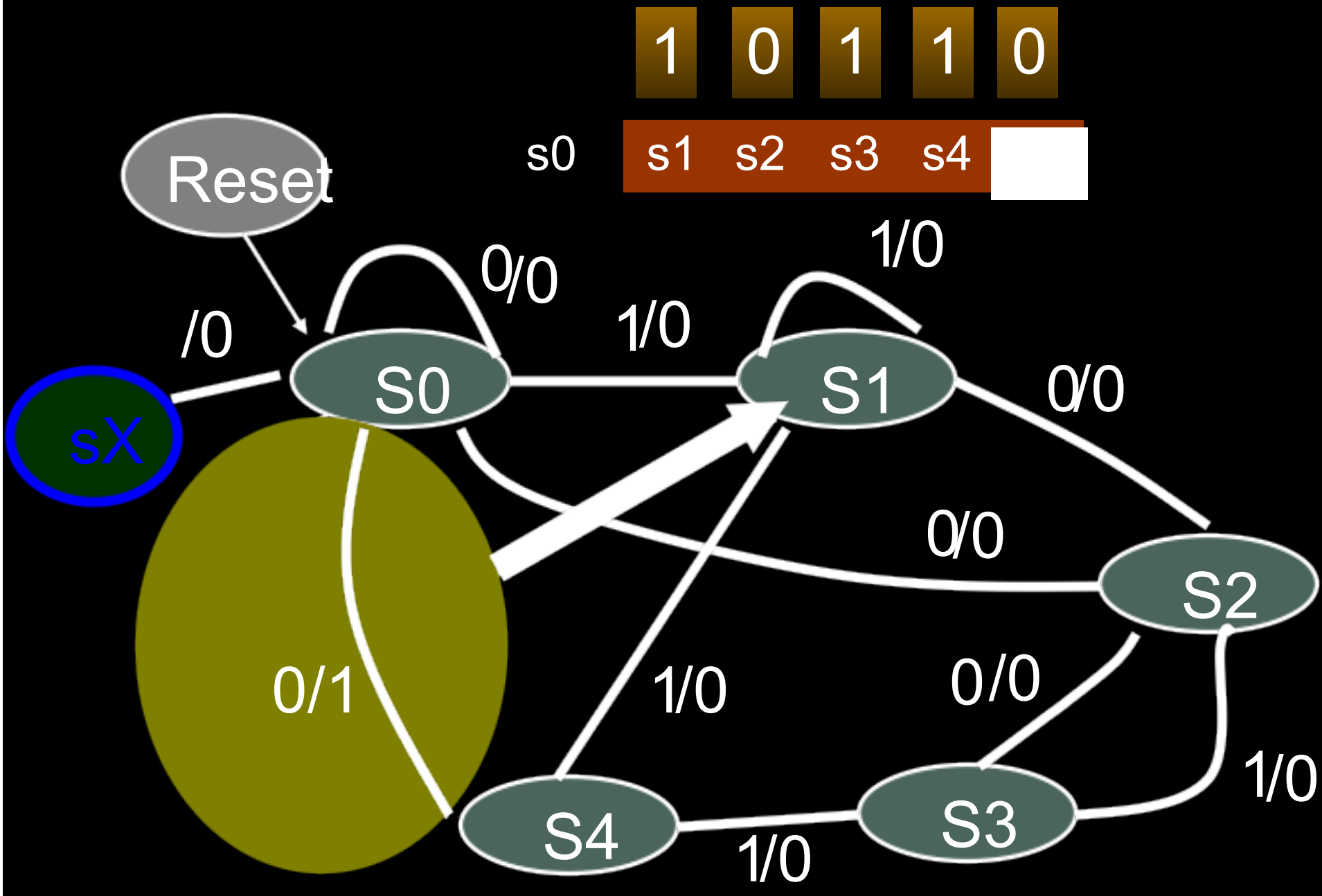
- 波形如下图所示



问题：如果需要将输出脉冲往前推一个时钟周期，该如何修改设计？

发现当当前壮态为s4，并且输入为0时，输出为1。

状态转换图设计(Mealy)



