

# 第 11 章

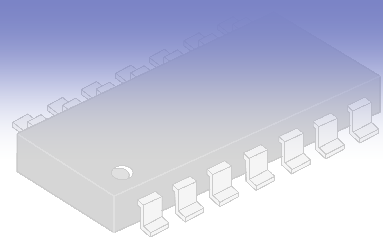
## 时序电路设计

# 内容



- ❑ 时序电路
- ❑ 锁存器和触发器
- ❑ 同步时序电路
- ❑ 异步时序电路
- ❑ 寄存器
  - ❑ 多位寄存器
  - ❑ 移位寄存器
- ❑ 存储器
- ❑ 计数器
  - ❑ 计数器
  - ❑ 可逆十进制计数器
- ❑ 状态机
- ❑ 序列检测器
- ❑ 并行/串行转换

# 时序逻辑电路



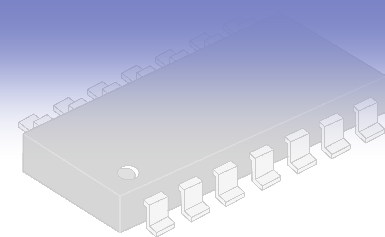
## □ 组合逻辑系统

- 输出完全由当前输入决定——一旦输入信号变化，输出随之改变。

## □ 时序逻辑电路

- 输出不仅取决于当前的输入，而且取决于过去的输入
- 过去的输入值
  - ◆ 系统内部的存储元件保持
  - ◆ 基本的存储元件——锁存器（latch）和 触发器（flip-flop）
- 时序——按照发生的先后顺序，将一个接一个的事件按时间排序

# 组合逻辑与时序逻辑



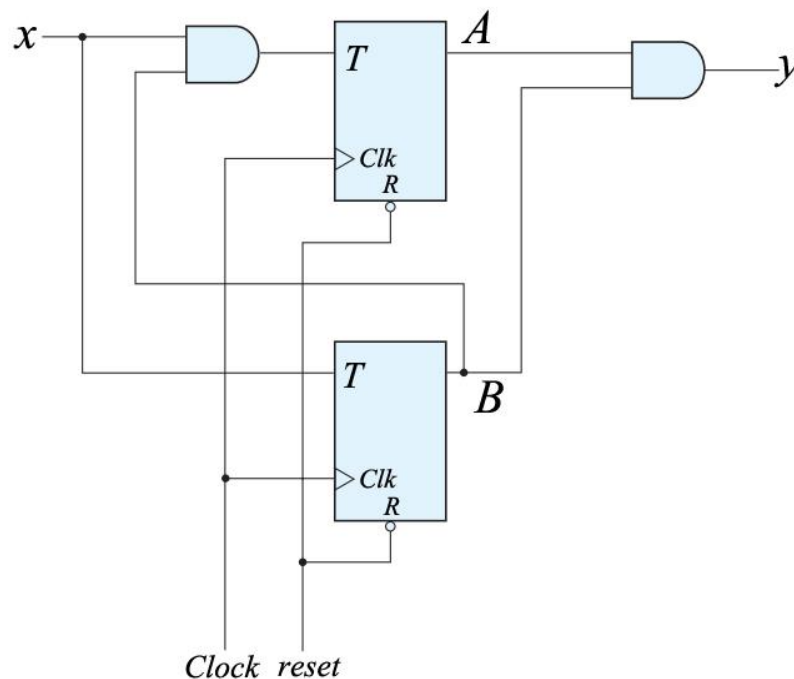
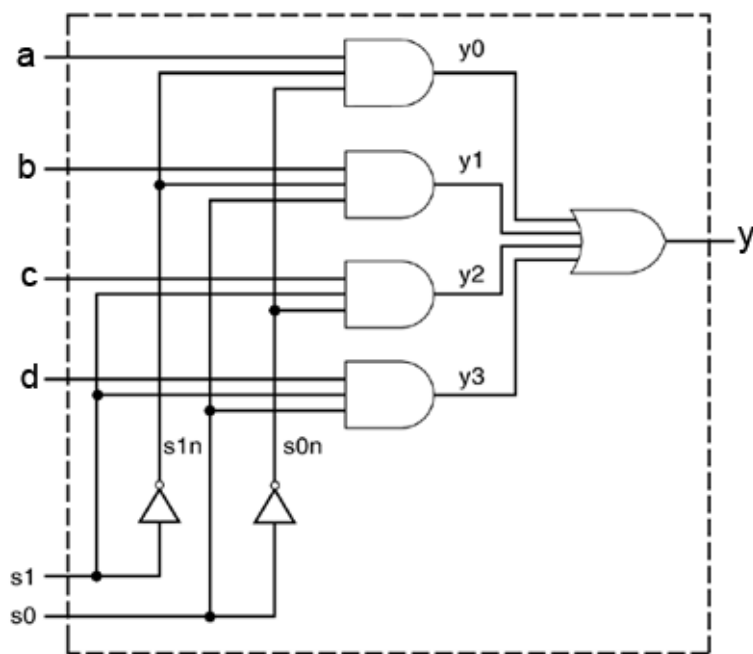
## □ 组合逻辑

- 输出完全由当前输入决定，一旦输入信号变化，输出随之改变

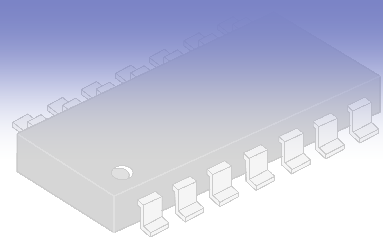
## □ 时序逻辑

- 输出不仅取决于当前的输入，而且取决于过去的输入
- 时序电路含有存储元件

## □ 例、



# 锁存器和触发器



□ 基本存储单元 —— 保持有两个稳定的状态 —— 0、1

□ 锁存器 (latch)

□ 电平敏感存储器

- ◆ 可以是任何输入信号的电平值
- ◆ 不一定是时钟信号

□ 常见的有两种 —— *SR* 锁存器、*D* 锁存器

□ 触发器

□ 由时钟跳变沿触发的存储器

□ 如果一个触发器对时钟的上升沿敏感，则其为上升沿触发

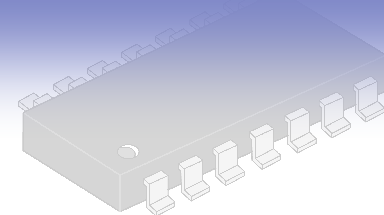
□ 如果一个触发器对时钟的下降沿敏感，则其为下降沿触发

□ 四类

- ◆ *SR* 触发器、*D* 触发器、*JK* 触发器、*T* 触发器

□ 最常用的触发器 —— *D* 触发器

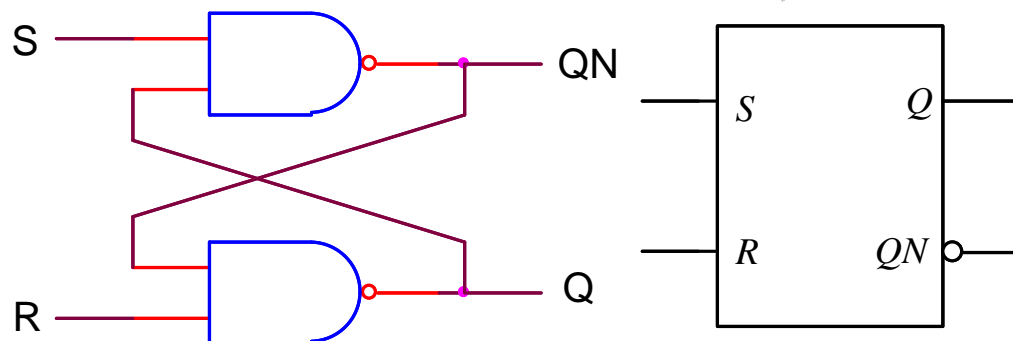
# S-R(set-reset) 锁存器



## □ 电平敏感存储器

## □ 两个输入

- S —— 置位 (set) 输入
- R —— 复位 (reset) 输入
- 当输出 Q 为逻辑 1 时
  - ◆ 锁存器 “置位”
- 当输出 Q 为逻辑 0 时
  - ◆ 锁存器 “复位”



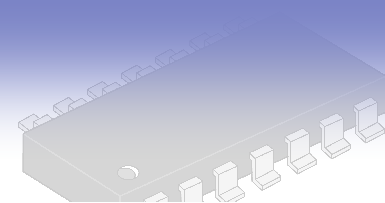
```
module SRLatch(output reg q, qbar,  
               input s, r );  
    always @(*) begin  
        case ({s,r})  
            2'b01: {q, qbar} <= 2'b01;  
            2'b10: {q, qbar} <= 2'b10;  
            2'b11: {q, qbar} <= 2'bx;  
            default ;  
        endcase  
    end  
endmodule
```

## □ RS锁存器真值表

S	R	Q	QN
0	0	Q	QN
0	1	0	1
1	0	1	0
1	1	X	X

能否使用阻塞赋值？

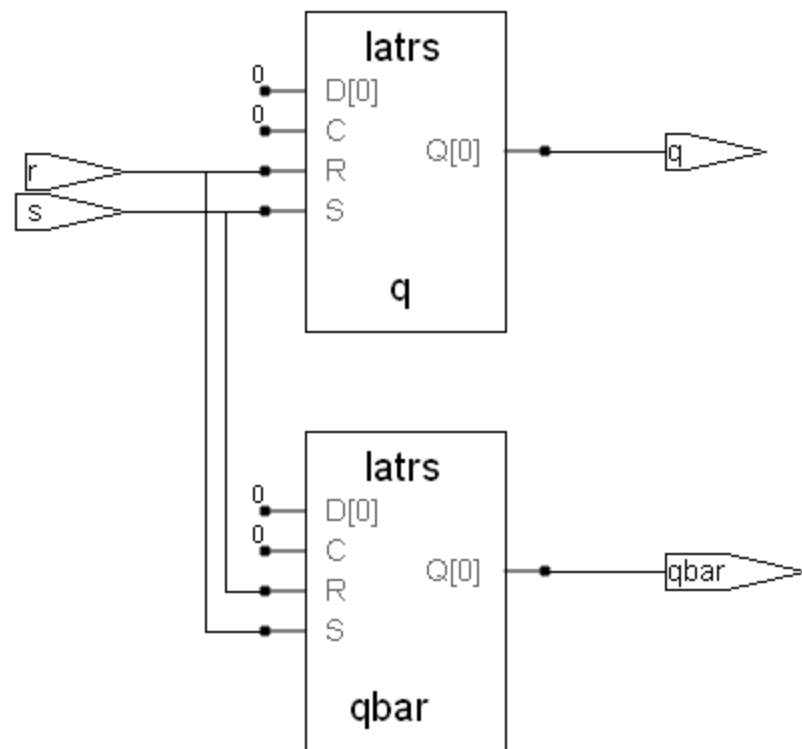
# S-R 锁存器设计综合结果



## 综合工具

- Synplify Pro 9.6.2
- FPGA:
  - ◆ Xilinx Spartan-3E XC3S500E

```
module SRLatch(output reg q, qbar,  
               input s, r );  
    always @(*) begin  
        case ({s,r})  
            2'b01: {q, qbar} <= 2'b01;  
            2'b10: {q, qbar} <= 2'b10;  
            2'b11: {q, qbar} <= 2'bx;  
            default ;  
        endcase  
    end  
endmodule
```



# S-R(set-reset) 锁存器仿真测试

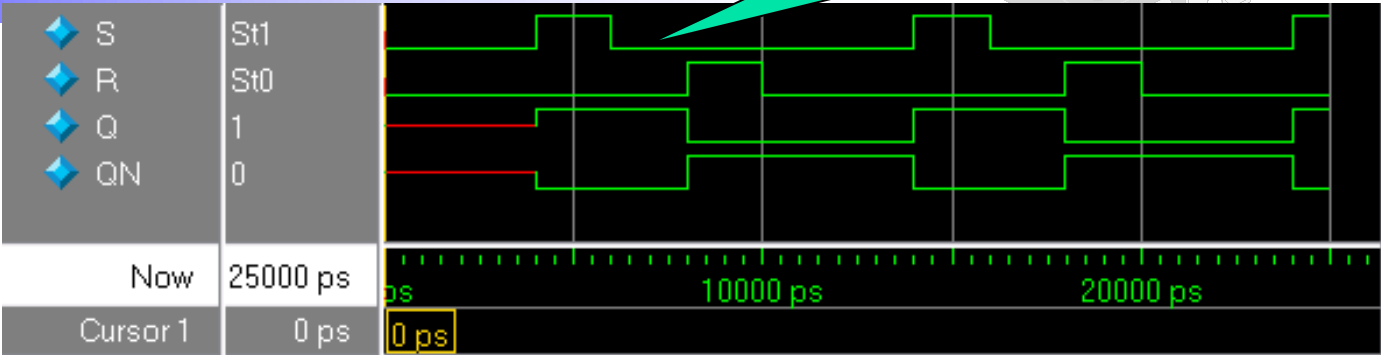
记忆最终输入为 1 的输入

```
`timescale 1ns / 100ps
`include "SRLatch.v"
module srlatch_tb();
  wire p_q, p_qbar;
  reg p_s, p_r;
```

```
  initial begin
    p_s = 0;
    forever begin
      #8 p_s = 1'b1; #2 p_s = 1'b0;
    end
  end

  initial begin
    p_r = 0;
    forever begin
      #8 p_r = 1'b1; #2 p_r = 1'b0;
    end
  end
end
```

```
SRLatch u0(.q(p_q), .qbar(p_qbar), .s(p_s), .r(p_r));
initial
  $monitor( "At time %t, S = %b, R = %b, Q = %b, QN = %b",
    $time, p_s, p_r, p_q, p_qbar);
endmodule
```

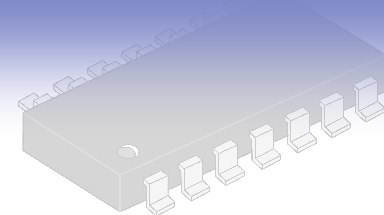


```
module SRLatch(output reg q, qbar,
               input s, r );
  always @(*) begin
    case ({s,r})
      2'b01: {q, qbar} <= 2'b01;
      2'b10: {q, qbar} <= 2'b10;
      2'b11: {q, qbar} <= 2'bx;
      default ;
    endcase
  end
endmodule
```

S	R	Q	QN
0	0	Q	QN
0	1	0	1
1	0	1	0
1	1	X	X



# D 锁存器

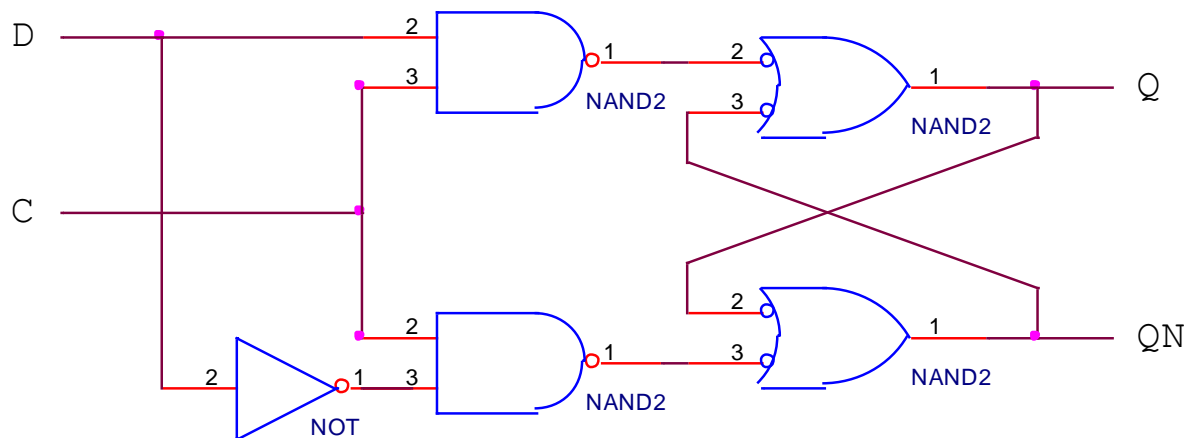
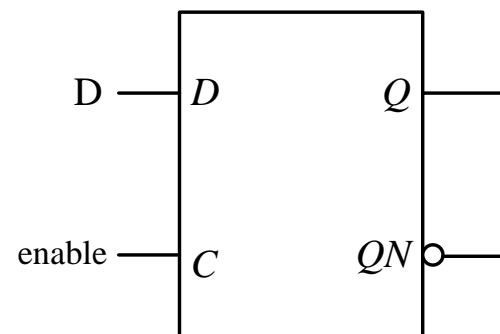


## □ 存储一根信号线上的二进制位

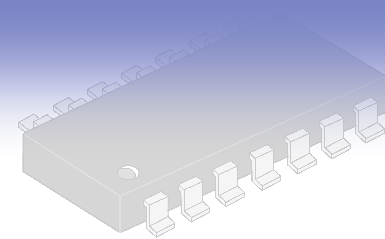
- 在具有使能端的 S-R 锁存器的基础上，通过一个反相器，由一个 D 输入产生
- D —— 数据输入端
- C —— 使能端，当  $C = 1$  时有效

D 锁存器真值表（功能表）

C	D	Q	QN
1	0	0	1
1	1	1	0
0	X	上一个 Q 值	上一个 QN 值



# D 锁存器的 Verilog 模块和仿真波形



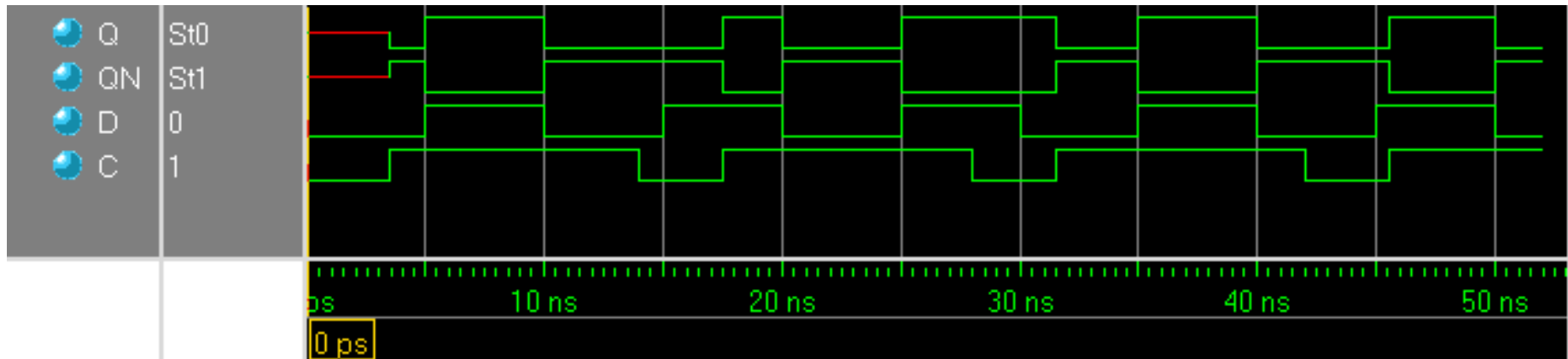
```
// D Latch
module DLatch(output reg Q, output QN,
              input D, input C );

    always @(*)
        if (C) Q <= D;

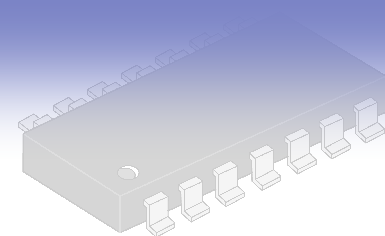
    assign QN = ~Q;
endmodule
```

D 锁存器真值表（功能表）

C	D	Q	QN
1	0	0	1
1	1	1	0
0	X	上一个 Q 值	上一个 QN 值

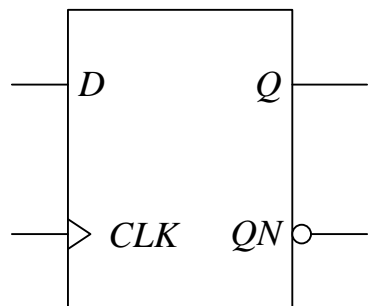


# 边沿触发式 D 触发器





## □ 正边沿触发式D触发器（positive-edge-triggered flip-flop）

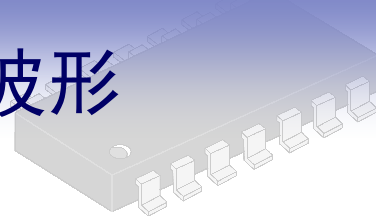
- 在控制时钟上升沿到来的时刻，采样D输入信号，并据此改变Q和QN的输出
- 保持最后一次储存的值
- 现代电路设计最常用的触发器



