有限状态机

为什么要设计有限状态机

应用的需要，复杂系统的需要

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，并记住自己目录所处的状态。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操作。

什么是状态机 ？

有限状态机(简称状态机)相当于一个控制器，它将一项功能的完成分解为若干步，每一步对应于二进制的一个状态，通过预先设计的顺序在各状态之间进行转换，状态转换的过程就是实现逻辑功能的过程。

有限状态机三要素

FSM:Finite State Machine

1. 状态（当前状态，下一个状态）
2. 输入信号（事件）
3. 输出控制信号（相应操作）

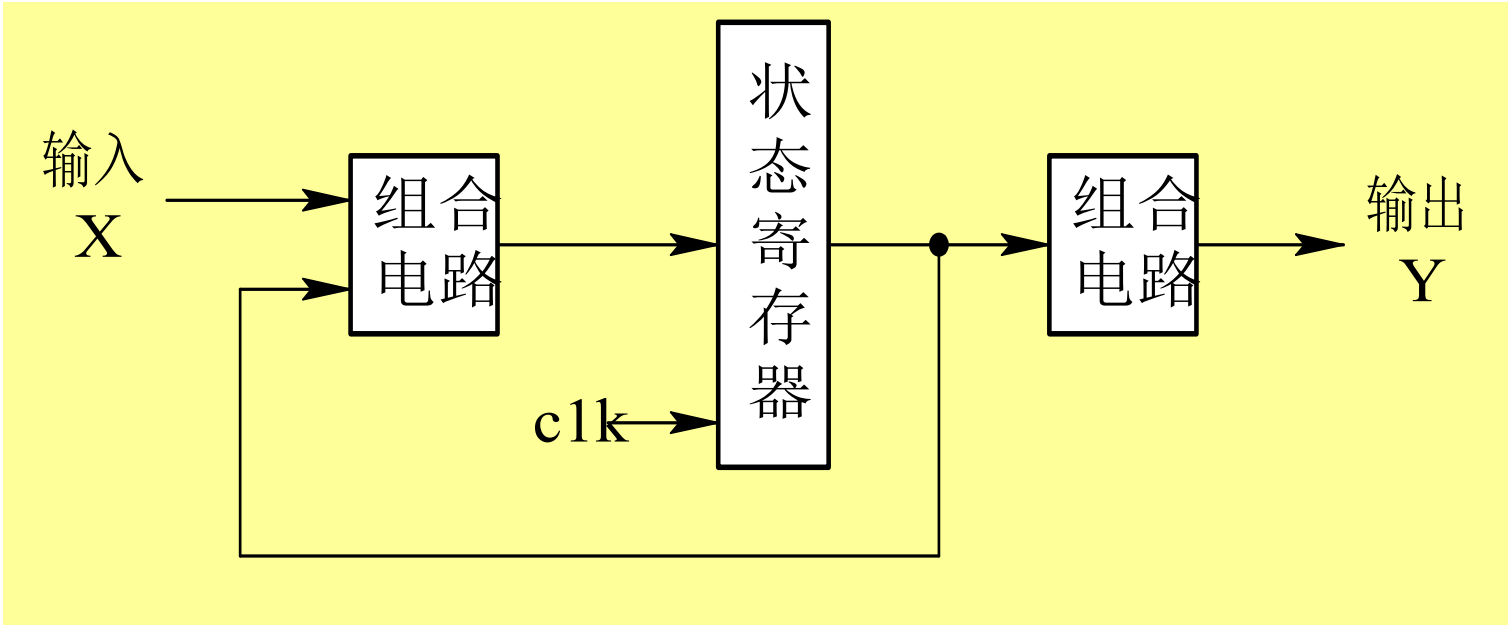
什么是有限状态机

有限状态机是由寄存器组和组合逻辑构成的硬件时序电路

解释说明

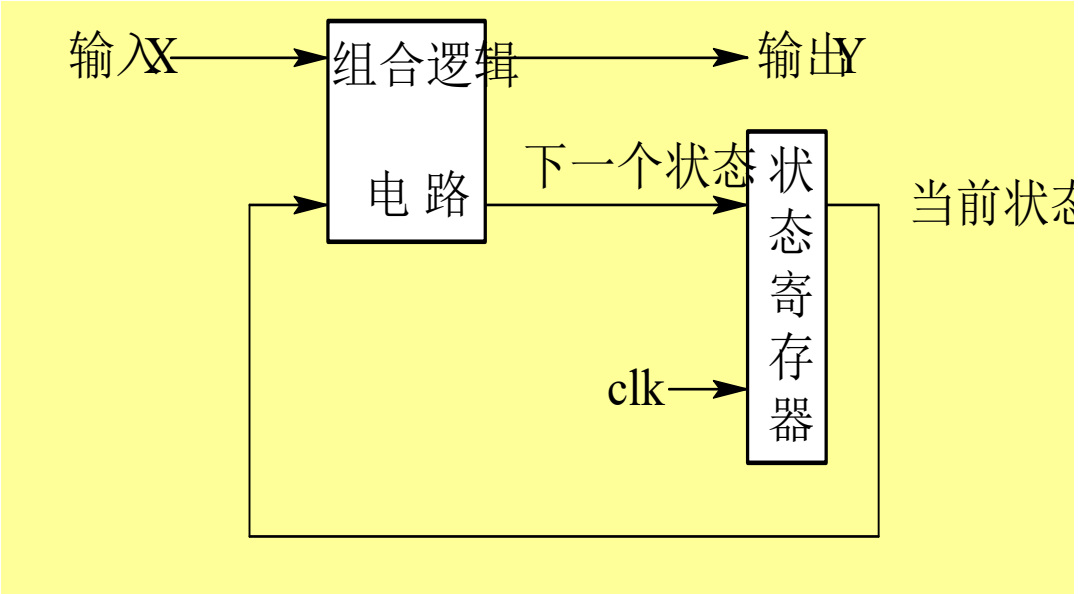
1. 其状态（即由寄存器组1和0的组合状态所构成的有限个状态）只能在同一时钟跳变沿的情况下，才能从一状态转向另一个状态。
2. 究竟转向哪一个状态不但取决于各个输入值，还取决于当前状态
3. 状态机可产生在时钟跳变沿时刻进行开关的复杂的控制逻辑，是数字逻辑的控制核心。

Moore型状态机设计方法



摩尔型状态机的典型结构

Mealy型状态机设计方法



Mealy型状态机的典型结构

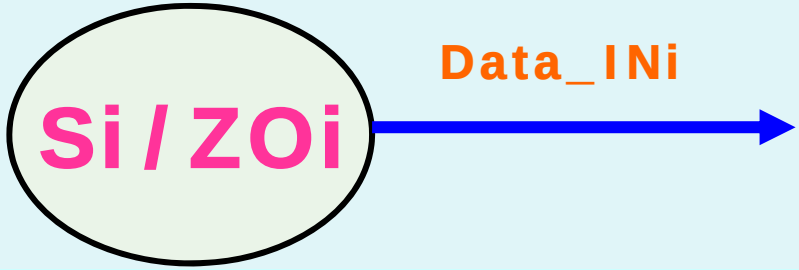
Moore型状态机设计方法

Moore型状态机输出仅和状态机的当前状态有关，与外部输入无关，即：

外部输出是内部状态的函数。

输入信号的变化决定当前状态的下一状态，即次态。

状态转移图描述方式：



Moore型状态机设计方法

例：设计一个序列检测器。要求检测器连续收到串行码{1101}后，输出检测标志1，否则输出0。

状态机设计步骤：

① 分析设计要求，列出全部可能状态；

② 画出状态转移图；

③ 用HDL语言描述状态机。

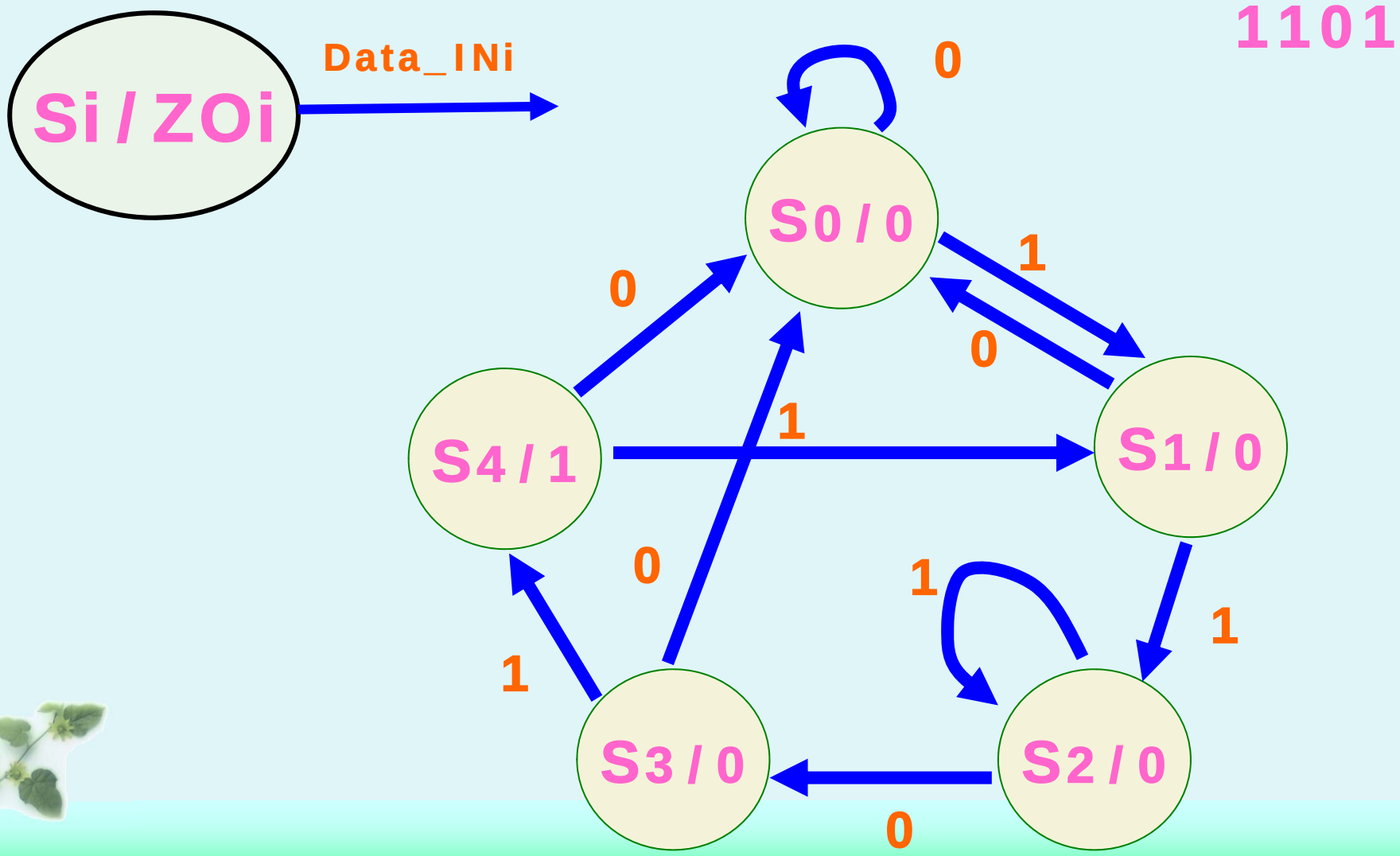
Moore型状态机设计方法

（1） 分析设计要求，列出全部可能状态：

未收到一个有效位（0） ：S0
收到一个有效位（1） ：S1
连续收到两个有效位（11） ：S2
连续收到三个有效位（110） ：S3
连续收到四个有效位（1101） ：S4

