# 模拟与数字电路

# Analog and Digital Circuits

课程主页 扫一扫

第 十二 讲: **状态机的应用**

Lecture 12: **Finite State Machine - II**

主　　讲：陈 迟 晓

Instructor : Chixiao Chen

# 提纲

- 复习
  - 状态机模型有哪两种，他们的区别是什么？

- 基于Verilog的状态机设计流程
- 序列检测电路
- 按键消抖动电路
- 其他时序电路——寄存器与存储器（I）

# 基于Verilog的状态机通用设计流程

- 设计步骤
  - 确定状态的逻辑功能
  - 画出状态转移图
  - 符号化的状态转移表（真值表）
  - 对于状态进行合并，简化和编码
  - 利用Verilog描述转移逻辑关系

**风格良好**的状态机Verilog具有以下特点：

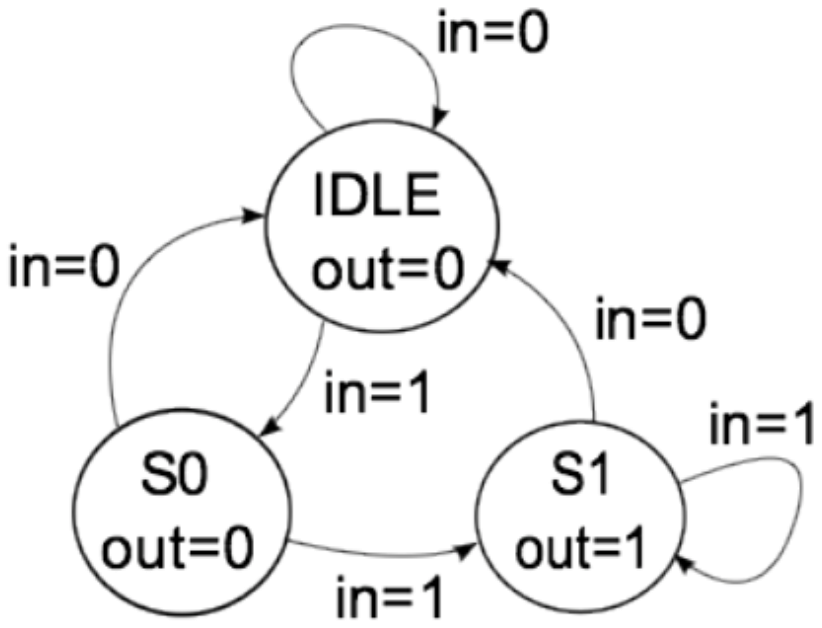
✓ 在Verilog中利用变量代表状态编码

✓ 在always块中使用阻塞赋值实现组合逻辑

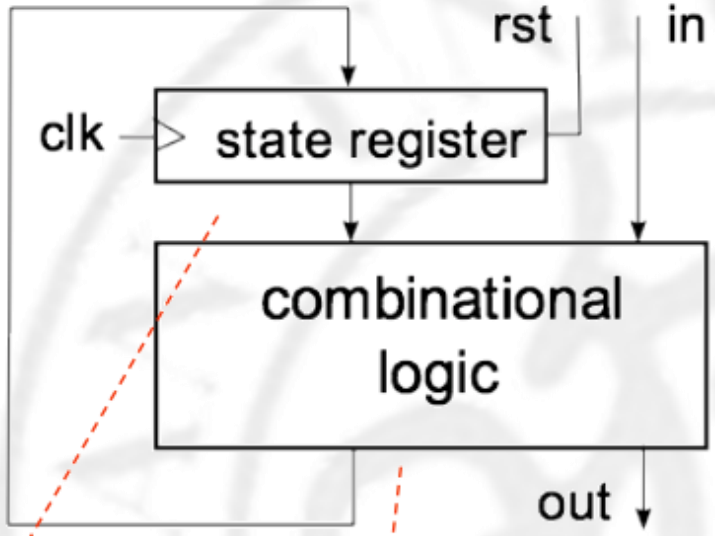
✓ 利用case语句完成转移表与Verilog的直接转化

# 状态转移与预计硬件实现

State Transition Diagram




Implementation Circuit Diagram




*Holds a symbol to keep track of which bubble the FSM is in.*

*CL functions to determine output value and next state based on input and current state.*

*out = f(in, current state)*

*next state = f(in, current state)*

画出上述状态转移图对应的表

# 状态机的Verilog实现 - I

**状态转移代码**

```
reg [1:0] state_reg, state_next;
always @(posedge clk, posdge reset)
if (reset)
    state_reg <= IDLE;
else
  state_reg <= state_next;
```

```
module FSM1(clk, rst, in, out);
input clk, rst;
input in;
output out;

// Defined state encoding:
localparam IDLE = 2'b00;
localparam S0 = 2'b01;
localparam S1 = 2'b10;

reg out;
reg [1:0] next_state;
wire [1:0] present_state;

// state register
REGISTER_R #(.N(2), .INIT(IDLE)) state
(.q(present_state), .d(next_state), .rst(rst));
```

