

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。

② 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。

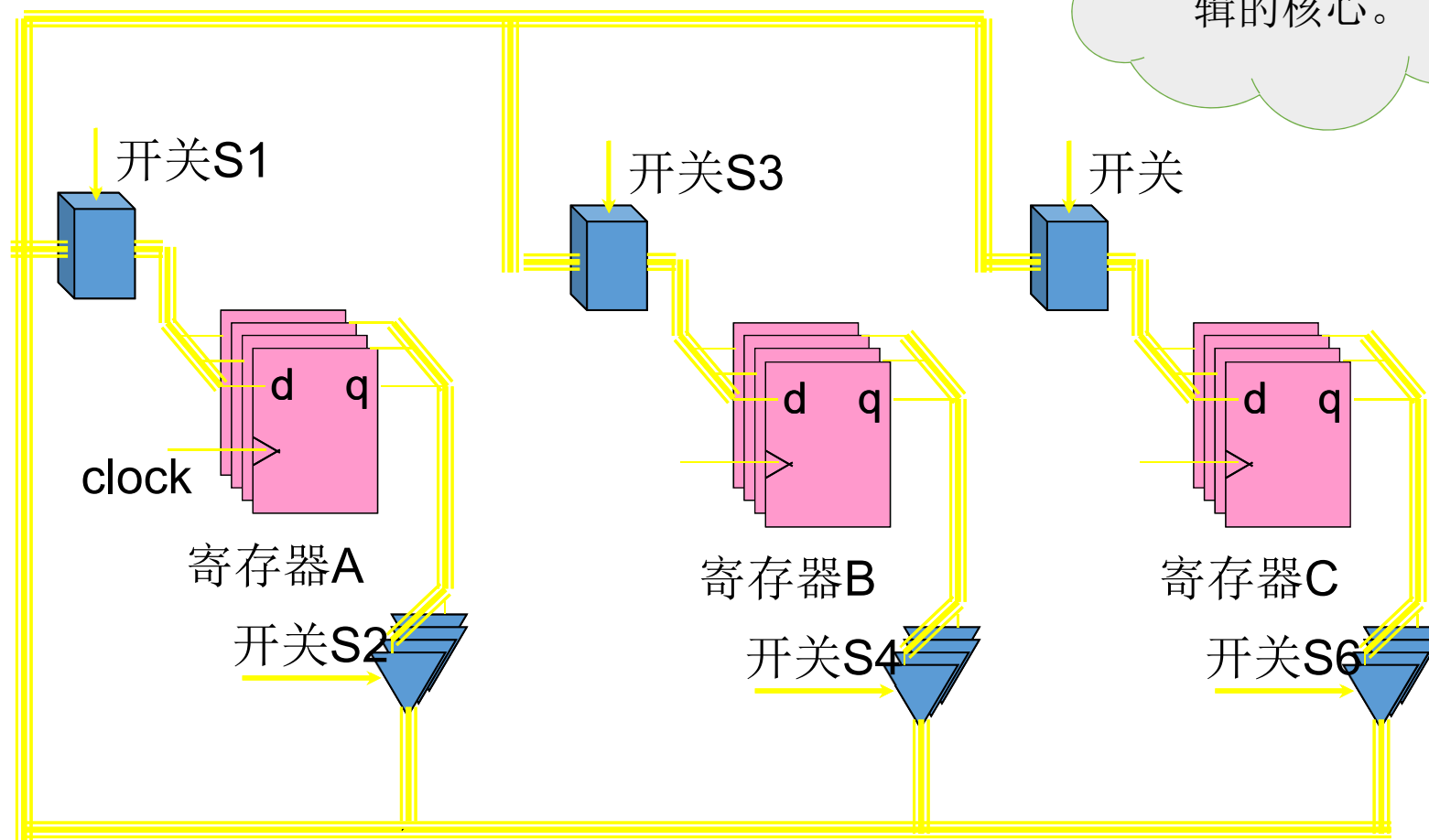
- 有限状态机适用于可编程逻辑器件。通过恰当的Verilog语言描述和EDA工具综合，可以生成性能优越的有限状态机，在执行时间、运行速度和占用资源等方面优于CPU实现的设计方案。

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。

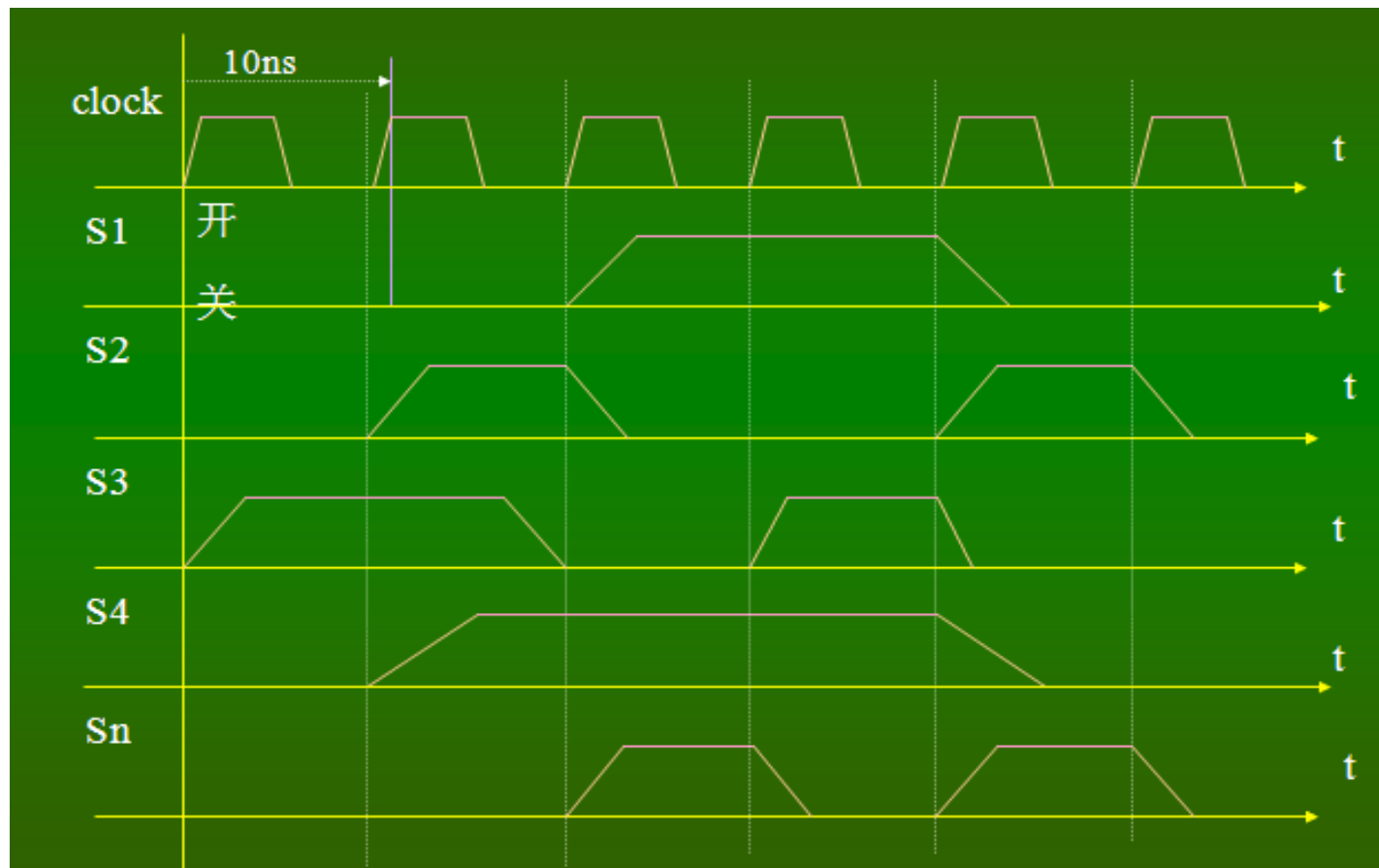
■ 数据流动控制

生成与时钟精确配合的开关时序是计算逻辑的核心。



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，安排好时序，来操作逻辑系统中每一个开关 s_i ，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。

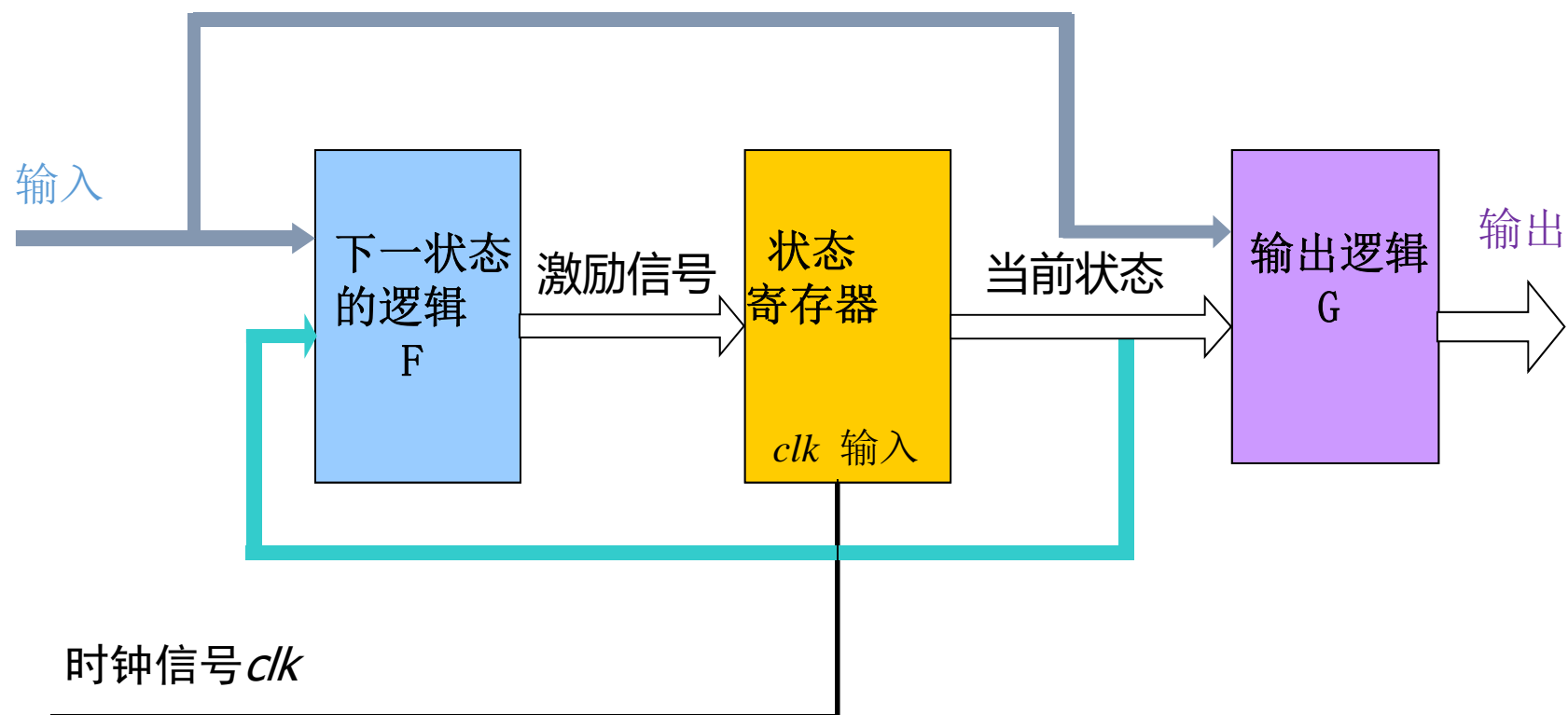
- 有了以上电路，我们就不难设计出复杂的控制序列来操纵数字系统的控制开关阵列。能达到要求的电路就是时序和组合电路互相结合的产物：**同步有限状态机**和由状态和时钟共同控制的**开关逻辑阵列**。
- 只要掌握有限状态机的基本设计方法，加上对基本电路的掌握，再加上对数据处理的过程的细致了解，就可以规避由于逻辑器件和布线延迟产生的“竞争冒险”现象所造成的破坏，设计出符合要求的复杂数字逻辑系统。