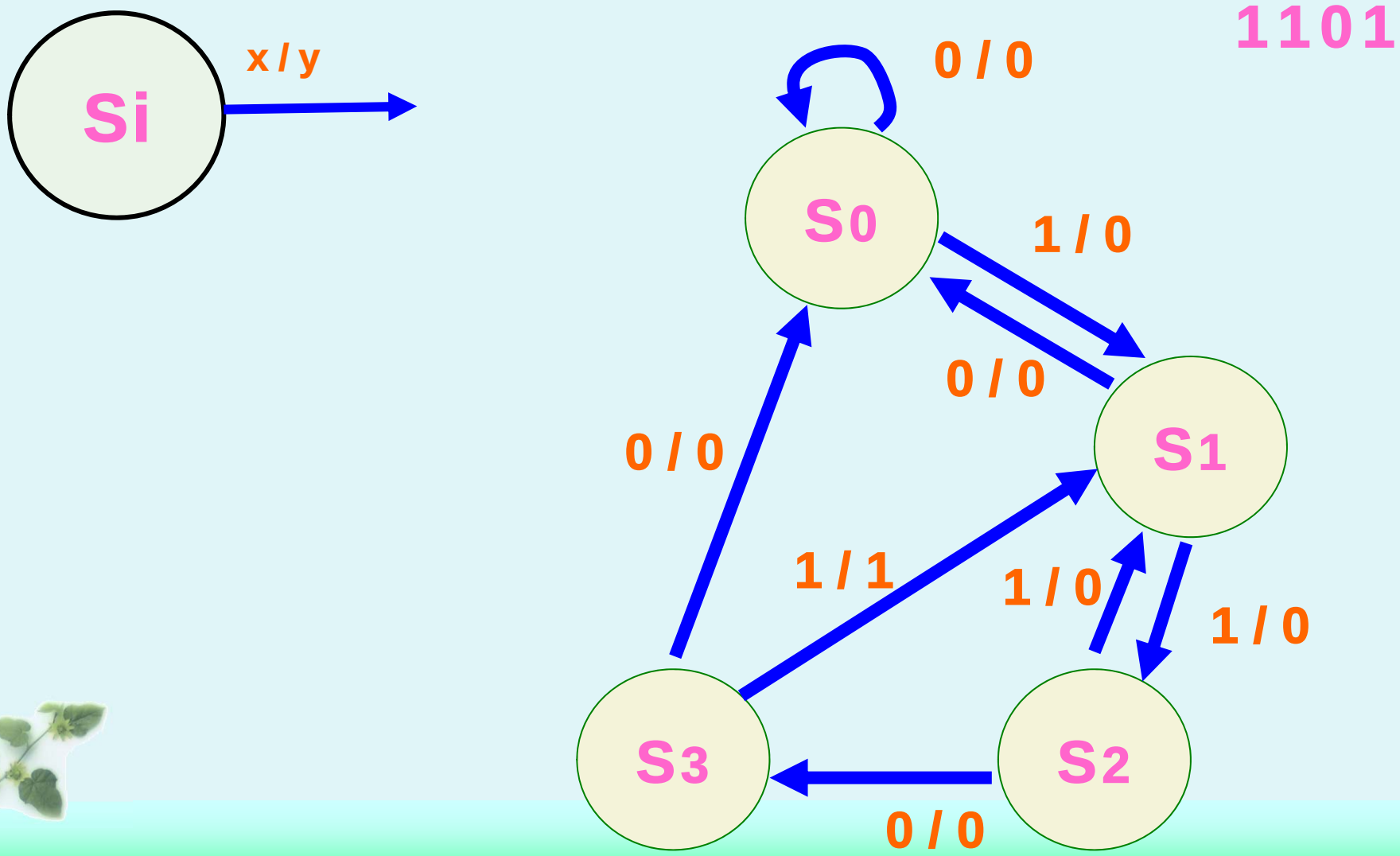
Moore型状态机设计方法

（2） 状态转移图:



Mealy型状态机设计方法

（2） 状态转移图：



4 Moore型状态机设计方法

例：设计一个序列检测器。要求检测器连续收到串行码{111}后，输出检测标志1，否则输出0。

状态机设计步骤：

① 分析设计要求，列出全部可能状态；

② 画出状态转移图；

③ 用HDL语言描述状态机。

Moore型状态机设计方法

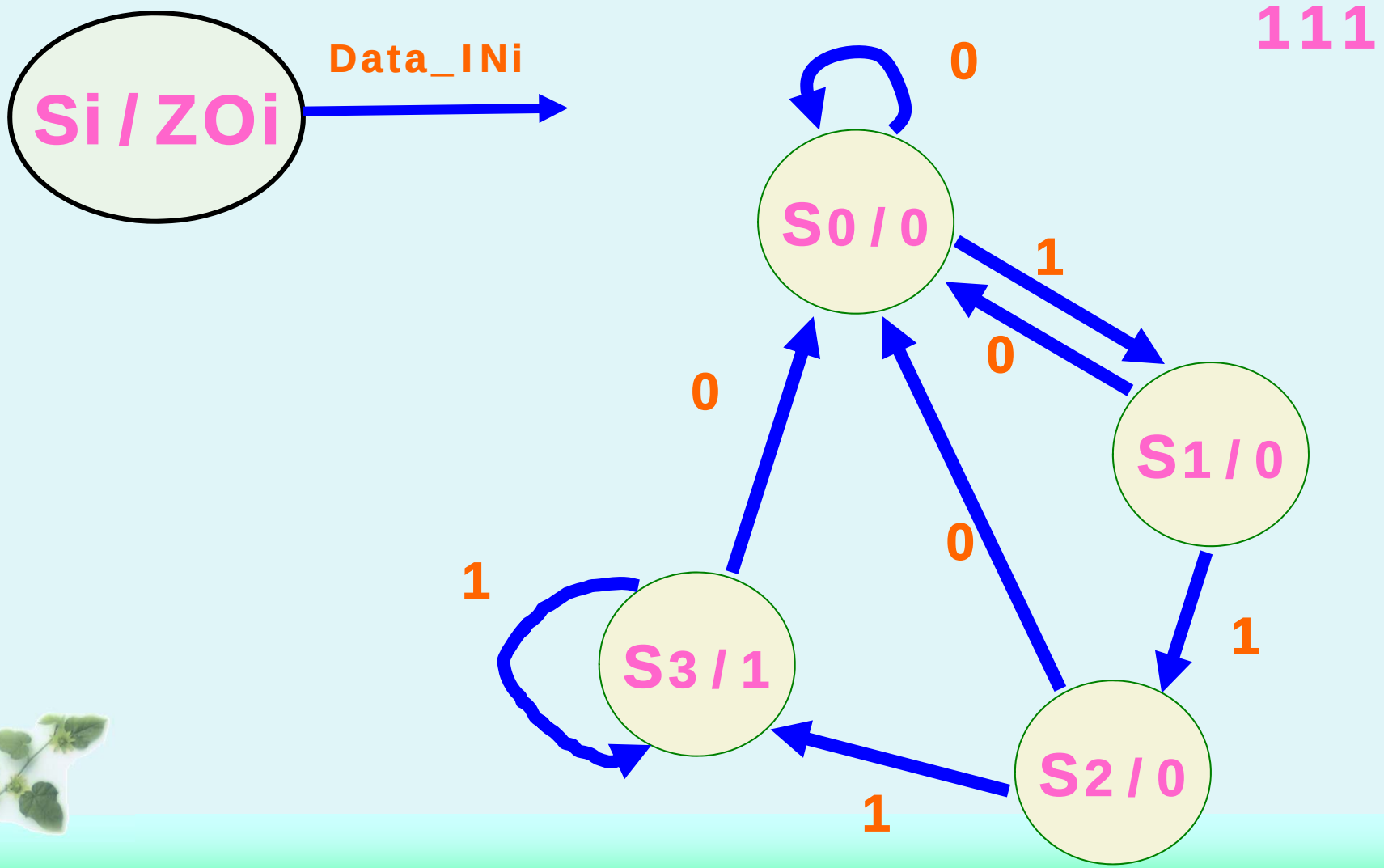
（1） 分析设计要求，列出全部可能状态:

未收到一个有效位（0） ：S0
收到一个有效位（1） ：S1
连续收到两个有效位（11） ：S2
连续收到三个有效位（111） ：S3

现态	次态		输出Y
	x=0	x=1	
S0	S0	S1	0
S1	S0	S2	0
S2	S0	S3	1
S3	S0	S3	1

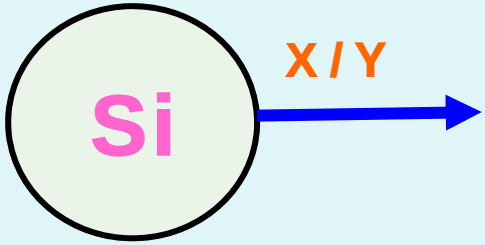
Moore型状态机设计方法

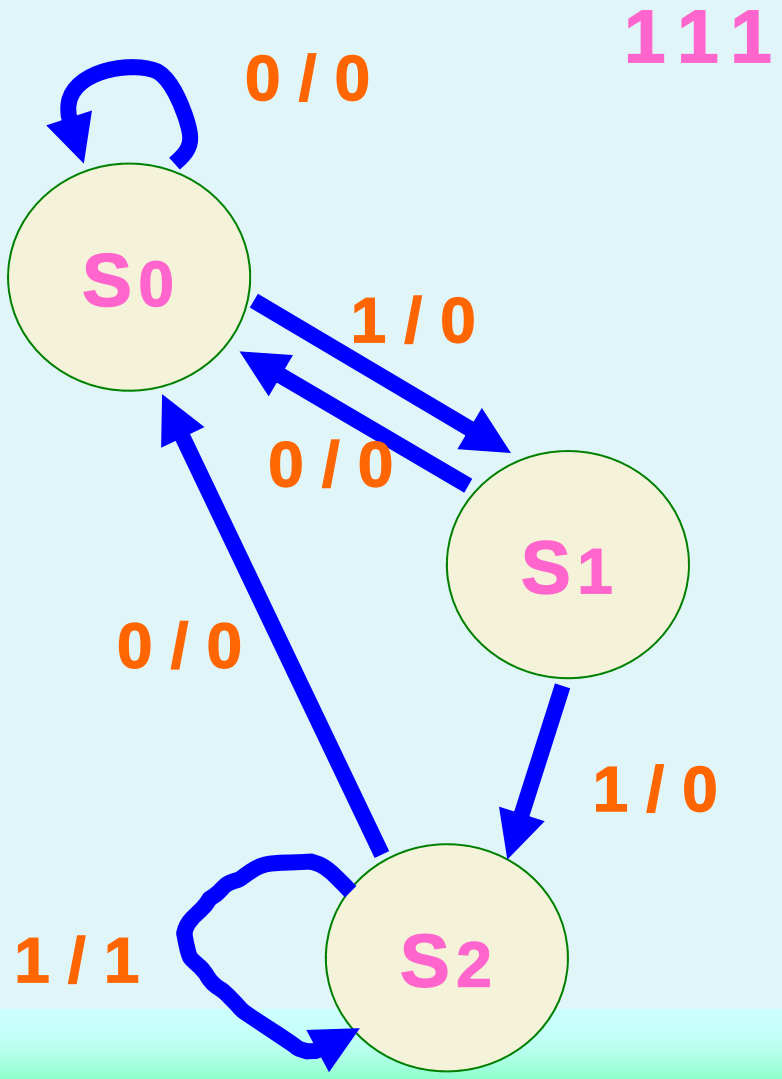
（2） 状态转移图：



Mealy型状态机设计方法

（2） 状态转移图：



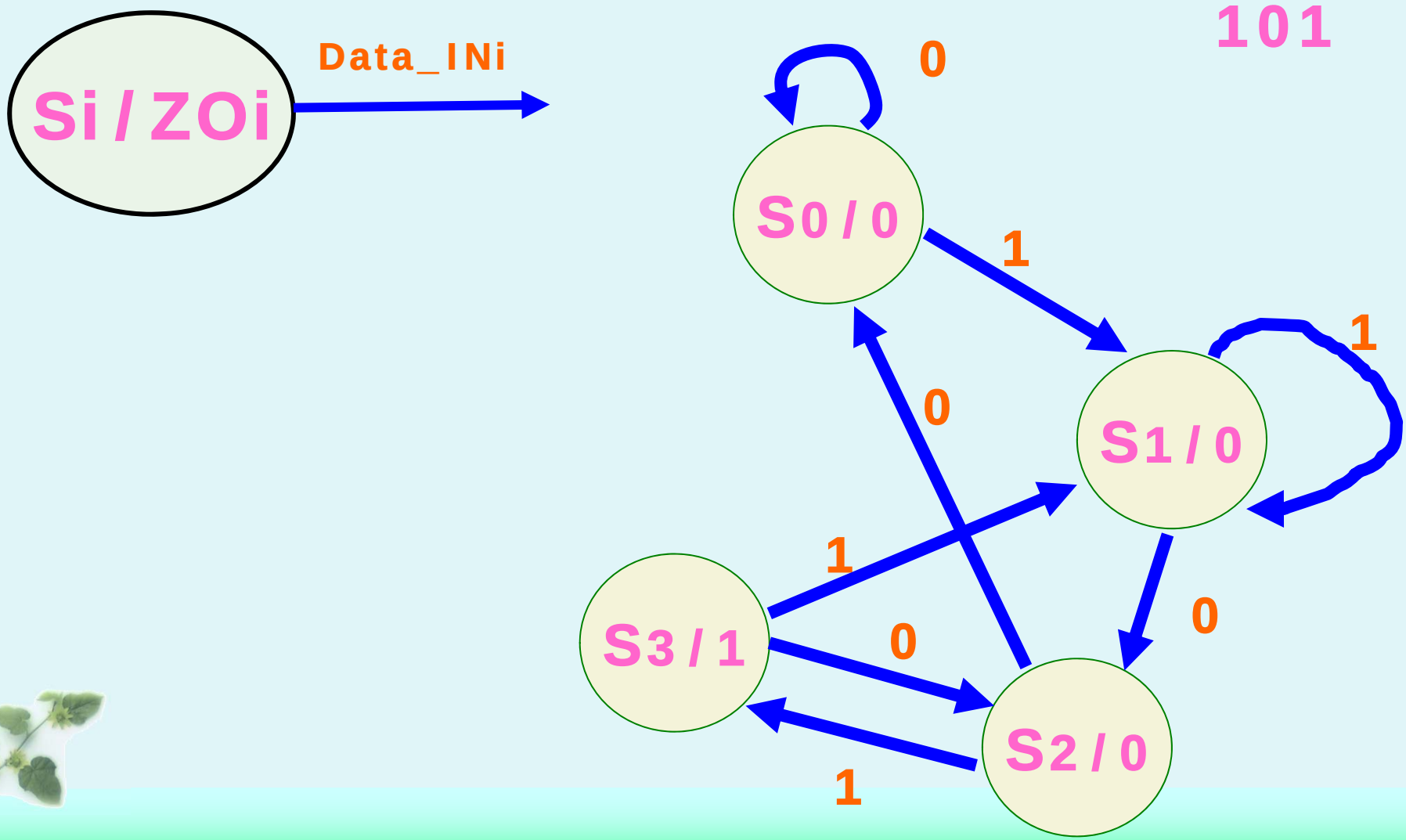


（3） 状态转移表

现态	次态		输出y	
	X=0	X=1	X=0	X=1
S0	S0	S1	0	0
S1	S0	S2	0	0
S2	S0	S2	0	1

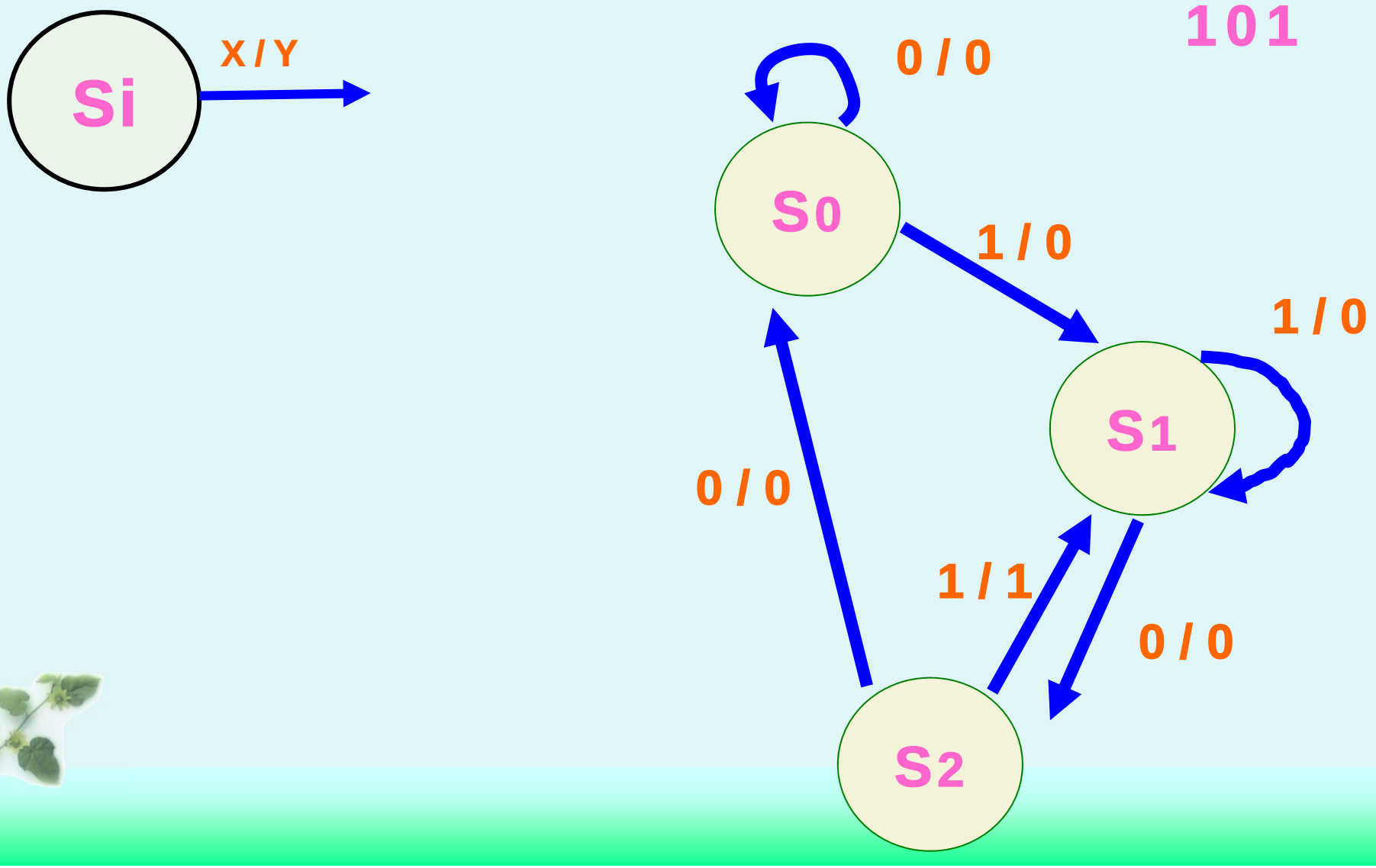
Moore型状态机设计方法

（2） 状态转移图：

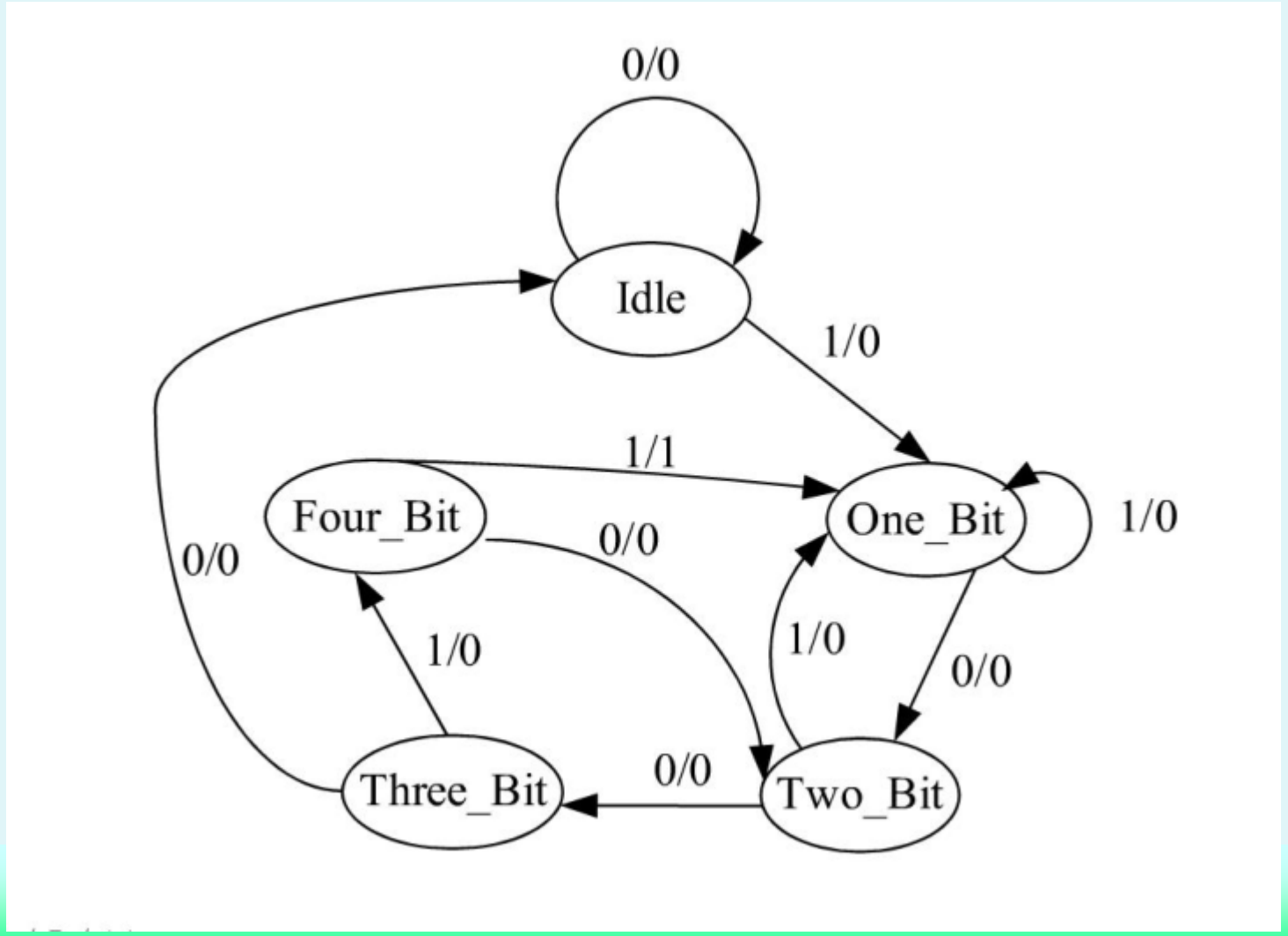


Mealy型状态机设计方法

（2） 状态转移图：



clk	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	…
data_in	1	1	0	0	1	1	1	1	0	0	1	1	0	0	1	1	0	1	…
out	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	1	0	…



Moore型状态机设计方法

例：位于十字路口的交通灯，在A方向和B方向各有红、黄、绿三盏灯，每10秒变换一次。变换顺序如下表：

A方向	B方向
绿	红
黄	红
红	绿
红	黄

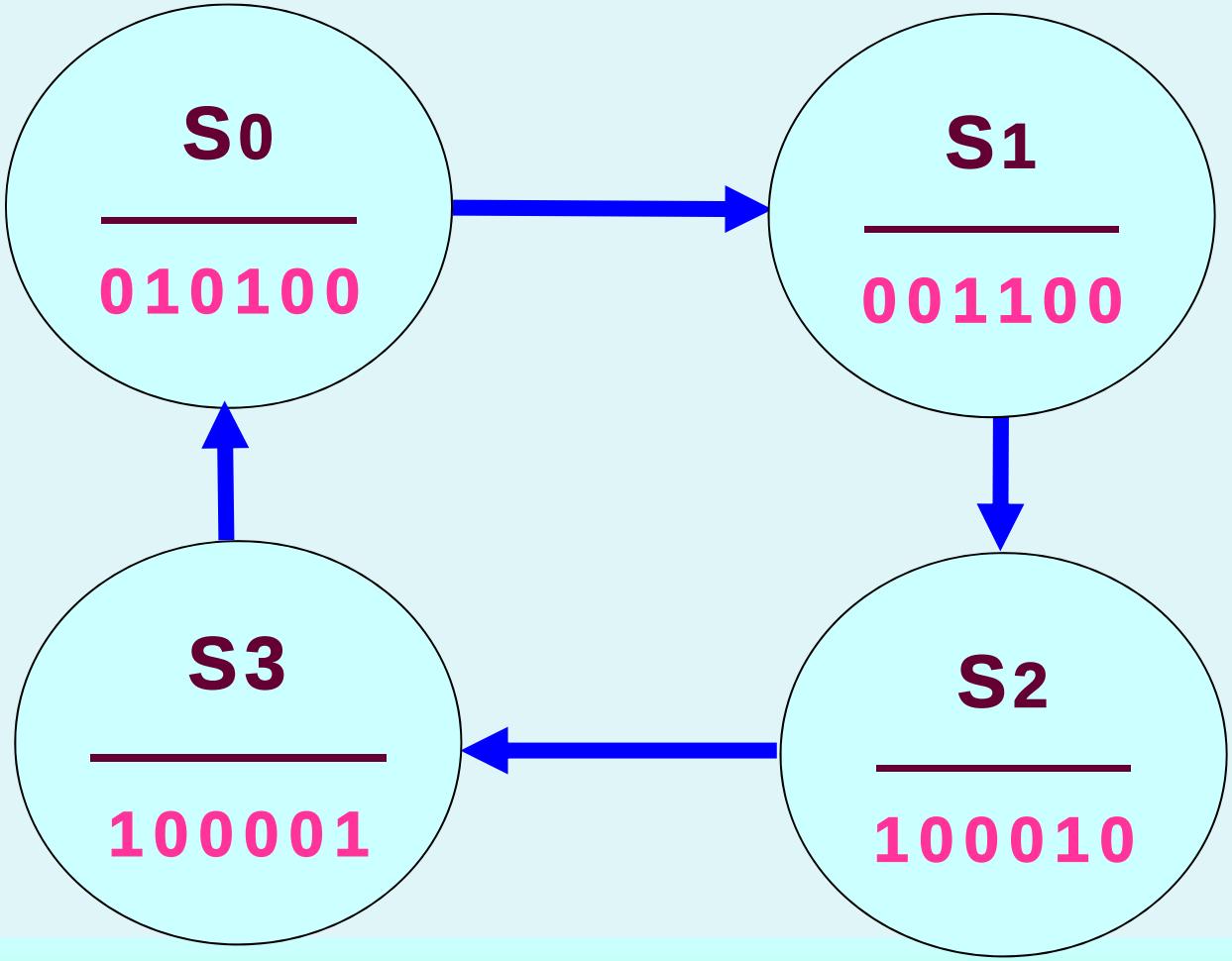
Moore型状态机设计方法

交通灯的全部状态及输出：

状态	A方向（红绿黄）	B方向（红绿黄）
S0	0 1 0	1 0 0
S1	0 0 1	1 0 0
S2	1 0 0	0 1 0
S3	1 0 0	0 0 1

Moore型状态机设计方法

画出状态转移图:

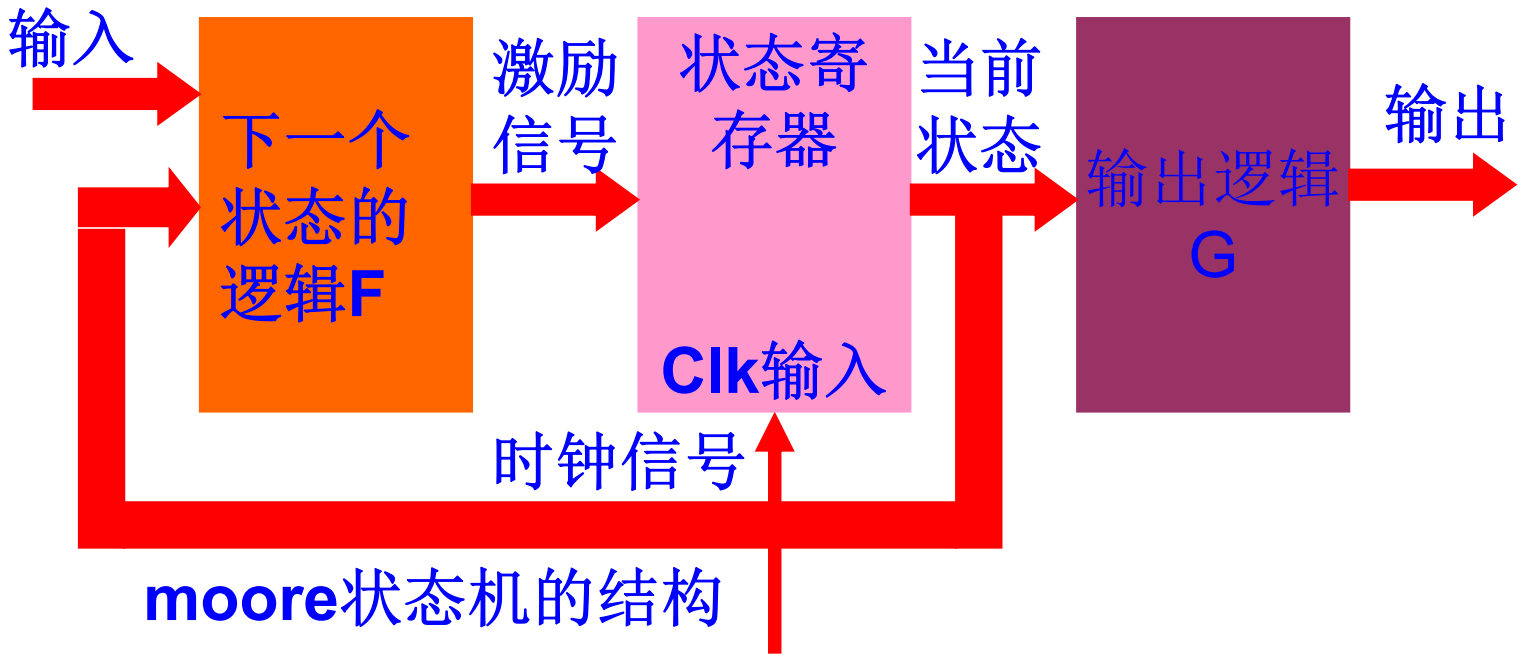


Moory有限状态机建模

Moore有限状态机（FSM)的输出只依赖于状态而不依赖于其输入信号

下一个状态=F（当前状态，输入信号）

输出信号=G（当前状态）

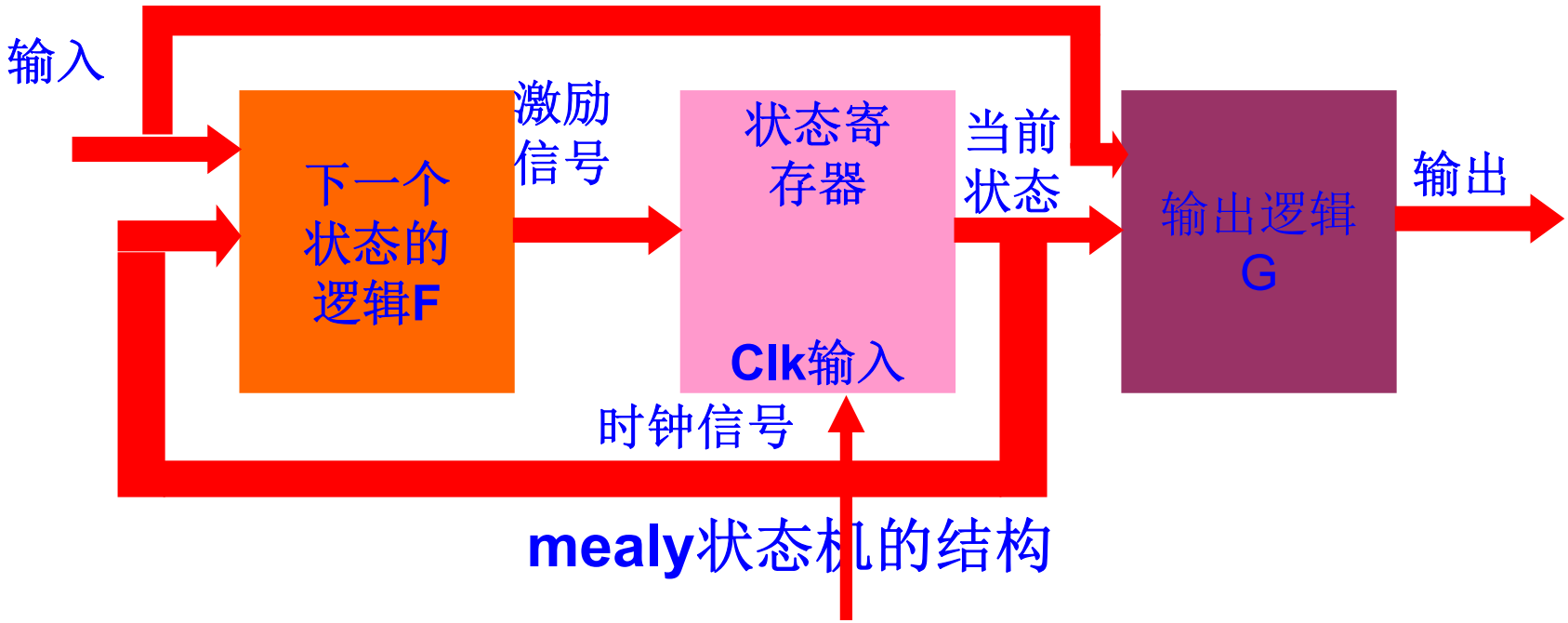


Mealy有限状态机建模

Mealy有限状态机的输出不仅依赖状态而且依赖于它的输入

下一个状态=F（当前状态，输入信号）

输出信号=G（当前状态，输入信号）

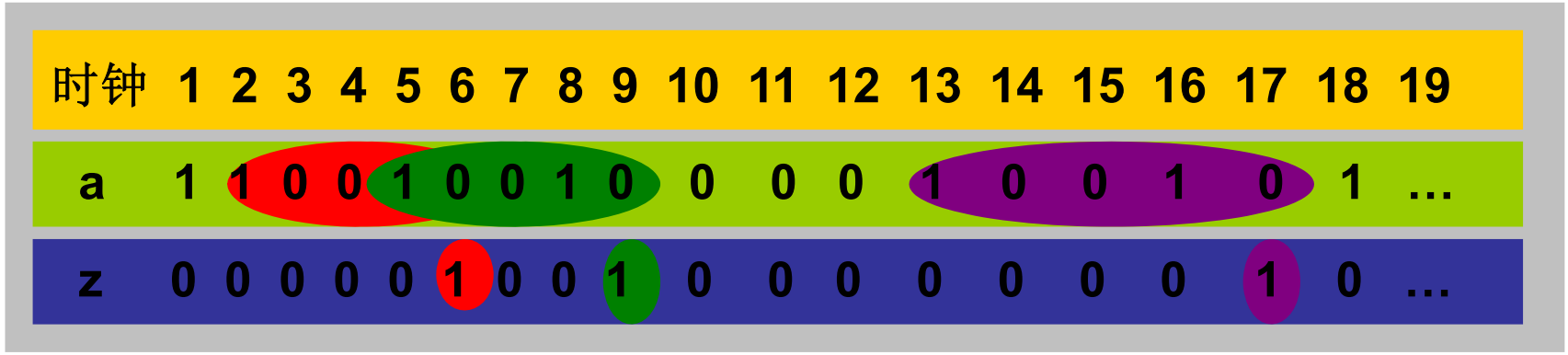


mealy状态机的结构

有限状态机应用举例—序列检测器

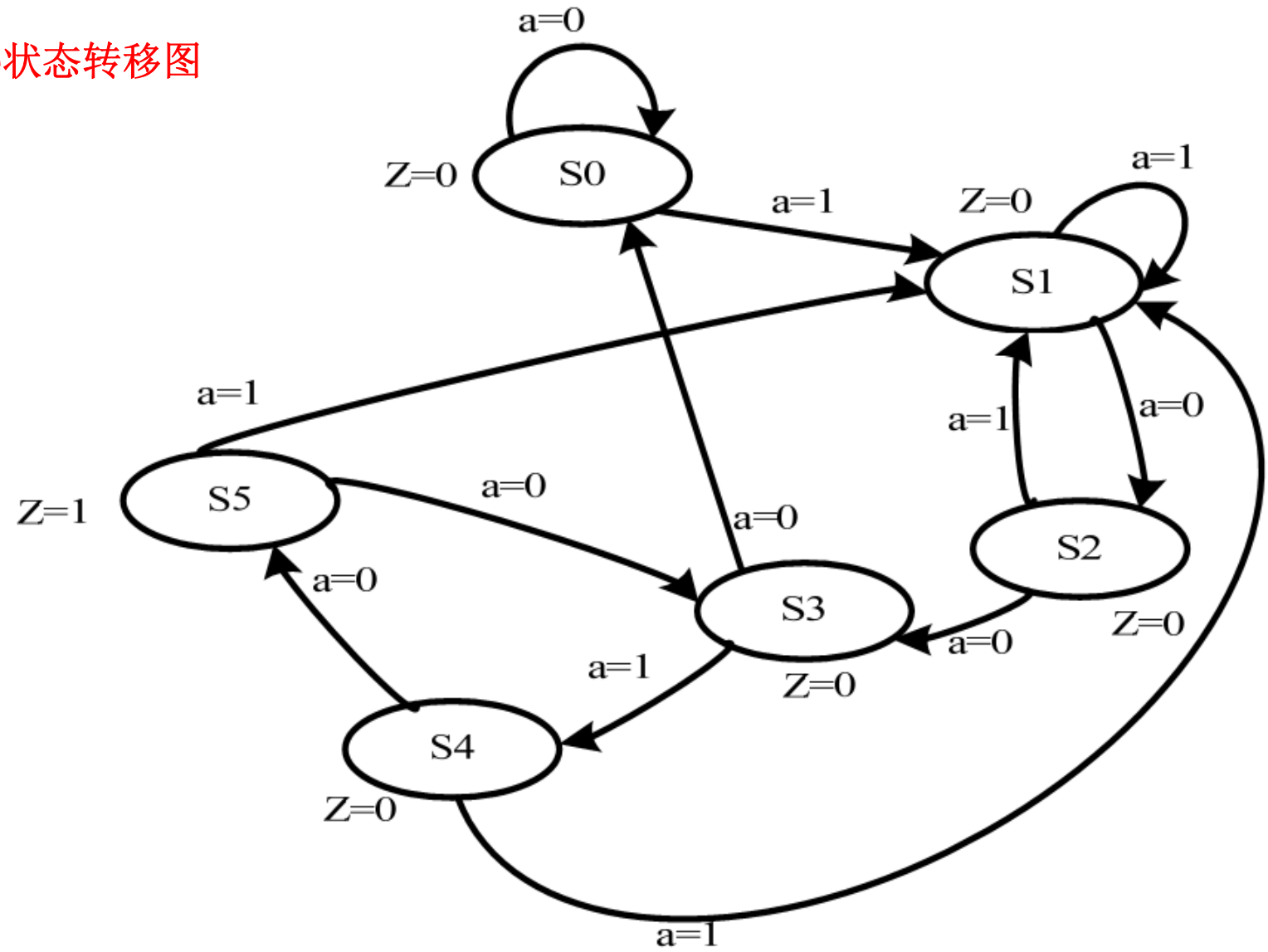
用FSM实现10010串的检测，画出其状态图，并用verilog语言实现

逻辑功能分析



用FSM实现10010串的检测，画出其状态图，并用verilog语言实现

Moore状态转移图



有限状态机应用举例—序列检测器

Moore型verilog源代码

```
module moorefsm(clk,rst,a,z);
    input   clk,rst;
    input   a;
    output  z;
    reg     z;
    reg [3:0]  currentstate,nextstate;
    parameter S0 = 4'b0000;
    parameter S1 = 4'b0001;
    parameter S2 = 4'b0010;
    parameter S3 = 4'b0011;
    parameter S4 = 4'b0100;
    parameter S5 = 4'b0101;
```