有限状态机（FSM）设计举例

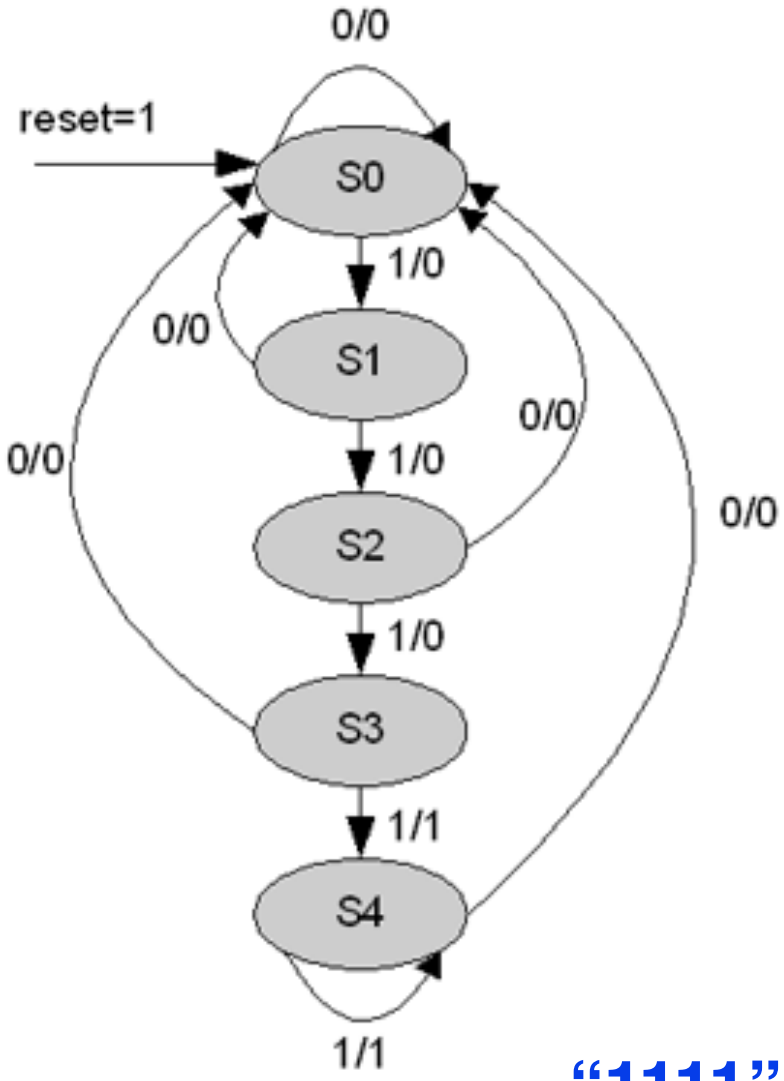
用状态机设计一个二进制序列检测器，其功能是检测一个4位二进制序列“1111”，即输入序列中如果有4个或4个以上连续的“1”出现，输出为1，其它情况下，输出为0。

其输入输出如下所示：

输入x：000 101 010 110 111 101 111 110 101

输出z：000 000 000 000 000 100 001 110 000

有限状态机（FSM）设计



“1111”序列检测器状态转换图

“1111”序列检测器的Verilog描述

```
module fsm_seq(x,z,clk,reset,state);
input x,clk,reset;
output z;
output[2:0] state;
reg z;
parameter s0=0,s1=1,s2=2,s3=3,s4=4;
reg [2:0] current_state, next_state;

assign state=current_state;
always @(posedge clk or posedge reset)
begin
 if(reset)
      current_state<=s0;
  else
      current_state<=next_state;
end
```

```
always @(current_state or x)
begin
    casex(current_state)
```

```
s0: begin
  if(x==0)  begin  next_state<=s0; z<=0;  end
   else    begin  next_state<=s1; z<=0;  end
   end
s1: begin
  if(x==0) begin  next_state<=s0; z<=0; end
       else   begin  next_state<=s2; z<=0;end
  end
s2: begin
  if(x==0)  begin  next_state<=s0; z<=0; end
       else    begin  next_state<=s3; z<=0;end
  end
s3: begin
     if(x==0) begin  next_state<=s0; z<=0; end
    else   begin  next_state<=s4; z<=1; end
  end
s4: begin
  if(x==0) begin  next_state<=s0; z<=0;  end
   else    begin  next_state<=s4; z<=1;  end
  end
       default:   begin  next_state<=s0;   end
endcase
end
endmodule
```