■ 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。

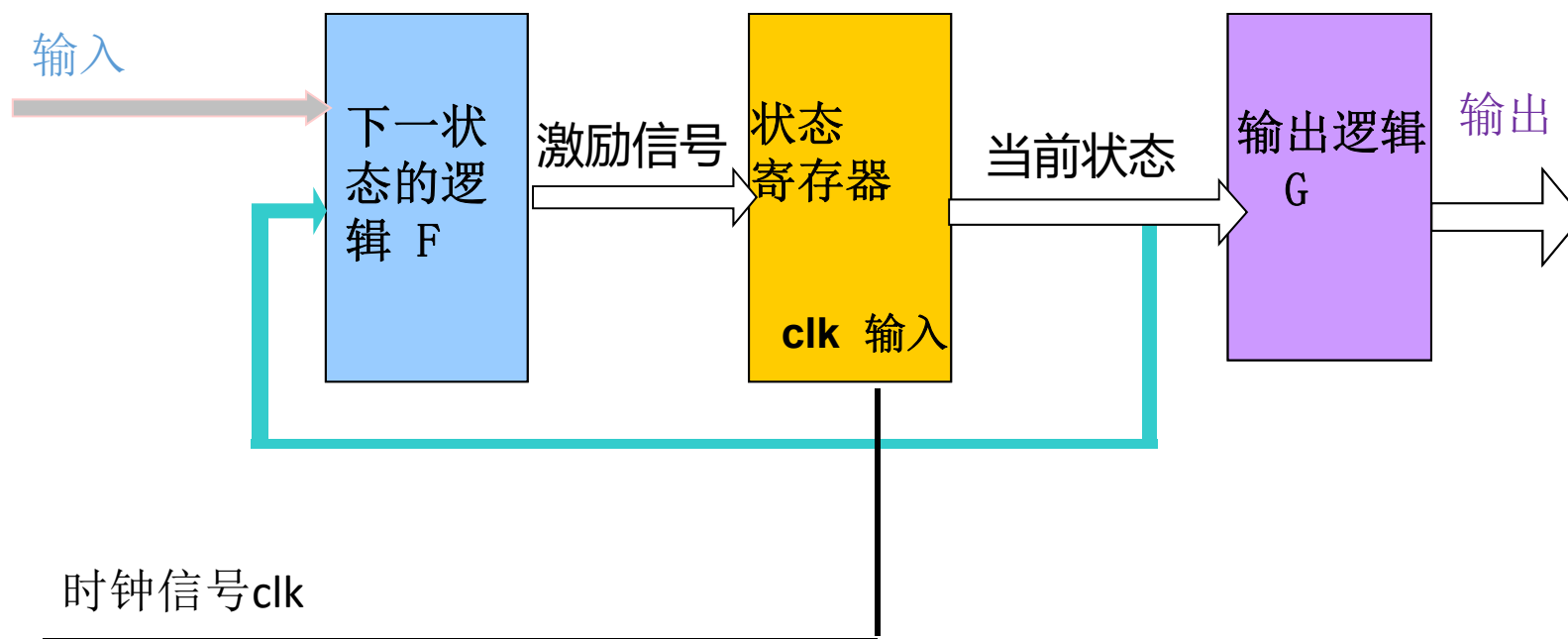
■ Mealy状态机

- 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

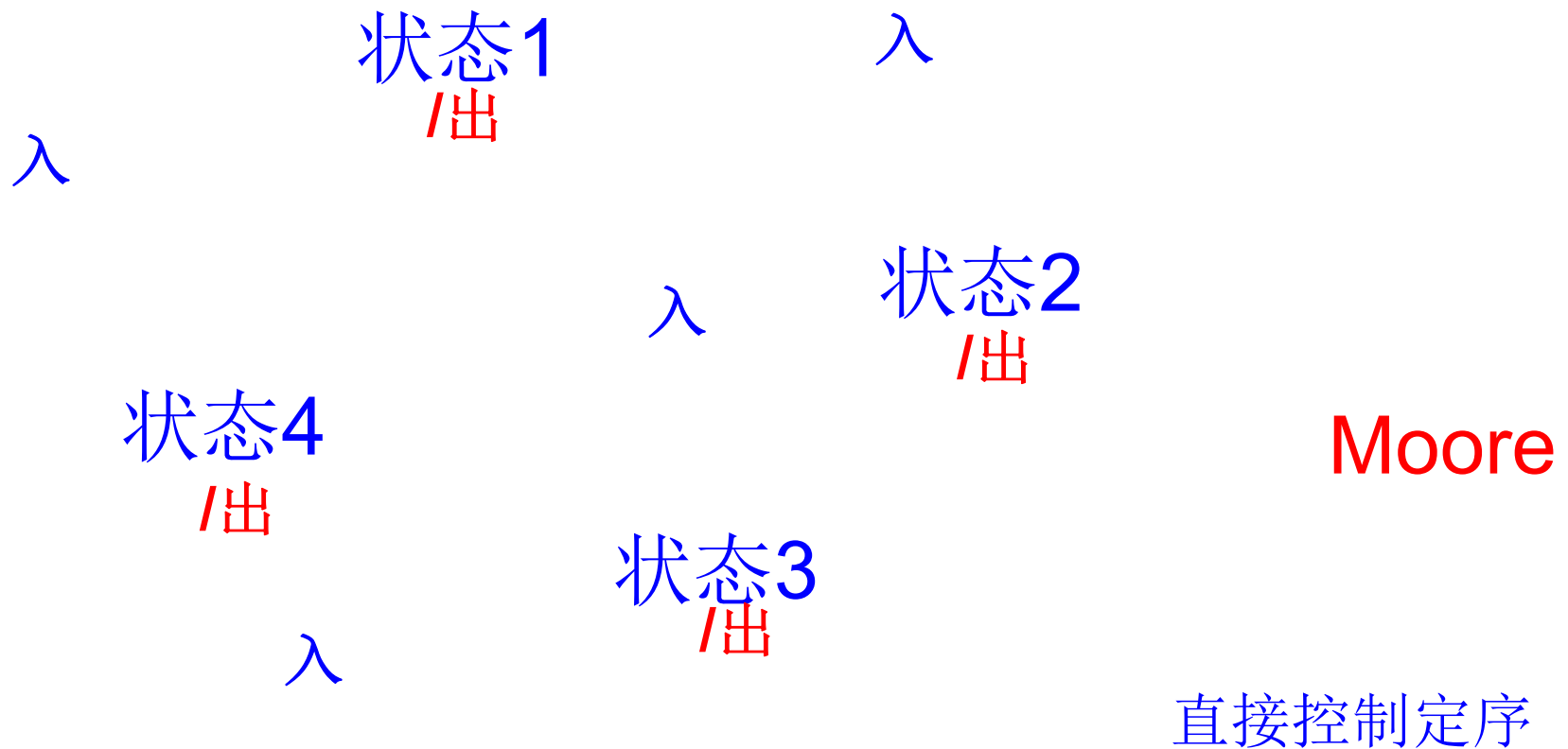
方法与软件程序的流程图类似

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

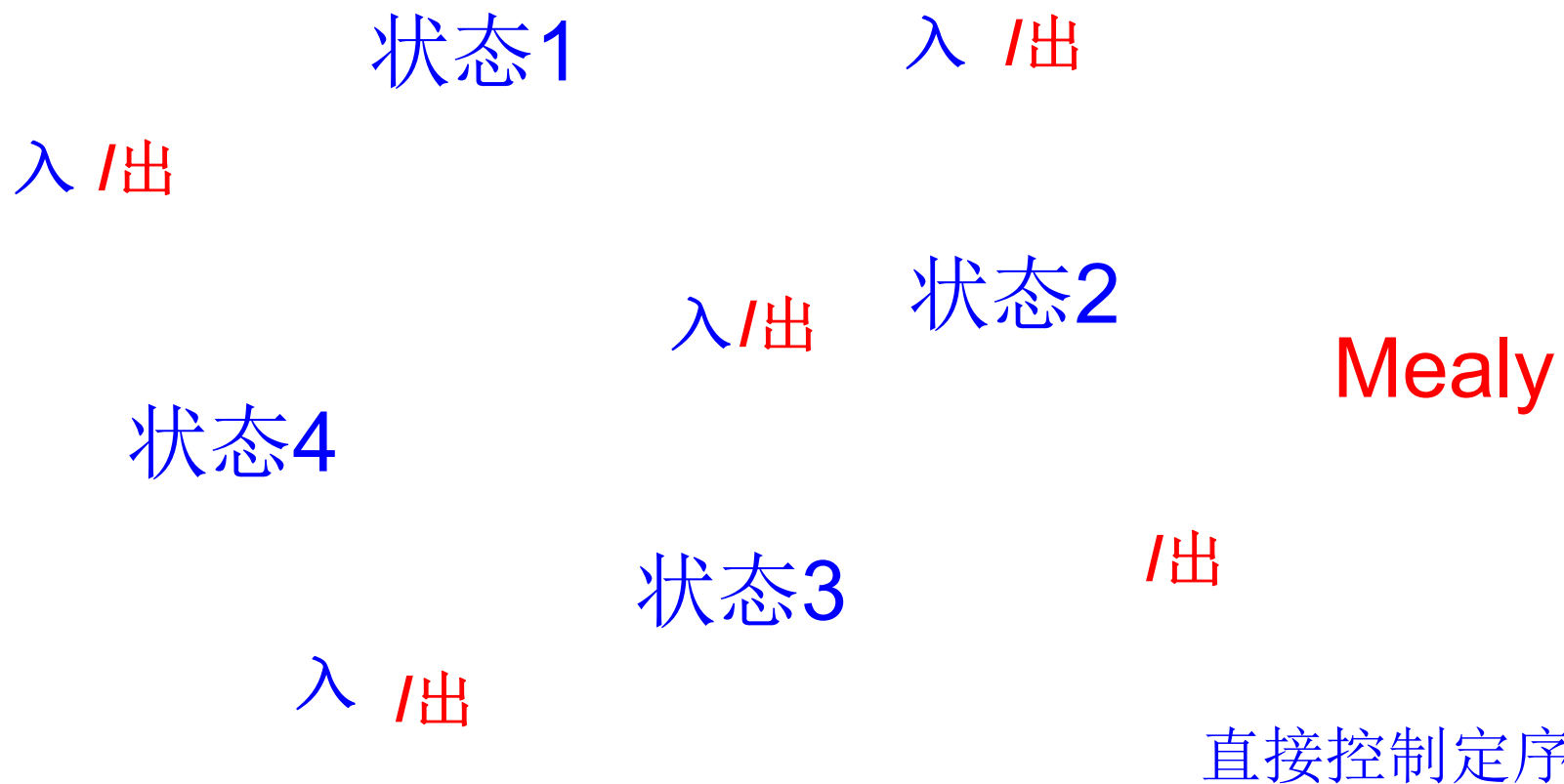
条件控制定序



状态机的表示方法3

●方法三：状态转换图

条件控制定序



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）。

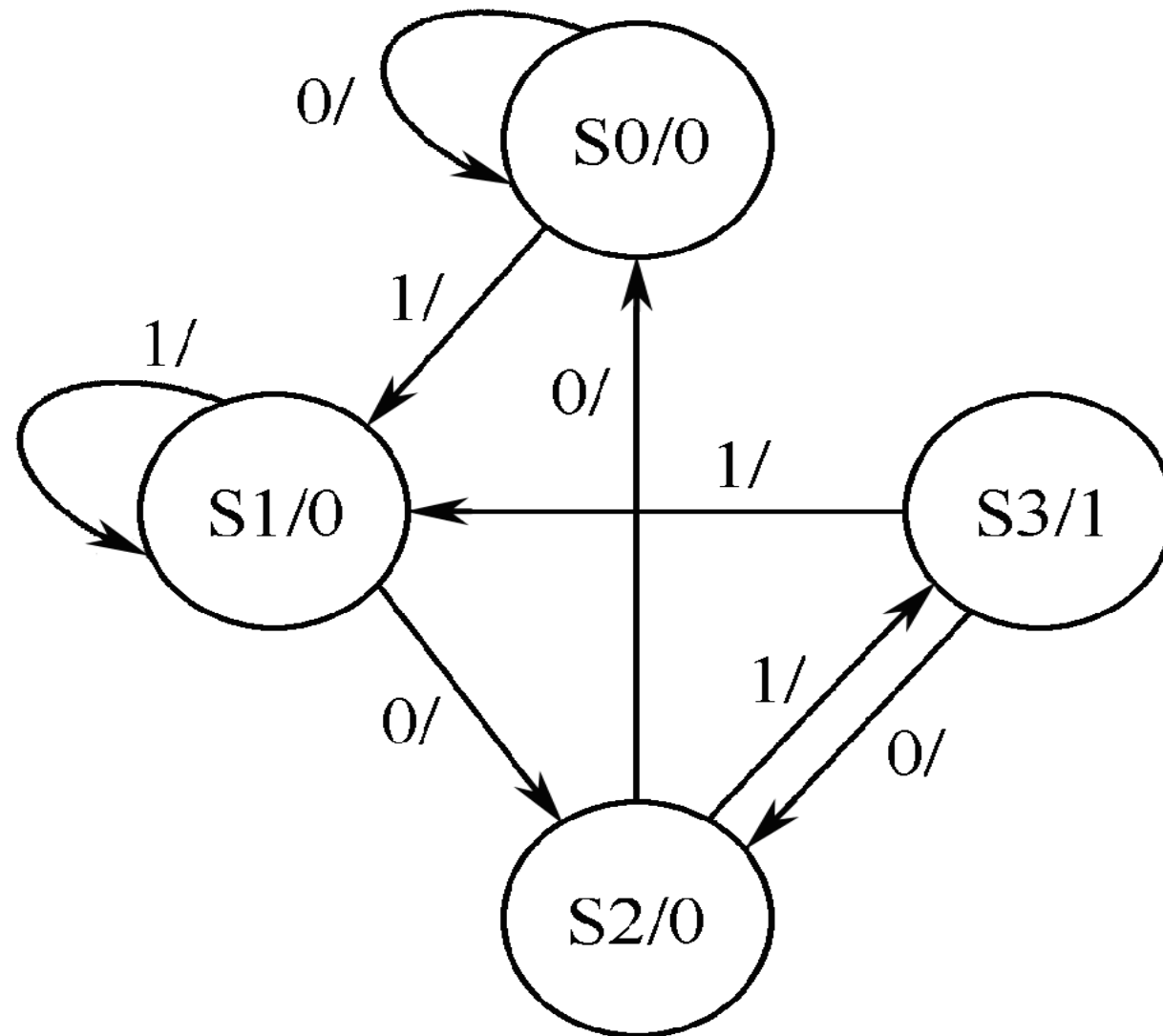
(3) 双过程描述（CS、NS+OL双过程描述）：一个过程用来描述现态（CS）；另一个过程描述次态和输出逻辑（NS+OL）。