正边沿触发式 D 触发器真值表（功能表）

D	CLK	Q	QN
0		0	1
1		1	0
X	0	上一个 Q 值	上一个 QN 值
X	1	上一个 Q 值	上一个 QN 值



# 正边沿触发式D触发器的Verilog 模块及仿真波形



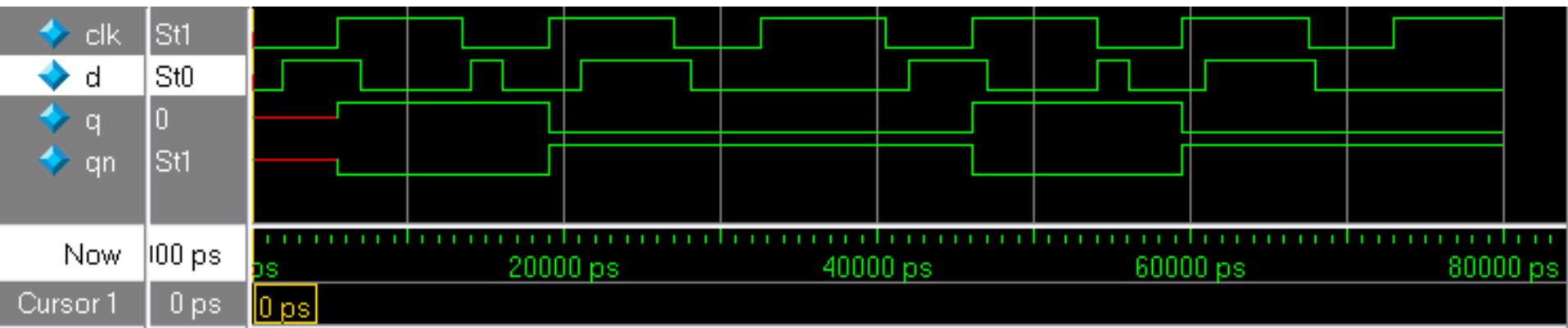
```
// Positive-edge_triggered D flip-flop
module dff (output reg q, output qn,
            input d, input clk);

    always @(posedge clk) q <= d;
    assign qn = ~q;
endmodule
```

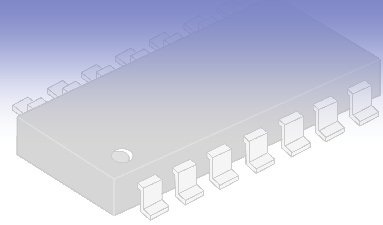
正边沿触发式D触发器真值表（功能表）

D	CLK	Q	QN
0		0	1
1		1	0
X	0	上一个Q 值	上一个QN 值
X	1	上一个Q 值	上一个QN 值

保持最后一次储存的值



# D 触发器与 D 锁存器



## □ D触发器

```
// Positive-edge_triggered D flip-flop
module dff (output reg q, output qn,
            input  d, input  clk);

    always @(posedge clk) q <= d;
    assign qn = ~q;
endmodule
```

## □ D锁存器

```
// D Latch
module DLatch(output reg Q, output QN,
              input D, input C );

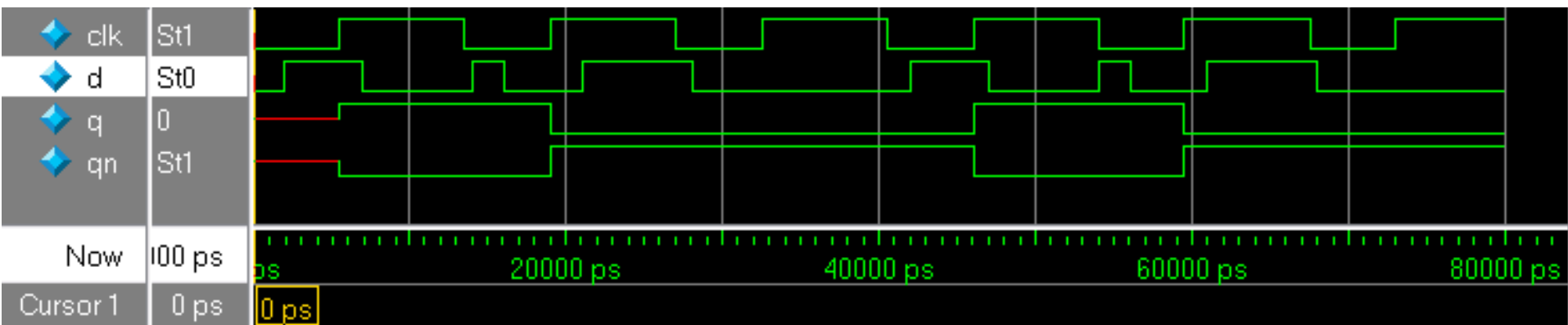
    always @(*)
        if (C) Q <= D;

    assign QN = ~Q;
endmodule
```

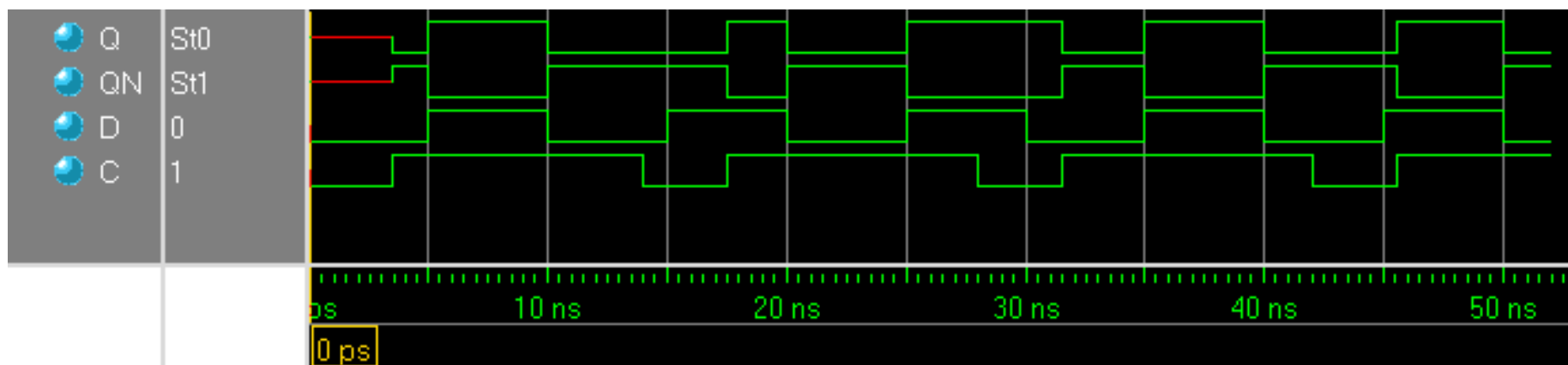
# D 触发器与 D 锁存器



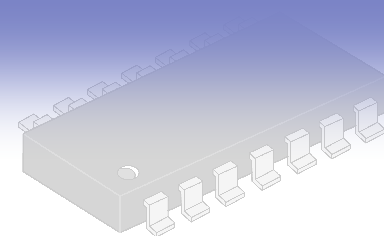
## □ D 触发器



## □ D 锁存器





# 负边沿触发式 D 触发器

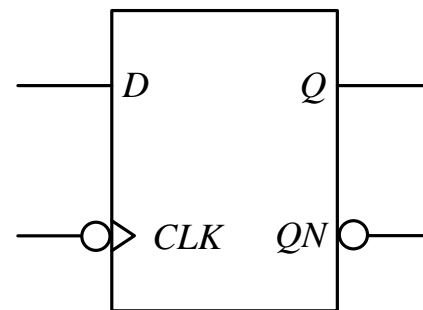


## ❑ 负边沿触发式D触发器（negative-edge-triggered flip-flop）

- ❑ 当下降沿到来时，采样D输入信号，并据此改变Q和QN的输出

负边沿触发式 D 触发器真值表（功能表）

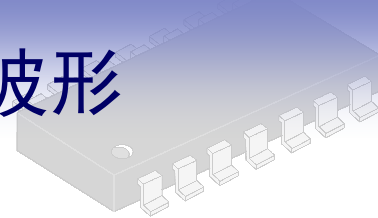
D	CLK	Q	QN
0		0	1
1		1	0
X	0	上一个 Q 值	上一个 QN 值
X	1	上一个 Q 值	上一个 QN 值



```
// Negative-edge_triggered D flip-flop
module dff_ne( output reg q, output qn,
               input  d, input  clk);

    always @(negedge clk) q <= d;
    assign qn = ~q;
endmodule
```

# 负边沿触发式D触发器的Verilog 模块及仿真波形



```
`timescale 1ns / 100ps
```

```
`include "dff_ne.v"
```

```
module dff_ne_tb;
```

```
    wire p_q, p_qn;
```

```
    reg p_d, p_clk;
```

```
    initial begin
```

```
        p_d = 0;
```

```
        forever begin
```

```
            #2 p_d = 1'b1;
```

```
            #5 p_d = 1'b0;
```

```
            #7 p_d = 1'b1;
```

```
            #2 p_d = 1'b0;
```

```
            #5 p_d = 1'b1;
```

```
            #7 p_d = 1'b0;
```

```
            #12 p_d = 1'b0;
```

```
        end
```

```
    end
```

```
    initial begin
```

```
        p_clk = 0;
```

```
        forever begin
```

```
            #5.5 p_clk = 1'b1;
```

```
            #8 p_clk = 1'b0;
```

```
        end
```

```
    end
```

```
    dff_ne dff_ne_inst(.q(p_q), .qn(p_qn), .d(p_d), .clk(p_clk));
```

```
    initial
```

```
        $monitor( "At time %t, D = %b, CLK = %b, Q = %b, QN = %b",
```

```
            $time, p_d, p_clk, p_q, p_qn);
```

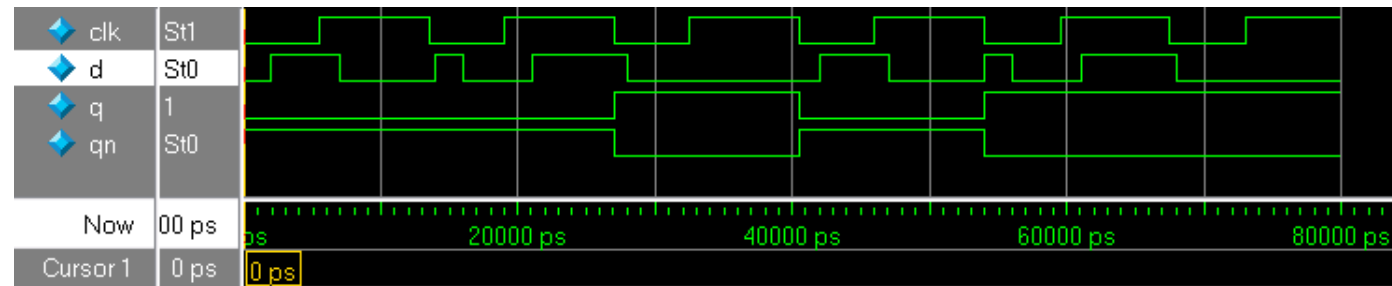
```
endmodule
```

```
// Negative-edge_triggered D flip-flop
module dff_ne( output reg q, output qn,
               input d, input clk);
```

```
    always @(negedge clk) q <= d;
```

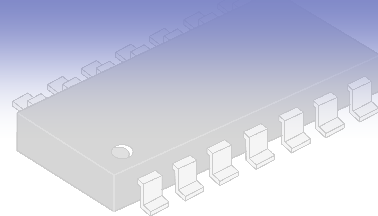
```
    assign qn = ~q;
```

```
endmodule
```





# 具有使能端的正边沿触发式 D 触发器

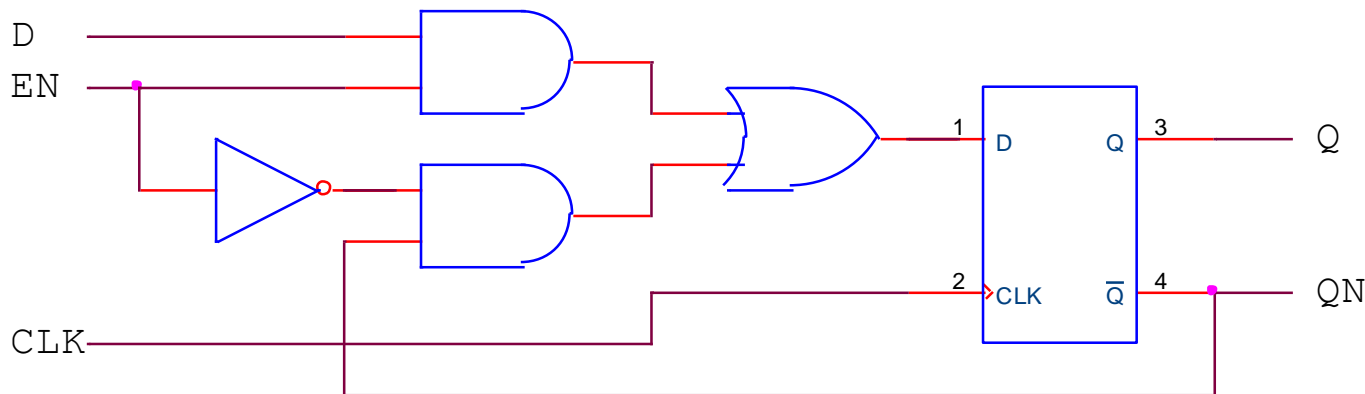
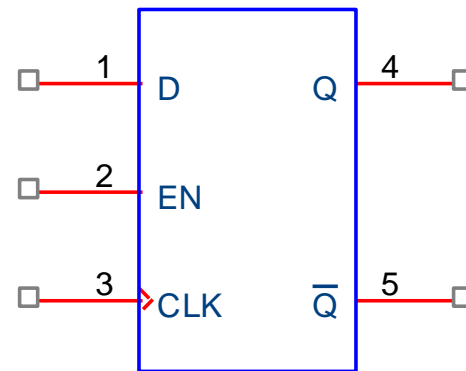


## 增加一个使能输入 (enable input) : EN

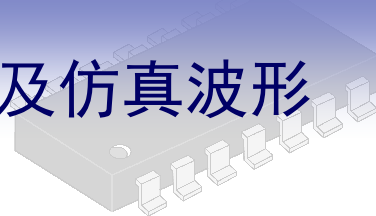
- 如果 EN 有效, 选择的是外部 D 端的输入
- 如果 EN 无效, 选择的是触发器现在的输出

具有使能端的正边沿触发式 D 触发器真值表

D	EN	CLK	Q	QN
0	1		0	1
1	1		1	0
X	0		上一个 Q 值	上一个 QN 值
X	X	0	上一个 Q 值	上一个 QN 值
X	X	1	上一个 Q 值	上一个 QN 值



# 具有使能端的正边沿触发式 D 触发器的Verilog 模块及仿真波形






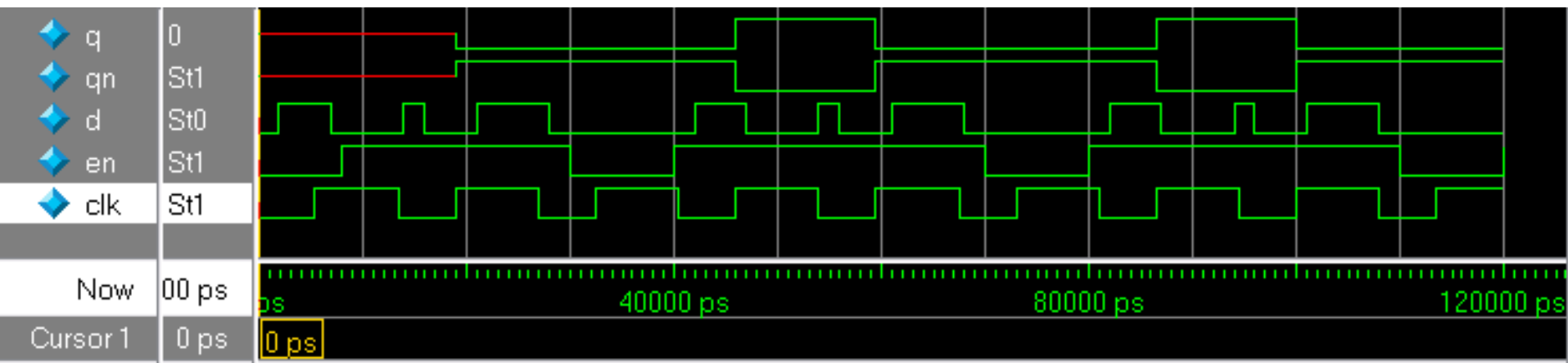
```
module dff_en(output reg q, output qn,  
             input d, input en, input clk);
```

```
    always @(posedge clk)  
        if ( en ) q <= d;
```

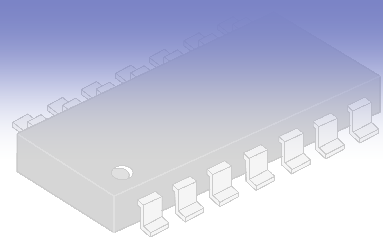
```
    assign qn = ~q;  
endmodule
```

具有使能端的正边沿触发式 D 触发器真值表

D	EN	CLK	Q	QN
0	1		0	1
1	1		1	0
X	0		上一个 Q 值	上一个 QN 值
X	X	0	上一个 Q 值	上一个 QN 值
X	X	1	上一个 Q 值	上一个 QN 值



# 正边沿触发式 J-K' 触发器



正边沿触发式 J-K' 触发器真值表

J	K	CLK	Q	QN
X	X	0	上一个Q值	上一个QN值
X	X	1	上一个Q值	上一个QN值
0	0		上一个Q值	上一个QN值
0	1		0	1
1	0		1	0
1	1		上一个QN值	上一个Q值

```
module JKFF( output reg q, qn,  
             input clk, j, k);
```

```
    always @ (posedge clk)
```

```
        case ({j,k})
```

```
            2'b01:    {q, qn} <= 2'b01;
```

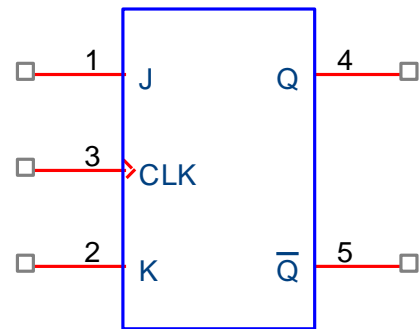
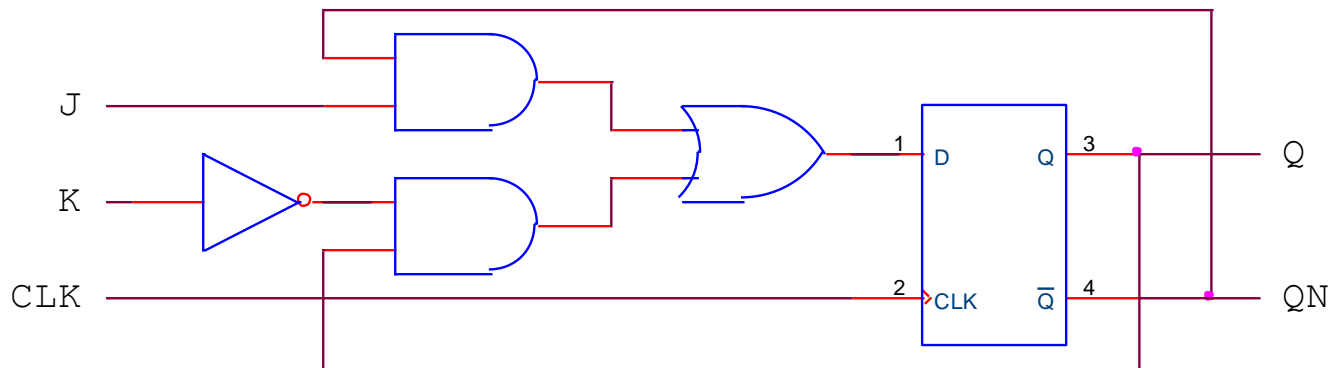
```
            2'b10:    {q, qn} <= 2'b10;
```

```
            2'b11:    {q, qn} <= {qn, q};
```