设计饮料自动投币售卖机的核心控制电路

要求

简化考虑，假设饮料只有一种价格为2.5元。硬币有0.5元和1.0元两种，考虑找零，用Verilog描述其控制电路，并用FPGA实现

设计饮料自动投币售卖机的核心控制电路

设计步骤分解

1	分析输入输出端口信号；
2	状态转移图；
3	根据状态转移图进行Verilog 语言描述；
4	测试代码编写，仿真；
5	FPGA实现。

分析输入输出信号

输入信号： clk， rst；

输入信号：操作开始: op_start； //定义1开始操作

输入信号：投币币值: coin_val；//定义2'b01表示0.5元； 2'b10表示1元

输入信号：取消操作指示 : cancel_flag; //定义1为取消操作

输出信号：机器是否占用: hold_ind；//定义0为不占用，可以使用

输出信号：取饮料信号: drinktk_ind；//定义1为取饮料

输出信号：找零与退币标志信号: charge_ind; //定义1为找零

输出信号：找零与退币币值： charge_val;

//定义3'b001表示找0.5元； 3'b010表示找1元

// 3'b011表示找1.5元; 3'b100表示找2.0元;

状态确定

S0： 初始态；

S1： 已投币0.5元

S2： 已投币1.0元

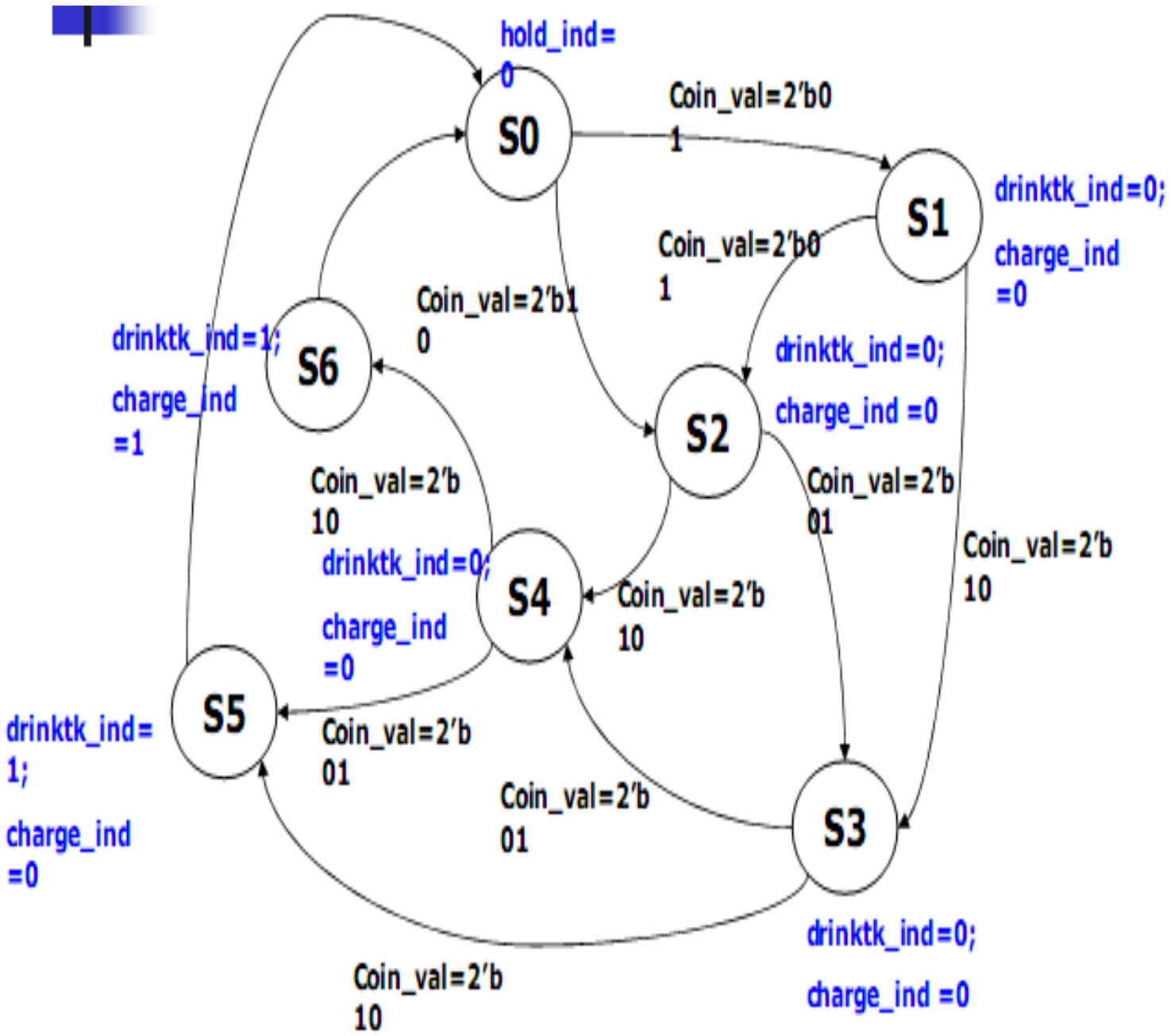
S3： 已投币1.5元

S4：已投币2.0元；

S5： 已投币2.5元

S6： 已投币3.0元

状态转移图



状态说明：

S0 ： 初始态；

S1 ： 已投币0.5元；

S2 ： 已投币1.0元；

S3 ： 已投币1.5元；

S4： 已投币2.0元；

S5 ： 已投币 2.5元；

S6 ： 已投币3.0元；

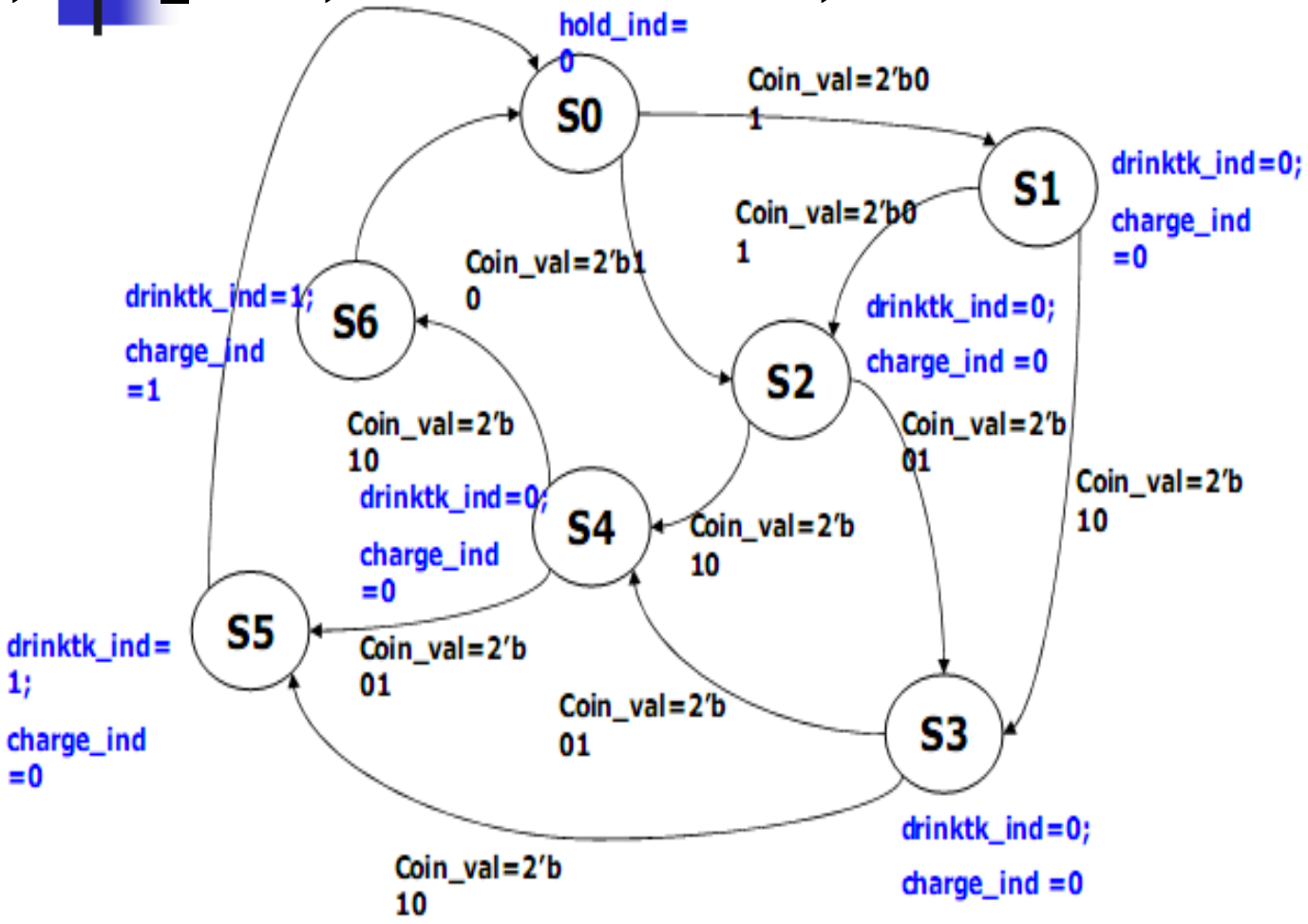
说明：1.在S0 状态下，如果检测到op_start=1，开始检测是否有投币，如果有，一次新的售货操作开始；