(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。

“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

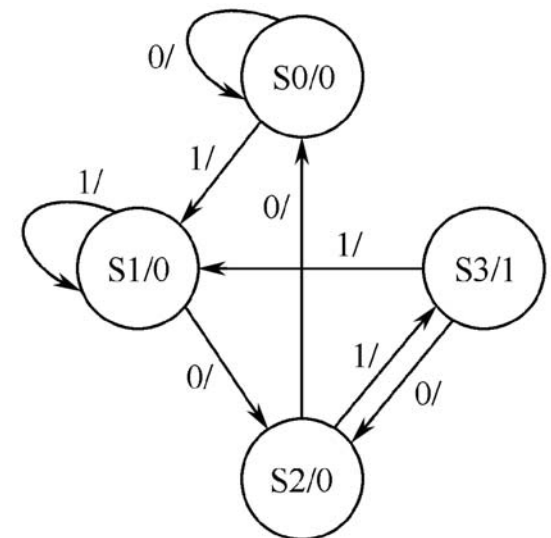
“101”序列检测器的Verilog描述（三个过程）



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）

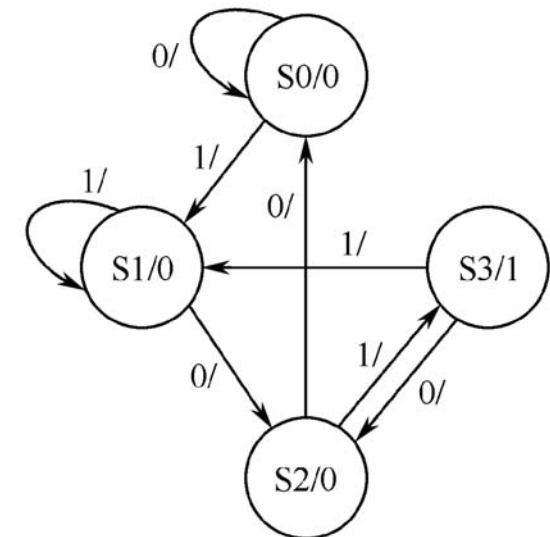
```
module fsm1_seq101(clk, clr, x, z);  
input clk, clr, x;  
output reg z;  
reg[1:0] state, next_state;  
parameter S0=2' b00, S1=2' b01, S2=2' b11, S3=2' b10;  
        /*状态编码，采用格雷（Gray）编码方式*/  
always @(posedge clk or posedge clr) /*该过程定义当前状态*/  
begin if(clr) state<=S0;           //异步复位，s0为起始状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

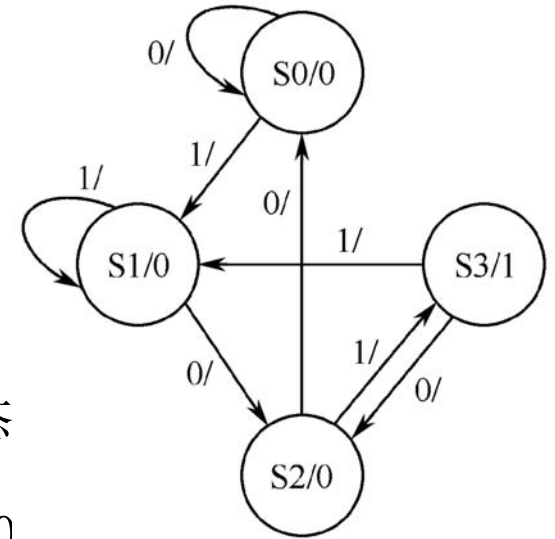
```
S2:begin
    if(x) next_state=S3;
    else next_state=S0;
end
S3:begin
    if(x) next_state=S1;
    else next_state=S2;
end
default: next_state=S0;
/*default语句*/
endcase
end
```

```
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：

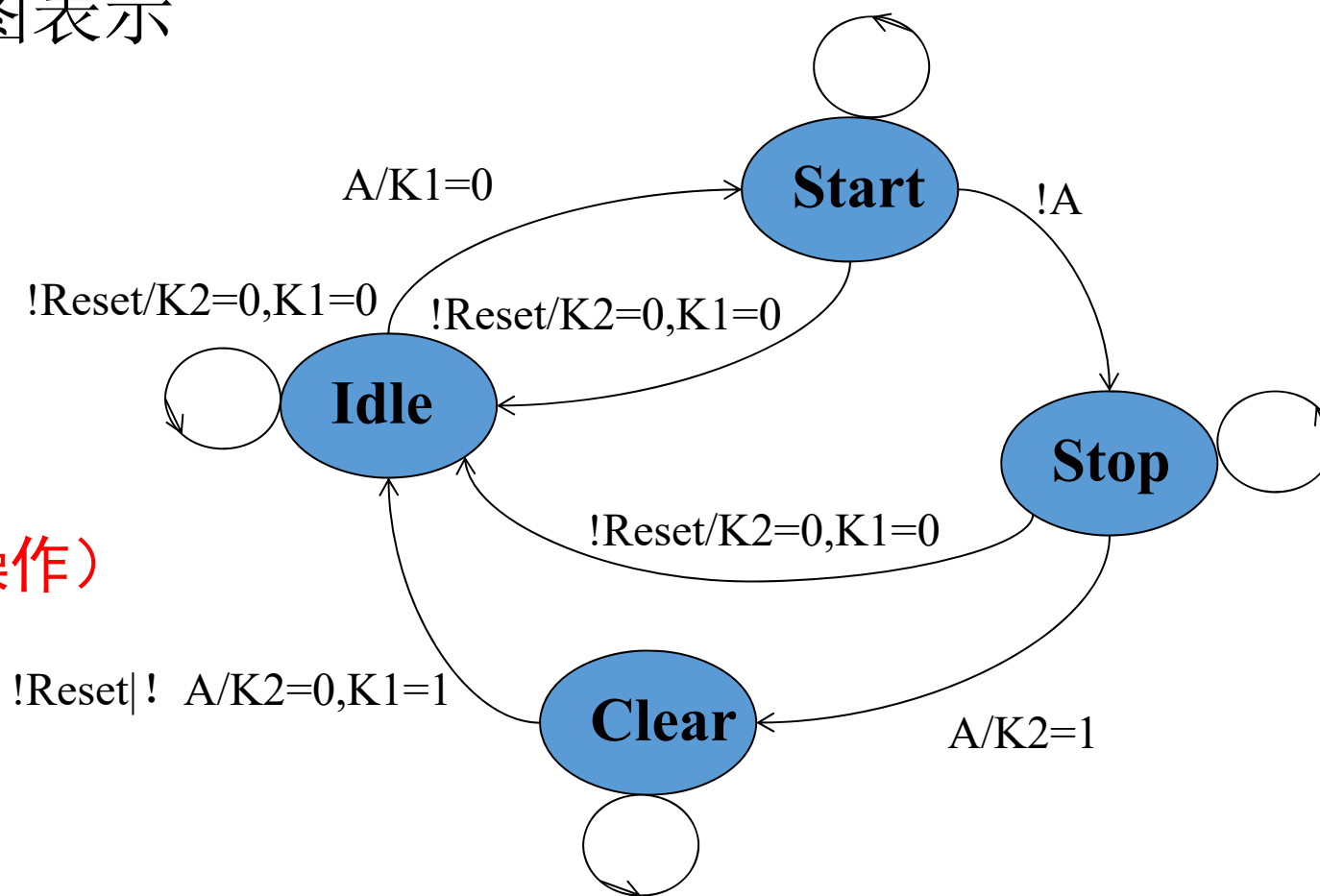
- （1）在case语句中用default分支决定如果进入无效状态所采取的措施；
- （2）编写必要的Verilog源代码明确定义进入无效状态所采取的行为。

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）

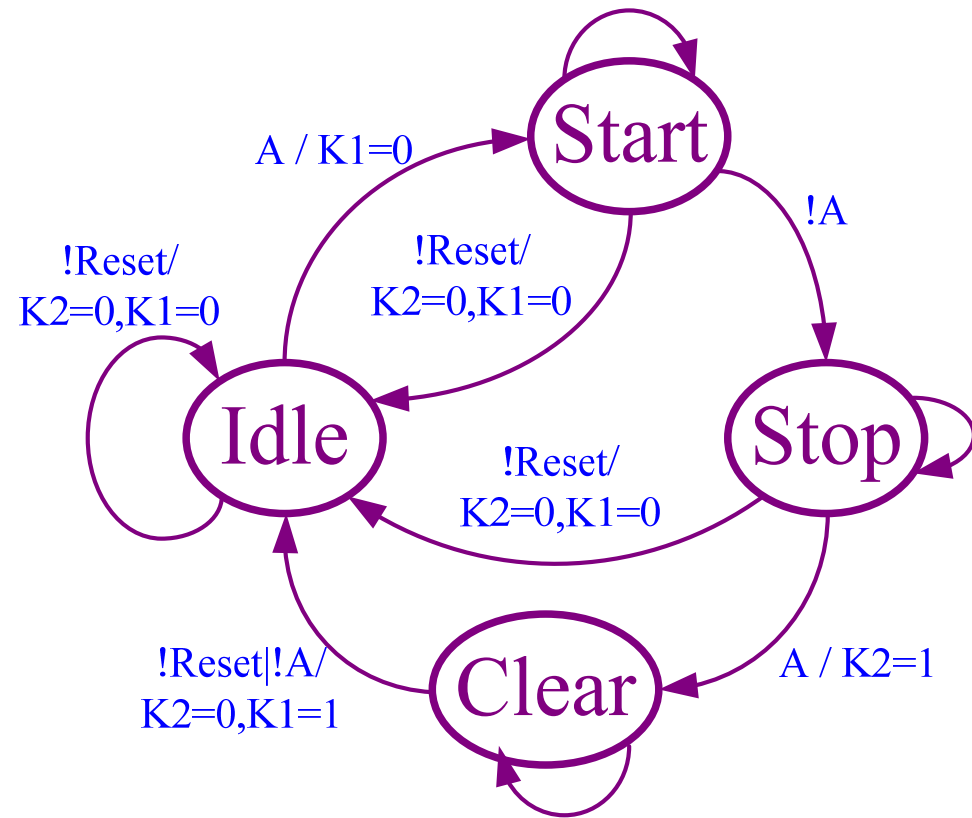


■ 有限状态机的Verilog描述

- 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
```

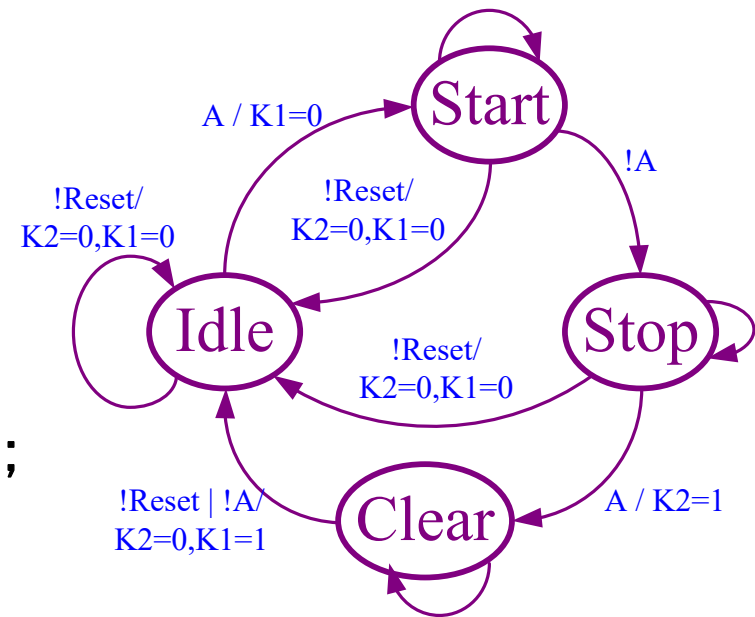
■ 有限状态机的Verilog描述：（方法一）

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

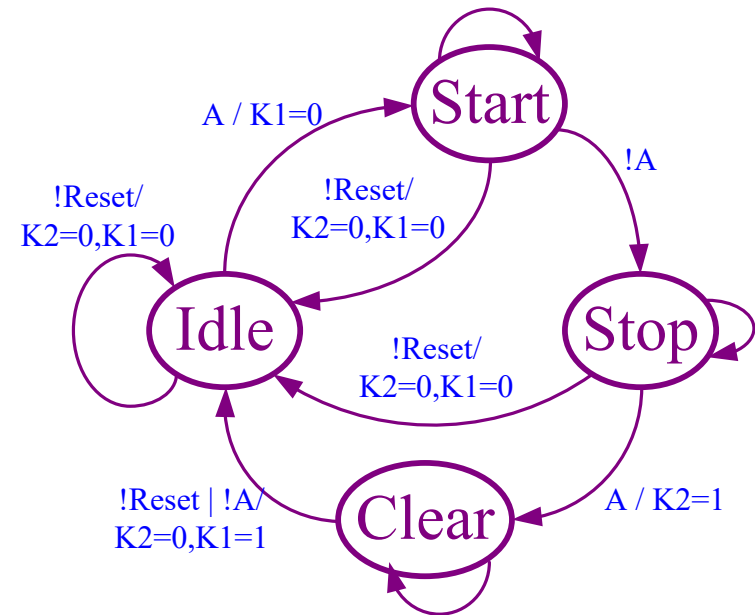
```
    if(!A | !Reset) begin
        state <= Idle;
        K2<=0; K1<=1;
    end
```