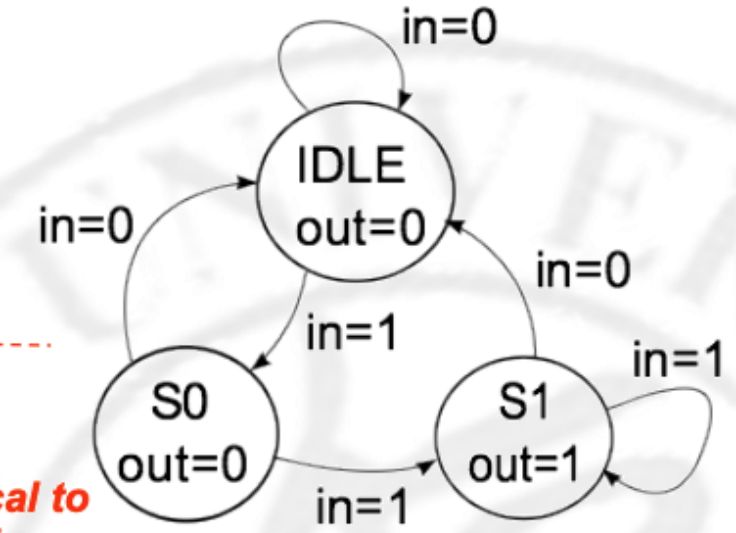
Must use reset to force to initial state.

reset not always shown in STD

Constants local to this module.

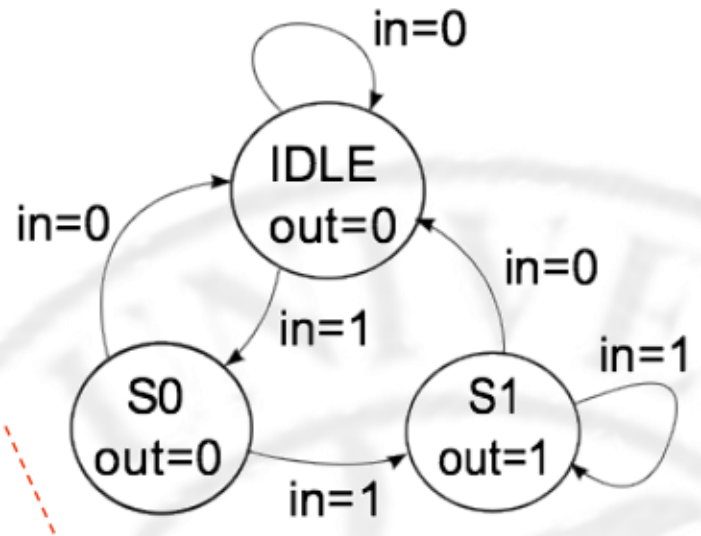
out not a register, but assigned in always block

Combinational logic signals for transition.



# 状态机的Verilog实现 - II

```
// always block for combinational logic portion
always @(present_state or in)
case (present_state)
// For each state def output and next
  IDLE    : begin
              out = 1'b0;
              if (in == 1'b1) next_state = S0;
              else next_state = IDLE;
            end
  S0      : begin
              out = 1'b0;
              if (in == 1'b1) next_state = S1;
              else next_state = IDLE;
            end
  S1      : begin
              out = 1'b1;
              if (in == 1'b1) next_state = S1;
              else next_state = IDLE;
        end
  default: begin
              next_state = IDLE;
              out = 1'b0;
            end
endcase
endmodule
```




*Each state becomes a case clause.*

*For each state define:*

*Output value(s)*

*State transition*

*Use "default" to cover unassigned state. Usually unconditionally transition to reset state.*

这是moore模型还是Mealy模型

# 状态机的Verilog实现 - III

阻塞赋值的Verilog代码优化

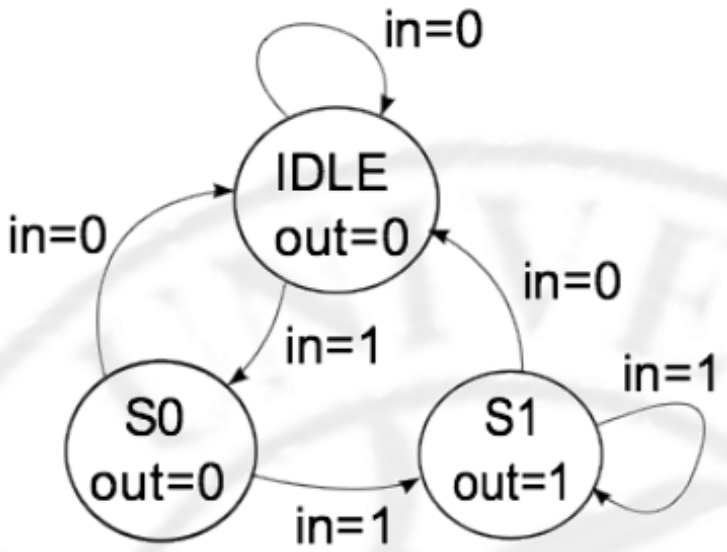
```
always @*
  begin
   next_state = IDLE;
   out = 1'b0;
   case (state)
    IDLE    : if (in == 1'b1) next_state = S0;
    S0      : if (in == 1'b1) next_state = S1;
    S1      : begin
                out = 1'b1;
                if (in == 1'b1) next_state = S1;
              end
    default: ;
   endcase
  end
Endmodule
```