2.在状态S1/S2/S3/S4下，如果检测到cancel_flag=1，则取消操作，状态回S0，并退回相应的币值；

3.在状态S5下，卖出饮料不找零；在状态S6下，卖出饮料并找零；

4.在状态S5和S6 操作完后，都返回状态S0，等待下一轮新的操作开始；

5.只有在S0 状态下，hold_ind=0，可以发起新一轮操作，其它状态下都为1



设计饮料自动投币售卖机的核心控制电路

源代码

```
module softdrinkFSM(clk,rst,op_start,cancel_flag,coin_val,
                    hold_ind,charge_ind,drinktk_ind,charge_val);
input   clk,rst;
input   op_start,   cancel_flag;
input   [1:0] coin_val;
output hold_ind,   charge_ind,drinktk_ind;
output [2:0] charge_val;
reg hold_ind,   charge_ind, drinktk_ind;
reg [2:0] charge_val;
reg [2:0]  currentstate, nextstate;
parameter  S0=3'b000;    parameter  S1=3'b001;
parameter  S2=3'b010;    parameter  S3=3'b011;
parameter  S4=3'b100;    parameter  S5=3'b101;
parameter  S6=3'b110;
```

```
always @(posedge clk or posedge rst)
  if  (rst)
      currentstate <= S0;
  else
     currentstate <= nextstate;
always@ (currentstate or rst or op_start or cancel_flag or coin_val)
  if(rst)    nextstate=S0;
  else  case (currentstate)
       S0:     if(op_start)
              if(coin_val==2'b01) nextstate=S1;
              else  if(coin_val==2'b10)  nextstate=S2;

            S1:   if(cancel_flag)  nextstate=S0;
                  else if(coin_val==2'b01)   nextstate=S2;
                  else if(coin_val==2'b10)  nextstate=S3;
            S2:   if(cancel_flag) nextstate=S0;
                  else if(coin_val==2'b01)  nextstate=S3;
                  else if(coin_val==2'b10)  nextstate=S4;
            S3:  if(cancel_flag) nextstate=S0;
                 else  if(coin_val==2'b01) nextstate=S4;
                 else  if(coin_val==2'b10)   nextstate=S5;
```

```
  S4: if(cancel_flag)  nextstate=S0;
      else  if(coin_val==2'b01)  nextstate=S5;
      else  if(coin_val==2'b10)  nextstate=S6;
  S5: nextstate=S0;
  S6: nextstate=S0;
  default:   nextstate=S0;
  endcase


always @ (currentstate)
  if (currentstate == S0) hold_ind = 1'b0;
    else   hold_ind = 1'b1;
always @ (currentstate)
  if ((currentstate == S5 )||(currentstate == S6))
      drinktk_ind = 1'b1;
  else drinktk_ind = 1'b0;
always @ (currentstate or cancel_flag)
  if(currentstate == S0) charge_ind= 1'b0;
  else  if(currentstate == S6)  charge_ind= 1'b1;
  else if(cancel_flag)  charge_ind= 1'b1;
  else   charge_ind= 1'b0;
```

```
always @ (currentstate or cancel_flag)
  if(currentstate == S0)       charge_val= 3'b000
  else  if(currentstate == S6) charge_val= 3'b001
  else  if(cancel_flag)
  begin
      case (currentstate)
          S1: charge_val= 3'b001;
          S2: charge_val= 3'b010;
          S3: charge_val= 3'b011;
          S4: charge_val= 3'b100;
          default: charge_val= 3'b000;
      endcase
   end
   else   charge_val= 3'b000;
endmodule
```

testbench

```
module  softdrink_testbench;
    reg rst, clk;
    reg op_start;
    reg cancel_flag;
    reg [1:0] coin_val;
    wire  hold_ind;
    wire charge_ind;
    wire drinktk_ind;
    wire [2:0] charge_val;
    initial  clk=0；
    always #500 clk=~clk;
```

```
initial
 begin
     rst=0;
    op_start=0; cancel_flag=0;
    coin_val=2'b00;
    #25 rst=1;
    #25 rst=0;
  //第一次：依次投入一元、一元、一元的硬币，看结果如何?
    #50 op_start=1;
    #300     coin_val=2'b10;
    #1000    coin_val=2'b10;
    #1000    coin_val=2'b10;
    #1000    op_start=0;
```