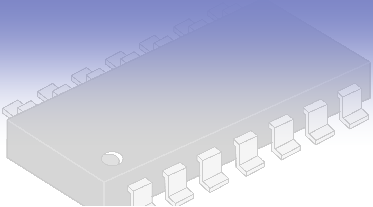
```
            default: ;
```

```
        endcase
```

```
    endmodule
```



# 正边沿触发式 J-K' 触发器仿真波形

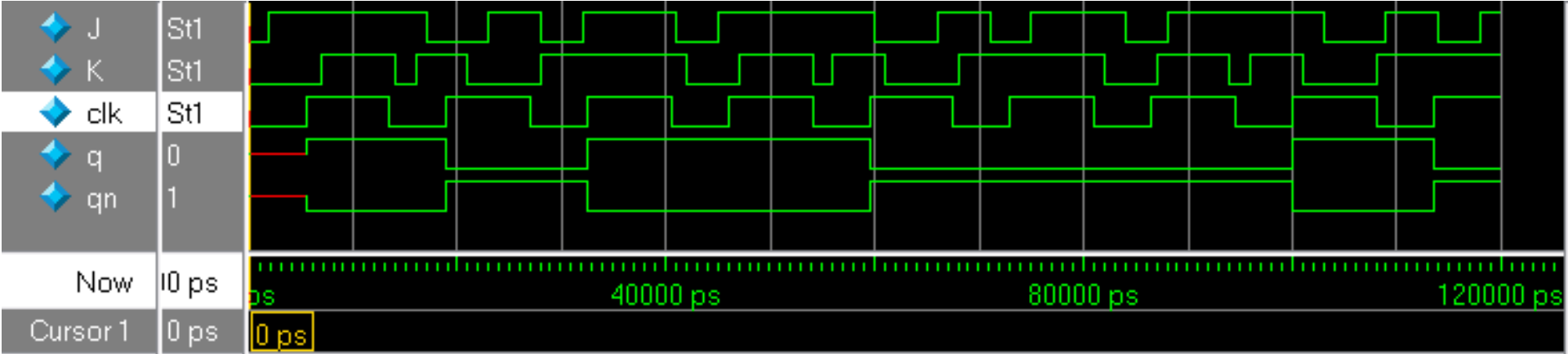


```
module JKFF( output reg q, qn,
             input clk, j, k);

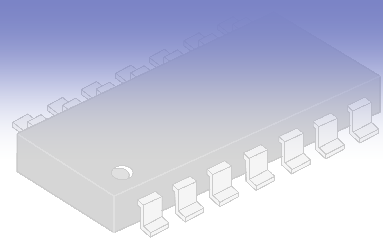
    always @ (posedge clk)
        case ({j,k})
            2'b01: {q, qn} <= 2'b01;
            2'b10: {q, qn} <= 2'b10;
            2'b11: {q, qn} <= {qn, q};
            default: ;
        endcase
endmodule
```

正边沿触发式 J-K' 触发器真值表

J	K	CLK	Q	QN
X	X	0	上一个 Q 值	上一个 QN 值
X	X	1	上一个 Q 值	上一个 QN 值
0	0		上一个 Q 值	上一个 QN 值
0	1		0	1
1	0		1	0
1	1		上一个 QN 值	上一个 Q 值



# 正边沿触发式 J-K' 触发器 (2)







```
module JK_FF ( output reg Q, output Q_b, input Clk, J, K );
    assign Q_b = ~ Q ;
    always @( posedge Clk)
        case ({J,K})
            2'b00: Q <= Q;
            2'b01: Q <= 1'b0;
            2'b10: Q <= 1'b1;
            2'b11: Q <= ~Q;
        endcase
endmodule
```

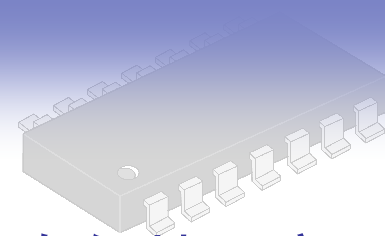
```
module JKFF( output reg q, qn, input clk, j, k);

    always @ (posedge clk)
        case ({j,k})
            2'b01: {q, qn} <= 2'b01;
            2'b10: {q, qn} <= 2'b10;
            2'b11: {q, qn} <= {qn, q};
            default: ;
        endcase
endmodule
```

正边沿触发式 J-K' 触发器真值表



J	K	CLK	Q	QN
X	X	0	上一个Q 值	上一个QN 值
X	X	1	上一个Q 值	上一个QN 值
0	0		上一个Q 值	上一个QN 值
0	1		0	1
1	0		1	0
1	1		上一个QN 值	上一个Q 值

# T 触发器



- T端有效时，在时钟边沿状态发生翻转，否则，状态保持不变

T 触发器的真值表

T	CLK	Q	QN
0		上一个 Q 值	上一个 QN 值
1		上一个 QN 值	上一个 Q 值

```
module TFF(output reg q, qn, input clk, rst_n, t);
```

```
    always @(posedge clk) begin
```

```
        if (~rst_n)
```

```
            {q, qn} <= 2'b01;
```

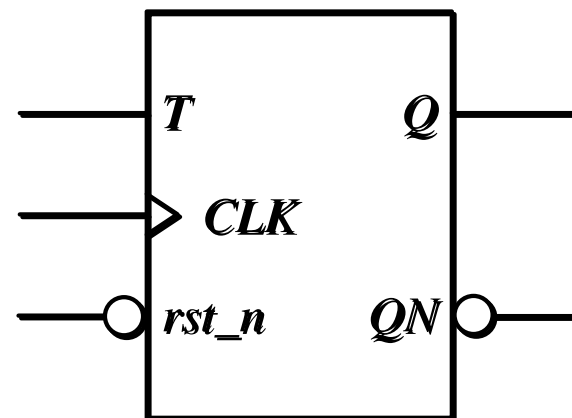
```
        else
```

```
            if (t)
```

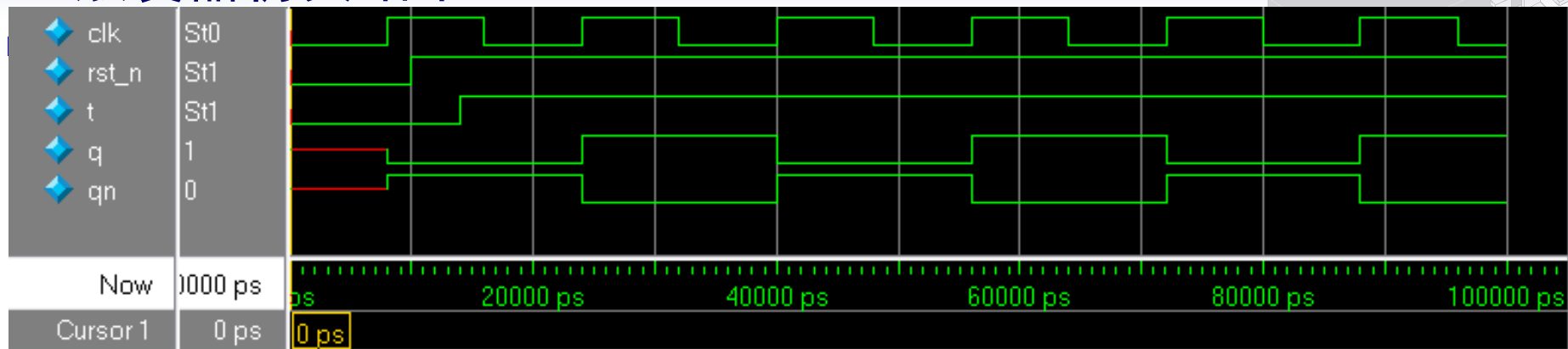
```
                {q, qn} <= {qn, q};
```

```
    end
```

```
endmodule
```



# T 触发器仿真结果



```

`timescale 1ns / 1ns
`include "TFF.v"
module TFF_tb;
    wire p_q, p_qn;
    reg p_clk, p_rst_n, p_t;

    initial begin
        p_rst_n = 1'b0; p_t = 1'b0;

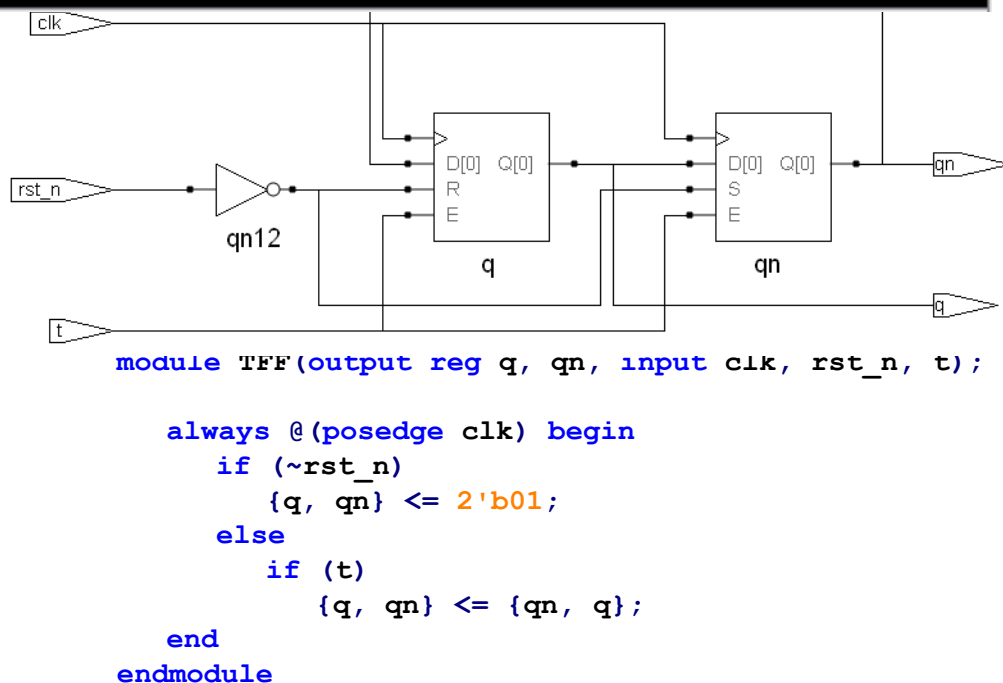
        #10 p_rst_n = 1'b1; #4 p_t = 1'b1;
    end

    initial begin
        p_clk = 1'b0;
        forever #8 p_clk = ~p_clk;
    end

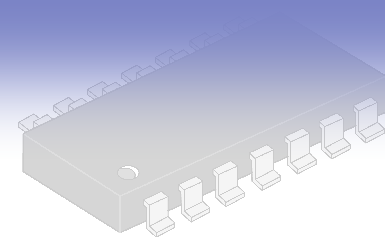
    TFF u0(.q(p_q), .qn(p_qn), .clk(p_clk), .rst_n(p_rst_n), .t(p_t));

    initial
        $monitor( "At time%t, T = %b,Q = %b, QN = %b", $time, p_t, p_q, p_qn);
endmodule

```



# 同步置位和复位



## □ 时序逻辑

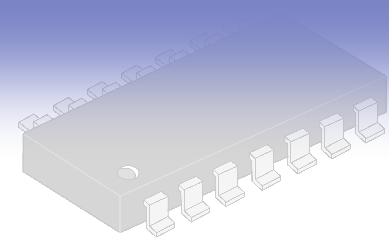
- 同步 —— 有时钟控制，根据同一个时钟信号改变状态
- 异步 —— 改变状态不需要使用同步时钟

## □ 同步

- 只有在时钟的有效跳变沿的时刻，置位和复位信号脉冲才能使触发器置位和复位
- **always** 语句的事件控制列表中不包括同步置位和复位信号
  - ◆ `always @(posedge CLK )`
  - ◆ `if (SET)`
  - ◆ `Q <= 1'b1;`
  - ◆ `else Q <= D;`
- **always** 语句块的执行只被有效时钟边沿控制
- 在**always** 语句块中首先检查 **set** 和 **reset** 信号的电平



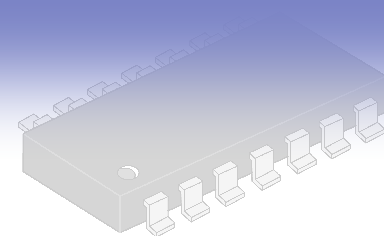
# 高电平有效同步置位和复位的事件控制举例



## □ 方法

```
◆ always @(posedge CLK)
◆     if (RESET)
◆         /* 复位输出 */
◆     else if (SET)
◆         /* 置位输出 */
◆         else
◆             case (state)
◆                 ...
◆                 /* 与时钟同步的逻辑 */
◆                 ...
◆             endcase
```

# 同步复位T触发器



## ❑ 复位信号低电平有效

❑ reset = 1'b0 时复位

```
module TFF(output reg q, qn, input clk, rst_n, t);
```

```
    always @(posedge clk) begin
```

```
        if (~rst_n)
```

```
            {q, qn} <= 2'b01;
```

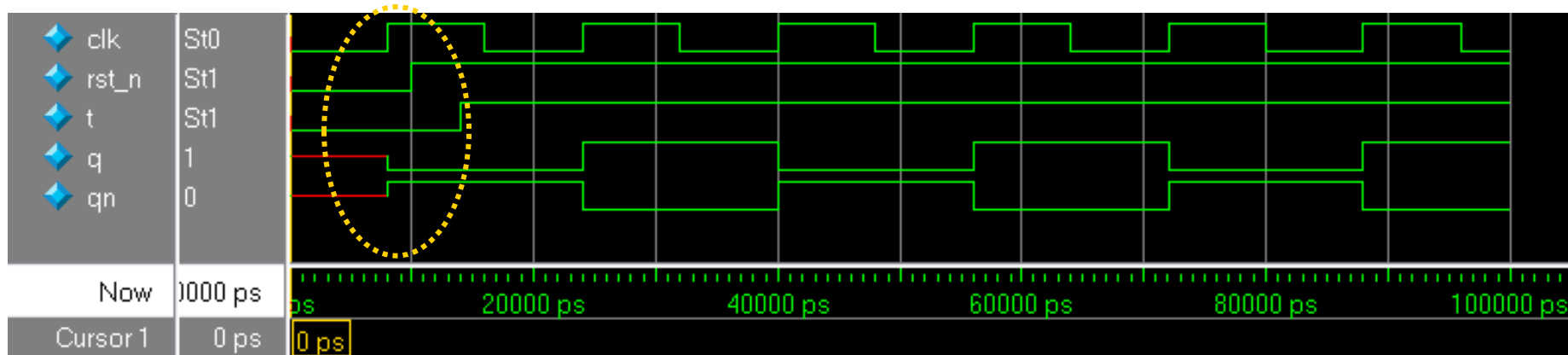
```
        else
```

```
            if (t)
```

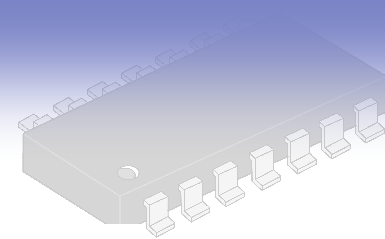
```
                {q, qn} <= {qn, q};
```

```
    end
```

```
endmodule
```

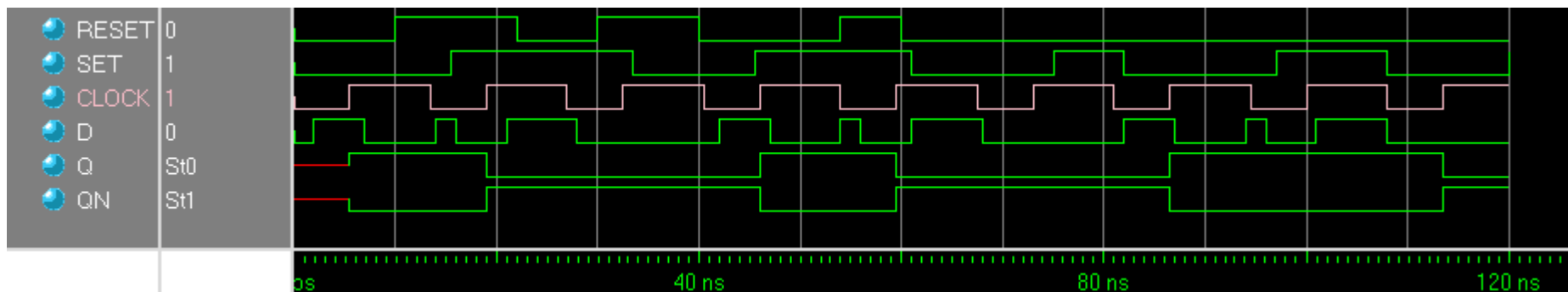


# 具有高电平有效同步置位和复位的 D 触发器

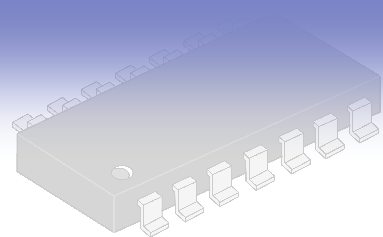


```
module DFF_sr( output reg q, qn,  
               input d, clk, reset, set );  
    always @(posedge clk)  
        if (reset)  
            {q, qn} <= 2'b01;  
        else if (set)  
            {q, qn} <= 2'b10;  
        else begin  
            {q, qn} <= {d, ~d};  
        end  
endmodule
```

- ❑ 复位优先于置位
- ❑ 置位优先于输入



# 异步置位的事件控制



## ❑ 异步——操作信号不受时钟脉冲（边沿）控制

- ❑ 当置位与复位脉冲到来时，立即将触发器的输出端置 1 或 0

## ❑ 异步高电平有效置位的方法

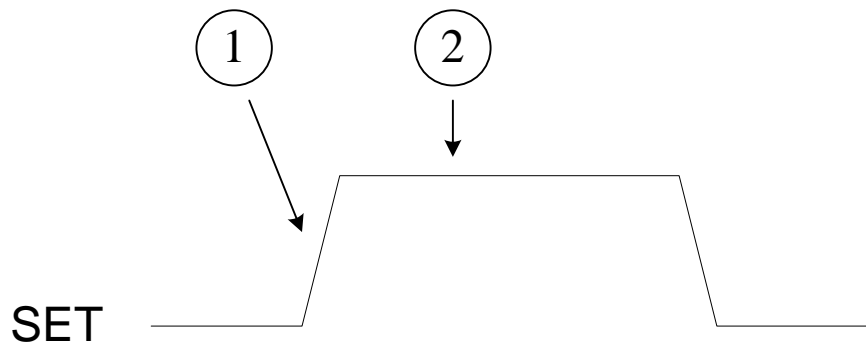
- ❑ 将置位和复位信号列入 `always` 语句的事件控制列表中，即：

```
◆ always @(posedge CLK or posedge SET)
◆     if (SET)
◆         Q <= 1'b1;
◆     else Q <= D;
```

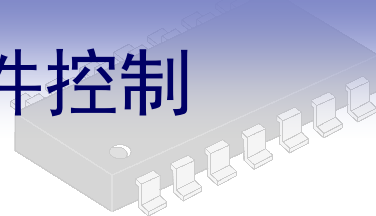
2

1

- ① 监测高电平置位的脉冲的到来 —— 监测正边沿
- ② 确定是否是高电平



# 异步高电平有效复位和低电平有效置位的事件控制



## □ 方法

```
◆ always @(posedge CLK or posedge RESET or negedge SET)
◆   if (RESET)
◆     /* 复位输出 */
◆   else if (!SET)
◆     /* 置位输出 */
◆     else
◆       case (state)
◆         ...
◆         /* 与时钟同步的逻辑 */
◆         ...
◆       endcase
```

- ① 监测高电平复位的脉冲的到来 —— 监测正边沿
- ② 监测低电平置位的脉冲的到来 —— 监测负边沿
- ③ 确定是否是高电平
- ④ 确定是否是低电平

# 高电平有效异步置位/复位 D 触发器



```
module DFF_asr( output reg q, qn,  
               input d, clk, reset, set );
```

```
// always @(posedge clk or posedge reset or posedge set )
```

```
always @(posedge clk, posedge reset, posedge set )
```

```
    if (reset)
```

```
        {q, qn} <= 2'b01;
```

```
    else if (set)
```

```
        {q, qn} <= 2'b10;
```