```
    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:
```

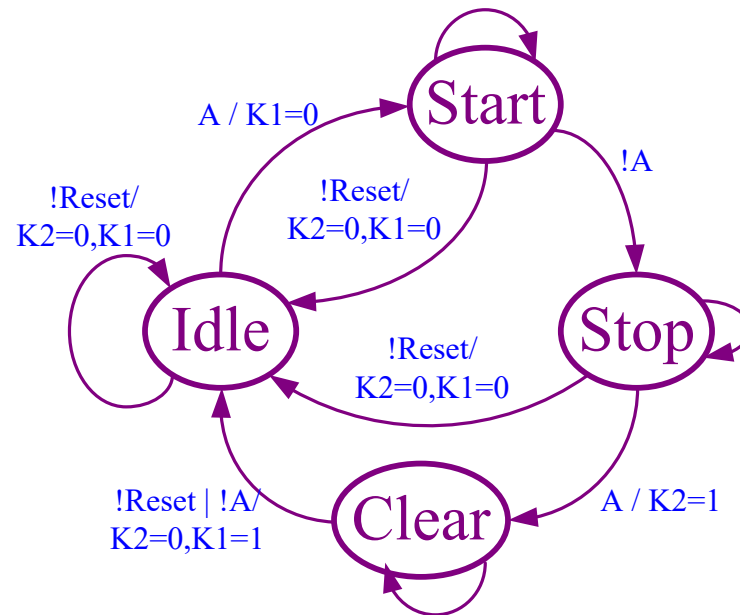
```
                if (A) begin
```

```
                    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:

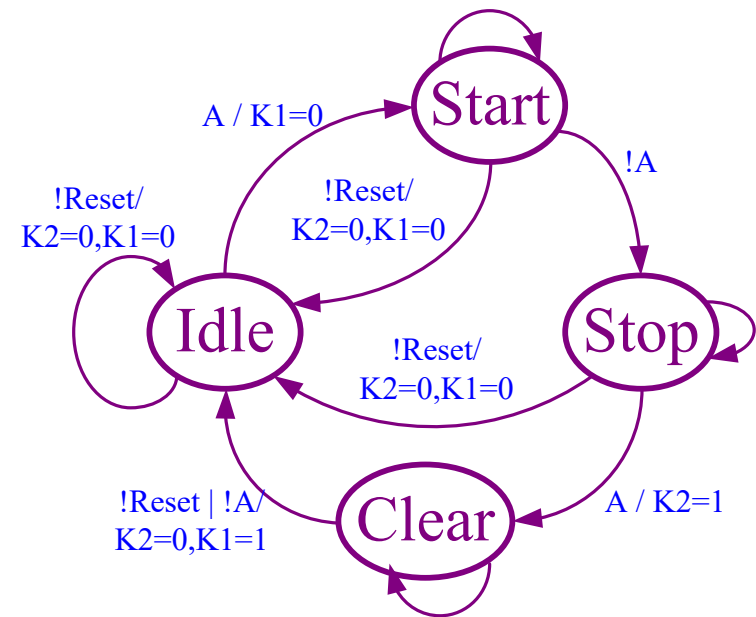
```
if (!A | !Reset) begin  
    state <= Idle;  
    K2 <= 0; K1 <= 1;  
end
```

```
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

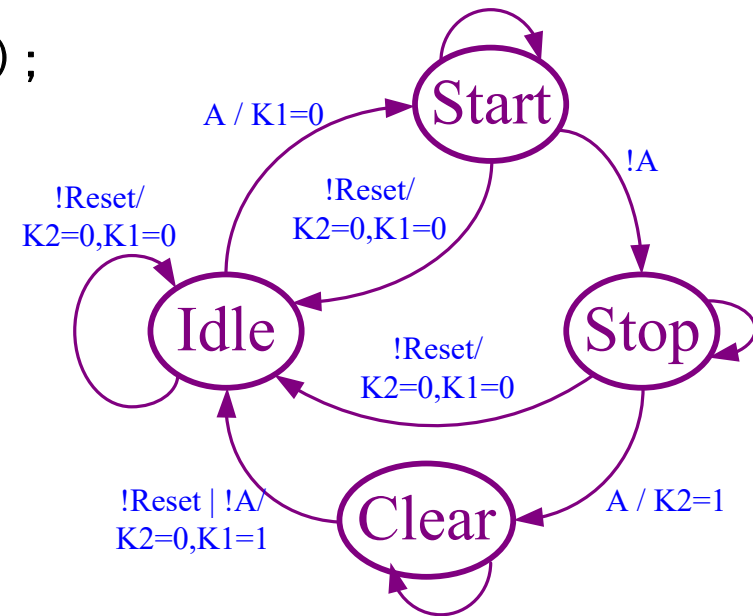
- 思路：把状态的变化与输出开关的控制分成两部分来考虑；为调试方便，常把每个开关写成独立的always组合块。
- 优点：在调试多输出状态机时，这样做比较容易发现问题和改正模块编写中出现的问题。

■ 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-----

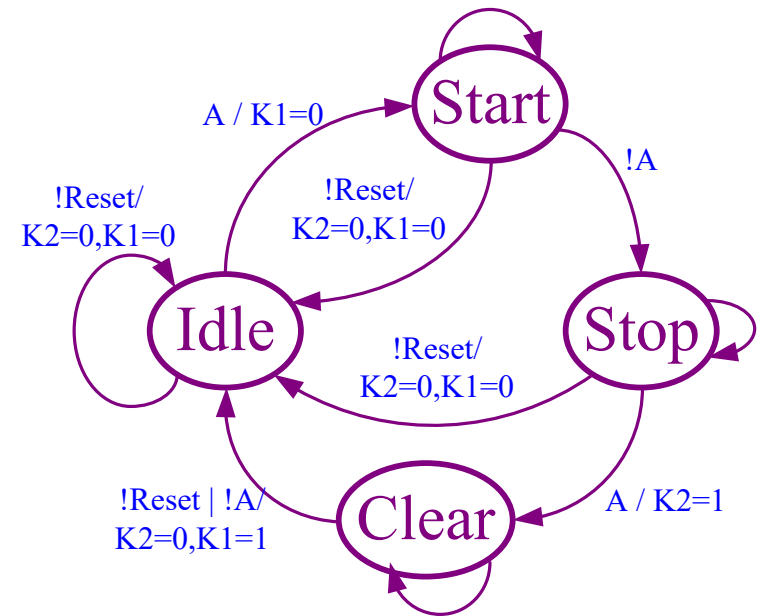
```
always @(posedge Clock)  
begin    if (!Reset)  
            state <= Idle;  
        else    state <= nextstate;  
end
```



■ 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

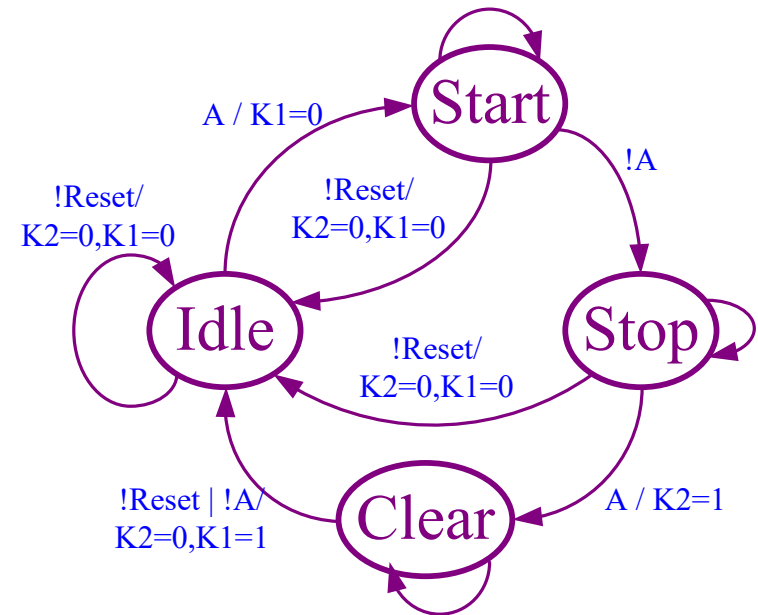
■ 有限状态机的Verilog描述：方法三（续）

//---- 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;
endmodule
```



■ 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

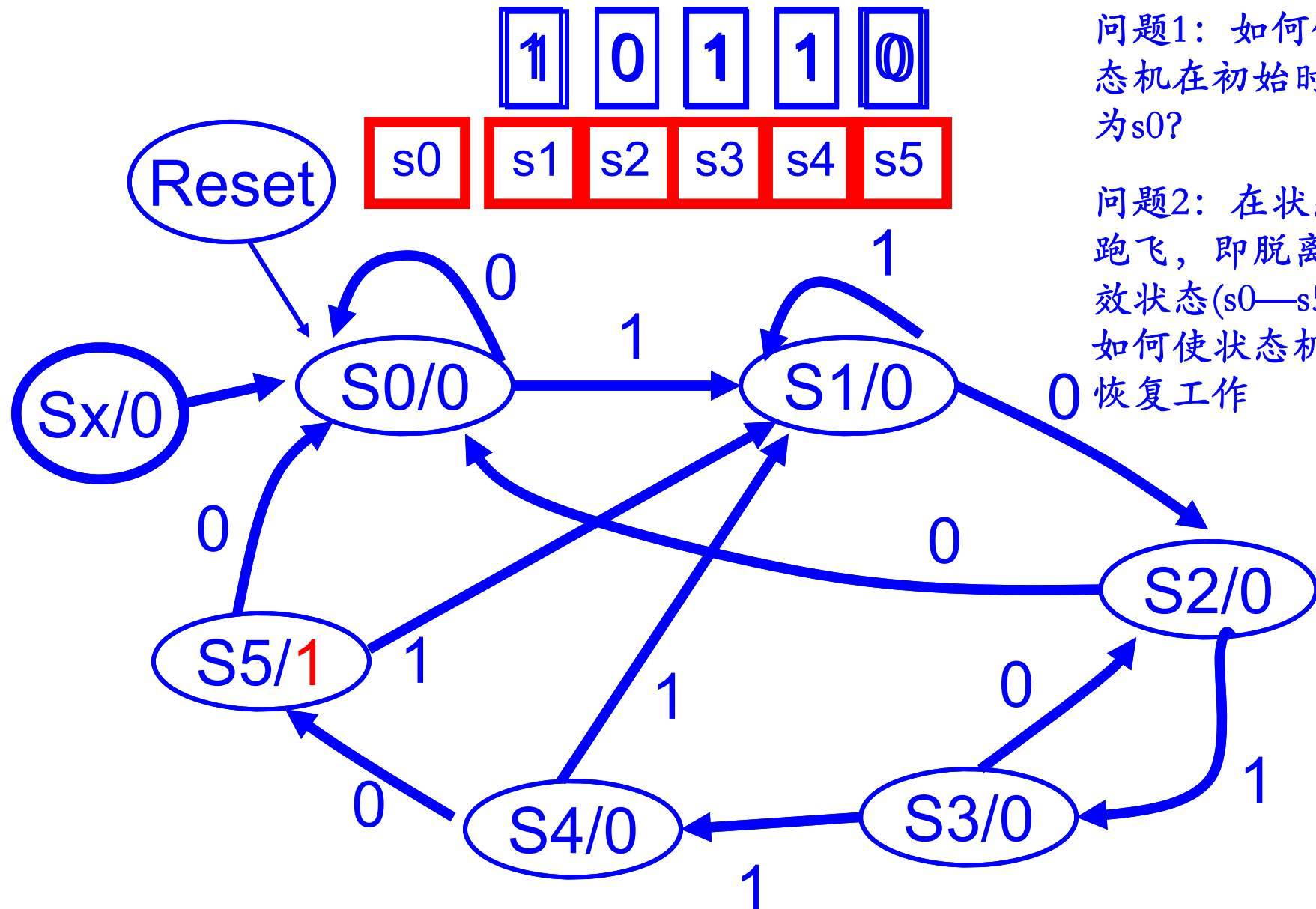
例：设计一个二进制序列检测器，当检测到**10110**序列时，就输出**1**(一个时钟周期的脉冲)。其他情况下输出**0**。