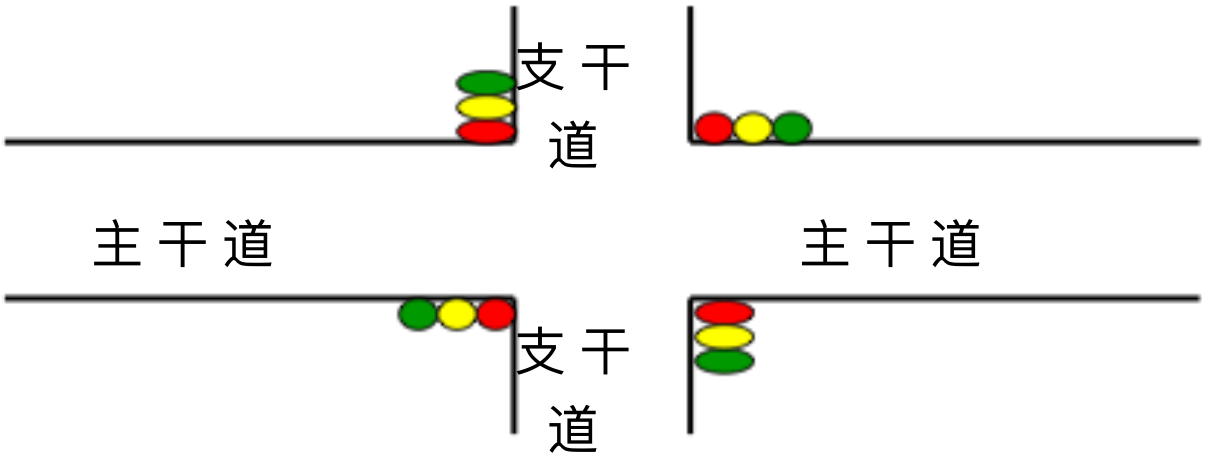
```
//第二次：依次投入0.5元、一元、一元的硬币，看结果又如何?
#2000 op_start=1; coin_val=2'b01;
#1000 coin_val=2'b10;
#1000 coin_val=2'b10;
#1000 op_start=0;
//第三次：依次投入0.5元、一元的硬币，然后取消操作，看结果如何?
#2000 op_start=1;
    coin_val=2'b01;
#1000 coin_val=2'b10;
#1000 cancel_flag=1'b1;
    op_start=0;
#1000 cancel_flag=0;
```

```
//第四次：依次投入0.5元、 0.5元、 0.5元、 0.5元、一元的硬
币，看结果如何?
#2000 op_start=1; coin_val=2'b01;
#1000 coin_val=2'b01;
#1000 coin_val=2'b01;
#1000 coin_val=2'b01;
#1000 coin_val=2'b10;
#1000 op_start=0;
#1000000 $stop;
end     //end initial
softdrinkFSM u0 (rst,clk,op_start,coin_val,cancel_flag,
hold_ind,charge_ind,drinktk_ind,charge_val);
endmodule
```

交通灯的核心控制电路

设计描述

用状态机设计交通灯控制器。有一条主干道和一条支干道的汇合点形成十字交叉路口，主干道为东西向，支干道为南北向。为确保车辆安全，迅速地通行，在交叉道口的每个入口处设置了红、绿、黄3色信号灯