Moore型verilog源代码

```
//*****************************************
  always@(posedge clk or negedge rst) //
     begin
        if(!rst)
           currentstate <= S0;
        else
           currentstate <= nextstate;
      end
  //*****************************************
```

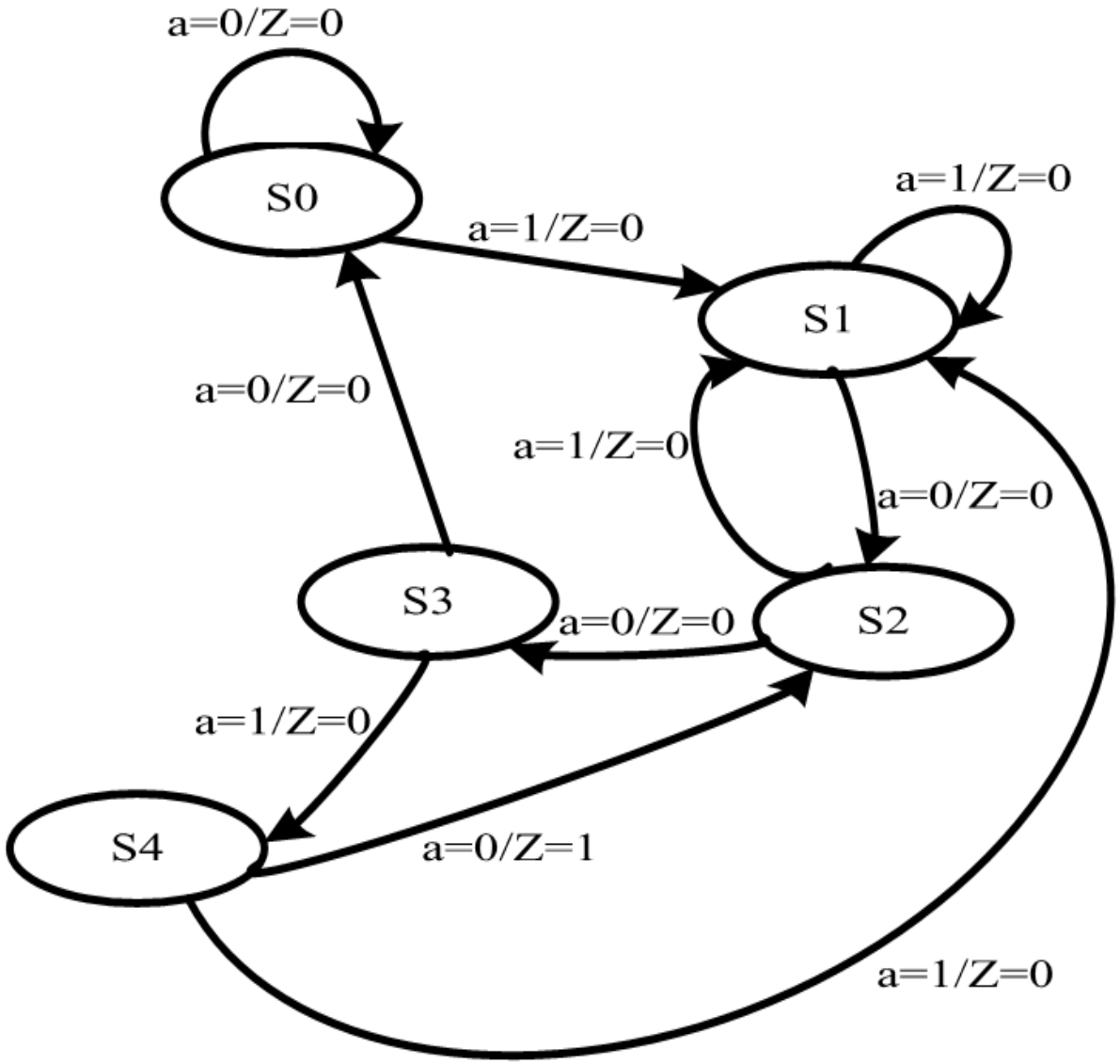
Moore型verilog源代码

```
always@(currentstate or a or rst)   //下一状态
   begin
      if(!rst)
        nextstate = S0;
      else
         case(currentstate)
            S0: nextstate = (a==1)?S1:S0;
            S1: nextstate = (a==0)?S2:S1;
            S2: nextstate = (a==0)?S3:S1;
            S3: nextstate = (a==1)?S4:S0;
            S4: nextstate = (a==0)?S5:S1;
            S5: nextstate = (a==0)?S3:S1;
            default: nextstate = S0;
          endcase
    end
```

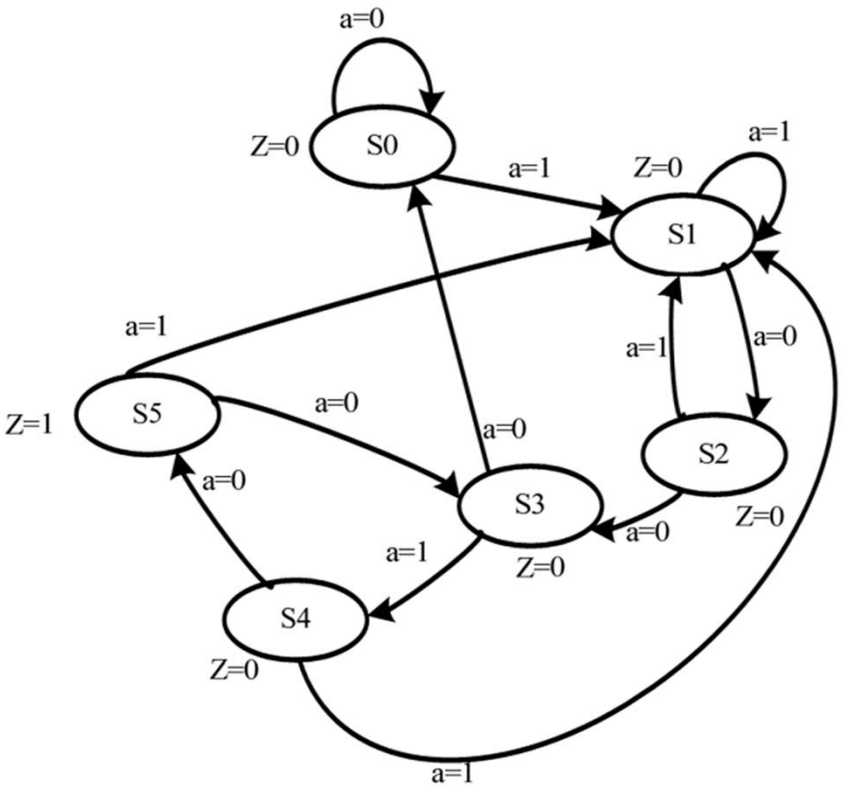
Moore型verilog源代码

```
always@(rst or currentstate)   // 输出
  begin
     if(!rst)
              z = 0;
     else
       case(currentstate)
          S0: z = 0;
          S1: z = 0;
          S2: z = 0;
          S3: z = 0;
          S4: z = 0;
          S5: z = 1;
      default: z = 0;
       endcase
   end
endmodule
```

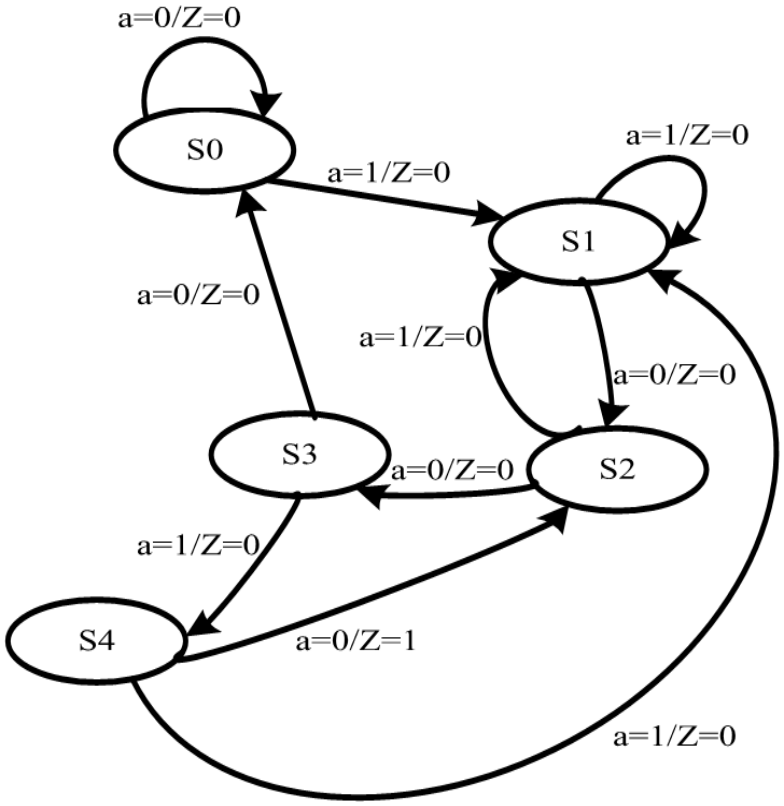
Mealy型FSM实现10010串的检测，画出其状态图，并用verilog语言实现



Moore状态转移图



Mealy状态转移图



Mealy型verilog源代码

```
module mealyfsm(clk,rst,a,z);
    input     clk;
    input     rst;
    input     a;
    output    z;
    reg       z;
    reg [3:0] temp_z;
    reg [3:0]  currentstate,nextstate;
    parameter S0 = 4'b0000;
    parameter S1 = 4'b0001;
    parameter S2 = 4'b0010;
    parameter S3 = 4'b0011;
    parameter S4 = 4'b0100;
```

```
//*****************************************
  always@(posedge clk or negedge rst)
   if(!rst)
        currentstate <= S0;
   else
        currentstate <= nextstate;
  //*****************************************

always@(currentstate or a or rst)
   if(!rst)
       nextstate = S0;
   else
    case(currentstate)
        S0: nextstate = (a == 1)? S1 : S0;
        S1: nextstate = (a == 0)? S2 : S1;
        S2: nextstate = (a == 0)? S3 : S1;
        S3: nextstate = (a == 1)? S4 : S0;
        S4: nextstate = (a == 0)? S2 : S0;
     default:nextstate = S0;
    endcase
```

```
always@(rst or currentstate or a) //输出
    if(!rst)
          temp_z = 0;
    else
       case(currentstate)
            S0: temp_z = 0;
            S1: temp_z = 0;
            S2: temp_z = 0;
            S3: temp_z = 0;
            S4: temp_z = (a == 0)? 1 : 0;
       default:temp_z = 0;
      endcase
```

```
always@(posedge clk or negedge rst)
    if(!rst)
        z <= 0;
    else
      begin
         if((temp_z == 1)&&(nextstate == S2))
             z <= 1;
        else
             z <= 0;
      end
endmodule
```