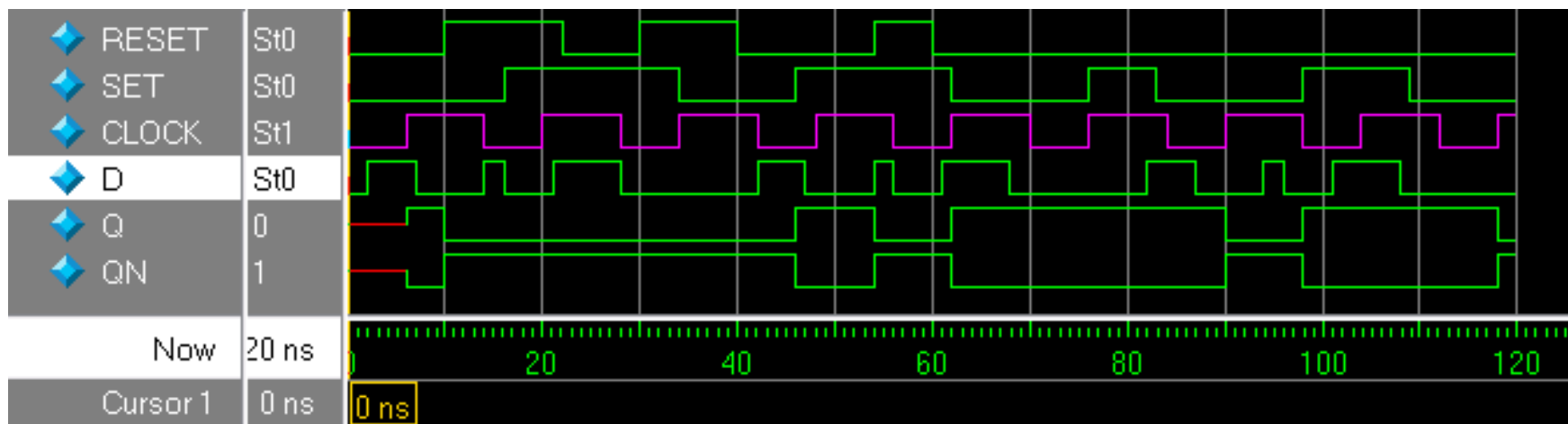
```
    else
```

```
        {q, qn} <= {d, ~d};
```

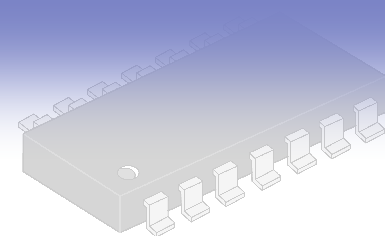
```
endmodule
```

❑ 复位优先于置位

❑ 置位优先于输入



# 异步和同步的比较



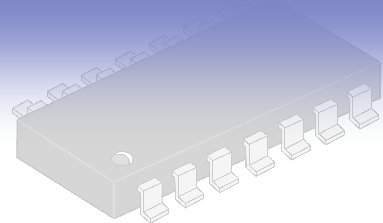
## □ 同步

```
◆ always @(posedge CLK)
◆     if (RESET)
◆         /* 复位输出 */
◆     else if (SET)
◆         /* 置位输出 */
◆     else
◆         case (state)
◆             ...
◆             /* 与时钟同步的逻辑 */
◆             ...
◆         endcase
```

## □ 异步

```
◆ always @(posedge CLK or posedge RESET or negedge SET)
◆     if (RESET)
◆         /* 复位输出 */
◆     else if (!SET)
◆         /* 置位输出 */
◆     else
◆         case (state)
◆             ...
◆             /* 与时钟同步的逻辑 */
◆             ...
◆         endcase
```

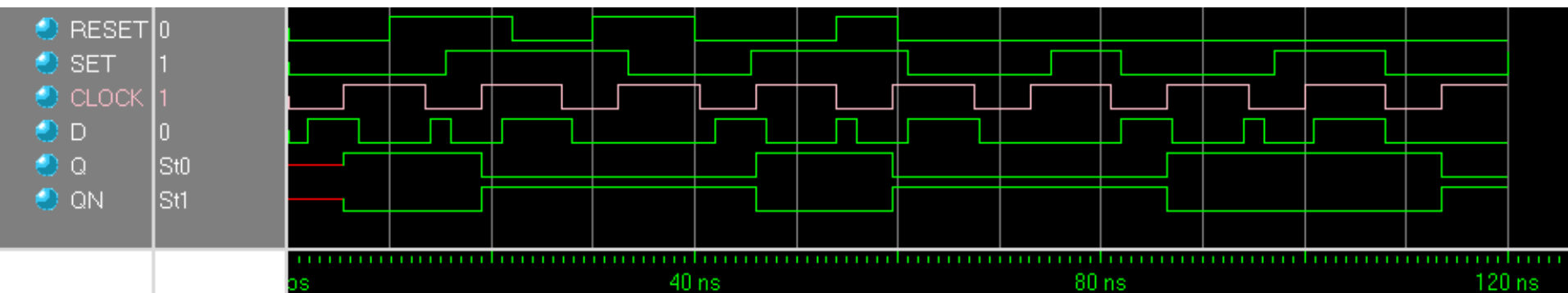
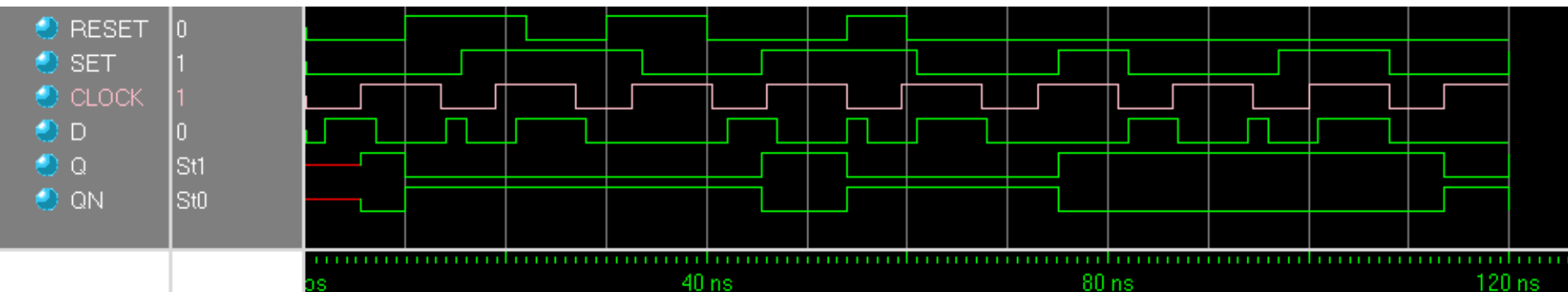
# 异步和同步的比较



## □ 高电平有效置位和复位的 D 触发器

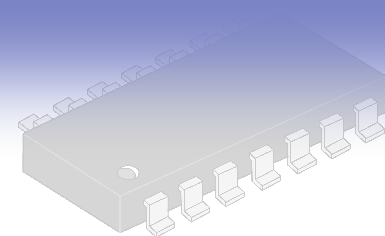
□ 上图：异步

□ 下图：同步





# 多位寄存器



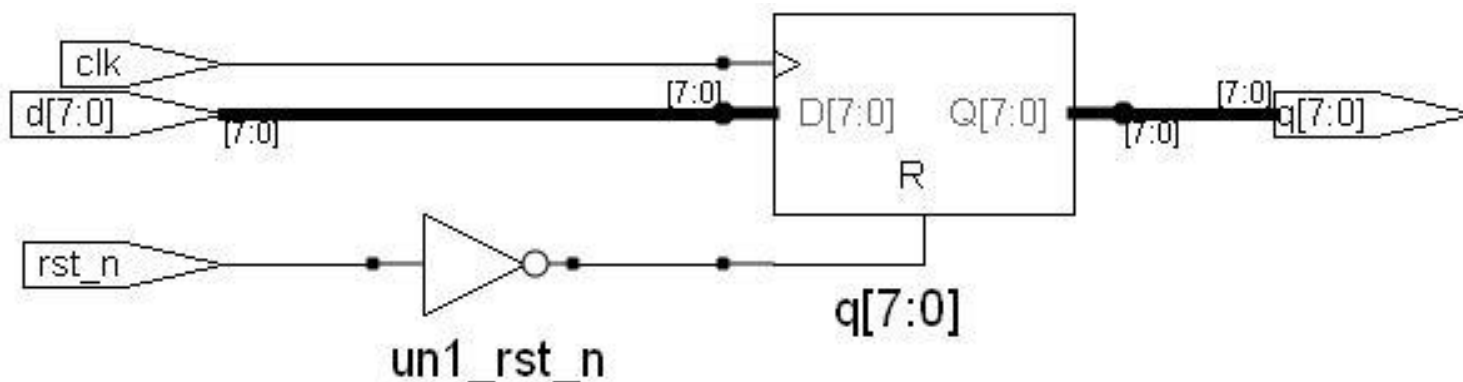
□ 用一组D触发器描述，例：

▣ 缺省为 8 位的寄存器

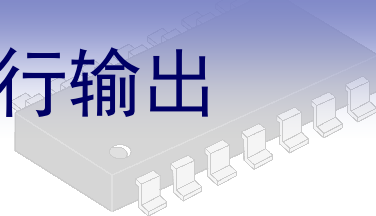
```
module DFFn #(parameter N=8) ( output reg [N-1:0] q,  
                                input [N-1:0] d,  
                                input clk, rst_n);
```

```
    always @(posedge clk, negedge rst_n)  
        if ( ~rst_n) q <= 0;  
        else  q <= d;
```

```
endmodule
```

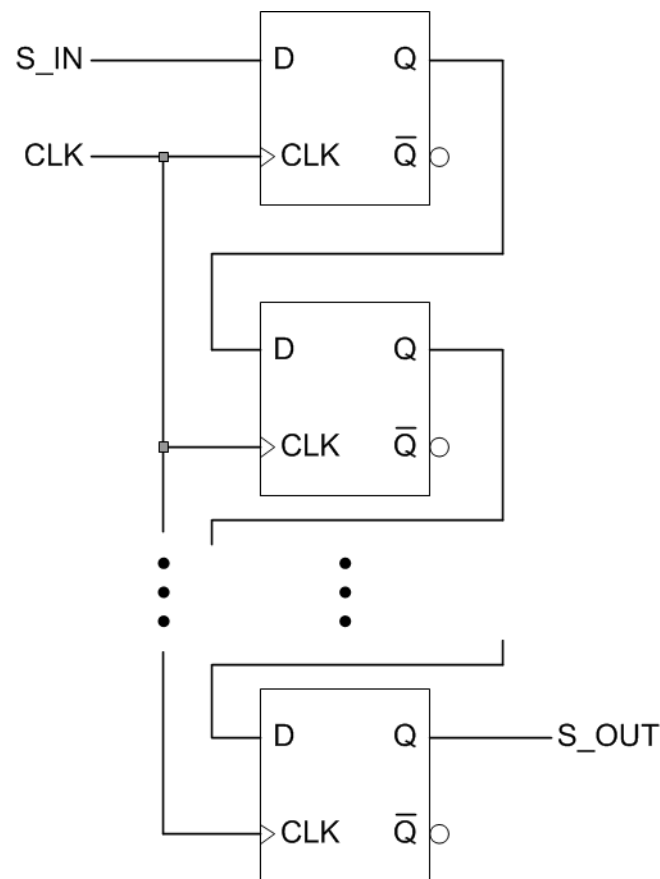


# 移位寄存器 (shift register) —— 串行输入/串行输出

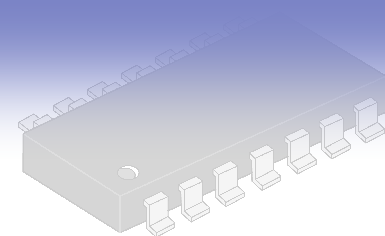


## □ 串行输入/串行输出

- 在每一个时钟触发沿到来时
  - ◆ 1位新的数据移入寄存器最末端的数据位中
  - ◆ 同时存储的所有数据向前移 1 位
  - ◆ 最前端的数据位移出出现在串行输出



# 串行输入/串行输出 4 位移位寄存器设计



```
module shiftreg4b( output reg dout,  
                  input clk, reset, din);
```

```
    reg [3:0] r;
```

```
    always @( posedge clk or posedge reset )
```

```
        if (reset) r <= 4'b0000;
```

```
        else begin
```

```
            r <= {r[2:0], din};
```

```
            dout <= r[3];
```

```
        end
```

```
    endmodule
```

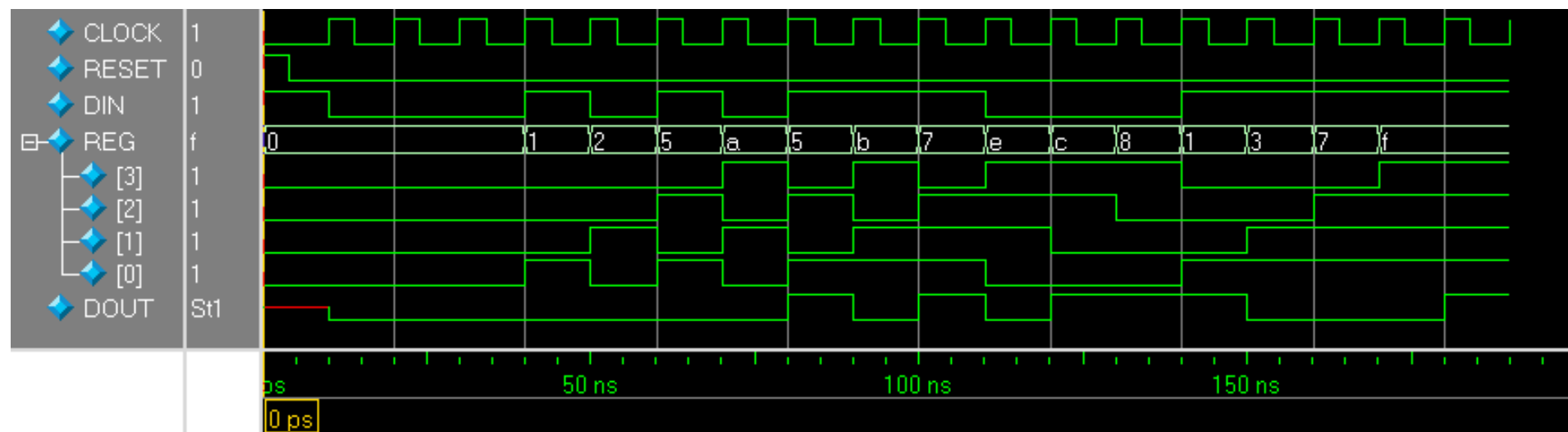
## □ 输入/输出变量

### □ 3 个输入变量:

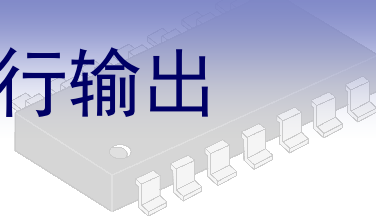
◆ clk, reset, din

### □ 1 个输出变量: dout

## □ 采用非阻塞赋值

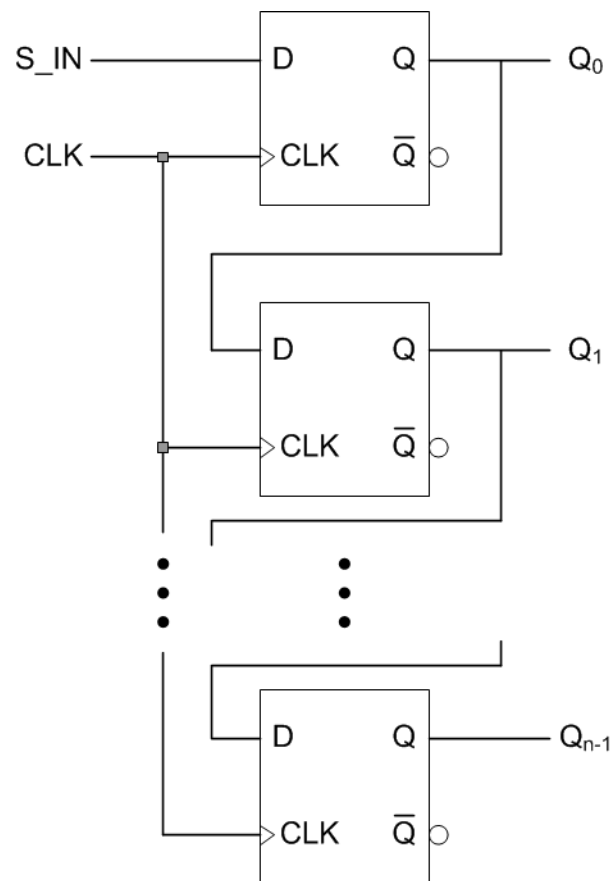


# 移位寄存器（shift register）——串行输入/并行输出

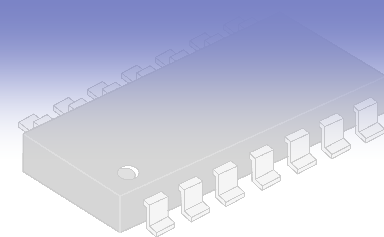


## □ 串行输入/并行输出

- 在时钟触发沿到来时，1位新的数据移入寄存器最末端的数据位中
- 每一个存储位都对应一个输出
- 串-并转换



# 串行输入/并行输出 4 位移位寄存器设计



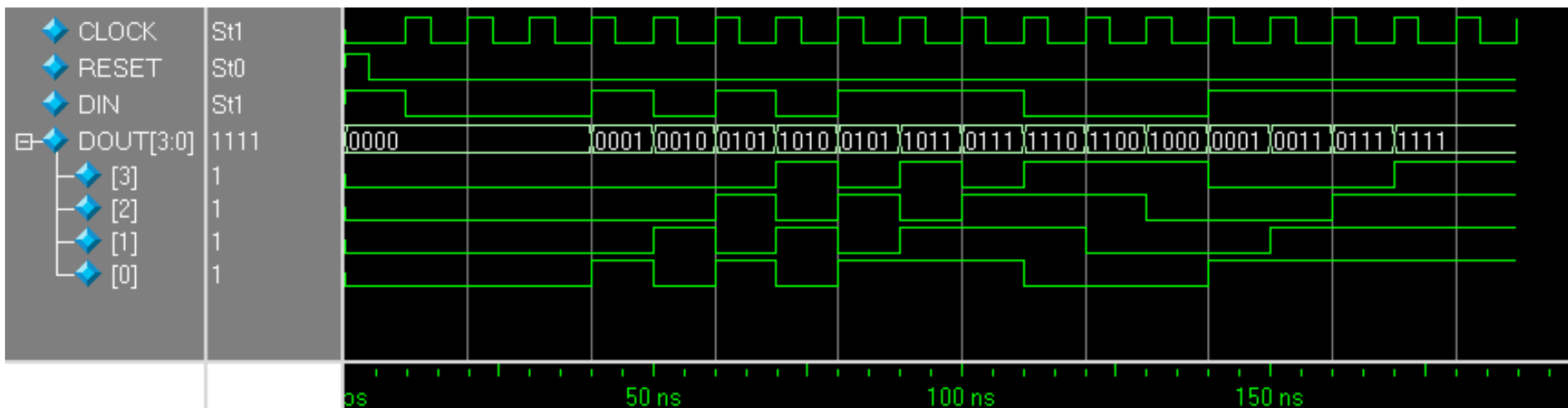
## □ 输入/输出变量

□ 3 个输入变量: clk、reset, din, 1 个输出变量: dout[3:0]

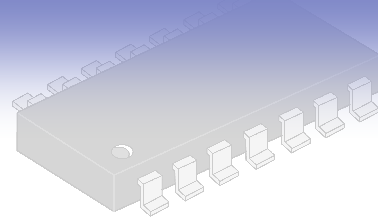
```
module shiftreg4b( output reg [3:0] dout,  
                  input clk, reset, din);
```

```
    always @(posedge clk or posedge reset)  
        if (reset) dout <= 4'h0;  
        else dout <= {dout[2:0], din};
```

```
endmodule
```



# 通用移位寄存器



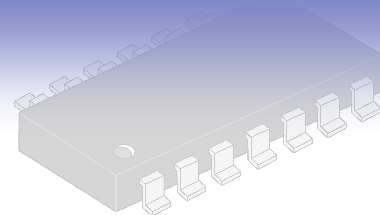
## □ 左移、右移、加载

```
module UShiftReg #(parameter N=8) ( output reg [N-1:0] q,  
                                     input [N-1:0] d,  
                                     input [1:0] s,  
                                     input Lin, Rin,  
                                     input clk, rst_n );
```

```
    always @ (posedge clk)  
        if (~rst_n)  
            q <= 0;  
        else  
            case (s)  
                2'b11:    q <= d;  
                2'b10:    q <= {q[N-2:0], Lin};  
                2'b01:    q <= {Rin, q[N-1:1]};  
                default:;  
            endcase  
        endmodule
```