* for sensitivity list

Normal values: used unless specified below.

Within case only need to specify exceptions to the normal values.

Note: The use of "blocking assignments" allow signal values to be "rewritten", simplifying the specification.



# 状态机设计实例 – 序列检测器设计

序列检测器可用于检测一组或多组由二进制码组成的脉冲序列信号，当序列检测器连续收到一组串行二进制码后，如果这组码与检测器中预先设置的码相同，则输出1，否则输出0。

**关键步骤：**正确码的接收必须是连续的，要求检测器必须记住前一次的正确码及正确序列，直到在连续的检测中所收到的每一位码都与预置数的对应码相同。

我们利用Moore状态机和Mealy状态机来分别实现对输入序列数"1101"的检测

# 状态机设计实例 – 序列检测器设计

## Moore状态机序列检测器设计_1

**解题分析：**

如果现态是s0，输入为0，那么下一状态还是停留在s0；如果输入1，则转移到状态s1。

在状态s1，如果输入为0，则回到状态s0；如果输入为1，那么就转移到s2。

在s2状态，如果输入为1，则停留在状态s2；如果输入为0，那么下一状态为s3。

在s3状态，如果输入为1，则转移到状态s4，输出1；如果输入为0，则返回状态s0。

在s4状态，如果输入为0，回到初始状态s0；如果输入为1，下一状态为s1。

定义以下状态

s0:未检测到‘1’输入

s1:收到‘1’

s2:收到’11‘

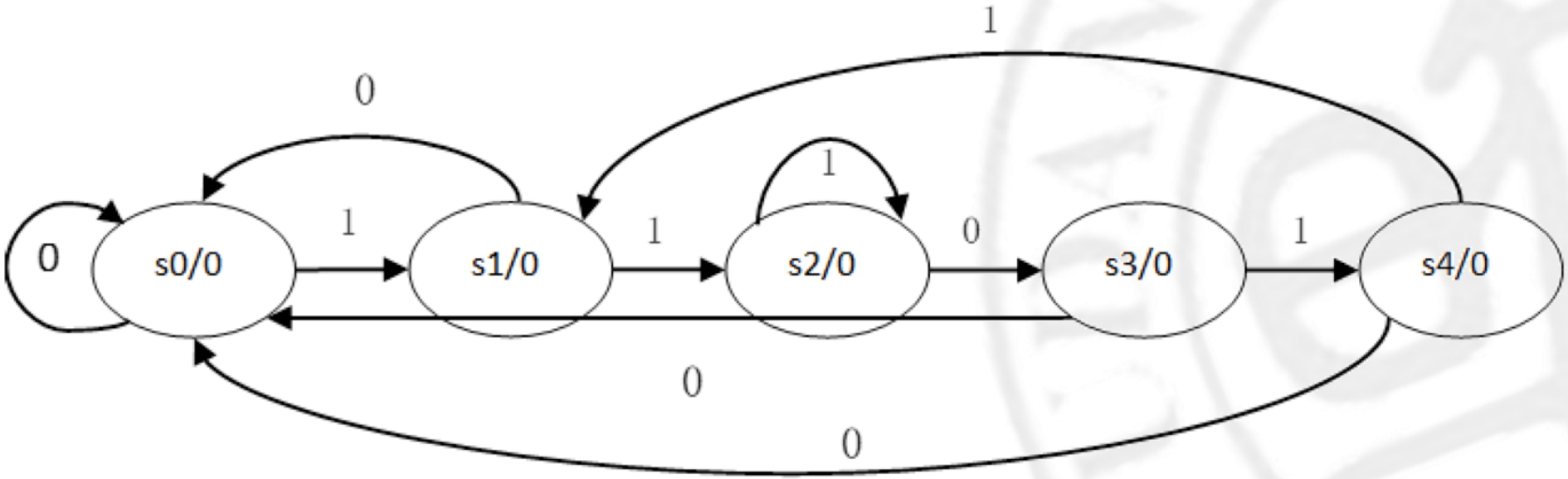
s3:收到‘110’

s4:收到‘1101

# 状态机设计实例 – 序列检测器设计

**Moore状态机序列检测器设计_2**

状态转移图：



# 状态机设计实例 – 序列检测器设计

## Moore状态机序列检测器设计_3

状态声明代码

```
localparam[ 2:0 ]
  s0 = 3'b000 ,
  s1 = 3'b001 ,
  s2 = 3'b010 ,
  s3 = 3'b011 ,
  s4 = 3'b100;
```