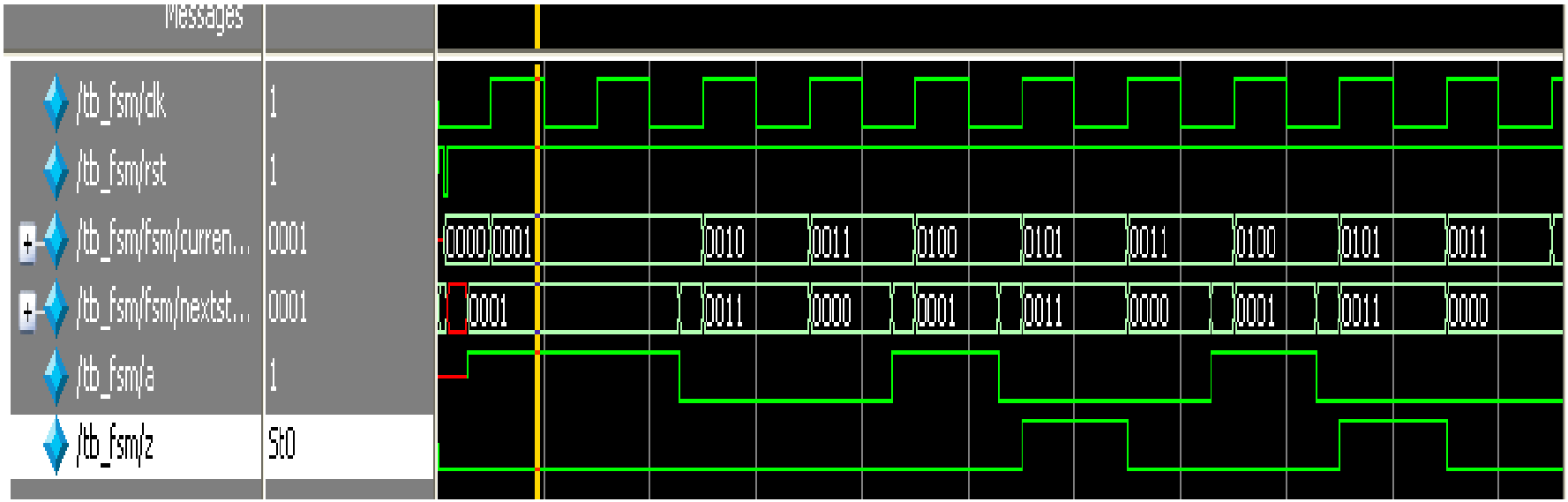
Moore fsm测试模块testbench

```
module tb_fsm;
  reg clk,rst;
  reg  a;
  wire z;
  moorefsm
    fsm(.clk(clk),.rst(rst),.a(a),.z(z));
  initial
    begin
      clk = 0;
      rst = 1;
      #5 rst = 0;
      #3 rst = 1;
      #20 a = 1;
      #100 a = 1;
      #100 a = 0;
      #100 a = 0;
      #100 a = 1;
      #100 a = 0;
      #100 a = 0;
      #100 a = 0;
      #100 a = 0;
      #100 a = 0;
      #100 a = 1;
      #100 a = 0;
      #100 a = 0;
      #100 a = 0;
      #100 a = 1;
      #100 a = 0;
      #100 a = 1;
      #100 a = 0;
    end
  always #50 clk = ~clk;
endmodule
```

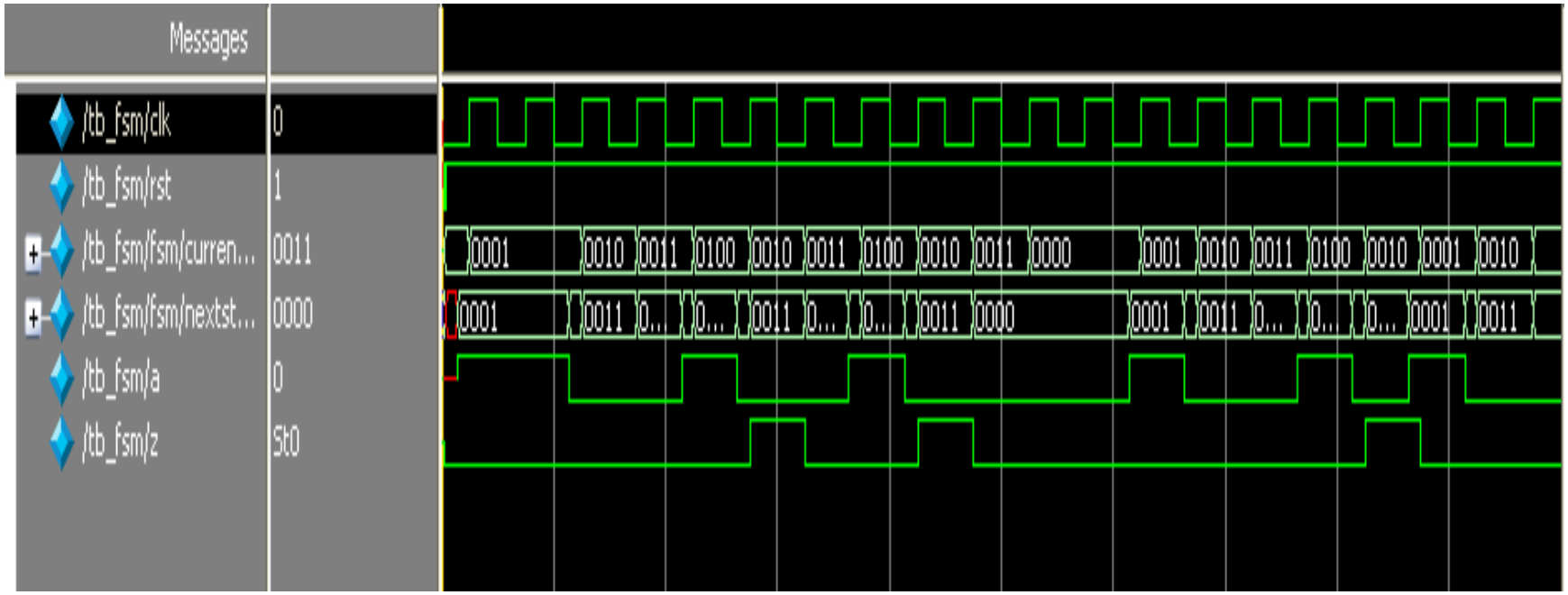
mealyfsm测试模块testbench

```
module tb_fsm;
  reg clk,rst;
  reg  a;
  wire z;
  mealyfsm
    fsm(.clk(clk),.rst(rst),.a(a),.z(z));
  initial
    begin
      clk = 0;
      rst = 1;
      #5 rst = 0;
      #3 rst = 1;
      #20 a = 1;
      #100 a = 1;
      #100 a = 0;
      #100 a = 0;
      #100 a = 1;
      #100 a = 0;
      #100 a = 0;
      #100 a = 0;
      #100 a = 0;
      #100 a = 0;
      #100 a = 1;
      #100 a = 0;
      #100 a = 0;
      #100 a = 0;
      #100 a = 1;
      #100 a = 0;
      #100 a = 1;
      #100 a = 0;
    end
  always #50 clk = ~clk;
endmodule
```