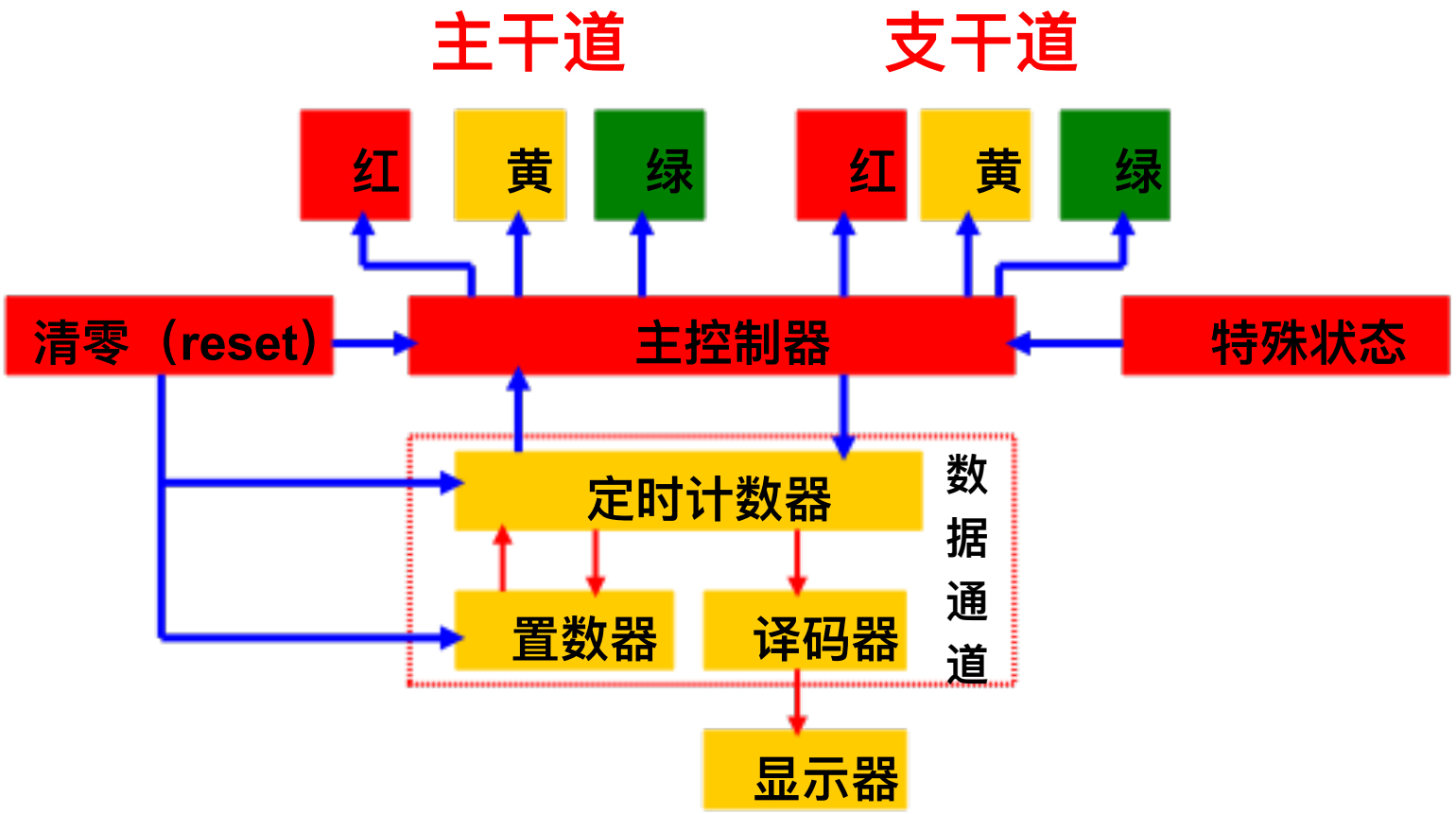
设计要求

1	主干道绿灯亮时，支干道红灯亮，反之亦然，主干道每次放行35s，支 干道每次放行25s.每次由绿灯变为红灯的过程中，亮光的黄灯作为过 渡，时间5s
2	能实现正常的倒计时显示功能
3	能实现总体清零功能；计数器由初始状态开始计数，对应状态的 指示灯亮
4	能实现特殊状态的功能显示；进入特殊状态时，东西、南北路口 均显示红灯状态

状态转换表

状态	主干道	支干道	时间
0	红灯亮	红灯亮	
1	绿灯亮	红灯亮	35s
2	黄灯亮	红灯亮	5s
3	红灯亮	绿灯亮	25s
4	红灯亮	黄灯亮	5s

系统框图



分析输入输出信号

输入信号

时钟
clock

复位清零信号reset（reset=1表示系统复位）

紧急状态输入信号sensor1（sensor1=1表示进入紧急状态）

定时计数器的输入信号sensor2（由sensor2【2】、 sensor2【1】、
sensor2【2】三位组成，该信号为高电平时，分别表示35s,5s，25s的计时完成）