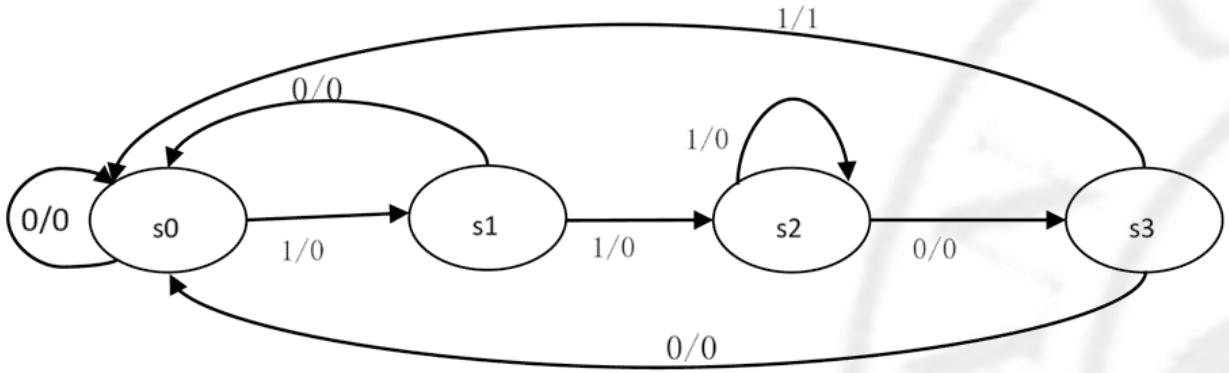
次级逻辑代码

```
    case  (cs )
      s0:
        if (din == 1'b1 ) nst = s1 ;
else nst = s0 ;
      s1:
        if (din == 1'b1 ) nst = s2 ;
else nst = s0 ;
      s2:
        if (din == 1'b0 ) nst = s3 ;
else nst = s2 ;
      s3:
        if (din == 1'b1 ) nst = s4;
else nst = s0 ;
      s4:
        if (din == 1'b0 ) nst = s1 ;
else nst = s0;
     default : nst = s0 ;
endcase
```

# 状态机设计实例 – 序列检测器设计

## Mealy状态机序列检测器设计_1

状态转移图：




对比Mealy状态机与Moore状态机的状态图可知：

Moore状态机的检测结果输出是与时钟同步的；而Mealy状态机的检测结果输出是异步的，当输入发生变化时，输出就立即变化。因此Mealy状态机的输出比Moore状态机状态的输出提前一个周期。

# 状态机设计实例 – 序列检测器设计

## Mealy状态机序列检测器设计_2

状态声明代码

```
localparam[ 1:0 ]
  s0 = 2'b00 ,
  s1 = 2'b01 ,
  s2 = 2'b10 ,
  s3 = 2'b11;
```

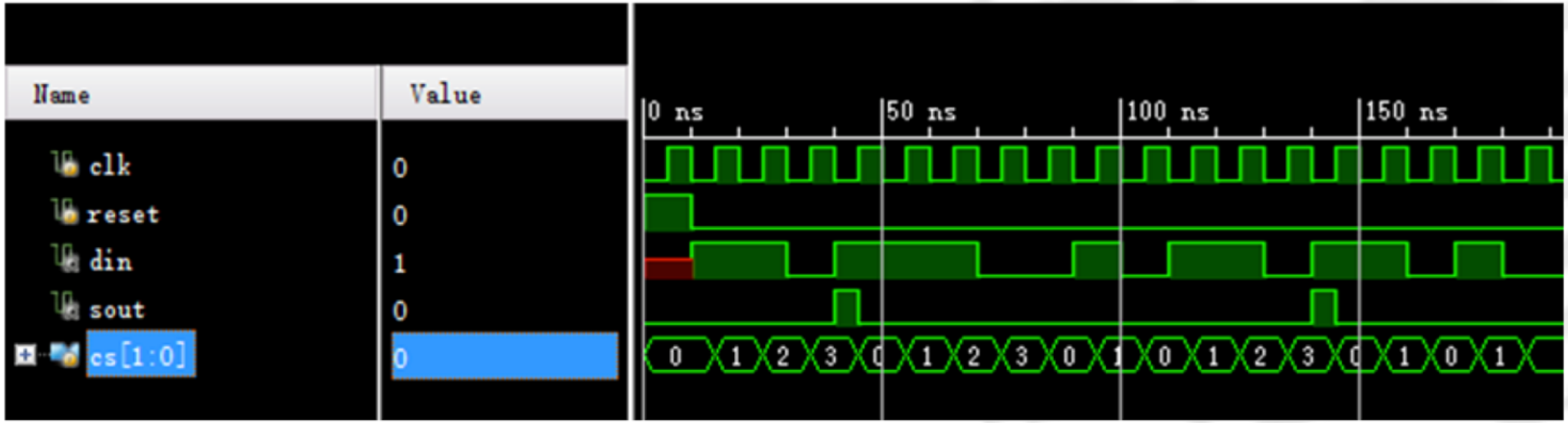
次级逻辑代码

```
case  (cs )
  s0:
    if (din == 1'b1 ) nst = s1 ;
else nst = s0 ;
  s1:
    if (din == 1'b1 ) nst = s2 ;
else nst = s0 ;
  s2:
    if (din == 1'b0 ) nst = s3 ;
else nst = s2 ;
  s3: nst = s0 ;
 default : nst = s0 ;
endcase
```