Moorefsm的Modelsim仿真波形图



Mealyfsm的Modelsim仿真波形图



FSM可综合编写风格

1. VerilogHDL状态机的状态分配

VerilogHDL描述状态机必须由parameter分配好状态。

2. 组合逻辑和时序逻辑要分开用不同的过程块

组合逻辑包括状态译码和输出，时序逻辑则是状态寄存器的切换；必须包括对所有状态都处理，不能出现无法处理的状态，使状态机失控。

3. Mealy机的输出与输入有关，而Moore机的输出与输入无关。

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

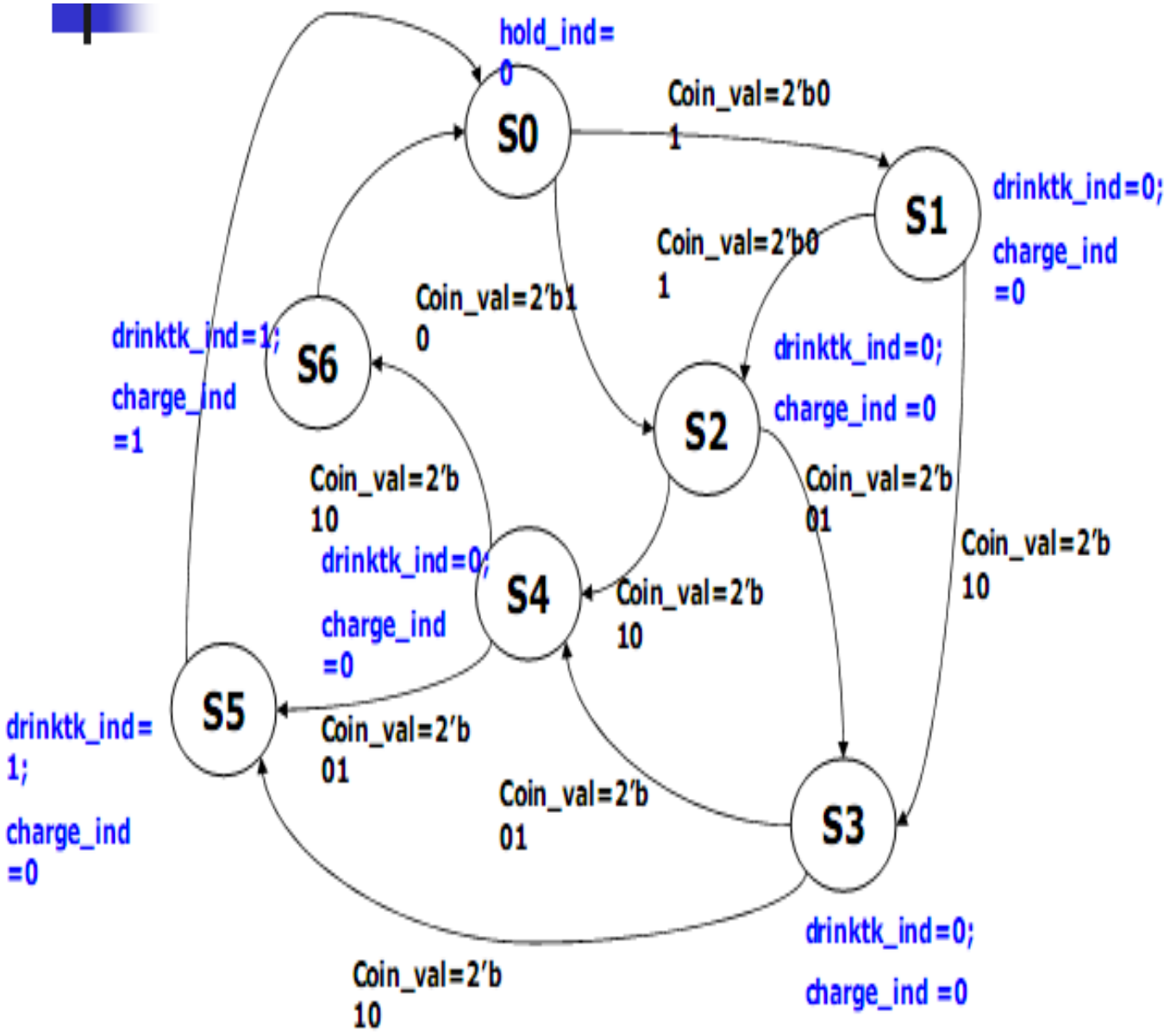
S0：初始态；　　**S1**：已投币**0.5**元

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；

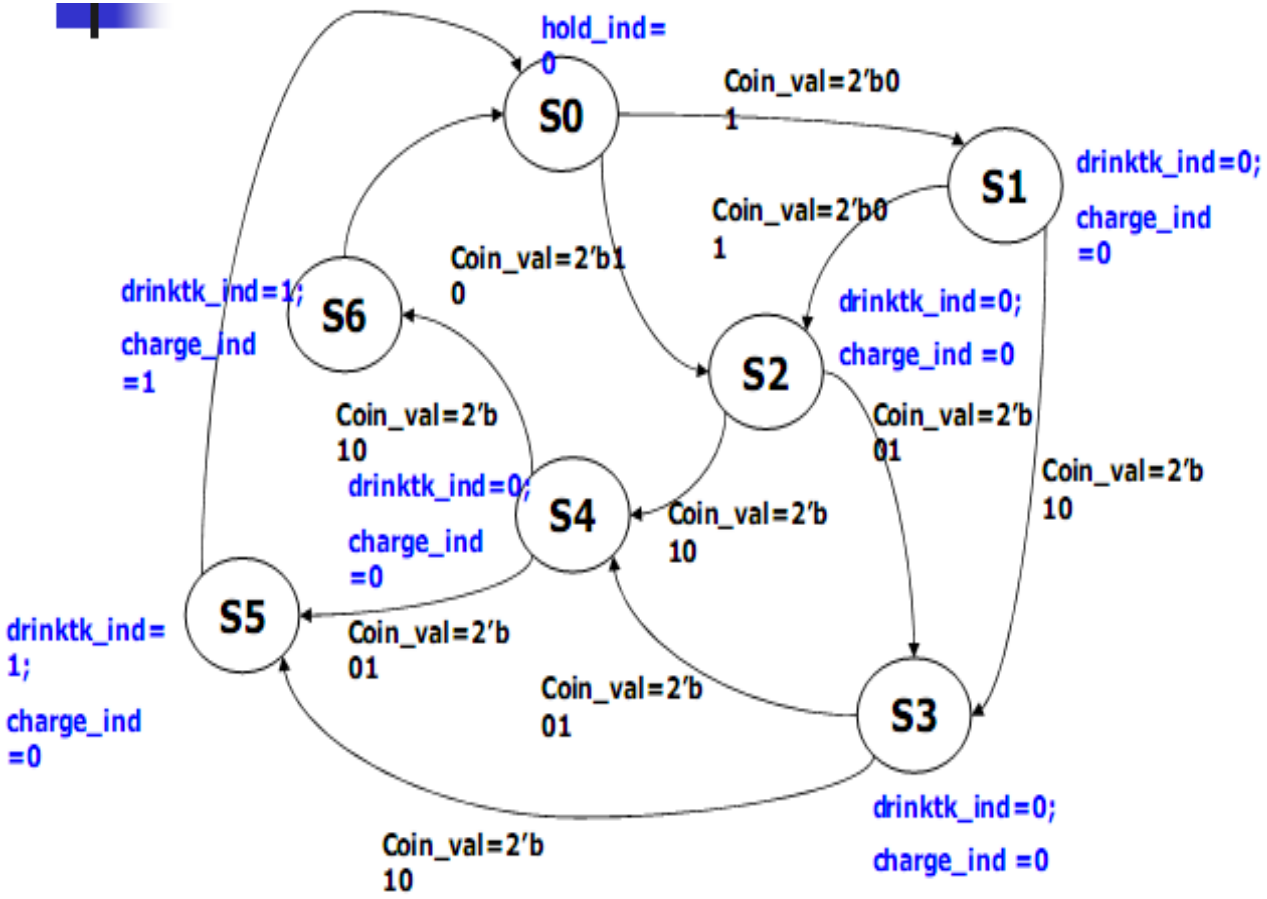
S3 ： 已投币**1.5**元；

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;
```

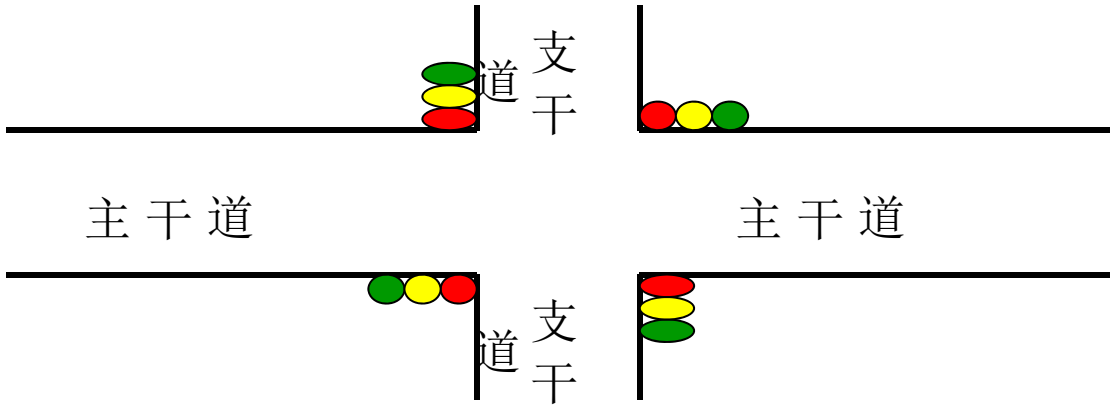
```
//第二次：依次投入0.5元、一元、一元的硬币，看结果又如何？
#2000 op_start=1; coin_val=2'b01;
#1000 coin_val=2'b10;
#1000 coin_val=2'b10;
#1000 op_start=0;
//第三次：依次投入0.5元、一元的硬币，然后取消操作，看结果
如何？
#2000 op_start=1;
      coin_val=2'b01;
#1000 coin_val=2'b10;
#1000 cancel_flag=1'b1;
      op_start=0;
#1000 cancel_flag=0;
```

```
//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



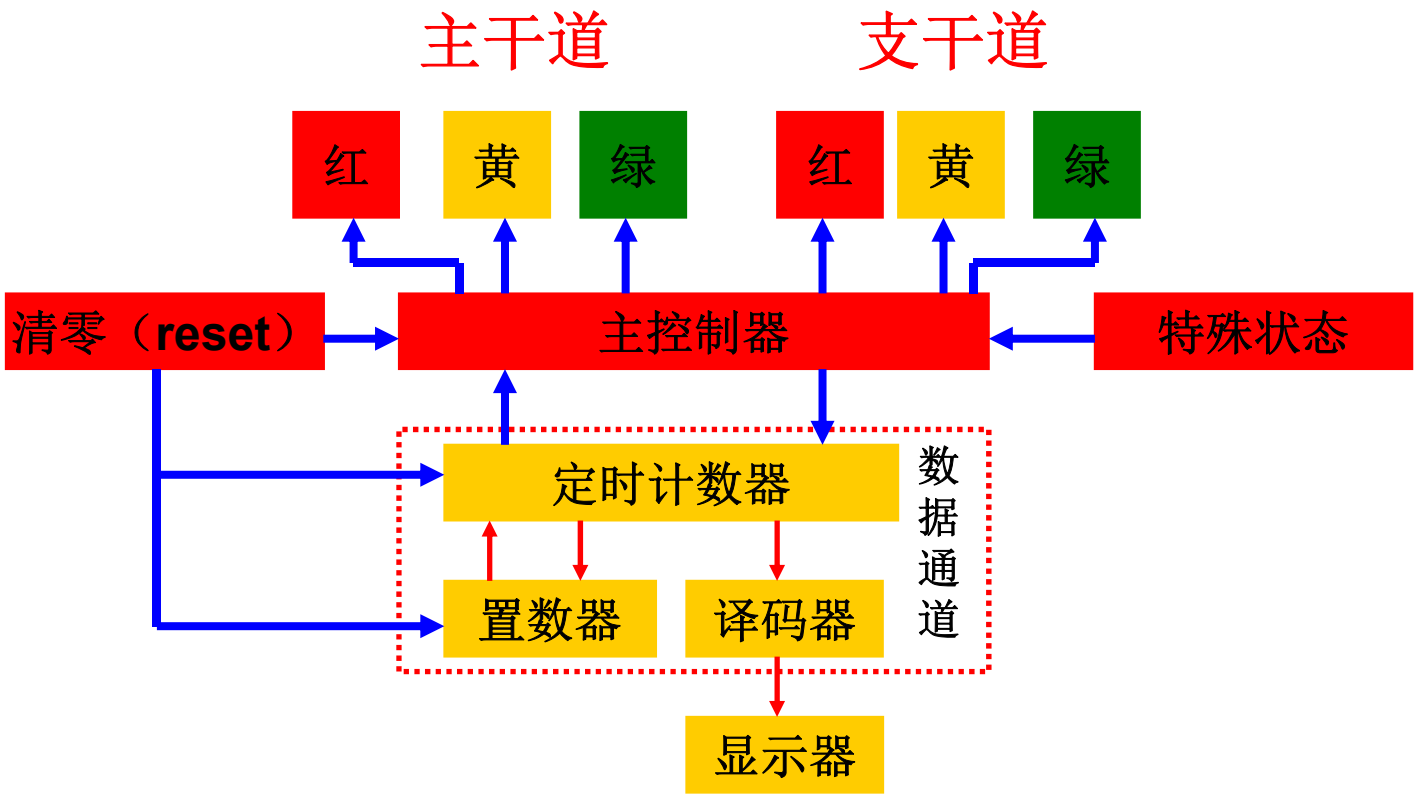
设计要求

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

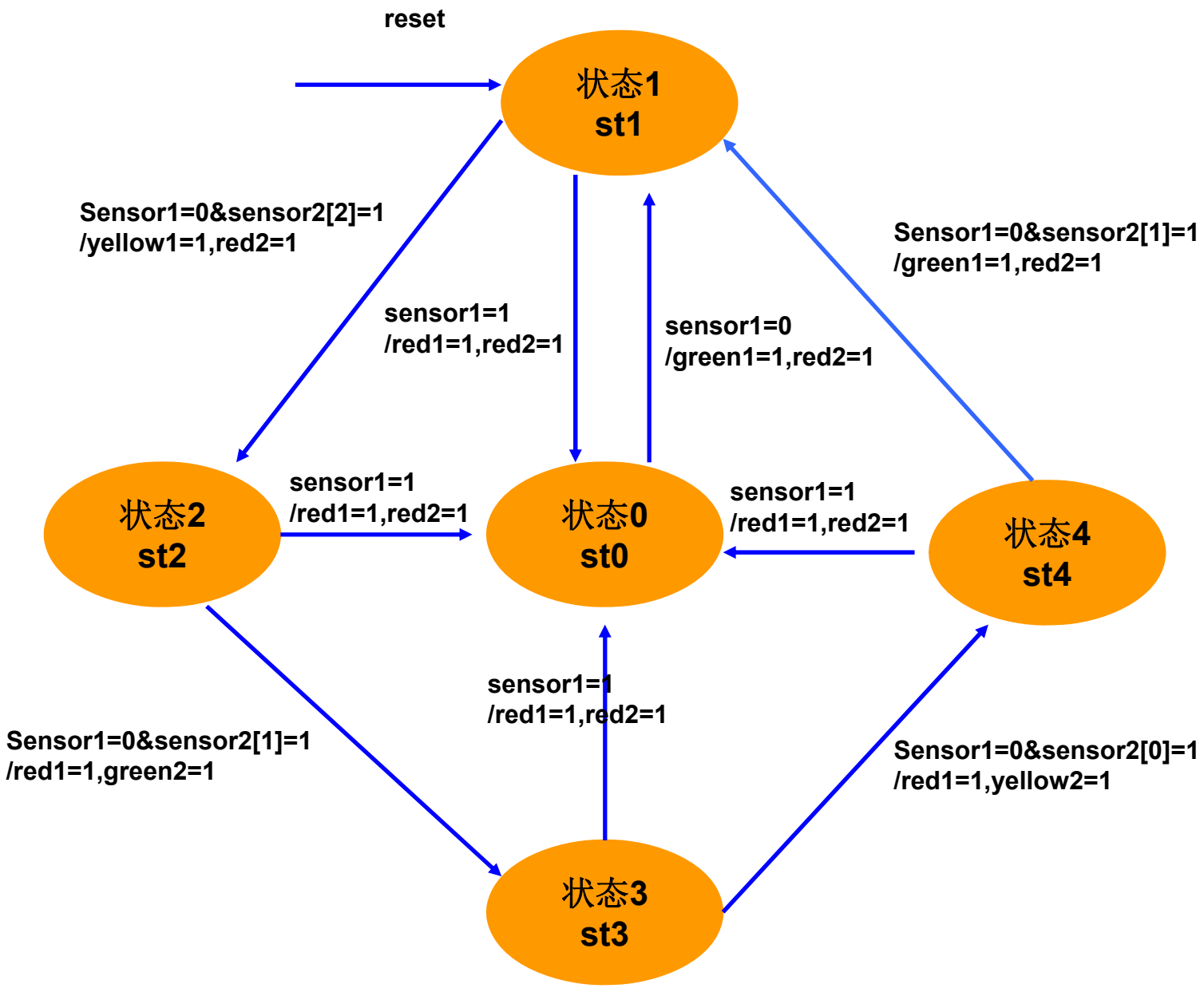
输出信号

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**35s,25s,5s**计时）

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

有限状态机的Verilog描述

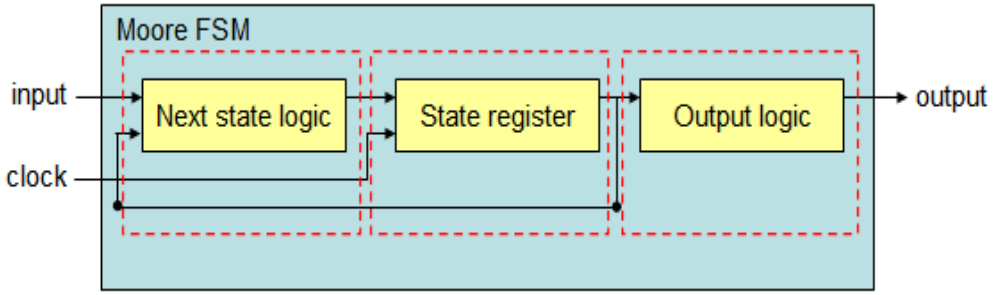
（1）用三个过程描述：即现态（CS）、次态（NS）、输出逻辑（OL）各用一个always过程描述。