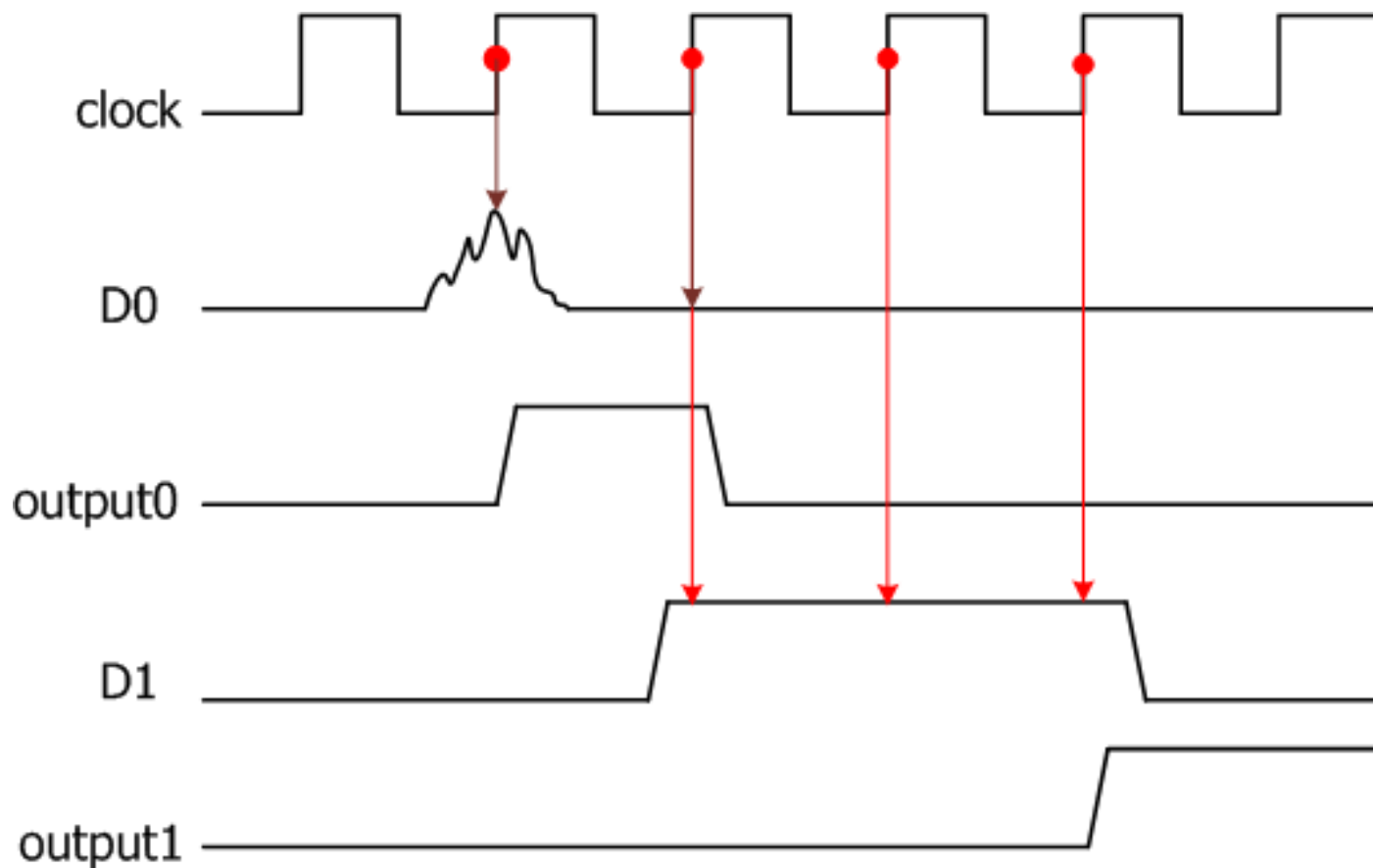
S[1:0]	动作
00	保持
01	右移位
10	左移位
11	并行加载

# 除去输入信号的噪声脉冲

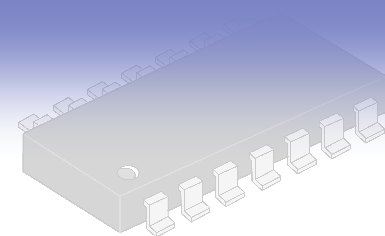


## ❑ 电路中出现的信号毛刺

- ❑ 对连续3个时钟周期内输入信号采样，当数据相同时输出  $y$  才能发生变化



# 移位寄存器数字滤波器



## ❑ 除去输入信号的噪声脉冲

- ❑ 当连续3个时钟周期内输入数据数据相同，输出 y 才能发生变化
- ❑ 输入信号平稳时间少于3个时钟周期时，不会引起输出 y 发生变化

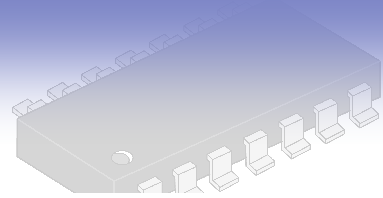
```
module filter(output reg y, input clk, rst_n, din);
    reg [3:0] q;

    always @(posedge clk) begin
        if (!rst_n) begin
            q <= 4'b0;
            y <= 1'b0;
        end
        else begin
            if ( &q[3:1]) y <= 1'b1; else
            if (~|q[3:1]) y <= 1'b0;

            q <= {q[2:0], din};
        end
    end
endmodule
```



# 移位寄存器数字滤波器仿真（1）



```
`timescale 1ns/1ns
`include "filter.v"
module filter_tb;
    wire p_y;
    reg p_clk, p_rst, p_d;

    filter u0(.y(p_y), .clk(p_clk), .rst_n(p_rst), .din(p_d) );

    initial begin
        p_clk = 0;
        forever #10 p_clk = ~p_clk;
    end

    initial begin p_rst = 1'b0; #50 p_rst = 1'b1; end

    localparam N=40;
    reg [N-1:0] sig = 40'b1101_0010_0000_0111_1001_1111_0000_0001_1100_0010;
    integer k;
    initial begin
        p_d = 1'b0;
        #60;
        for (k=0; k<N; k=k+1) begin
            #20 p_d = sig[N-1];
            sig = sig << 1;
        end
    end

    initial
        $monitor( "At time %4t, rst_n=%1b, din=%1b, y=%1b", $time, p_rst, p_d, p_y);
endmodule
```

## 移位寄存器数字滤波器仿真 (2)

```

module filter(output reg y, input clk, rst_n, din);
    reg [3:0] q;

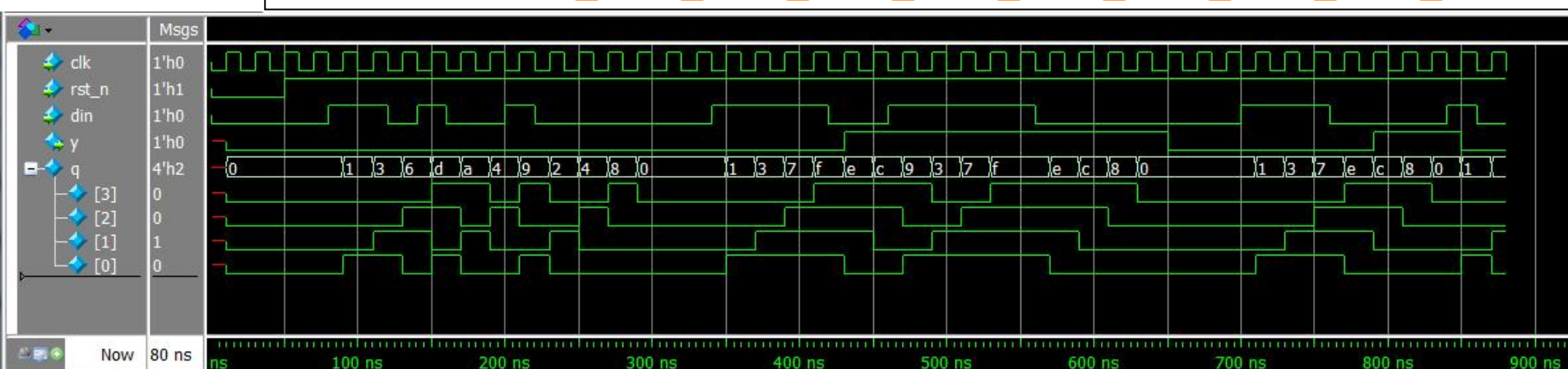
    always @(posedge clk) begin
        if (!rst_n) begin
            q <= 4'b0;
            y <= 1'b0;
        end
        else begin
            if (&q[3:1]) y <= 1'b1; else
            if (~|q[3:1]) y <= 1'b0;

            q <= {q[2:0], din};
        end
    end
endmodule

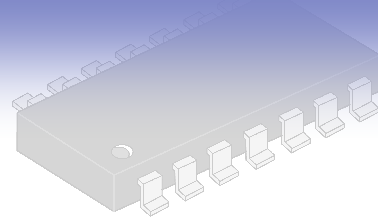
```

```
# At time      0, rst_n=0, din=0, y=x
# At time     10, rst_n=0, din=0, y=0
# At time     50, rst_n=1, din=0, y=0
# At time     80, rst_n=1, din=1, y=0
# At time    120, rst_n=1, din=0, y=0
# At time    140, rst_n=1, din=1, y=0
# At time    160, rst_n=1, din=0, y=0
# At time    200, rst_n=1, din=1, y=0
# At time    220, rst_n=1, din=0, y=0
# At time    340, rst_n=1, din=1, y=0
# At time    420, rst_n=1, din=0, y=0
# At time    430, rst_n=1, din=0, y=1
# At time    460, rst_n=1, din=1, y=1
# At time    560, rst_n=1, din=0, y=1
# At time    650, rst_n=1, din=0, y=0
# At time    700, rst_n=1, din=1, y=0
# At time    760, rst_n=1, din=0, y=0
# At time    790, rst_n=1, din=0, y=1
# At time    840, rst_n=1, din=1, y=1
# At time    850, rst_n=1, din=1, y=0
# At time    860, rst_n=1, din=0, y=0
```

```
sig = 40'b1101 0010 0000 0111 1001 1111 0000 0001 1100 0010;
```

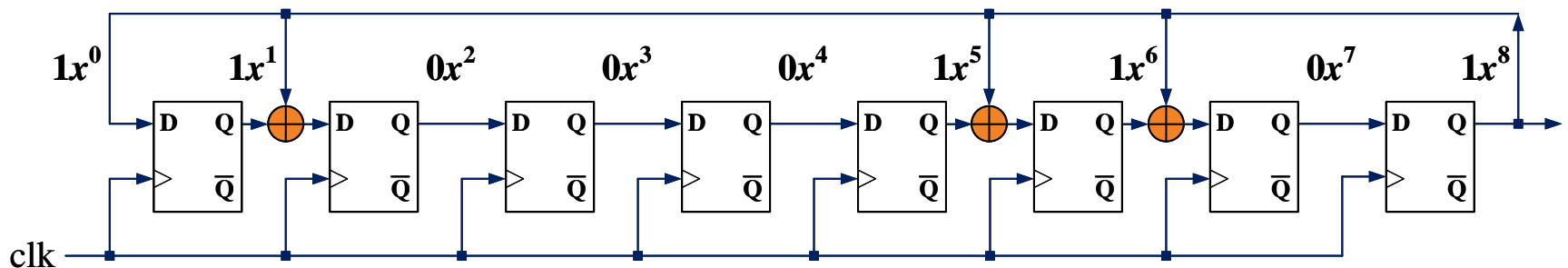


# 线性反馈移位寄存器 (LFSR)



- ❑ Linear Feedback Shift Register (LFSR)
- ❑ LFSR 产生周期序列
  - ❑ 必须从非零状态开始 —— 不产生全 0 模式
  - ❑ LFSR序列的最大长度  $2^n - 1$
- ❑ 本原多项式 ( primitive polynomial ) —— 产生最大长度序列的特征多项式
- ❑ 最大长度序列是一个伪随机数 (pseudo-random) 序列
- ❑ 例：本原多项式

$$P(x) = x^8 + x^6 + x^5 + x + 1$$



Internal Feedback LFSR

```

module LFSR( output reg [0:7] q, // 8 bit data output.
             input clk,          // Clock input.
             input rst_n,        // Synchronous reset input.
             input load,         // Synchronous load input.
             input [0:7] din     // 8 bit parallel data input.
           );

```

```

always @( posedge clk ) begin
    if ( ~rst_n )
        q <= 8'b0;
    else begin
        if (load)
            q <= (|din) ? din : 8'b0000_0001;
        else begin
            if ( q == 8'b0 )
                q <= 8'b0000_0001;
            else begin
                q[7] <= q[6];
                q[6] <= q[5] ^ q[7];
                q[5] <= q[4] ^ q[7];
                {q[4], q[3], q[2]} <= {q[3], q[2], q[1]};
                q[1] <= q[0] ^ q[7];
                q[0] <= q[7];
            end
        end
    end
end
endmodule

```

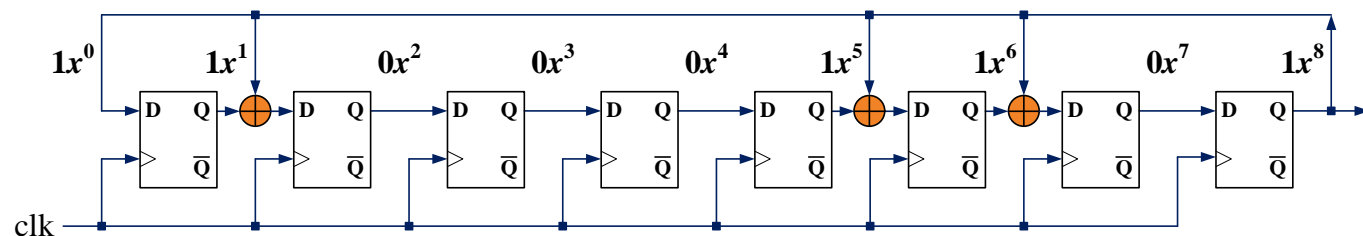
□ 输出8位伪随机数 q

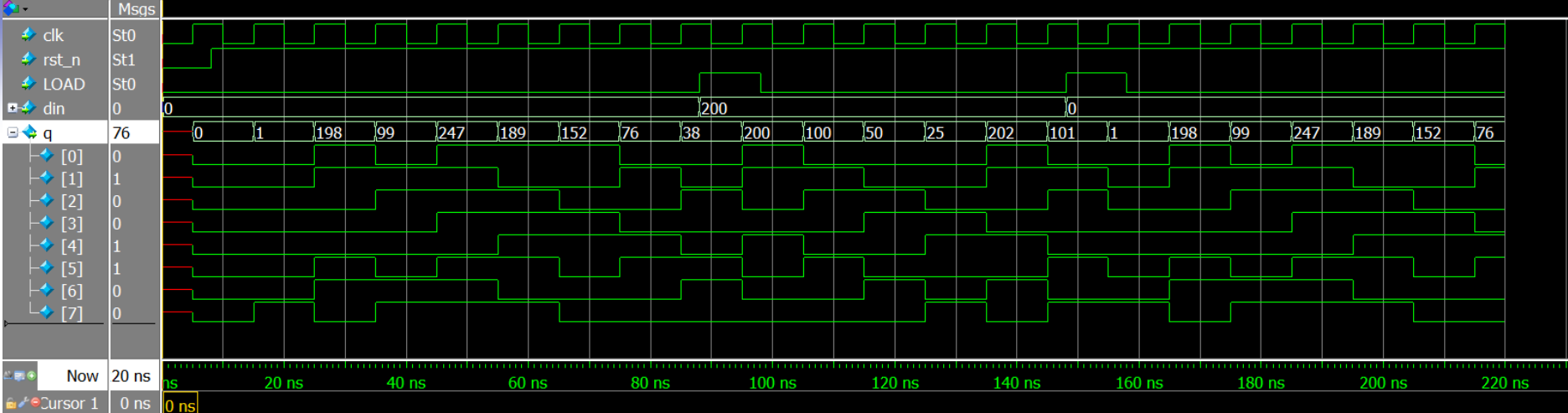
□ 输入:

- clk —— 时钟信号
- rst\_n —— 复位信号，低电平有效
- load —— 加载控制，当 load = 1'b1,
- 8位输入din不全为0时， $din \Rightarrow q$

## LFSR设计

$$P(x) = x^8 + x^6 + x^5 + x + 1$$





```
LFSR u0(.q(p_q), .clk(p_clk), .rst_n(p_rst_n),
        .load(p_load), .din(p_din));
```

```
initial begin
```

```
    p_clk = 0;
```

```
    forever #5 p_clk = ~p_clk;
```

```
end
```

```
initial begin
```

```
    p_rst_n = 0;
```

```
    #8 p_rst_n = 1;
```

```
end
```

```
initial begin
```

```
    p_load = 0;
```

```
    p_din = 8'b0;
```

```
    #88 p_load = 1'b1;
```

```
    p_din = 8'b1100_1000;
```

```
    #10 p_load = 1'b0;
```

```
    #50 p_load = 1'b1;
```

```
    p_din = 8'b0;
```

```
    #10 p_load = 1'b0;
```

```
end
```

## LFSR仿真结果

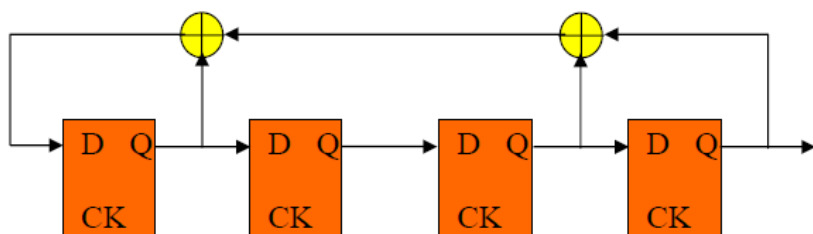
```
# At time t= 0, rst_n=0, load=0, din=00000000, q=xxxxxxxx, q= x
# At time t= 5, rst_n=0, load=0, din=00000000, q=00000000, q= 0
# At time t= 8, rst_n=1, load=0, din=00000000, q=00000000, q= 0
# At time t= 15, rst_n=1, load=0, din=00000000, q=00000001, q= 1
# At time t= 25, rst_n=1, load=0, din=00000000, q=11000110, q=198
# At time t= 35, rst_n=1, load=0, din=00000000, q=01100011, q= 99
# At time t= 45, rst_n=1, load=0, din=00000000, q=11110111, q=247
# At time t= 55, rst_n=1, load=0, din=00000000, q=10111101, q=189
# At time t= 65, rst_n=1, load=0, din=00000000, q=10011000, q=152
# At time t= 75, rst_n=1, load=0, din=00000000, q=01001100, q= 76
# At time t= 85, rst_n=1, load=0, din=00000000, q=00100110, q= 38
# At time t= 88, rst_n=1, load=1, din=11001000, q=00100110, q= 38
# At time t= 95, rst_n=1, load=1, din=11001000, q=11001000, q=200
# At time t= 98, rst_n=1, load=0, din=11001000, q=11001000, q=200
# At time t= 105, rst_n=1, load=0, din=11001000, q=01100100, q=100
# At time t= 115, rst_n=1, load=0, din=11001000, q=00110010, q= 50
# At time t= 125, rst_n=1, load=0, din=11001000, q=00011001, q= 25
# At time t= 135, rst_n=1, load=0, din=11001000, q=11001010, q=202
# At time t= 145, rst_n=1, load=0, din=11001000, q=01100101, q=101
# At time t= 148, rst_n=1, load=1, din=00000000, q=01100101, q=101
# At time t= 155, rst_n=1, load=1, din=00000000, q=00000001, q= 1
# At time t= 158, rst_n=1, load=0, din=00000000, q=00000001, q= 1
# At time t= 165, rst_n=1, load=0, din=00000000, q=11000110, q=198
# At time t= 175, rst_n=1, load=0, din=00000000, q=01100011, q= 99
# At time t= 185, rst_n=1, load=0, din=00000000, q=11110111, q=247
# At time t= 195, rst_n=1, load=0, din=00000000, q=10111101, q=189
# At time t= 205, rst_n=1, load=0, din=00000000, q=10011000, q=152
# At time t= 215, rst_n=1, load=0, din=00000000, q=01001100, q= 76
```