# 状态机设计实例 – 序列检测器设计

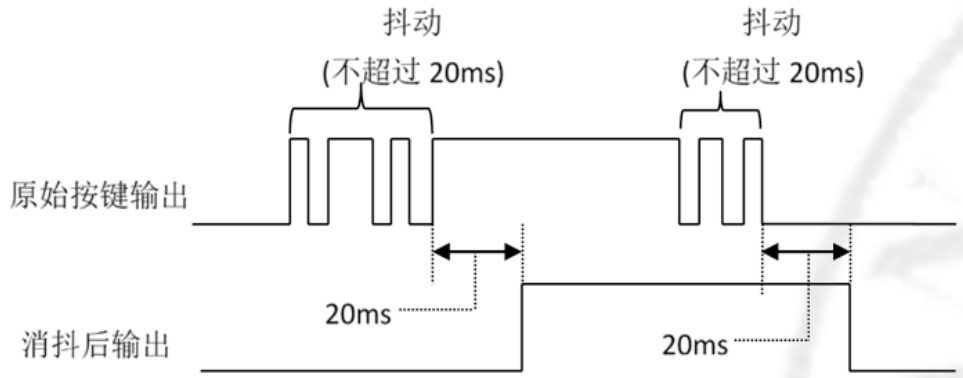
**Mealy状态机序列检测器设计_3**

仿真结果：



# 状态机设计实例 – 按键消抖电路设计

## 按键消抖电路设计_1




基于FSM的设计消抖电路，利用一个10ms的非同步定时器和有限状态机，计时器每10ms产生一个滴答使能周期信号，有限状态机利用此信号来确定输入信号是否稳定。

有限状态机将消除时间较短的抖动，当输入信号稳定20ms以后才改变去抖动以后的输出值。

# 状态机设计实例－按键消抖电路设计

## 按键消抖电路设计_2

解题分析：

1. 假定系统的起始态是zero(one)态，当sw变为1(0)时，系统转换为wait1_1态。
2. 当处于wait1_1态时，有限状态机处于等待状态并将m_tick置为有效电平态。若sw变为0则表示1值所持续的时间过短有限状态机返回zero态。
3. 这个动作在wait1_2态和wait1_3态也将再重复2次。