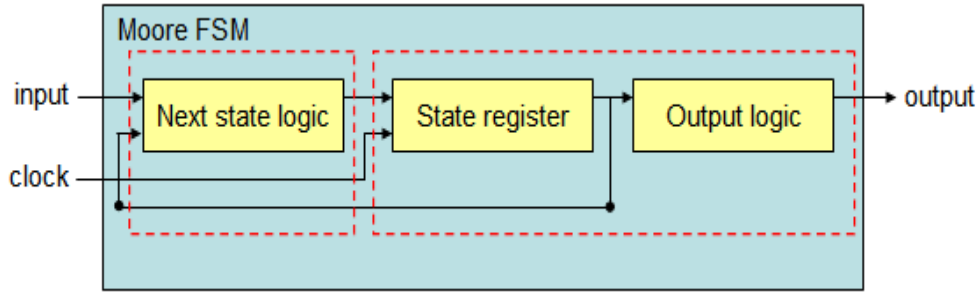
（2）双过程描述（CS+NS、OL双过程描述）：使用两个always过程来描述有限状态机，一个过程描述现态和次态时序逻辑（CS+NS）；另一个过程描述输出逻辑（OL）。

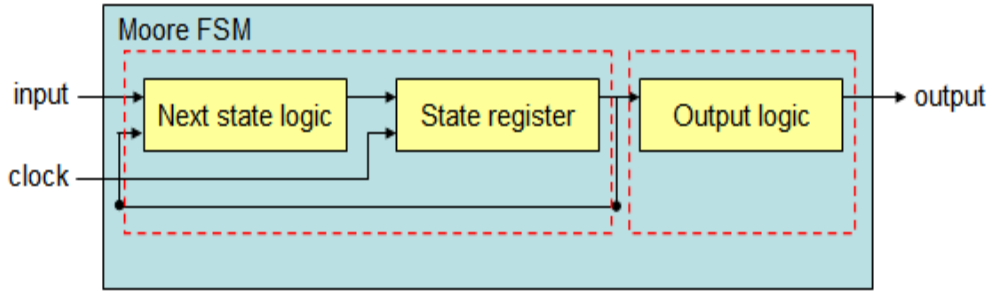
（3）双过程描述（CS、NS+OL双过程描述）：一个过程用来描述现态（CS）；另一个过程描述次态和输出逻辑（NS+OL）。

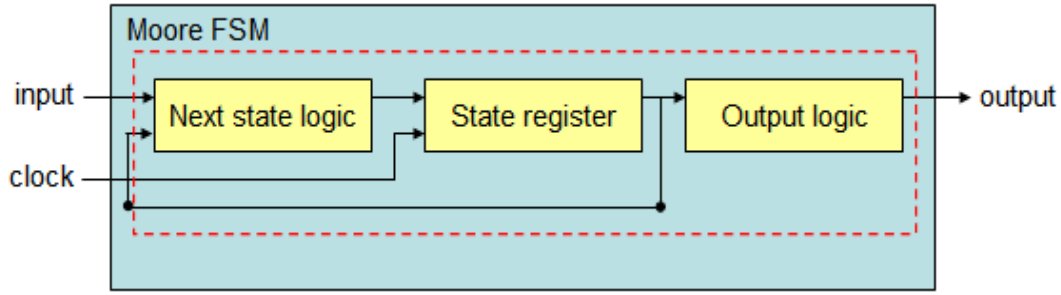
（4）单过程描述：在单过程描述方式中，将状态机的现态、次态和输出逻辑（CS+NS+OL）放在一个always过程中进行描述。

Verilog描述（三个过程）


Moore FSM
input
clock
Next state logic
State register
Output logic
output

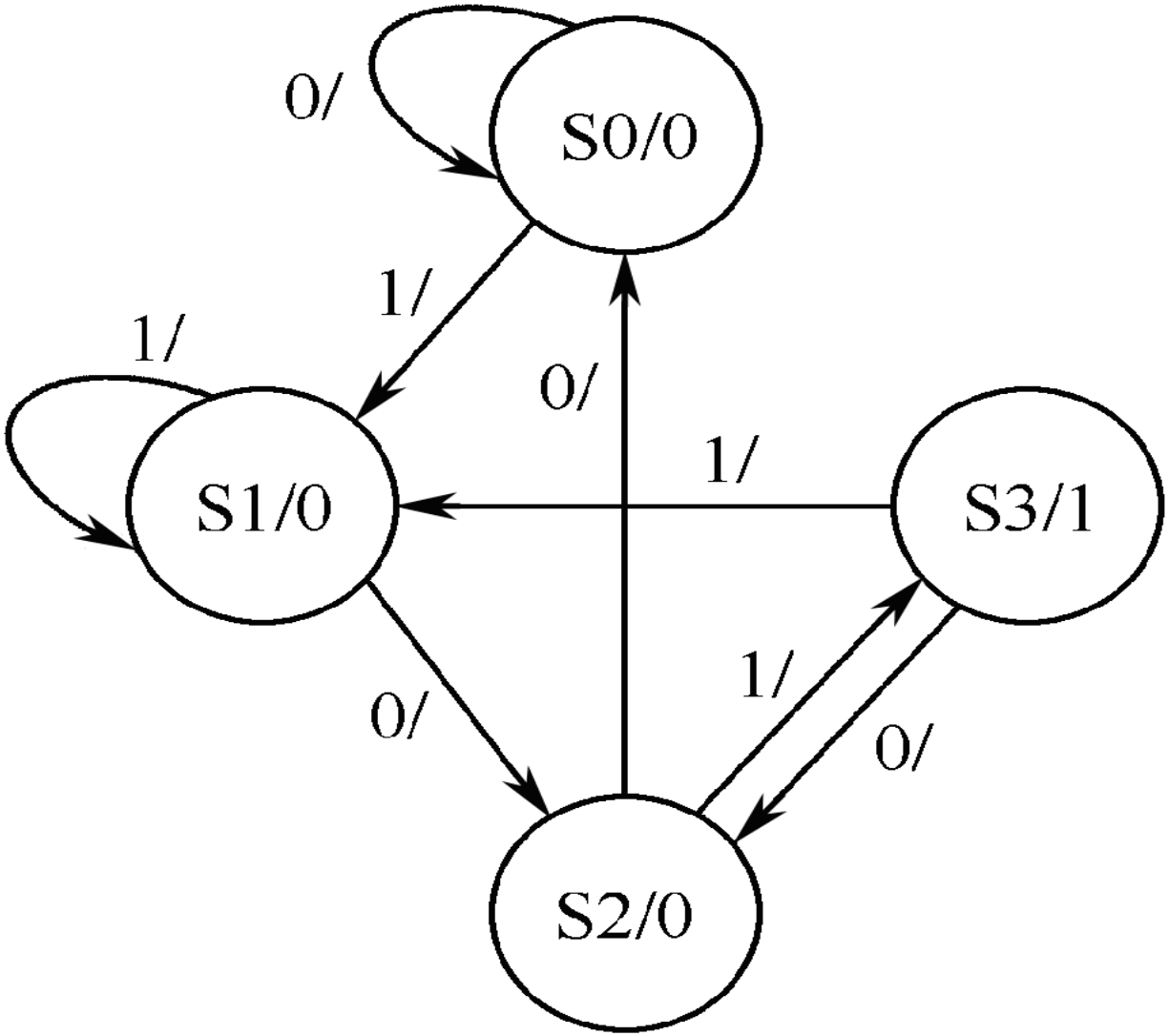

Moore FSM
input
clock
Next state logic
State register
Output logic
output


Moore FSM
input
clock
Next state logic
State register
Output logic
output


Moore FSM
input
clock
Next state logic
State register
Output logic
output

“101”序列检测器的Verilog描述（三个过程）

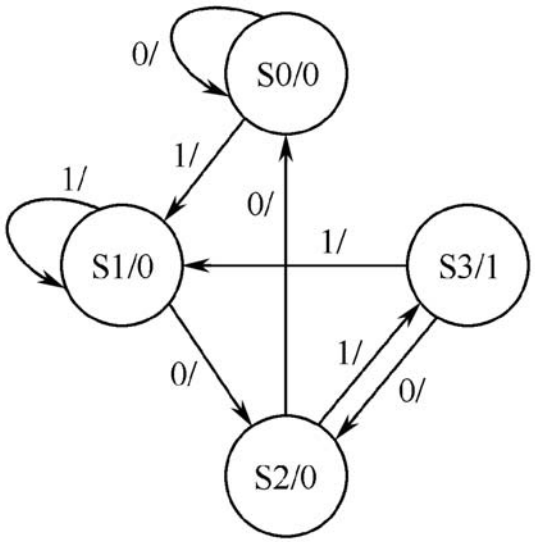
moore



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）

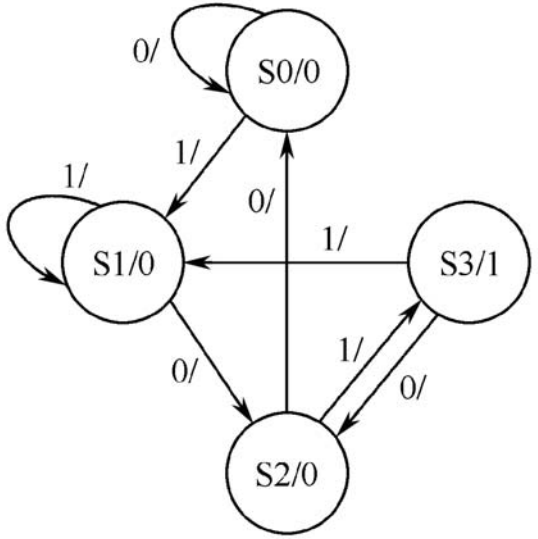
```
module fsm1_seq101(clk,clr,x,z);
input clk,clr,x;
output reg z;
reg[1:0] state,next_state;
parameter S0=2'b00,S1=2'b01,S2=2'b11,S3=2'b10;
                    /*状态编码，采用格雷（Gray）编码方式*/
always @(posedge clk or posedge clr) /*该过程定义当前状态*/
  begin if(clr) state<=S0;          //异步复位，s0为起始状态
          else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



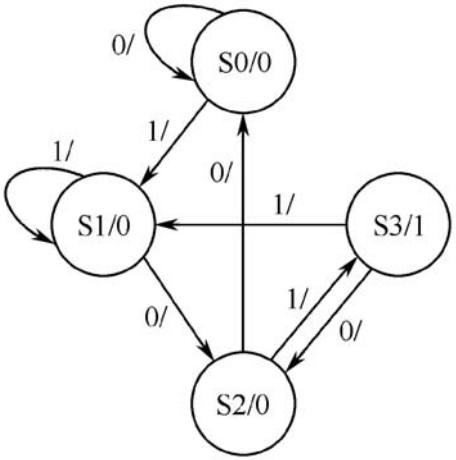
“101”序列检测器的Verilog描述（三个过程）

```
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
```

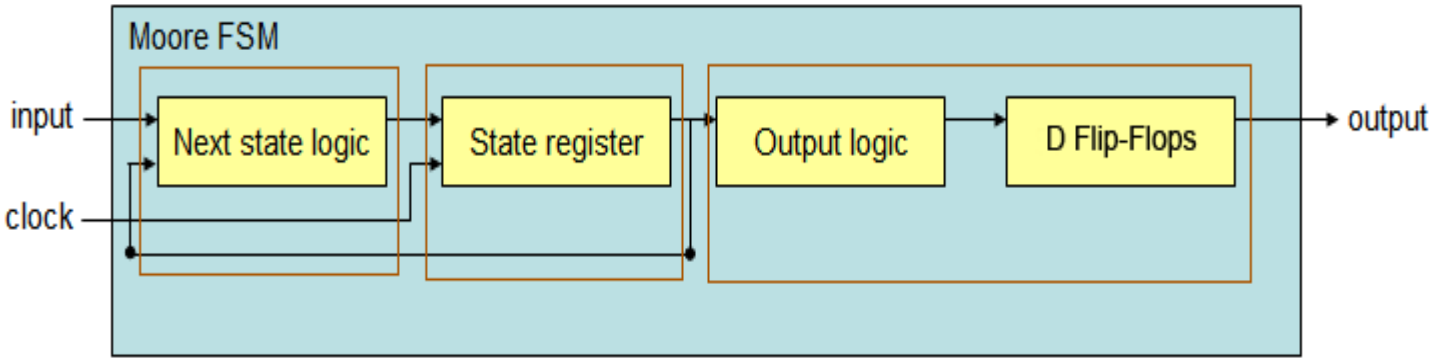


“101”序列检测器（单过程描述）




```
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
```

使用Mealy FSM少Moore FSM 1个状态，且输出早Moore FSM 1个周期，所以最后特别将输出增加一級延迟一个时钟周期，这样Mealy FSM就完全与Moore FSM一样。



在时序电路设计中Mealy型和Moore型电路选择原则是：当要求输出对输入改变快速做出响应及希望电路尽量简单时，选择Mealy型电路；而当要求输出序列稳定，能接受输出序列晚一个时钟周期，及选择Moore型电路并不增加电路复杂性时，适宜选择Moore型电路。