规定检测到一次之后，检测器复位到最初的状态，重新从头检测。如下所示：

输入：01**10110**1101100

输出：0000000**1**00000**1**

状态转换图设计 (Moore)

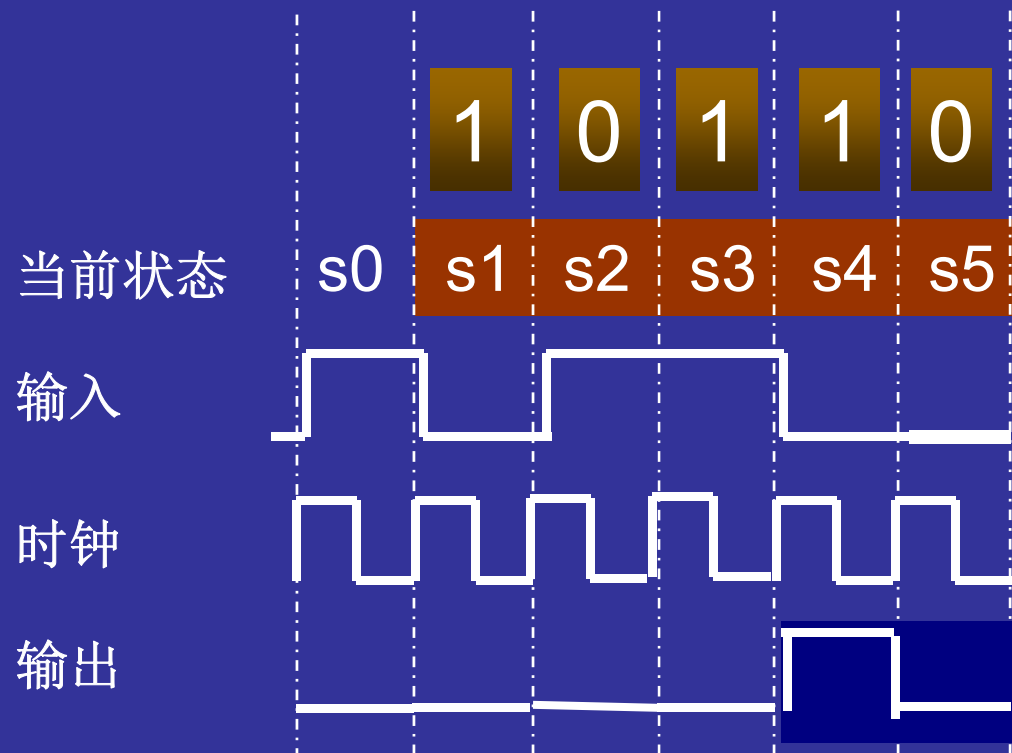


问题1: 如何保证状态机在初始时状态为 s_0 ?

问题2: 在状态机跑飞, 即脱离有效状态 (s_0-s_5) 时, 如何使状态机能恢复工作?

波形

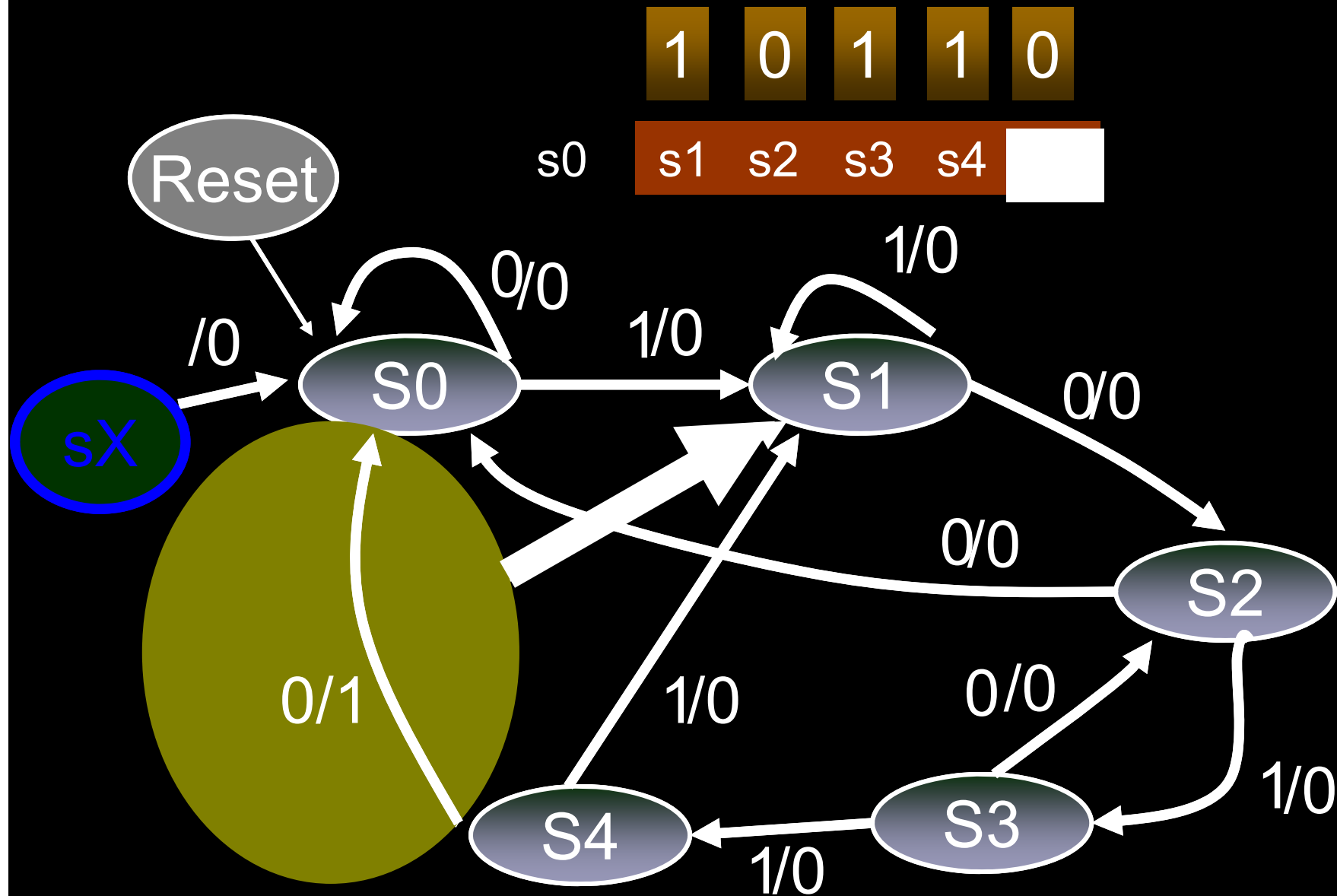
波形如下图所示



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？

发现当前状态为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

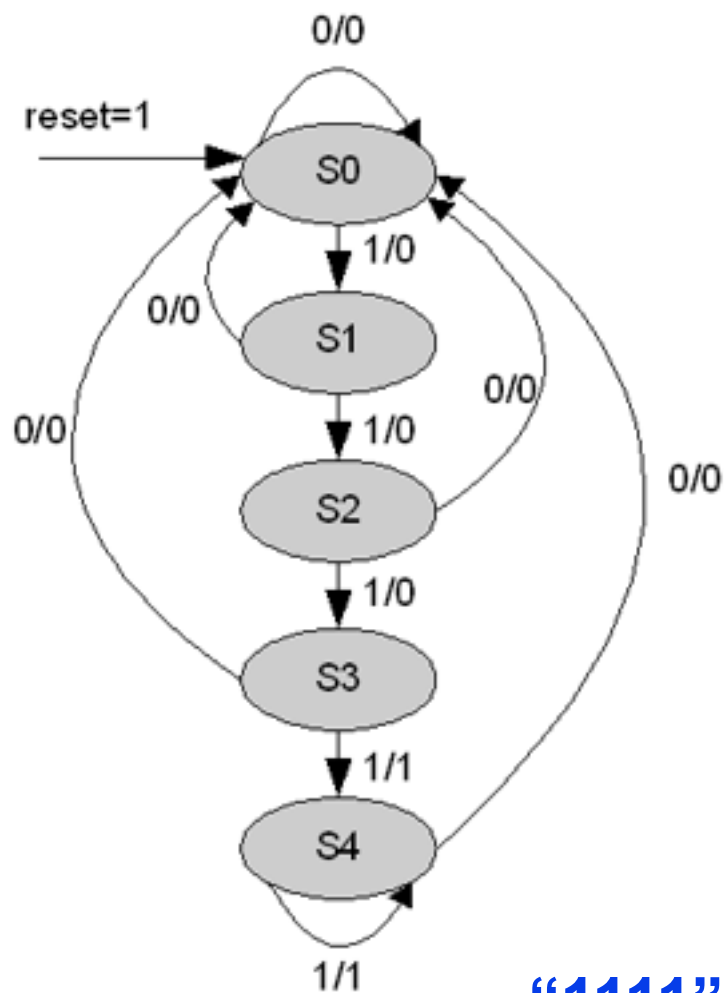
用状态机设计一个二进制序列检测器，其功能是检测一个4位二进制序列“1111”，即输入序列中如果有4个或4个以上连续的“1”出现，输出为1，其它情况下，输出为0。