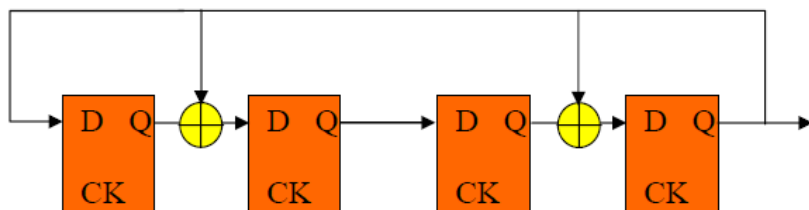
# 本原多项式

- 触发器的个数：
  - $n = \text{degree of polynomial}$
  - 不超过 4 个反馈连接
- 一些本原多项式
- 两种实现方式 —— 内部反馈更快

External Feedback LFSR



Internal Feedback LFSR



序号	Degree(n)	Polynomial
1	2, 3, 4, 6, 7, 15, 22	$x^n + x + 1$
2	5, 11, 21, 29	$x^n + x^2 + 1$
3	8, 19	$x^n + x^6 + x^5 + x + 1$
4	9	$x^n + x^4 + 1$
5	10, 17, 20, 25, 28	$x^n + x^3 + 1$
6	12	$x^n + x^7 + x^4 + x^3 + 1$
7	13, 24	$x^n + x^4 + x^3 + x + 1$
8	14	$x^n + x^{12} + x^{11} + x + 1$
9	16	$x^n + x^5 + x^3 + x^2 + 1$
10	18	$x^n + x^7 + 1$
11	23	$x^n + x^5 + 1$
12	26, 27	$x^n + x^8 + x^7 + x + 1$
13	30	$x^n + x^{16} + x^{15} + x + 1$

# 具有异步复位的 4 位计数器



## □ 使用异步复位

### ▣ 输入/输出

- ◆ 2 个输入变量: clk、reset
- ◆ 1 个输出变量: q[3:0]

```
module counter #(parameter N=4) ( output reg [N-1:0] count,  
                                   input clk, rst_n);
```

```
    always @(posedge clk or posedge rst_n)
```

```
        if ( ~rst_n )
```

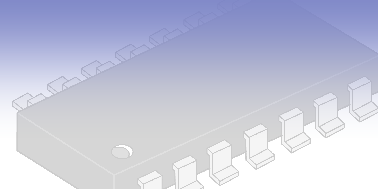
```
            count <= 0;
```

```
        else
```

```
            count <= count + 1;
```

```
endmodule
```

# 具有异步复位的 4 位计数器仿真结果



```
`timescale 1ns/1ns
`include "counter.v"

module counter_tb;
    wire [3:0] p_cnt;
    reg p_clk, p_rst_n;

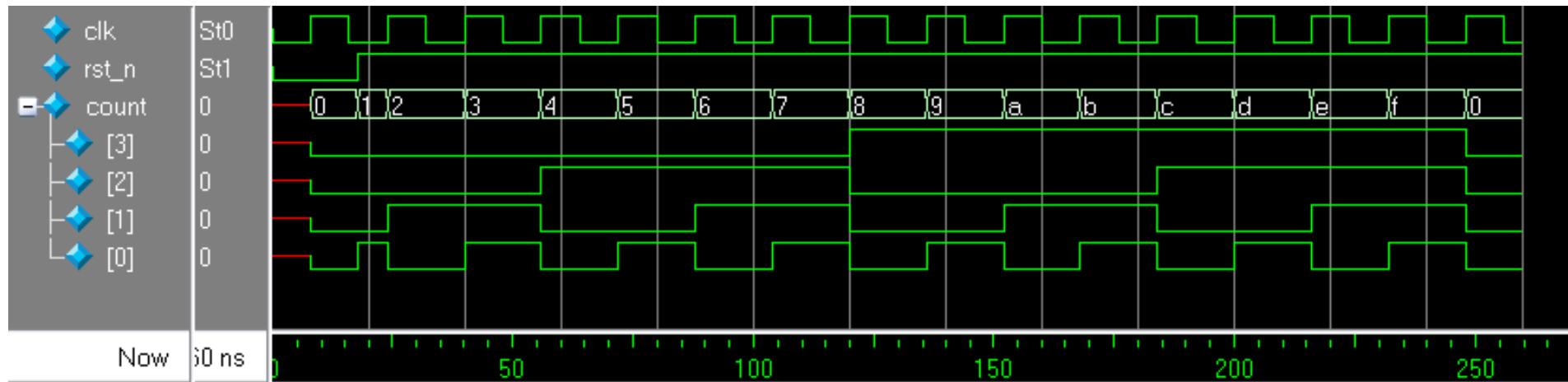
    initial begin
        p_clk = 0;
        forever #8 p_clk = ~p_clk;
    end

    initial begin
        p_rst_n = 0;
        #18 p_rst_n = 1;
    end

    counter #(4) u0(.count(p_cnt), .clk(p_clk), .rst_n(p_rst_n));
endmodule

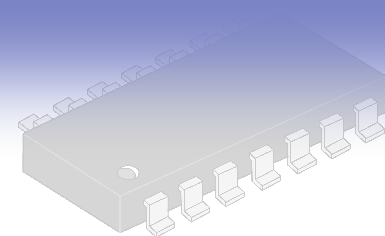
module counter #(parameter N=4) ( output reg [N-1:0] count,
    input clk, rst_n);

    always @(posedge clk or posedge rst_n)
        if ( ~rst_n )
            count <= 0;
        else
            count <= count + 1;
    endmodule
```





# 可逆十进制计数器



## ■ 功能

- 可以向上或向下计数，计数方向控制 **M**
  - ◆ **M = 0**: 向上计数，从 0 到 9；到达 9 后，标志 **SUP** 为 1，然后，重新开始
  - ◆ **M = 1**: 向下计数，从 9 到 0；到达 0 后，标志 **INF** 为 1，然后，重新开始
- 能够加载指定数据: **data**

## ■ Verilog 模块设计:

- 异步复位
- 同步加载

# 可逆十进制计数器模块

```
module decimal_counter #(parameter N = 4)
```

```
  ( output reg [N-1:0] count,  
    output reg sup, inf,  
    input clk, rst_n, load, dir,  
    input [N-1 : 0] data );
```

```
  always @ (posedge clk) begin
```

```
    if (!rst_n) begin count <= 4'd0; {inf, sup} <= 2'b0; end
```

```
    else
```

```
      if (load) count <= data;
```

```
      else
```

```
        if (!dir)
```

```
          if ( count < 4'd9) begin
```

```
            count <= count + 4'd1; {inf, sup} <= 2'b0;
```

```
          end
```

```
          else begin
```

```
            count <= 4'd0; {inf, sup} <= 2'b01;
```

```
          end
```

```
        else
```

```
          if ( count > 4'd0) begin
```

```
            count <= count - 4'd1; {inf, sup} <= 2'b0;
```

```
          end
```

```
          else begin
```

```
            count <= 4'd9; {inf, sup} <= 2'b10;
```

```
          end
```

```
    end
```

```
endmodule
```

- M = 0: 向上计数，从 0 到 9；到达 9 后，标志 SUP 为 1，然后，重新开始
- M = 1: 向下计数，从 9 到 0；到达 0 后，标志 INF 为 1，然后，重新开始

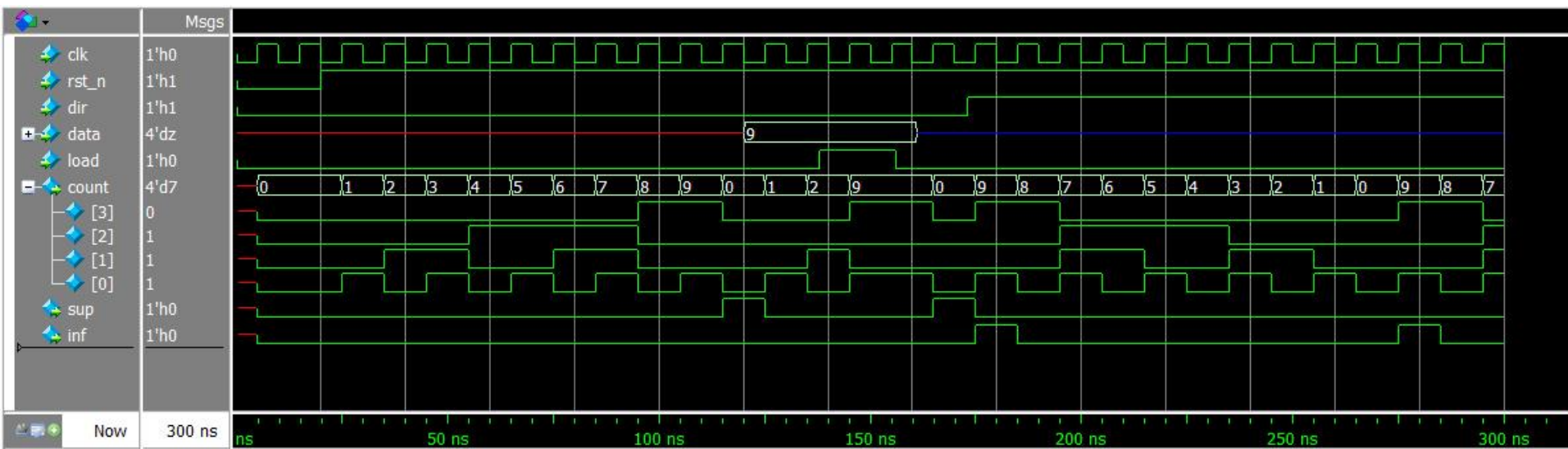
# 可逆十进制计数器模块的仿真波形

```
initial begin
    p_clk = 0;
    forever #5 p_clk = ~p_clk;
end
```

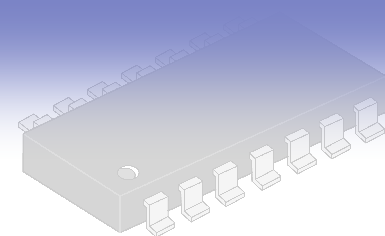
```
initial begin
    p_rst = 0;
    #20 p_rst = 1'b1;
end
```

```
initial begin
    p_load = 0;
    p_dir = 0;
    p_data = 4'bx;

    #120 p_data = 4'd9;
    #18 p_load = 1'b1;
    #18 p_load = 1'b0;
    #5 p_data = 4'dz;
    #12 p_dir = 1'b1;
end
```



# 存储器



## □ 多个多位锁存器或多个寄存器组成

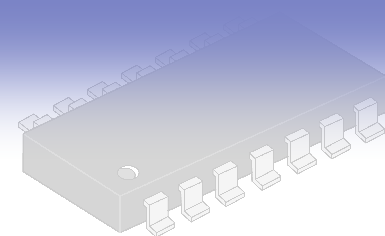
- 存储器是一个二维存储阵列 —— 与一维多位锁存器/寄存器等效
- 二维数组中每一行中的数据位组成一个字，每个字中的所有位同时被读写

## □ 存储器的大小

- $2^m \times n$  —— （深度  $\times$  宽度，字数  $\times$  每个字中的数据位数）
  - ◆  $m$  —— 地址的位数
  - ◆  $n$  —— 每个字中的数据宽度
- $2^m$  —— RAM depth

地址	数据							
0								
1								
2								
3								
4								
5								
6								
7								

# 设计ROM (1)



```
module ROM(output [7:0] data, input [3:0] address, input en);  
    assign data = (en) ? ROM_LOC(address) : 8'bz;
```

```
function [7:0] ROM_LOC( input [3:0] a );
```

```
    case (a)
```

```
        4'h0:    ROM_LOC = 8'b1010_1001;
```

```
        4'h1:    ROM_LOC = 8'b1111_1101;
```

```
        4'h2:    ROM_LOC = 8'b1110_1001;
```

```
        4'h3:    ROM_LOC = 8'b1101_1100;
```

```
        4'h4:    ROM_LOC = 8'b1011_1001;
```

```
        4'h5:    ROM_LOC = 8'b1100_0010;
```

```
        4'h6:    ROM_LOC = 8'b1100_0101;
```

```
        4'h7:    ROM_LOC = 8'b0000_0100;
```

```
        4'h8:    ROM_LOC = 8'b1110_1100;
```

```
        4'h9:    ROM_LOC = 8'b1000_1010;
```

```
        4'hA:    ROM_LOC = 8'b1100_1111;
```

```
        4'hB:    ROM_LOC = 8'b0011_0100;
```

```
        4'hC:    ROM_LOC = 8'b1100_0001;
```

```
        4'hD:    ROM_LOC = 8'b1001_1111;
```

```
        4'hE:    ROM_LOC = 8'b1010_0101;
```

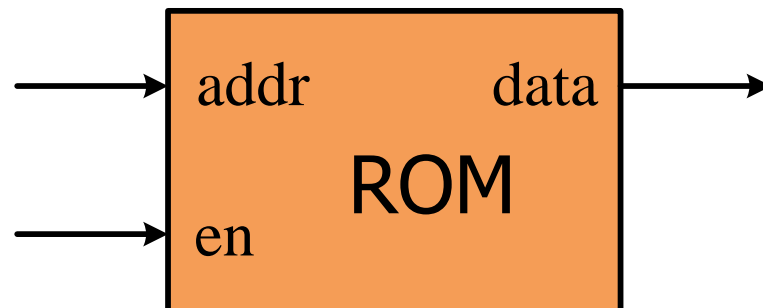
```
        4'hF:    ROM_LOC = 8'b0101_1100;
```

```
        default: ROM_LOC = 8'bx;
```

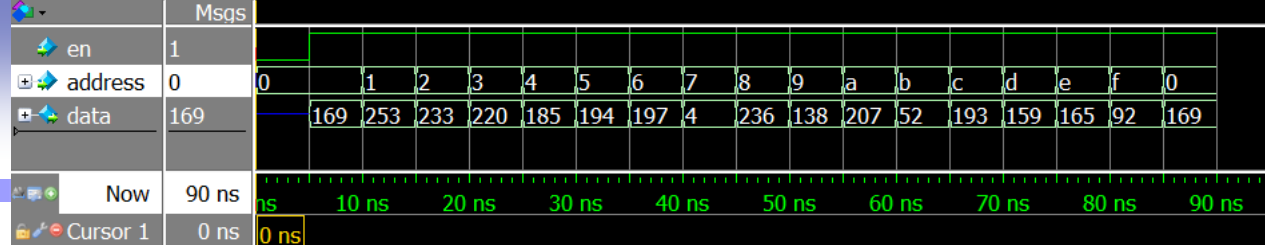
```
    endcase
```

```
endfunction
```

```
endmodule
```



# ROM仿真与综合



```
module ROM(output [7:0] data, input [3:0] address, input en);
    assign data = (en) ? ROM_LOC(address) : 8'bz;
```

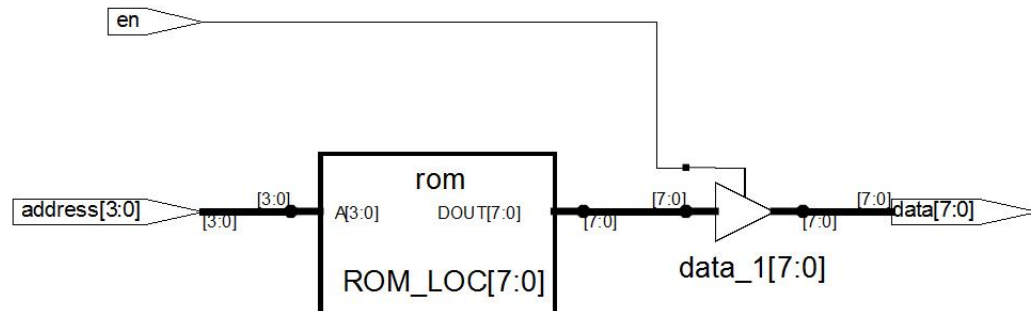
```
function [7:0] ROM_LOC( input [3:0] a );
    case (a)
        4'h0: ROM_LOC = 8'b1010_1001;
        4'h1: ROM_LOC = 8'b1111_1101;
        4'h2: ROM_LOC = 8'b1110_1001;
        4'h3: ROM_LOC = 8'b1101_1100;
        4'h4: ROM_LOC = 8'b1011_1001;
        4'h5: ROM_LOC = 8'b1100_0010;
        4'h6: ROM_LOC = 8'b1100_0101;
        4'h7: ROM_LOC = 8'b0000_0100;
        4'h8: ROM_LOC = 8'b1110_1100;
        4'h9: ROM_LOC = 8'b1000_1010;
        4'hA: ROM_LOC = 8'b1100_1111;
        4'hB: ROM_LOC = 8'b0011_0100;
        4'hC: ROM_LOC = 8'b1100_0001;
        4'hD: ROM_LOC = 8'b1001_1111;
        4'hE: ROM_LOC = 8'b1010_0101;
        4'hF: ROM_LOC = 8'b0101_1100;
        default: ROM_LOC = 8'bx;
    endcase
endfunction
endmodule
```

```
`timescale 1ns/1ns
`include "ROM.v"
module ROM_tb;
    wire [7:0] p_q;
    reg [3:0] p_addr;
    reg p_en;

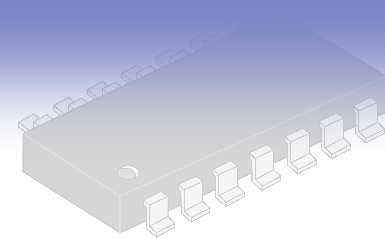
    ROM u0(.data(p_q), .address(p_addr), .en(p_en));

    integer k;
    initial begin
        p_en = 0;
        p_addr = 0;
        #5 p_en = 1;
        for( k=0; k<16; k=k+1)
            #5 p_addr = p_addr + 1'b1;
    end

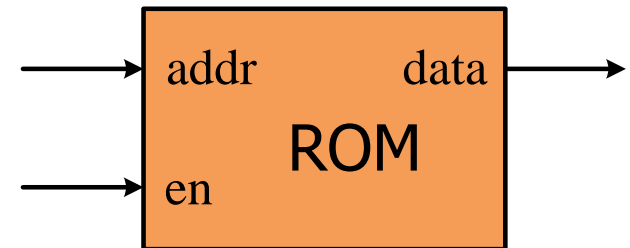
    initial
        $monitor("At time t=%4t, en=%b, address=%h, data=%b",
            $time, u0.en, u0.address, u0.data);
endmodule
```



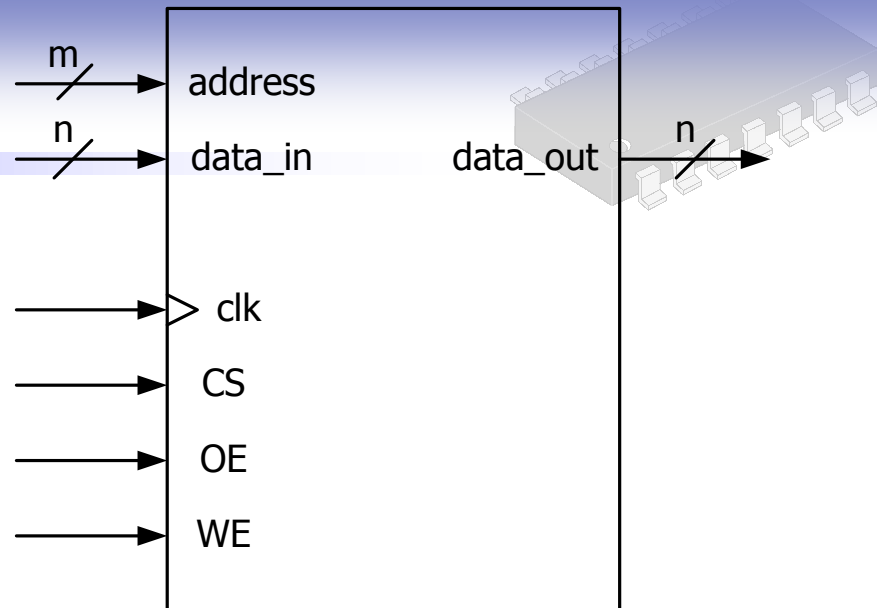
# 设计ROM (2)



```
module ROM ( output reg [3:0] address, // Address input
             input [7:0] data,         // Data output
             input en,                 // Read Enable
             input ce                  // Chip Enable
           );
  always @(*) begin
    if ( en && ce )
      case (address)
        0 : data = 8'ha;
        1 : data = 8'h37;
        2 : data = 8'hf4;
        3 : data = 8'h0;
        4 : data = 8'h9;
        5 : data = 8'hff;
        6 : data = 8'h11;
        7 : data = 8'h1;
        8 : data = 8'h10;
        9 : data = 8'h15;
        10 : data = 8'h1d;
        11 : data = 8'h25;
        12 : data = 8'h60;
        13 : data = 8'h90;
        14 : data = 8'h70;
        15 : data = 8'h90;
        default: data = 8'hz;
      endcase
    else data = 8'hz;
  end
endmodule
```