分析输入输出信号

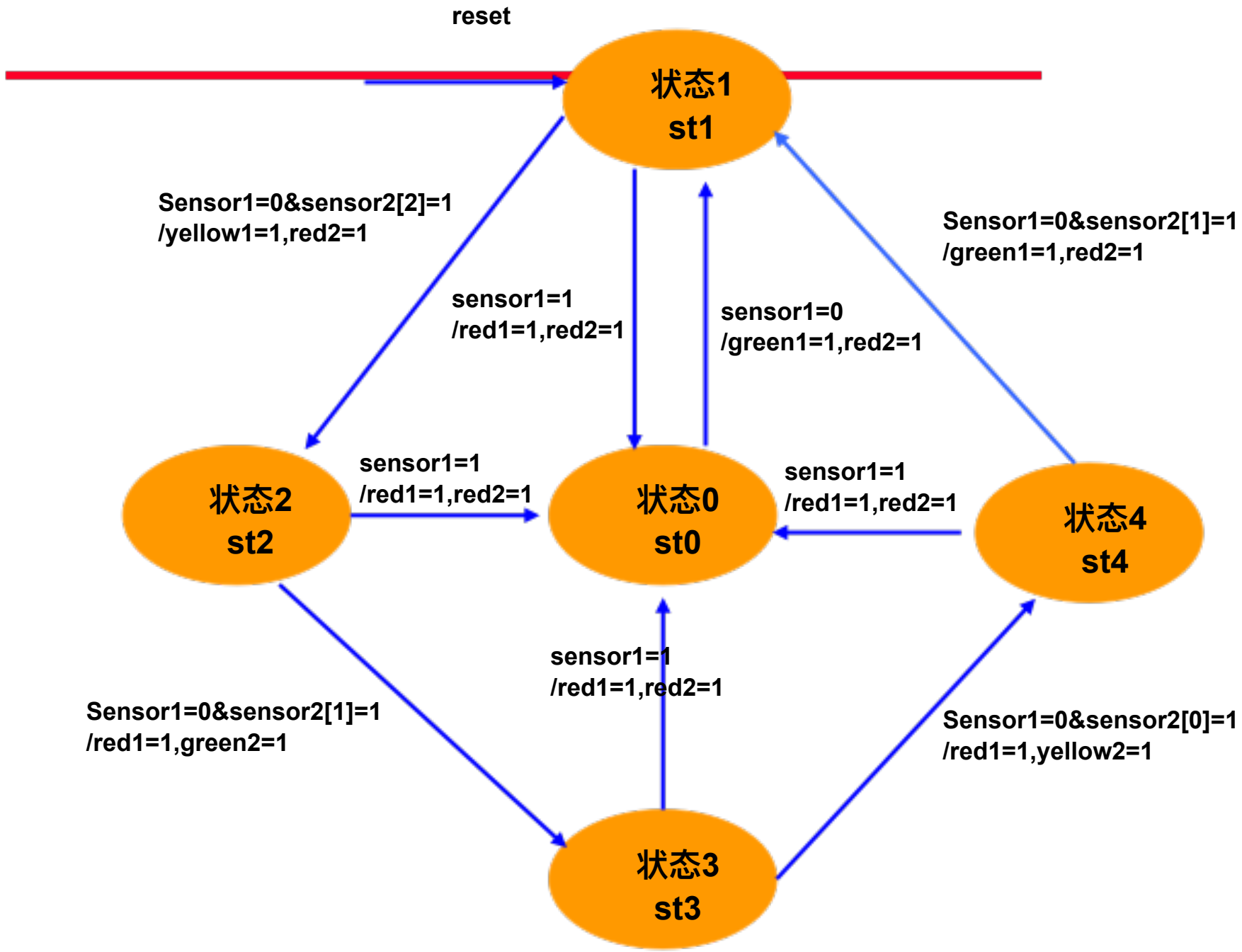
输出信号

主干道控制信号
（red1，yellow1，green1）

支干道控制信号
（red2，yellow2，green2）

控制状态信号state（输出到定时计数器，分别进行

状态转移图


reset
状态1
st1
Sensor1=0&sensor2[2]=1
/yellow1=1,red2=1
sensor1=1
/red1=1,red2=1
sensor1=0
/green1=1,red2=1
Sensor1=0&sensor2[1]=1
/green1=1,red2=1
状态2
st2
sensor1=1
/red1=1,red2=1
状态0
st0
sensor1=1
/red1=1,red2=1
状态4
st4
sensor1=1
/red1=1,red2=1
Sensor1=0&sensor2[1]=1
/red1=1,green2=1
Sensor1=0&sensor2[0]=1
/red1=1,yellow2=1
状态3
st3