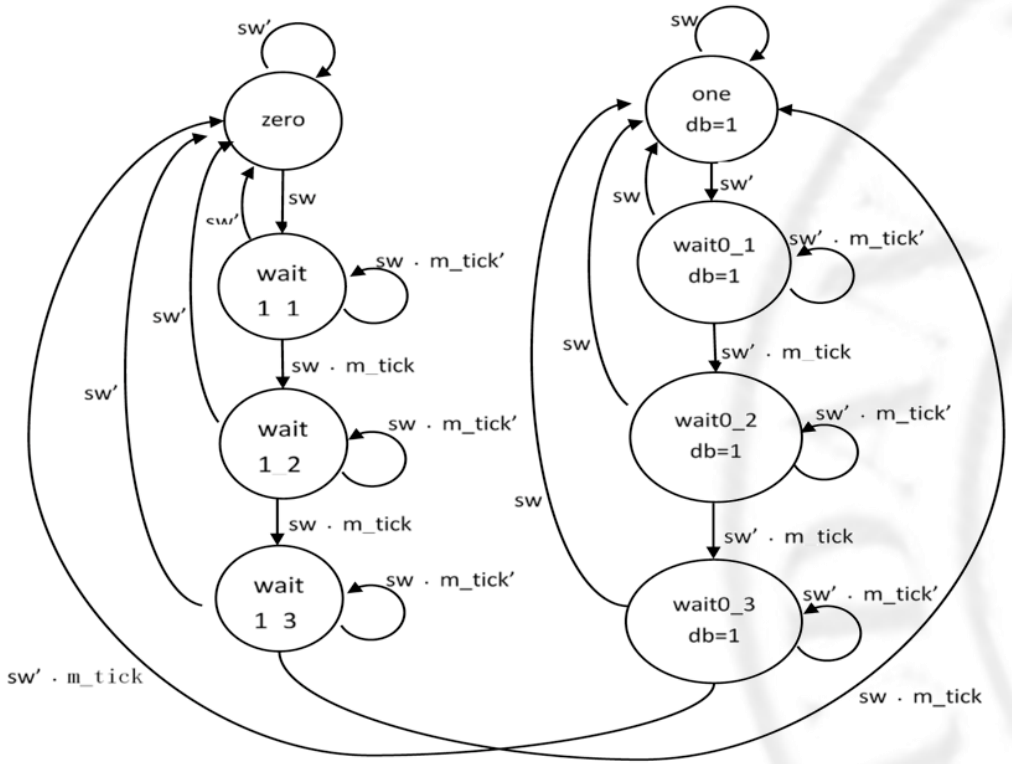
zero态： sw稳定在0值

one态：sw稳定在1值。

# 状态机设计实例 – 按键消抖电路设计

## 按键消抖电路设计_2

状态转移图：



# 状态机设计实例 – 按键消抖电路设计

## 按键消抖电路设计_3

状态声明代码

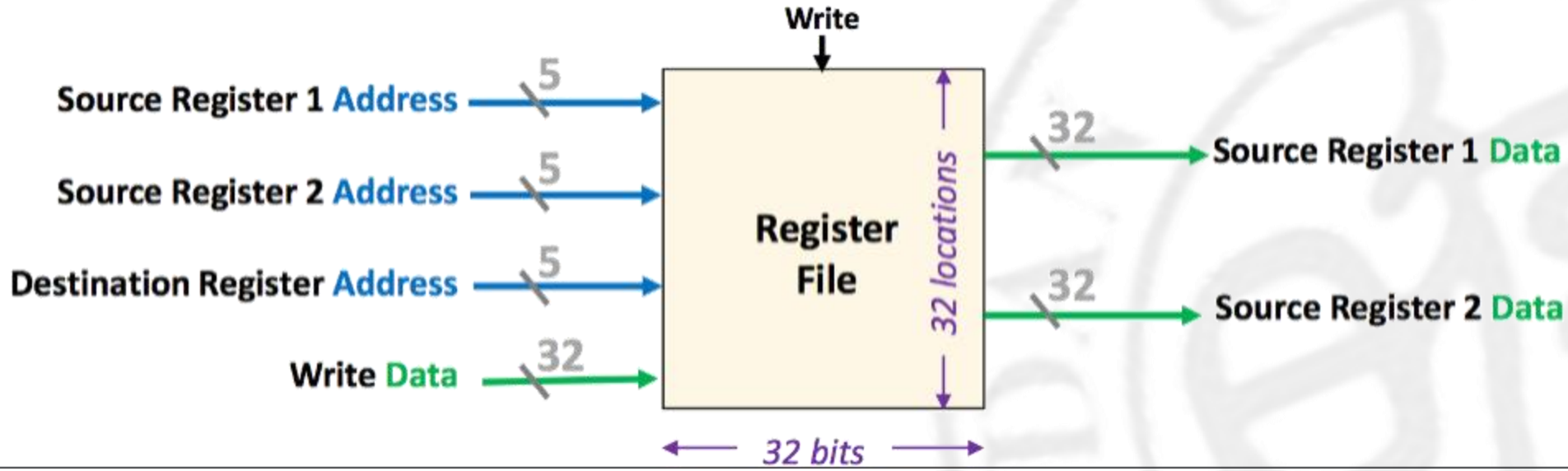
```
localparam[ 2:0 ]
  zero   = 3'b000 ,
  wait1_1 = 3'b001 ,
  wait1_2 = 3'b010 ,
  wait1_3 = 3'b011 ,
  one    = 3'b100 ,
  wait0_1 = 3'b101 ,
  wait0_2 = 3'b110 ,
  wait0_3  =  3'b111 ;
```

状态转移代码

```
case (state_reg )
  zero :
    if (sw )
        state_next = wait1_1 ;
  wait1_1 :
    if ( ~sw )
      state_next = zero ;
    else if (m_tick )
      state_next = wait1_2;
  wait1_2 :
    if ( ~sw )
      state_next = zero ;
    else if (m_tick )
      state_next = wait1_3 ;
  wait1_3 :
    if ( ~sw )
      state_next = zero ;
    else if (m_tick )
      state_next = one ;
  one :
    begin
      db = 1'b1 ;
      if ( ~sw )
      state_next = wait0_1 ;
    end
  wait0_1 :
    begin
      db = 1'b1 ;
      if(sw)
        state_next =one ;
      else if(m_tick )
        state_next = wait0_2;
    end
  wait0_2 :
    begin
      db = 1'b1 ;
      if(sw)
       state_next =one ;
      else if(m_tick )
        state_next = wait0_3 ;
    end
  wait0_3 :
    begin
      db = 1'b1 ;
      if(sw)
        state_next =one ;
      else if(m_tick )
       state_next = zero ;
    end
  default :state_next = zero ;
endcase
```