# 单端口RAM

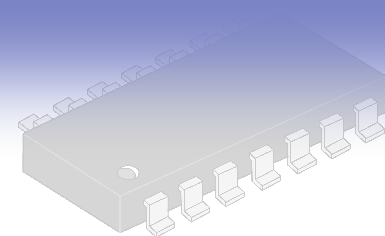


## □ 数据输入/输出和控制信号

- address: 数据存放地址
- data\_in: 数据输入
- data\_out: 数据输出
- clk: 时钟信号
- CS: 选片信号，总控制信号，若 CS 有效，则可对该 RAM 芯片操作
  - ◆ 对于 CS 无效的RAM，其数据输出端处于高阻状态
- OE: 输出使能，当 OE 有效，RAM 中的数据读到数据输出端口
- WE: 写使能，当 WE 有效，将输入端口的数据写入到 RAM 中
- 任何时候，**OE 和 WE 只能有一个有效**

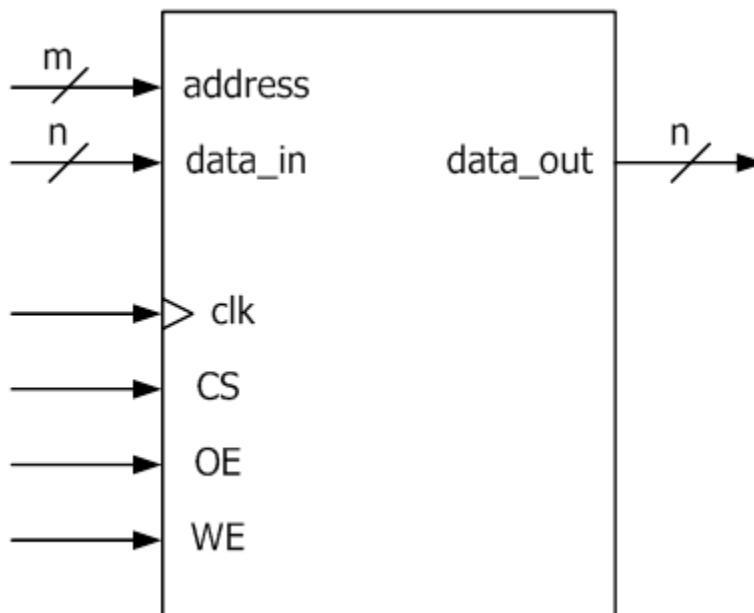


# 单端口RAM设计 (1a)



## □ 模块端口列表

```
module RAM_sync #( parameter DATA_WIDTH = 8, ADDR_WIDTH = 8 )
    ( output [DATA_WIDTH-1:0] data_out, // Output data
      input [ADDR_WIDTH-1:0] address, // Address Input
      input [DATA_WIDTH-1:0] data_in, // Input data
      input clk, // Clock Input
      input cs, // Chip Select
      input we, // Write Enable/Read Enable
      input oe // Output Enable
    );
```



# 单端口RAM设计 (1b)

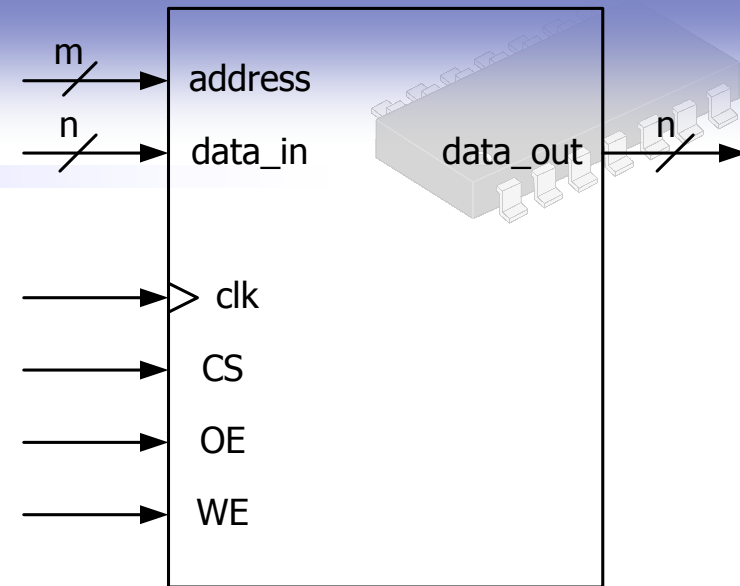
## □ (续) 模块实现部分

```
// DATA_WIDTH = 8, ADDR_WIDTH = 8
parameter RAM_DEPTH = 1 << ADDR_WIDTH;
// Internal variables
reg [DATA_WIDTH-1:0] mem [0:RAM_DEPTH-1];
reg [DATA_WIDTH-1:0] data ;

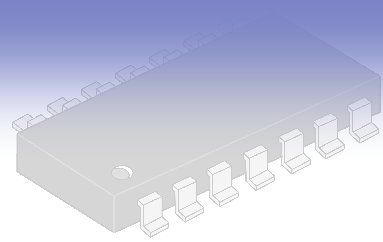
// output : When cs = 1, we = 0, oe = 1
assign data_out = ( cs && !we && oe ) ? data : 8'bz;

// Write Operation : When cs = 1, we = 1
always @ (posedge clk) begin : MEM_WRITE
    if ( cs && we )
        mem[address] <= data_in;
end

// Read Operation : When cs = 1, we = 0, oe = 1
always @ (posedge clk) begin : MEM_READ
    if ( cs && !we && oe )
        data <= mem[address];
    end
endmodule
```

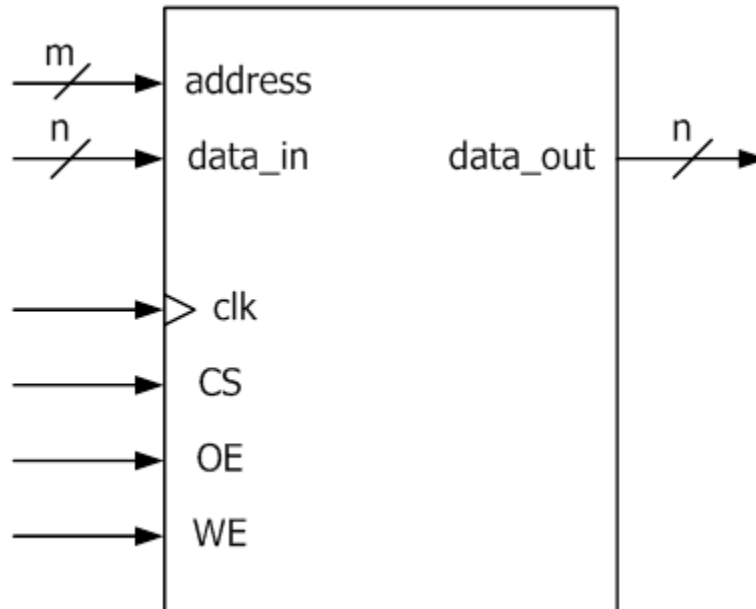


# 单端口RAM设计 (2a)



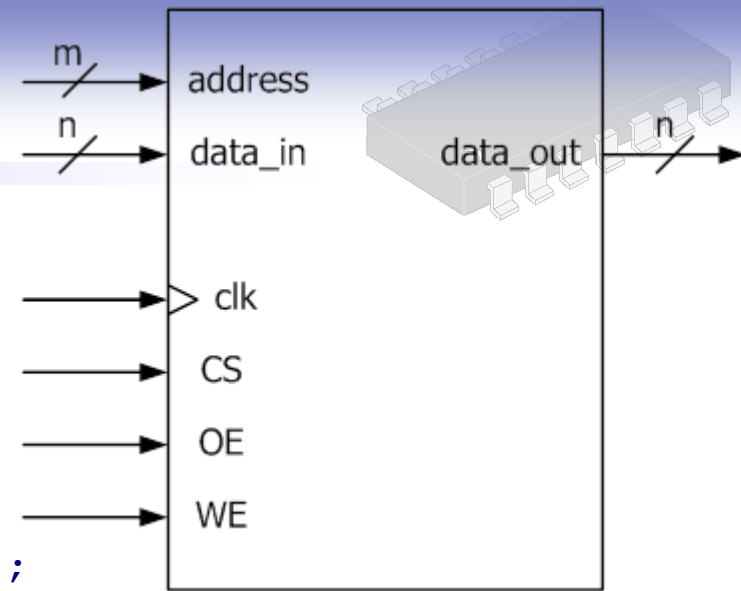
## □ 模块端口列表

```
module RAM_sync #( parameter DATA_WIDTH = 8, ADDR_WIDTH = 8 )
    ( output [DATA_WIDTH-1:0] data_out, // Output data
      input [ADDR_WIDTH-1:0] address, // Address Input
      input [DATA_WIDTH-1:0] data_in, // Input data
      input clk, // Clock Input
      input cs, // Chip Select
      input we, // Write Enable/Read Enable
      input oe // Output Enable
    );
```



# 单端口RAM设计 (2a)

## □ (续) 模块实现部分

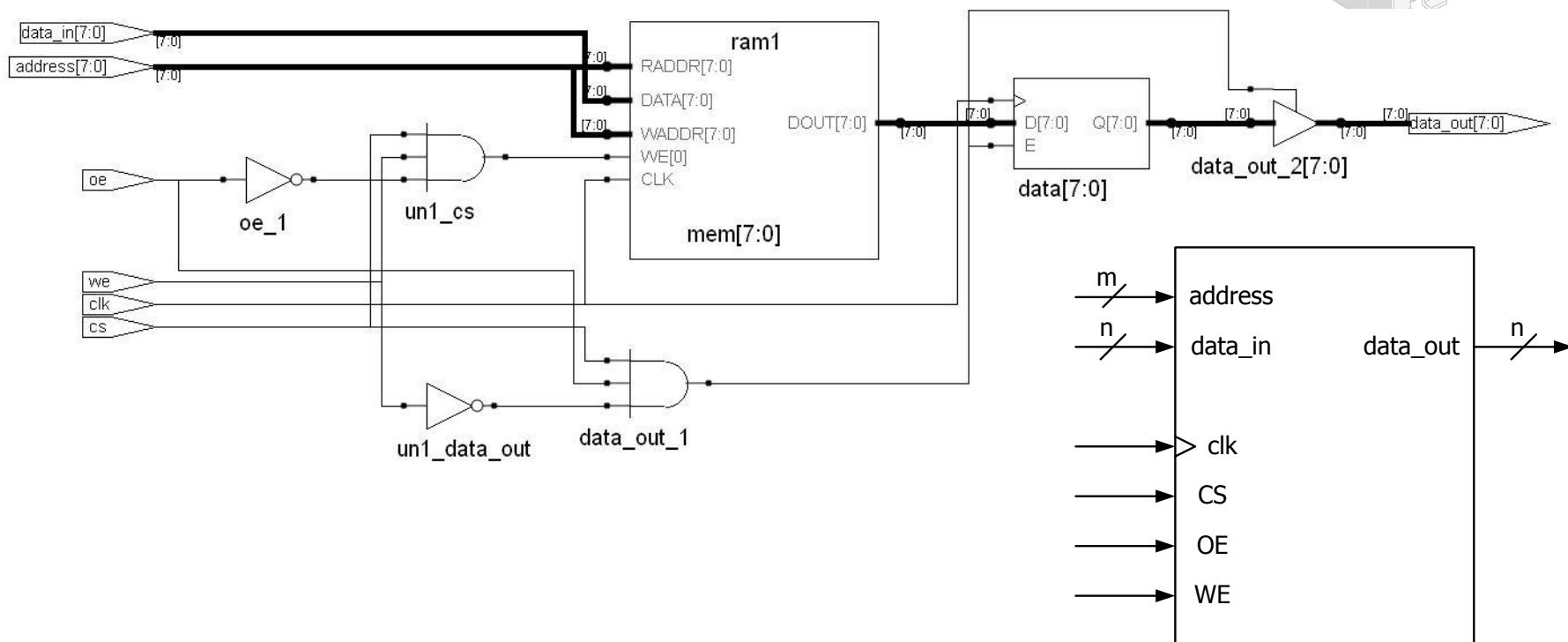
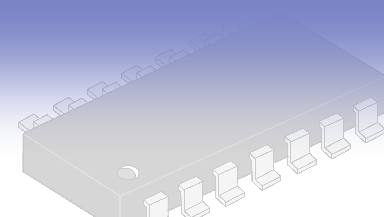


```
// DATA_WIDTH = 8, ADDR_WIDTH = 8
parameter RAM_DEPTH = 1 << ADDR_WIDTH;
// Internal variables
reg [DATA_WIDTH-1:0] mem [0:RAM_DEPTH-1];
reg [DATA_WIDTH-1:0] data ;

// Tri-State Buffer control
// output : When cs = 1, we = 0, oe = 1
assign data_out = ( cs && !we && oe ) ? data : 8'bz;

always @(posedge clk) begin
    if ( cs )    // When cs = 1
        case ({we, oe})
            2'b01:  data <= mem[address];    // 读: we = 0, oe = 1
            2'b10:  mem[address] <= data_in; // 写: we = 1, oe = 0
            default:;
        endcase
    end
endmodule
```

# 单端口RAM综合



Type	ID	Message
	CL134	Found RAM mem, depth=256, width=8
	CG364	Synthesizing module RAM
	MT206	Autoconstrain Mode is ON
	MF257	Gated clock conversion enabled
	MF249	Running in 32-bit mode.
	FX164	The option to pack flops in the IOB has not been specified

# 双端口存储器 (Dual Port RAM)



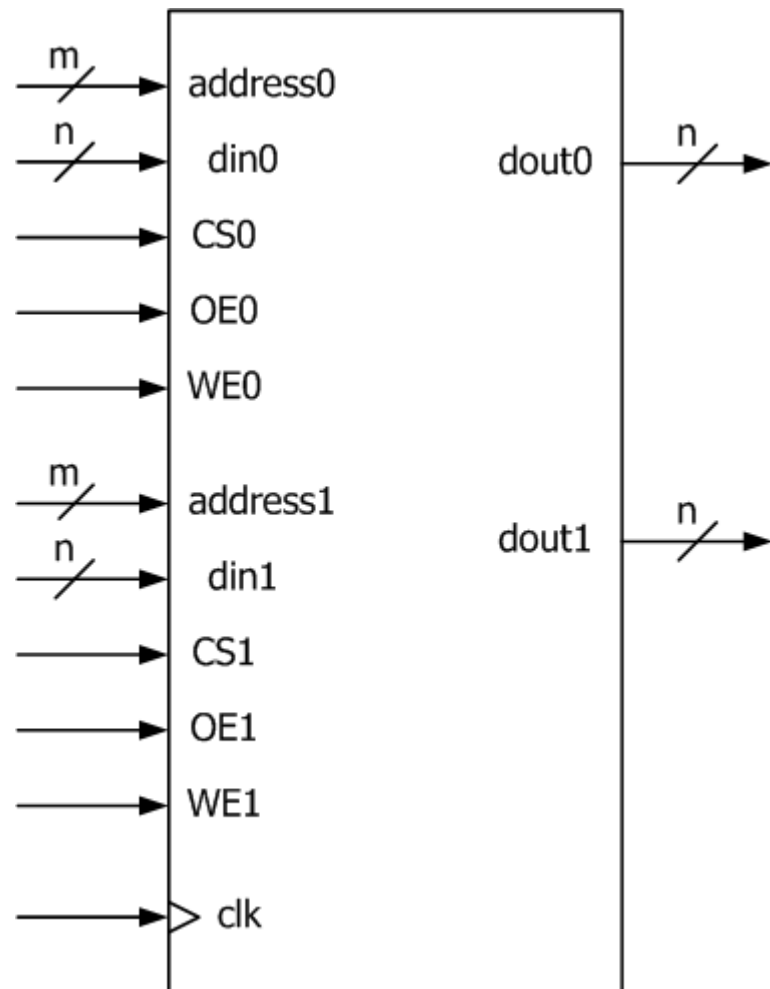
## □ 单一时钟信号

- clk: 时钟信号

## □ 端口 0 数据输入/输出和控制信号

- address0: 数据存放地址
- din0: 数据输入
- dout0: 数据输出
- CS0: Port\_0 选片信号
  - ◆ 若 CS0 有效, 可用 Port\_0 操作RAM
  - ◆ 当 CS0 无效, Port\_0 数据输出为高阻
- OE0: 输出使能
  - ◆ 当OE0有效, 读RAM中的数据到 dout0
- WE0: 写使能
  - ◆ 当WE0有效, 将输入数据写入RAM
- 任何时候, **OE0 和 WE0 中只能一个有效**

## □ 端口 1 数据输入/输出和控制信号与端口 0 完全相同



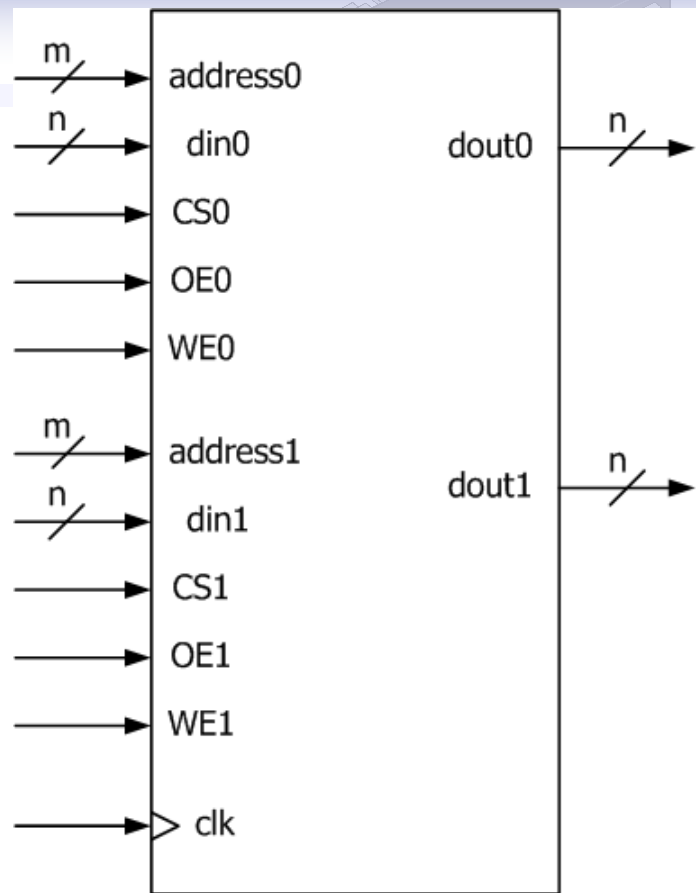
# 双端口存储器设计

## ❑ 不能同时写

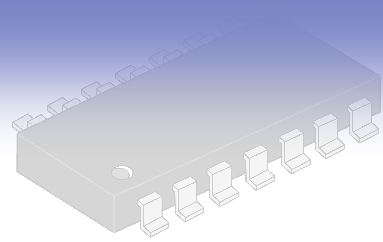
- ❑ 不允许两个端口不能同时向一个存储单元中写

## ❑ 不能写时读

- ❑ 任何时候，不能在写某个单元时候，同时读该单元



# 双端口存储器设计（1）



## □ 模块端口列表

```
module DualPortRAM #( parameter DATA_WIDTH = 8, ADDR_WIDTH = 8 )
(
    output [DATA_WIDTH-1:0] dout0,        // Port 0 output data
    input [ADDR_WIDTH-1:0] address0,      // Port 0 address Input
    input [DATA_WIDTH-1:0] din0,          // Port 0 input data
    input cs0,                            // Port 0 chip Select
    input we0,                            // Port 0 write enable/read Enable
    input oe0,                            // Port 0 output Enable
    //////////////////////////////////
    output [DATA_WIDTH-1:0] dout1,        // Port 1 output data
    input [ADDR_WIDTH-1:0] address1,      // Port 1 address Input
    input [DATA_WIDTH-1:0] din1,          // Port 1 input data
    input cs1,                            // Port 1 chip Select
    input we1,                            // Port 1 write enable/read Enable
    input oe1,                            // Port 1 output Enable
    //////////////////////////////////
    input clk                             // Clock Input
);
```