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)
```

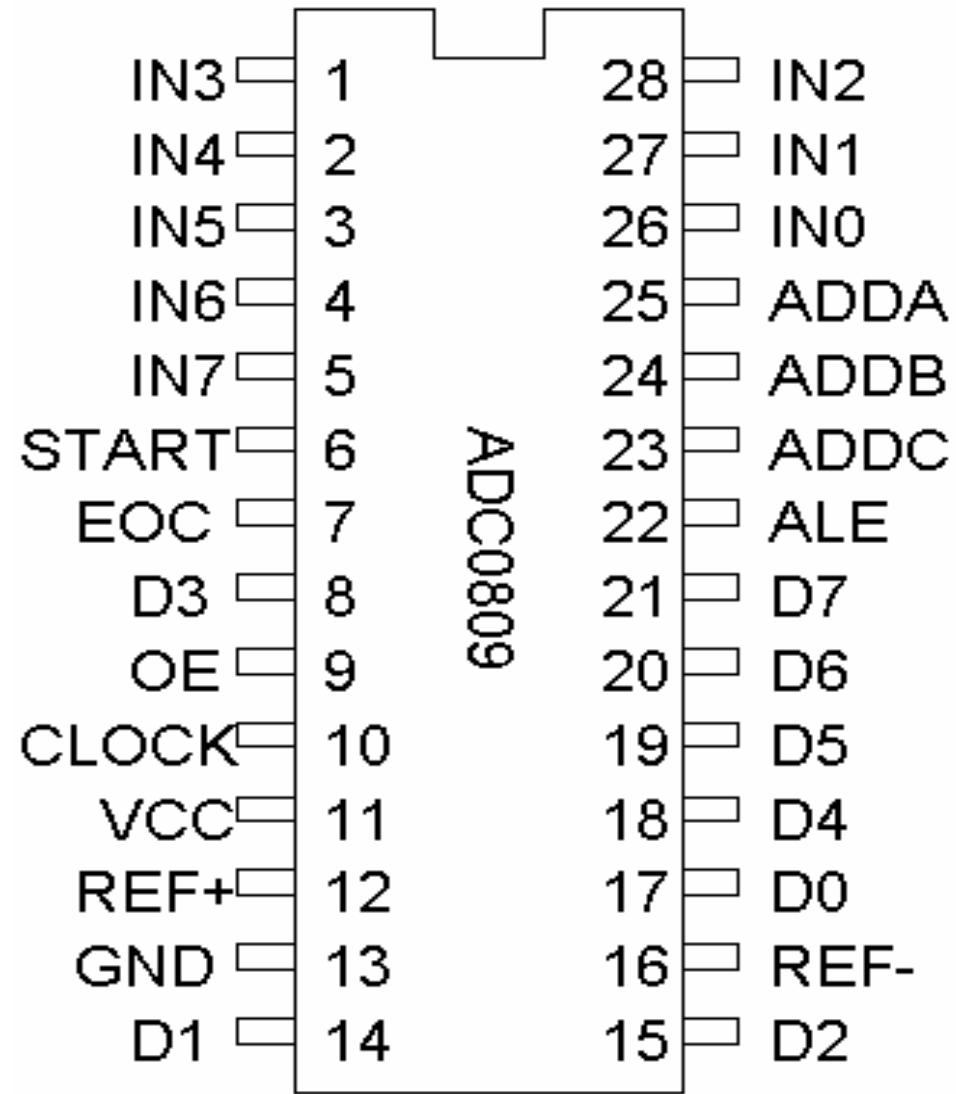
```
s0: begin
    if(x==0) begin next_state<=s0; z<=0; end
    else begin next_state<=s1; z<=0; end
end
s1: begin
    if(x==0) begin next_state<=s0; z<=0; end
    else begin next_state<=s2; z<=0;end
end
s2: begin
    if(x==0) begin next_state<=s0; z<=0; end
    else begin next_state<=s3; z<=0;end
end
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
```


状态机设计举例：数字电压表

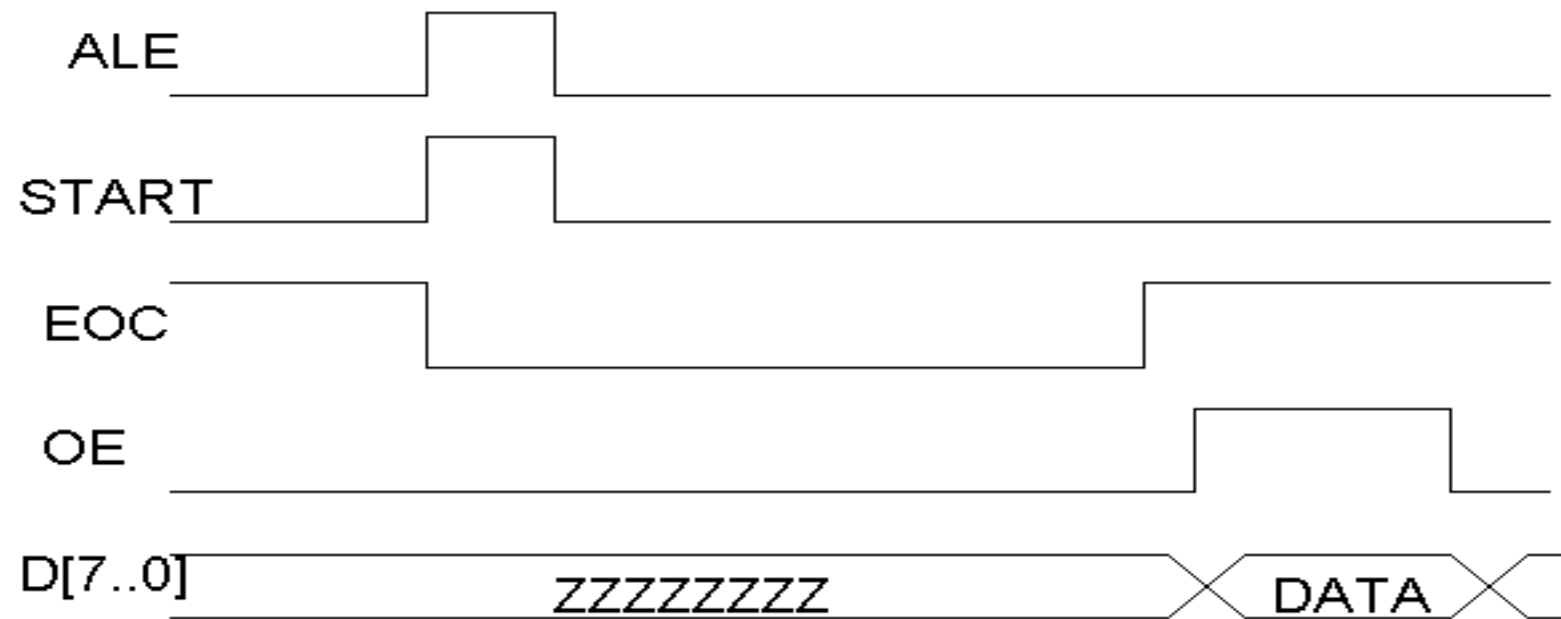
对A/D器件进行采样控制，用单片机完成编程简单，控制灵活，但控制周期长，速度慢。

使用FPGA的状态机来控制A/D采样，可以充分发挥高速采样的特性。

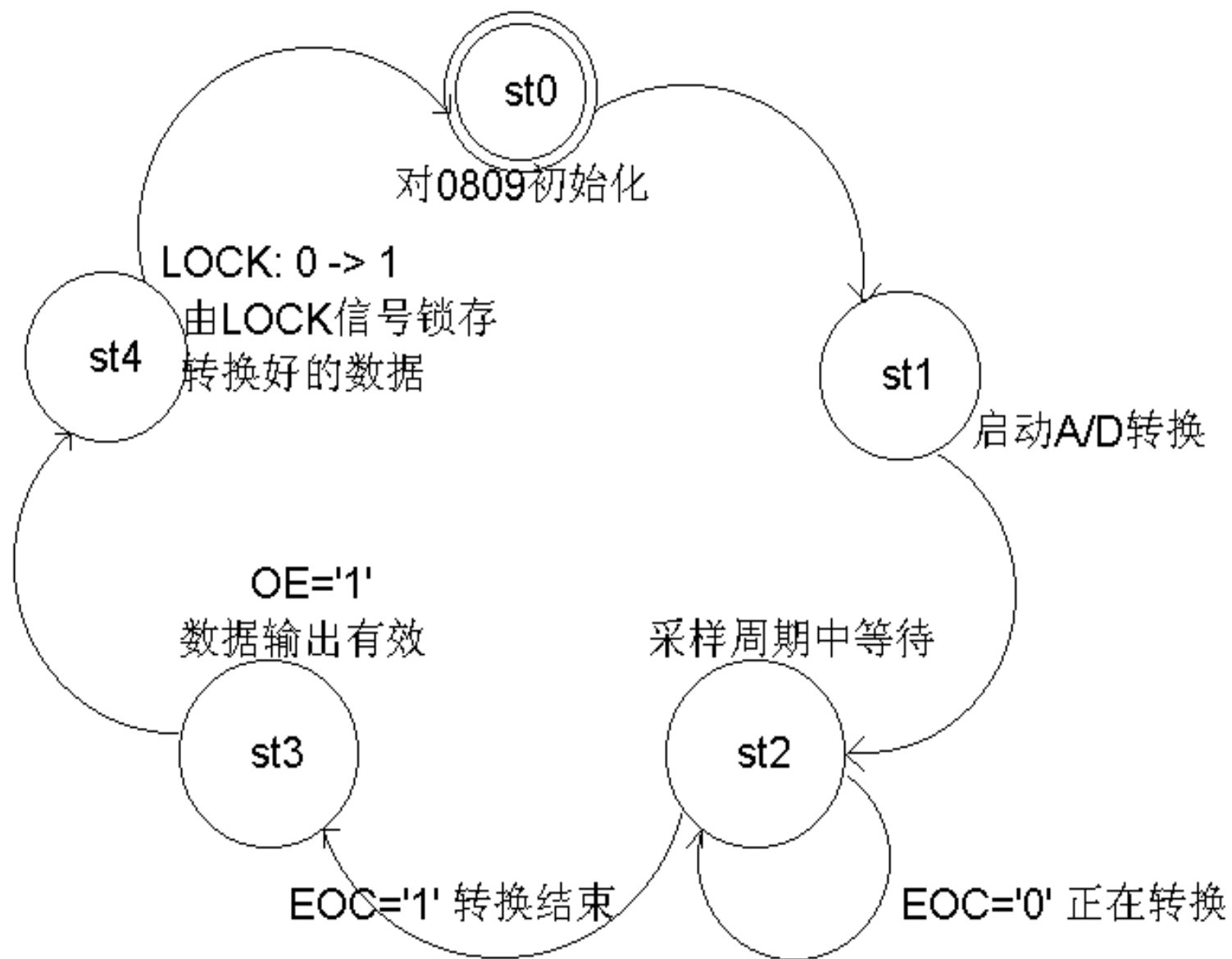
1. ADC0809引脚



2. ADC0809工作时序



3. 控制ADC0809采样状态图



状态位直接输出型编码

将状态码直接输出作为控制信号，每一位的编码值都赋予了实际的控制功能。

ADC0809采样控制信号状态编码表

状态	状态编码					
	START	ALE	OE	LOCK	EOC	功能说明
ST0	0	0	0	0	0	初始态
ST1	1	1	0	0	0	启动转换
ST2	0	0	0	0	1	若EOC=1，转ST3
ST3	0	0	1	0	0	输出转换好的数据
ST4	0	0	1	1	0	锁存转换好的数据

【例 8-2】 为了仿真方便观察，输出口增加了内部锁存信号 LOCK_T

```
module ADC0809 (D, CLK, EOC, RST, ALE, START, OE, ADDA, Q, LOCK_T);
    input[7:0] D;           //来自0809转换好的8位数据
    input CLK,RST;          //状态机工作时钟,和系统复位控制
    input EOC;              //转换状态指示，低电平表示正在转换
    output ALE;             //8个模拟信号通道地址锁存信号
    output START,OE ;       //转换启动信号，和数据输出三态控制信号
    output ADDA,LOCK_T ;    //信号通道控制信号和锁存测试信号
    output[7:0] Q;
    reg ALE, START, OE;
    parameter s0=0,s1=1,s2=2,s3=3,s4=4; //定义各状态子类型
    reg[4:0] cs , next_state ;           //为了便于仿真显示，现态名简为cs
    reg[7:0] REGL; reg LOCK;             // 转换后数据输出锁存时钟信号
    always @(cs or EOC) begin            // 组合过程，规定各状态转换方式
        case (cs)
            s0 : begin ALE=0 ; START=0 ; OE=0 ; LOCK=0 ;
```

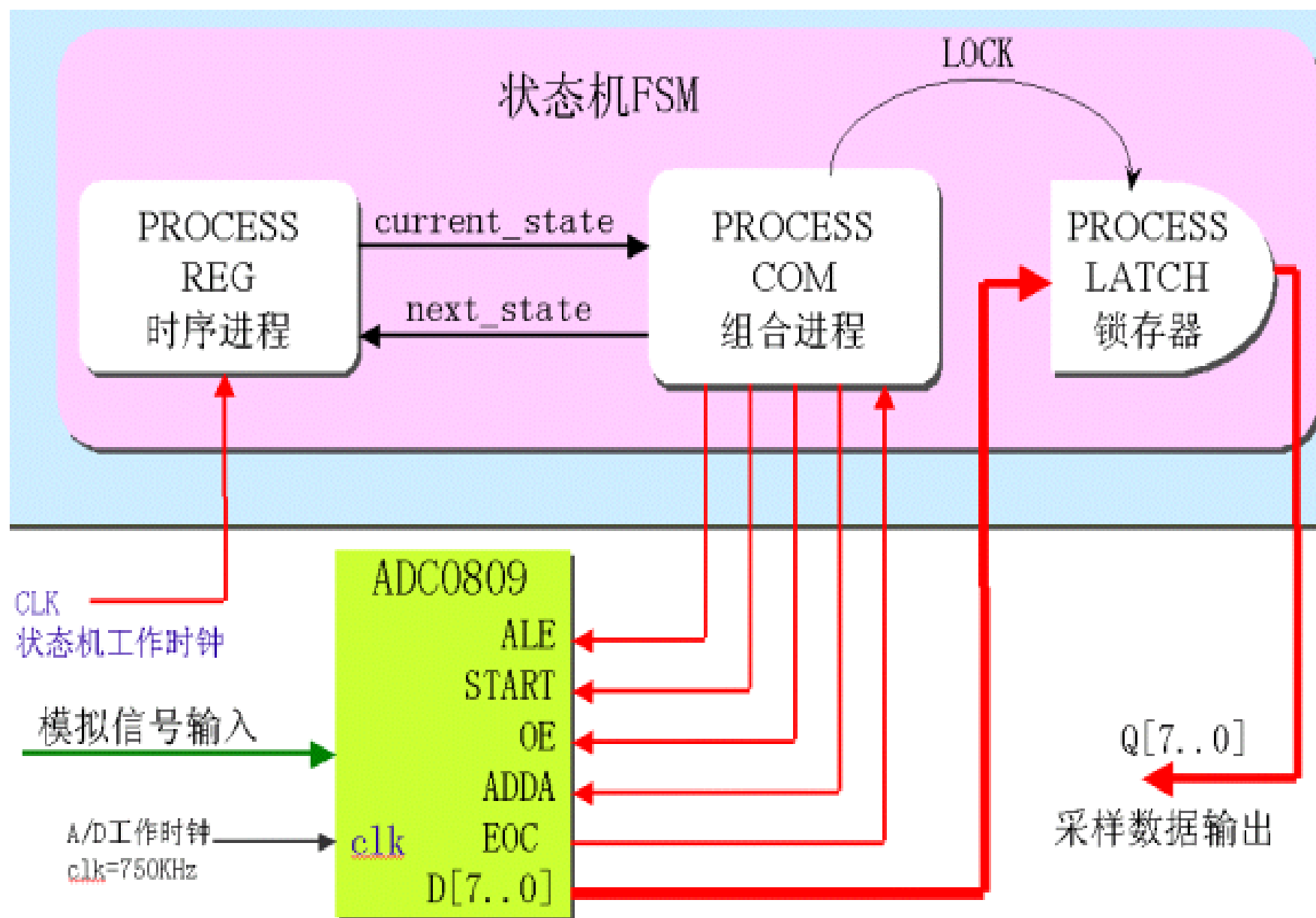
[接下页](#)

[接上页](#)