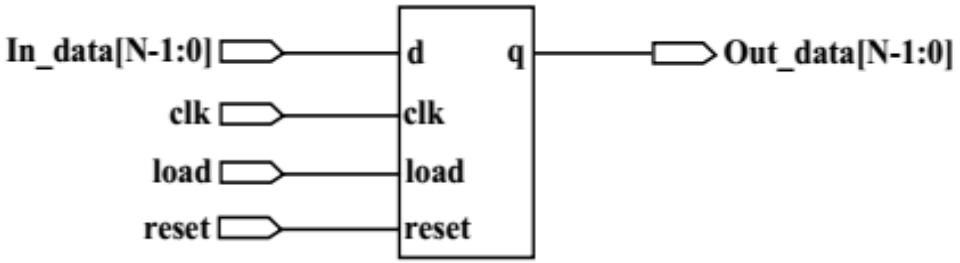
# 寄存器列表 Register File

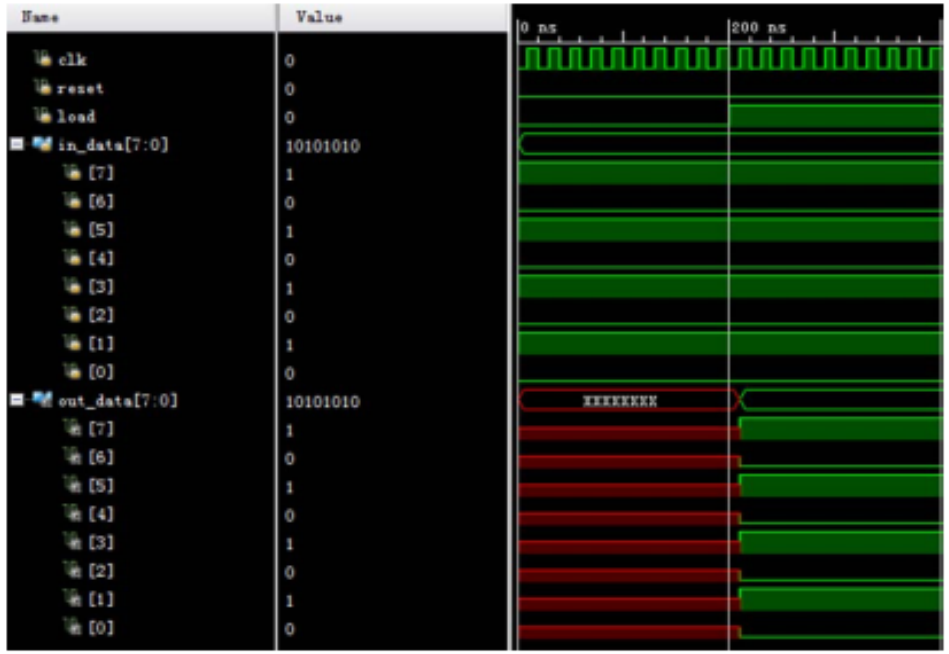
- 寄存器是RISC (load-store 架构)的核心单元
  - ALU 计算的输入、输出仅能是寄存器列表



# N位寄存器的功能描述

- 设计一个N位寄存器，它可以在需要时从输入线in_data加载一个值，我们给D触发器增加一根输入线load，当我们想要从in_data加载一个值时，就把load设置为1，那么在下一个时钟上升沿，in_data的值将被存储在q中。



# N位寄存器的Verilog功能

```
module reg_N
 #(parameter N = 8)
   (
      input  clk,
      input  reset,
      input [N-1:0] in_data,
      input   load,
      output  reg [N-1:0] out_data
   );
always @(posedge clk, posedge reset)
      if(reset)
            out_data <= 0;
      else if(load == 1)
            out_data <= in_data;
   endmodule
```

- 如果我们想修改寄存器的位宽，可以使用Verilog的实例化语句。我们可以实现一个如下所示的16位寄存器，称为fReg。

```
reg_N #(
       .N(16))
      fReg(.clk(clk),
      .reset(reset),
      .load(load),
      .in_data(indata),
      .out_data(out_data)
);
```

# 寄存器组

- 寄存器组是由一组拥有同一个输入端口和一个或多个输出端口的寄存器组成。写地址信号w_addr指定了数据存储位置，读取地址信号r_addr指定数据检索位置。
- 常用于快速、临时存储。