```

        next_state <= s1 ;    end           //0809初始化
s1 : begin  ALE=1 ; START=1 ; OE=0 ; LOCK=0 ;
        next_state <= s2 ;    end           //启动采样信号START
s2 : begin  ALE=0 ; START=0 ; OE=0 ; LOCK=0 ;
        if (EOC==1'b1) next_state = s3 ;    //EOC=0表明转换结束
        else next_state = s2 ; end           //转换未结束，继续等待
s3 : begin  ALE=0 ; START=0 ; OE=1; LOCK=0; //开启OE，打开AD数据口。
        next_state = s4 ;    end           //下一状态无条件转向s4
s4 : begin  ALE=0 ; START=0 ; OE=1; LOCK=1; //开启数据锁存信号
        next_state <= s0 ;    end
default : begin  ALE=0 ; START=0 ; OE=0 ; LOCK=0 ;
        next_state = s0 ;    end
endcase    end
always @(posedge CLK or posedge RST) begin //时序过程
    if (RST) cs <= s0 ;
    else cs <= next_state ; end // 由现态变量cs将当前状态值带出过程
always @(posedge LOCK)                //寄存器过程
    if (LOCK)    REGL <= D ; // 此过程中，在LOCK的上升沿将转换好的数据锁入
assign ADDA =0 ; assign Q = REGL ; //选择模拟信号进入通道IN0
assign LOCK_T = LOCK ; //将测试信号输出
endmodule
```

4. 采样状态机结构



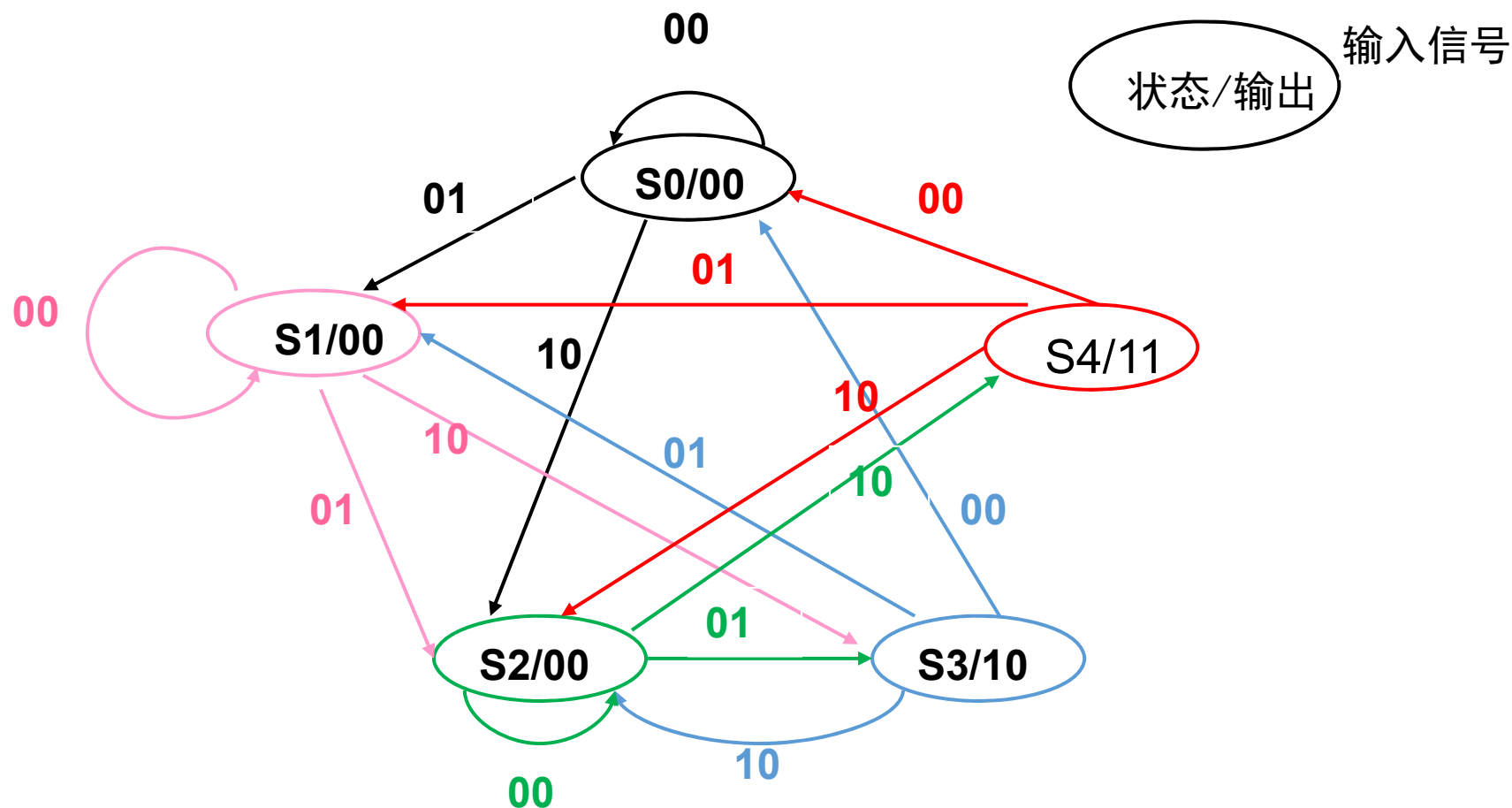
例1：投入1元5角硬币输出货物，投入2元硬币输出货物并找5角零钱的自动售货机。

状态：S0表示初态，S1表示投入5角硬币，
S2表示投入1元硬币，S3表示投入1元5角硬币，
S4表示投入2元硬币

输入：state_input(0) 投入1元硬币，
state_input(1) 投入5角硬币

输出：comb_outputs(0) 输出货物，
comb_outputs(1) 找5角零钱

根据需求列写状态图。



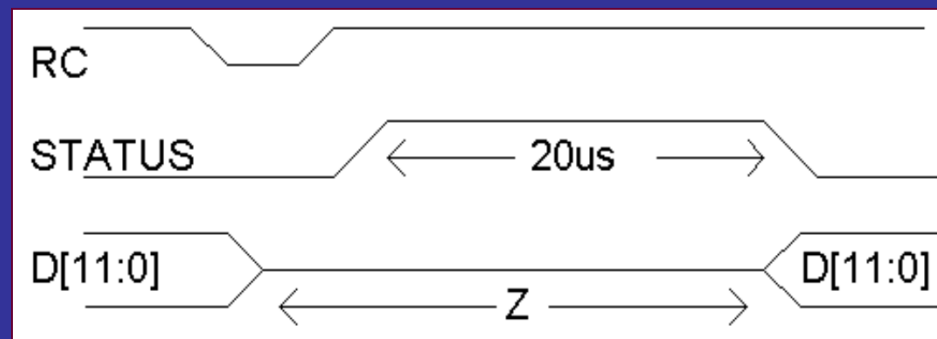
状态：S0、S1、S2、S3、S4；输入：state_inputs

输出：comb_outputs；输出仅与状态有关

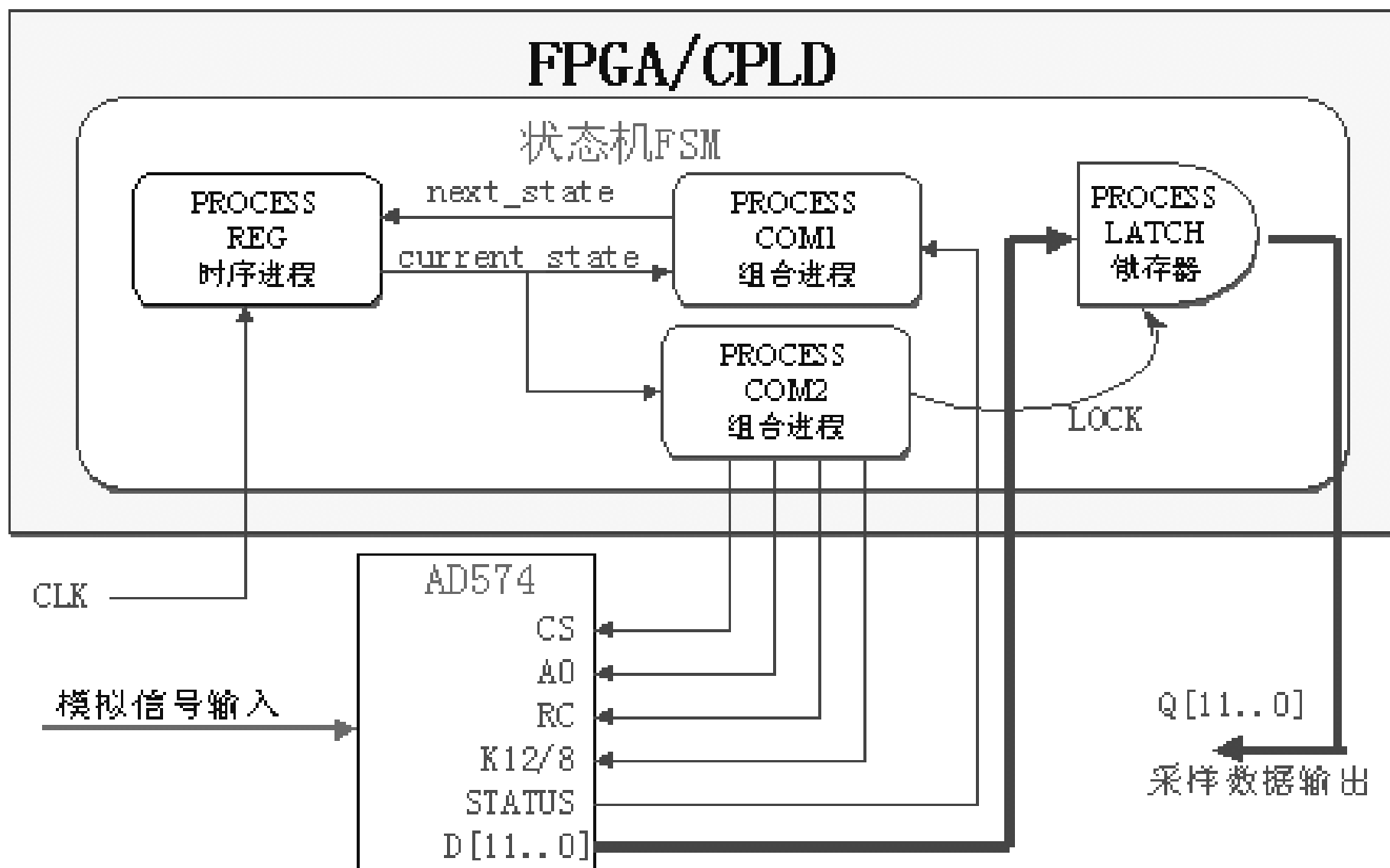
AD574真值表

CE	CS	RC	K12/8	A0	工 作 状 态
0	X	X	X	X	禁止
X	1	X	X	X	禁止
1	0	0	X	0	启动12位转换
1	0	0	X	1	启动8位转换
1	0	1	1	X	12位并行输出有效
1	0	1	0	0	高8位并行输出有效
1	0	1	0	1	低4位加上尾随4个0有效

AD574时序图

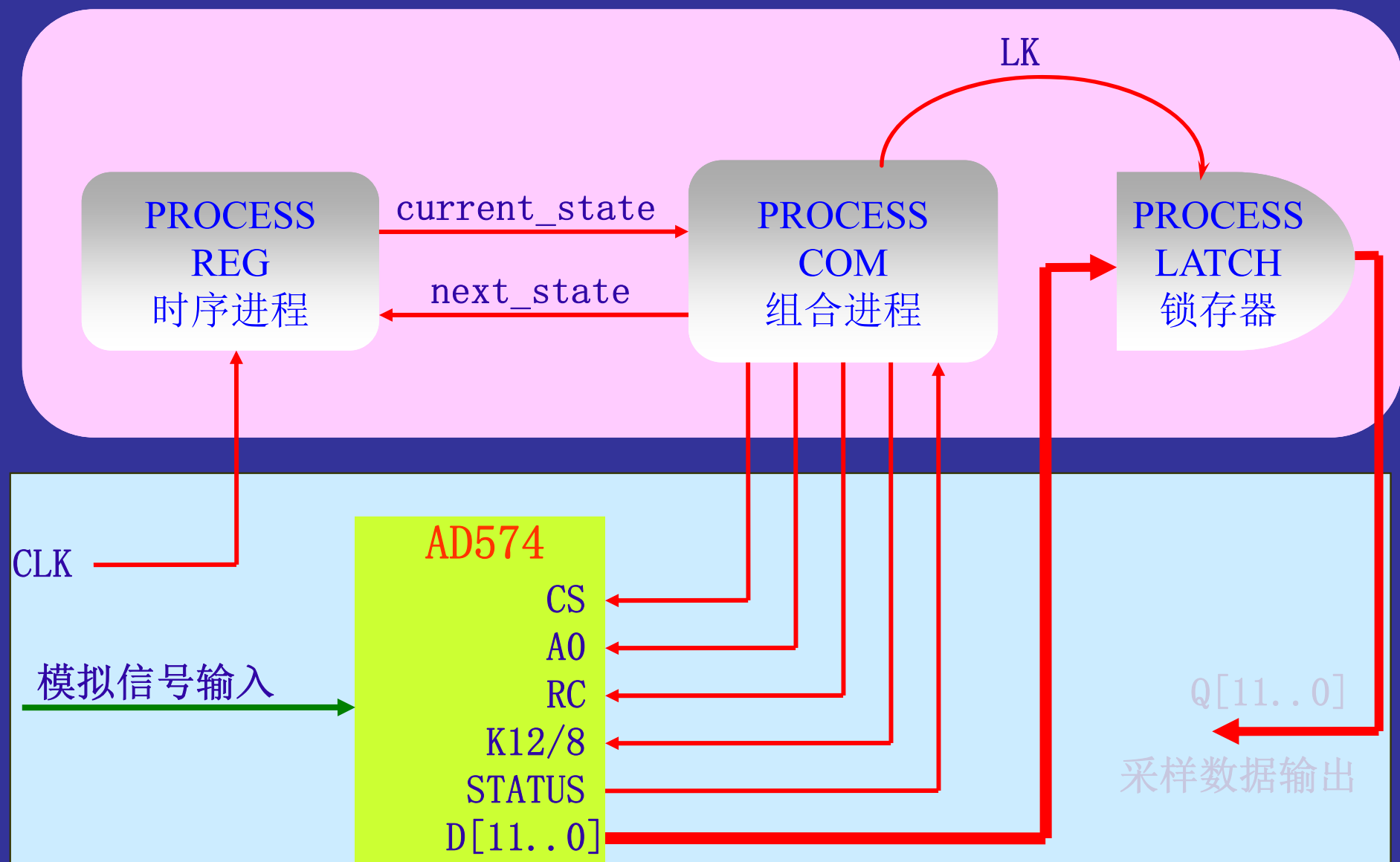


AD574采样控制状态机结构图

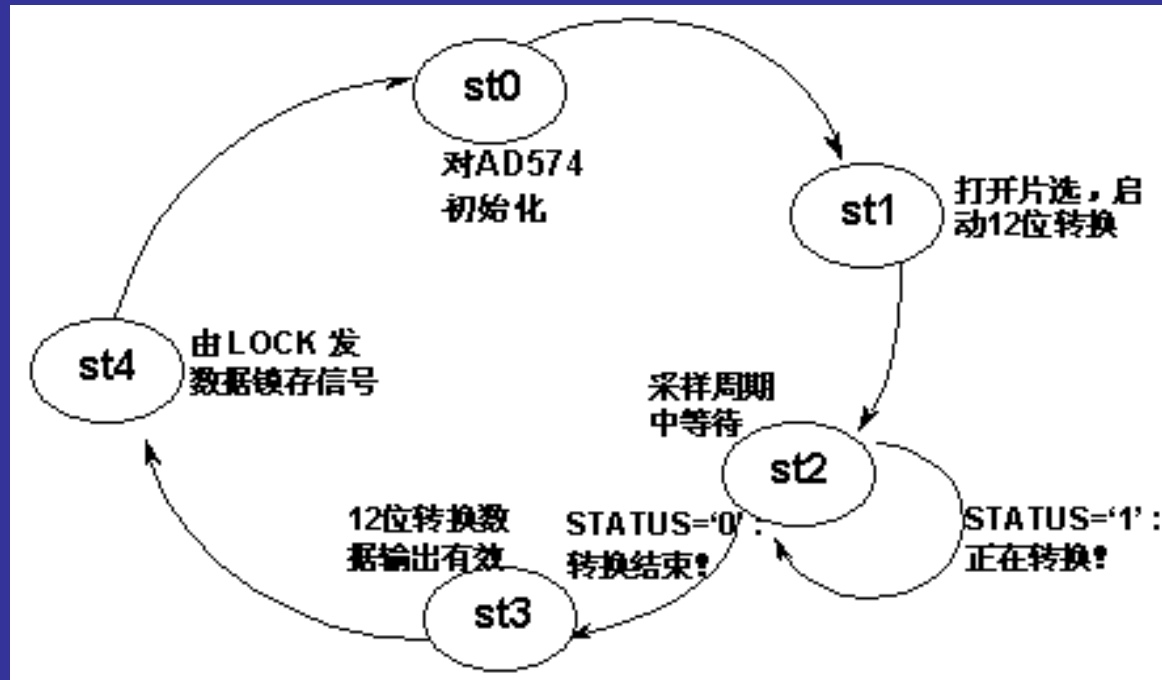


可以将两个组合进程合并为一个进程，将状态编码设计成输出信号。

AD574采样控制状态机结构图



AD574采样控制状态图

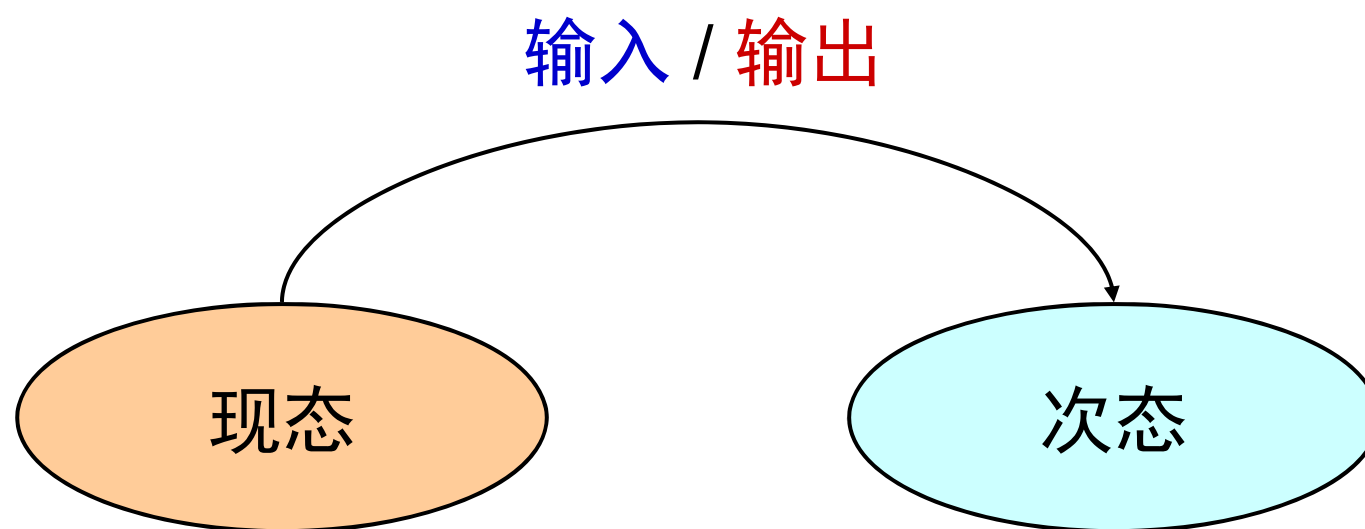


状态	状态编码out4 (3~0)					功能说明
ST	CS	A0	RC	LK	B	
ST0	1	1	1	0	0	初态
ST1	0	0	0	0	1	启动转换
ST2	0	0	0	0	0	若测得status=1, 转下一状态
ST3	0	0	1	0	0	输出转换后的数据
ST4	0	0	1	1	0	产生LK边沿, 将转换数据锁存

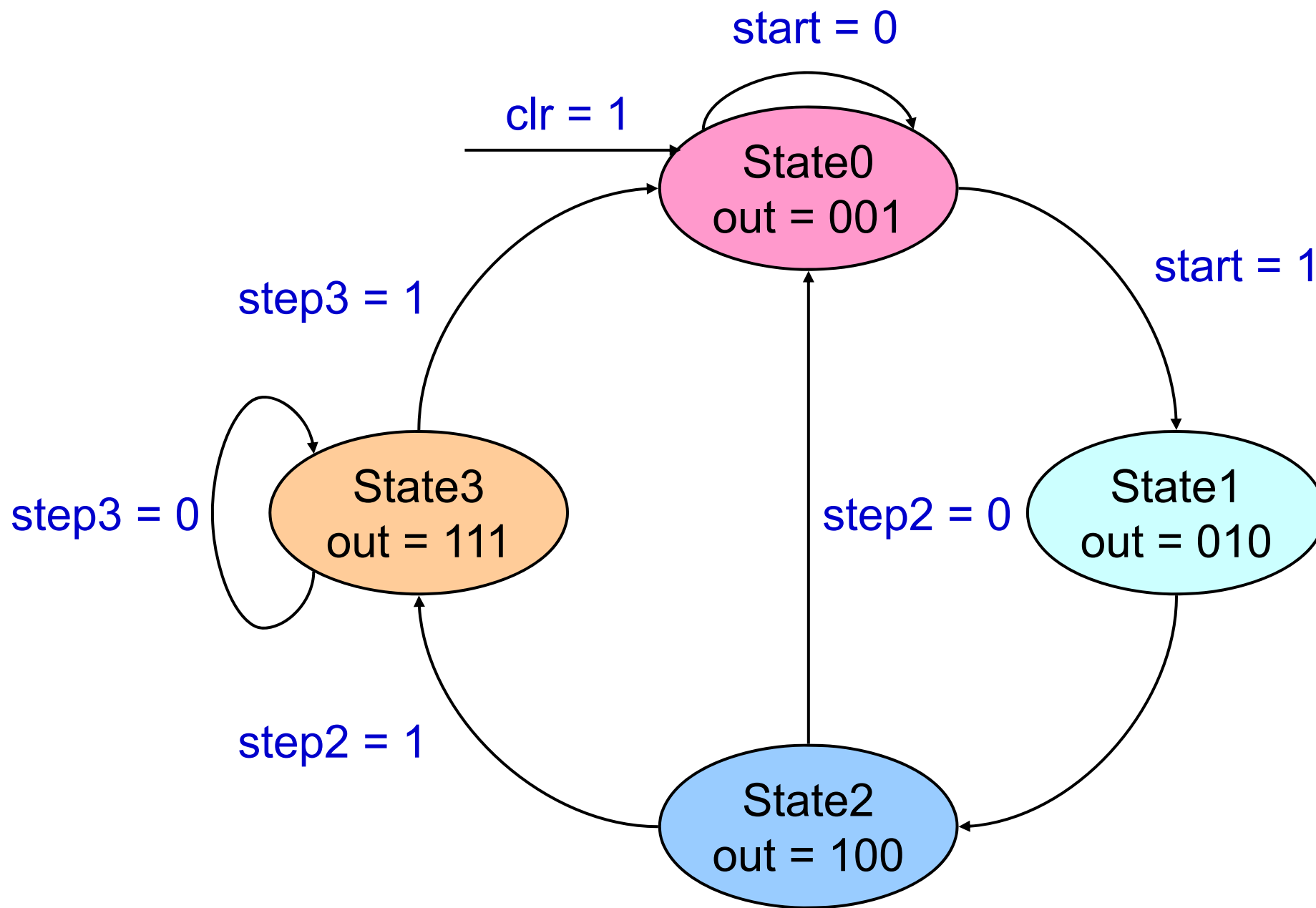
状态机设计举例：实用交通灯设计。

功能要求：主辅十字路口，分别有红、黄、绿和左转弯灯，等待时间，系统复位控制。灯持续时间及其要求如下：

主路灯				辅路灯				持续时间
R	Y	G	L	R	Y	G	L	(S)
0	0	1	0	1	0	0	0	50
0	1	0	0	1	0	0	0	5
1	0	0	1	1	0	0	0	15
1	0	0	0	0	0	1	0	40
1	0	0	0	0	1	0	0	5
1	0	0	0	1	0	0	1	10



Mealy型状态图



Moore型状态图

状态机设计举例

```
module FSM( clk, clr, out, start, step2, step3 );  
input clk, clr, start, step2, step3;  
output[2:0] out;  
reg[2:0] out;  
reg[1:0] state, next_state;  
parameter state0 = 2'b00, state1 = 2'b01, // 状态编码  
            state2 = 2'b11, state3 = 2'b10; // 格雷码  
  
always @( posedge clk or posedge clr )  
begin  
    if( clr )    state <= state0; // 定义初态  
    else        state <= next_state;  
end
```

```
always @( state or start or step2 or step3 ) // 状态转换
```

```
begin
```

```
    case (state)
```

```
        state0:
```

```
            begin
```

```
                if( start )    next_state <= state1;
```

```
                else          next_state <= state0;
```

```
            end
```

```
        state1:
```

```
            begin
```

```
                next_state <= state2;
```

```
            end
```

```
state2:
    begin
        if( step2 )    next_state <= state3;
        else           next_state <= state0;
    end
state3:
    begin
        if( step3 )    next_state <= state0;
        else           next_state <= state3;
    end
    default:           next_state <= state0;
endcase
end
```

```
always @( state ) // 状态译码及输出
```

```
begin
```

```
    case( state )
```

```
        state0:    out = 3'b001;
```

```
        state1:    out = 3'b010;
```

```
        state2:    out = 3'b100;
```

```
        state3:    out = 3'b111;
```

```
        default:   out = 3'b001;
```

```
    endcase
```

```
end
```

```
endmodule
```