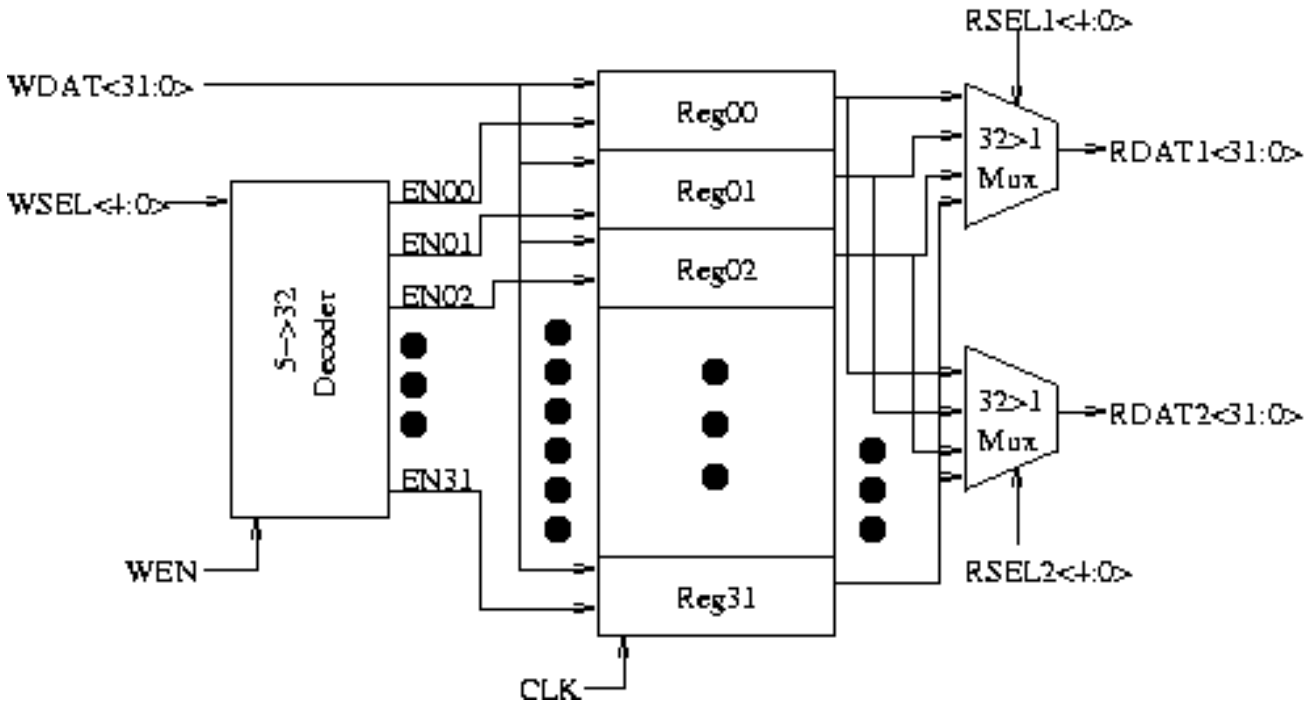
```
module reg_file
  #(
parameter N = 8，//比特数
W = 2) //地址比特数
      (
        input  clk,
        input  wr_en,
        input [W-1:0] w_addr,r_addr,
        input [BN-1:0]  w_data,
        output  [BN-1:0]  r_data
    );
     reg [BN-1:0] array_reg[2**W-1:0];
```

```
always @(posedge clk)
      if(wr_en)
        array_reg[w_addr] <= w_data;
        assign r_data = array_reg[r_addr];
  endmodule
```

一个信号被用作索引来访问数组中元素

**二维数组的数据类型：**表示array_reg变量是一个含有[2**W-1:0]个元素的数组，每个元素的数据类型是reg [B-1:0]

# 寄存器列表的Verilog描述




**Register File (2 read ports, 1 write port) Pg.2**

```
module regfile(input    clk,
               input    we3,
               input  [4:0]  ra1, ra2, wa3,
               input  [31:0] wd3,
               output [31:0] rd1, rd2);

reg [31:0] rf [31:0];                    //2-D (32x32) array

always @(posedge clk)
  if (we3) rf[wa3] <= wd3;                //write

assign rd1 = (ra1 != 0) ? rf[ra1] : 0;    //read 1
assign rd2 = (ra2 != 0) ? rf[ra2] : 0;    //read 2

Endmodule        // Note: rf[0] is always 0!
```

width

depth

outside of always

# 地址译码器

- 3-8译码器

```
module Vr74x138a(G1, G2A_L, G2B_L, A, Y_L);
  input G1, G2A_L, G2B_L;
  input [2:0] A;
  output [0:7] Y_L;
  reg [0:7] Y_L;

  always @ (G1 or G2A_L or G2B_L or A) begin
    if (G1 & ~G2A_L & ~G2B_L)
      case (A)
        0: Y_L = 8'b01111111;
        1: Y_L = 8'b10111111;
        2: Y_L = 8'b11011111;
        3: Y_L = 8'b11101111;
        4: Y_L = 8'b11110111;
        5: Y_L = 8'b11111011;
        6: Y_L = 8'b11111101;
        7: Y_L = 8'b11111110;
        default: Y_L = 8'b11111111;
      endcase
    else Y_L = 8'b11111111;
  end
endmodule
```