## 双端口存储器设计（2）

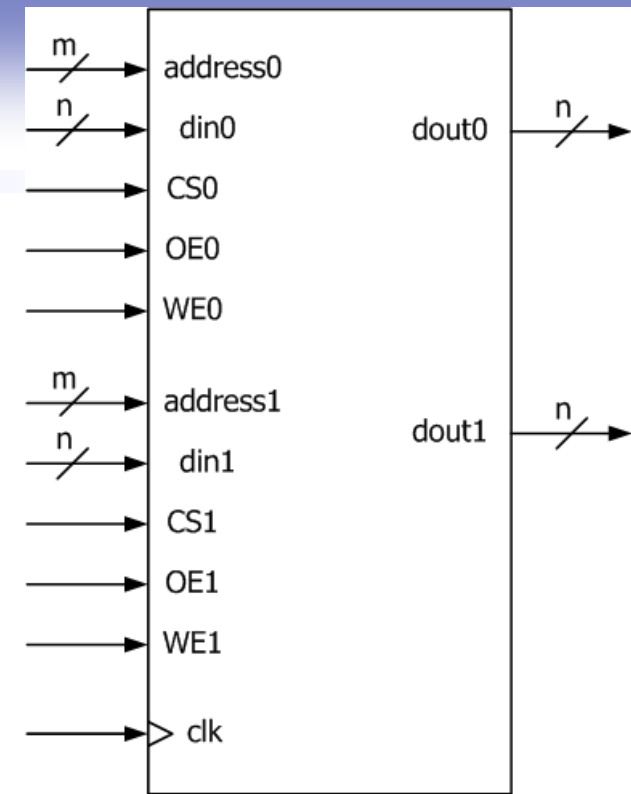
### □ （续）模块实现部分 —— 存储器写操作

```
// DATA_WIDTH = 8, ADDR_WIDTH = 8
parameter RAM_DEPTH = 1 << ADDR_WIDTH;

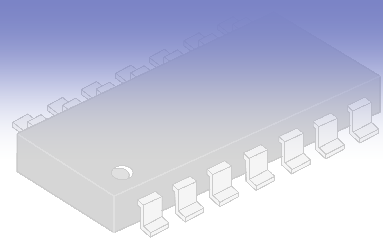
reg [DATA_WIDTH-1:0] mem [0:RAM_DEPTH-1];

reg [DATA_WIDTH-1:0] data0;
reg [DATA_WIDTH-1:0] data1;

// Memory Write Block
// Write Operation : When we0 = 1, cs0 = 1
//                   otherwise we1 = 1, cs1 = 1
always @ (posedge clk) begin : MEM_WRITE // 防止同时写
    if ( cs0 && we0 ) begin
        mem[address0] <= din0;
    end
    else if (cs1 && we1) begin
        mem[address1] <= din1;
    end
end
end
```



# 双端口存储器设计（3）



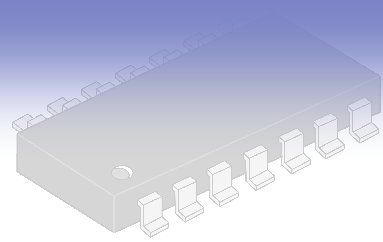
## □ （续）模块实现部分

### ▣ 存储器读操作

```
wire we = we0 | we1; // 防止写时读

// First Port of RAM output : When we = 0, oe0 = 1, cs0 = 1
assign dout0 = (cs0 && oe0 && !we) ? data0 : 8'bz;
// Read Operation : When we = 0, oe0 = 1, cs0 = 1
always @ (posedge clk) begin : MEM_READ_0
    if (cs0 && oe0 && !we) begin
        data0 <= mem[address0];
    end
    else begin
        data0 <= 8'b0;
    end
end
end
```

# 双端口存储器设计（3）



## ❑ （续）模块实现部分

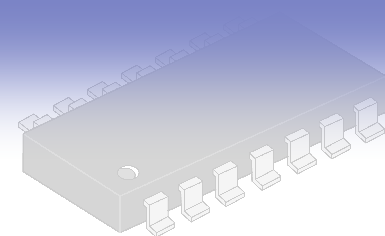
### ❑ 存储器读操作

```
// wire we = we0 | we1; ---- 防止写时读

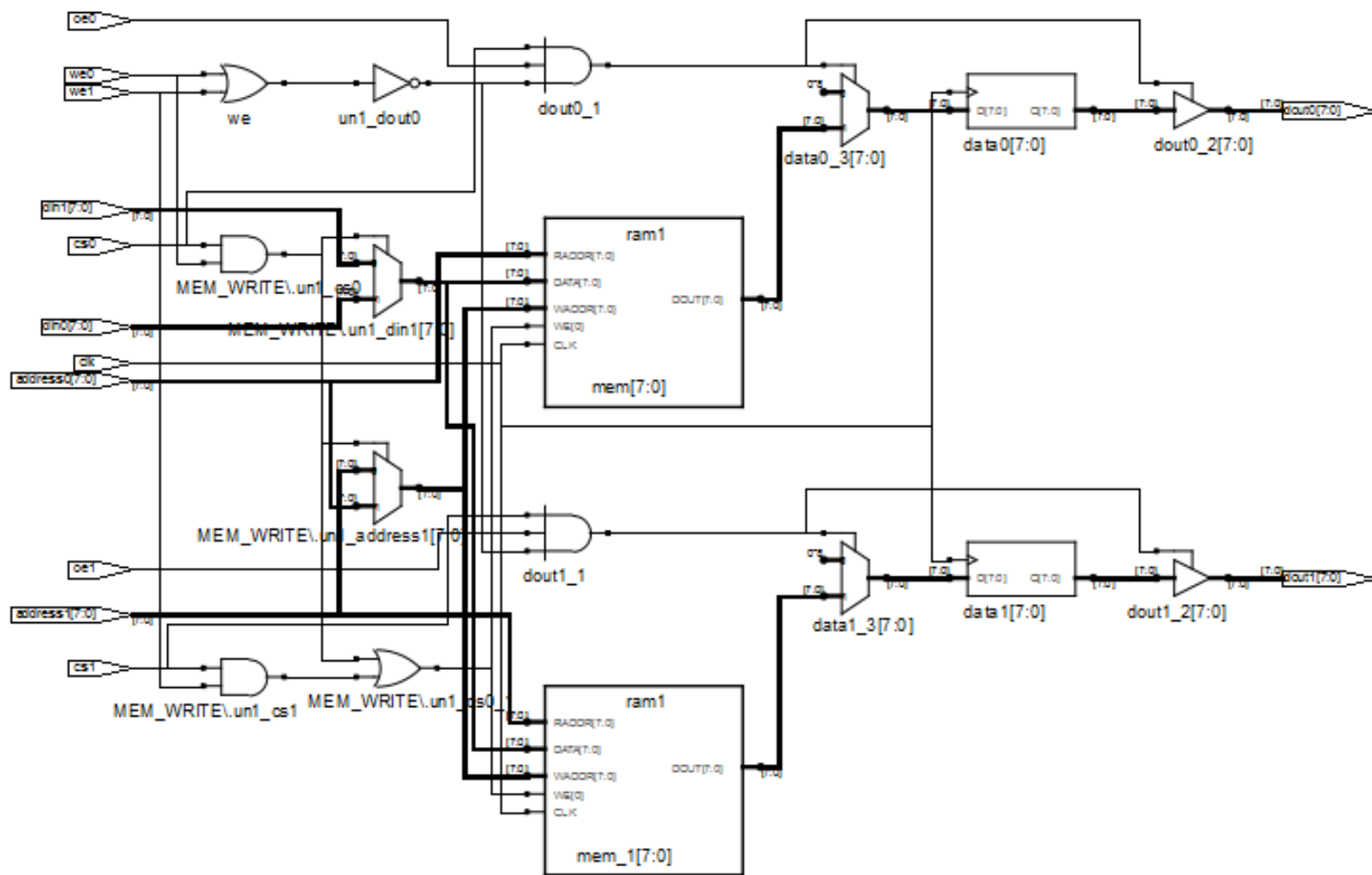
// Second Port of RAM output : When we = 0, oe1 = 1, cs1 = 1
assign dout1 = (cs1 && oe1 && !we) ? data1 : 8'bz;
// Read Operation : When we1 = 0, oe_1 = 1, cs_1 = 1
always @ (posedge clk) begin : MEM_READ_1
    if (cs1 && oe1 && !we ) begin
        data1 <= mem[address1];
    end
    else begin
        data1 <= 8'b0;
    end
end

endmodule // End of Module DualPortRAM
```

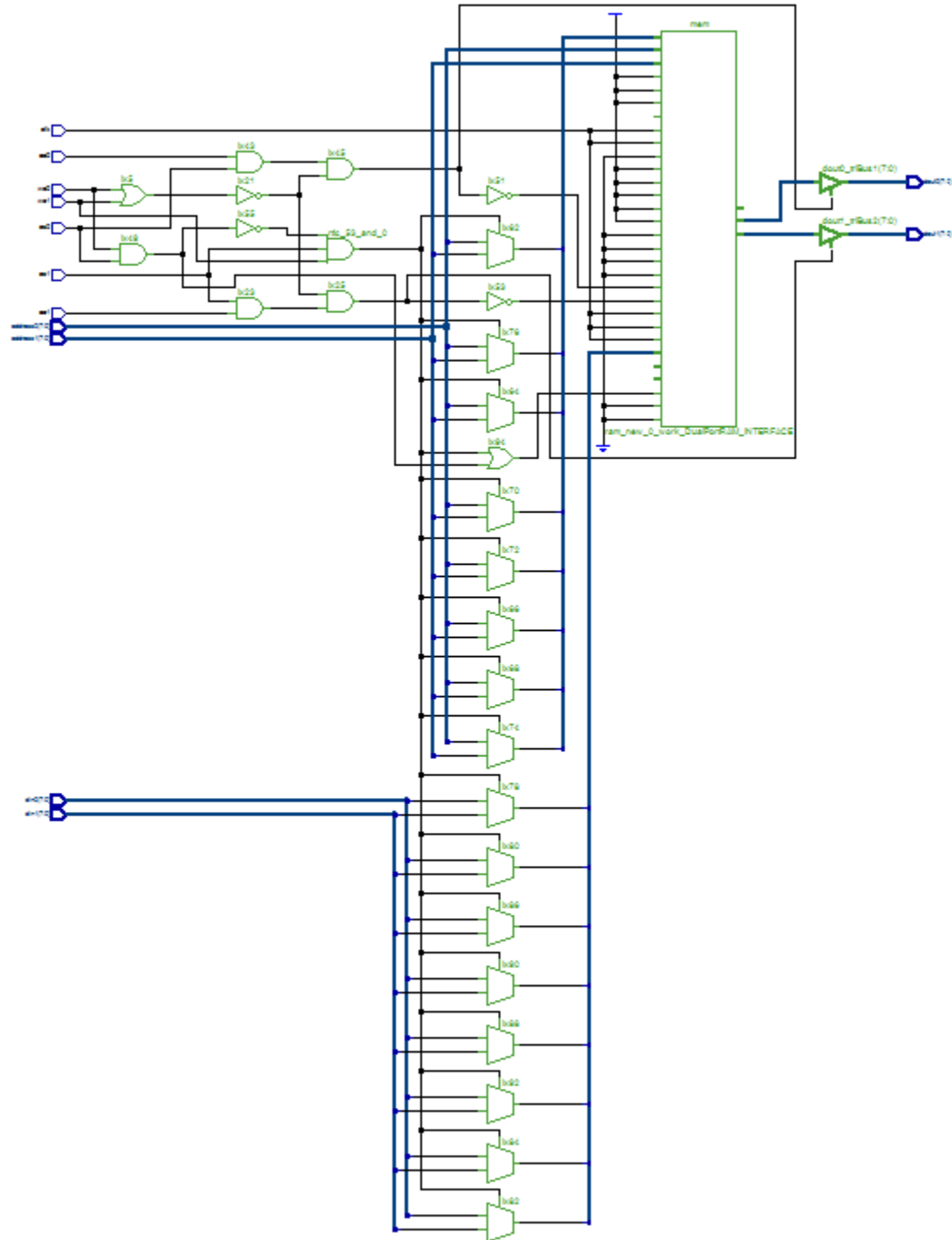
# 双端口RAM综合 (1)



## □ Synplify Pro E-2011.03-SP2



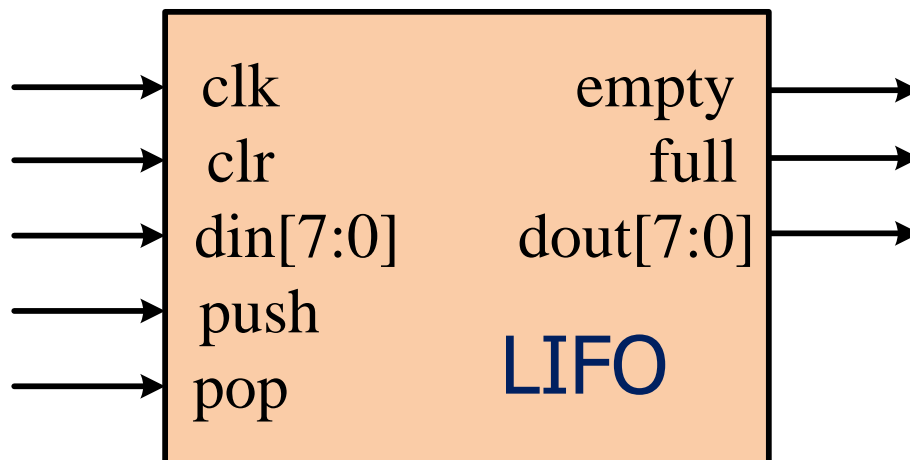
## 双端口RAM综合 (2)



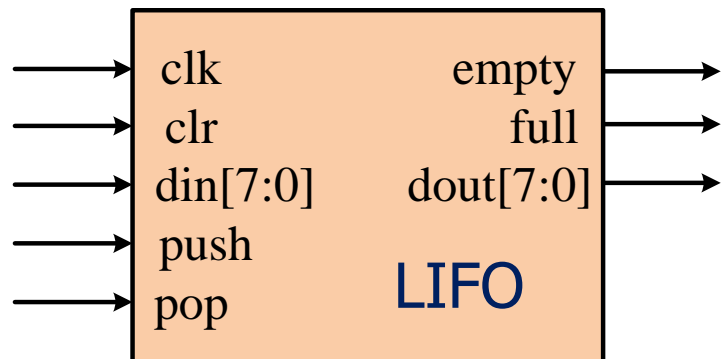
# 堆栈（LIFO）



- ❑ 存储器——实现“后入先出（last in, first out）”算法
  - ❑ 顺序将数据逐个存入存储区
  - ❑ 地址指针总指向最后一个压入堆栈的数据单元
  - ❑ 存放这个地址指针的寄存器就称为堆栈指示器
  - ❑ 开始放入数据的单元称为“栈底”
- ❑ 压入（push）
  - ❑ 在压栈的过程中，数据逐个地存入，
  - ❑ 每一个压入堆栈的数据，放在与前一个单元相连的后一个单元中
  - ❑ 堆栈指示器中的地址自动加1
- ❑ 弹出（pop）
  - ❑ 按照堆栈指示器中的地址读取数据，堆栈指示器中的地址数自动减1



# LIFO 设计



```

module LIFO(output reg [7:0] dout, // 输出数据
            output reg full,      // 栈满标志
            output reg empty,     // 栈空标志
            input clk,            // 时钟信号
            input clr,            // 清零
            input [7:0] din,      // 输入数据
            input push,           // 压栈
            input pop,            // 出栈
            );

```

```

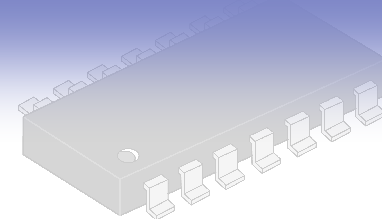
    reg [7:0] stack [0:7]; // 堆栈存储空间
    reg [3:0] addr;        // 堆栈指针, 栈底为 0

```

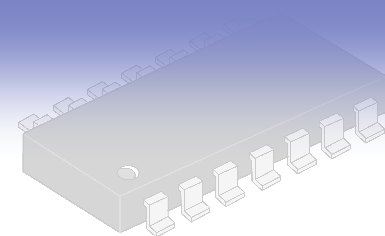
```

always @(posedge clk) begin
    if (clr) begin // 清零
        dout <= 8'h0;
        addr <= 4'h0;
        full <= 1'b0;
        empty <= 1'b1;
    end
    else begin
        case ( {pop, push} )
            2'b01: begin // 压栈
                if ( !full ) begin
                    if ( addr < 4'h8 ) begin
                        stack[addr] <= din; // 存入数据
                        addr <= addr + 1'b1; // 指针 + 1
                        empty <= 1'b0;
                    end
                end
            end
            2'b10: begin // 出栈
                if ( !empty ) begin
                    if ( addr > 4'h0 ) begin
                        dout <= stack[addr-1]; // 输出数据
                        addr <= addr - 1'b1; // 指针 - 1
                        full <= 1'b0;
                    end
                end
            end
            default:;
        endcase
    end
end
endmodule

```



# 有限状态机 (FSM, finite state machine)



## □ 状态机 —— 时序电路的通称

- 有限 —— 状态数是有限的
- 同步、异步

## □ 时钟同步状态机 (clocked synchronous state machine)

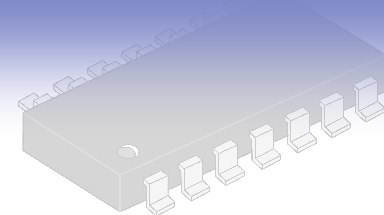
- 最常用
- 状态改变 —— 在时钟信号的触发边沿出现的时候, 才改变状态

## □ 状态机的结构

- 最常用的两种模型:
  - ◆ Mealy 机
  - ◆ Moore 机



# 时钟同步状态机结构 —— Mealy 机



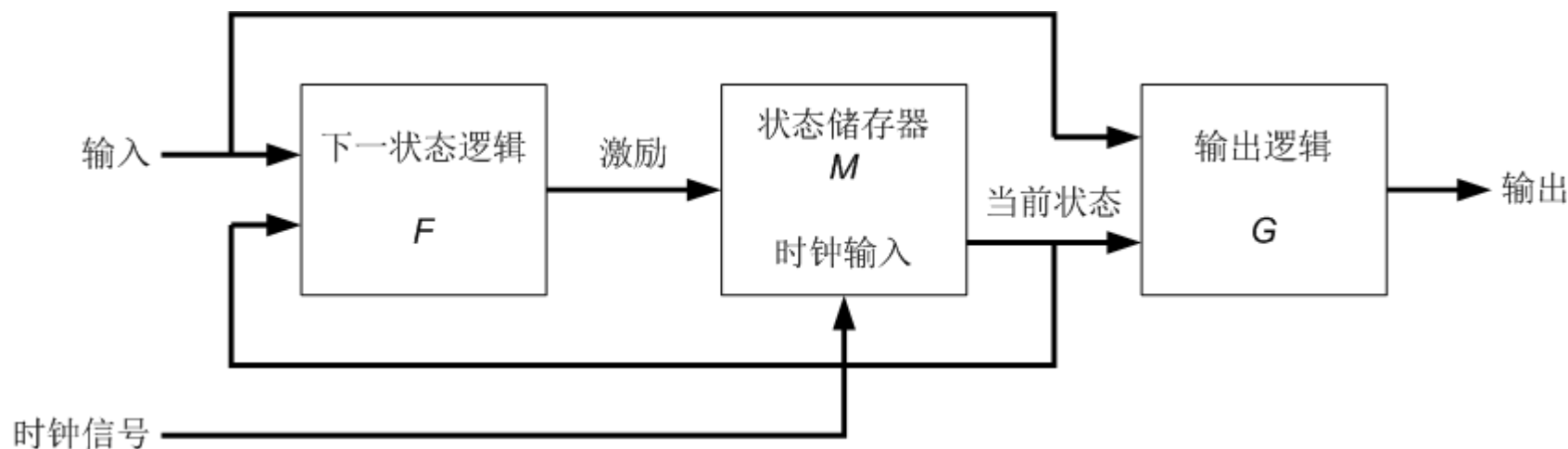
- 状态存储器
  - 储存状态机当前状态的一组触发器
    - ◆  $n$  个触发器具有  $2^n$  种不同的状态
  - 所有触发器都连接到一个公共时钟信号
- 下一状态逻辑 (next-state logic)  $F$ 
  - 输入变换 —— 当前状态和输入的函数
- 输出逻辑 (output logic)  $G$ 
  - 输出变换 —— 当前状态和输入的函数
- $F$  和  $G$  都是组合逻辑电路

Mealy 机:

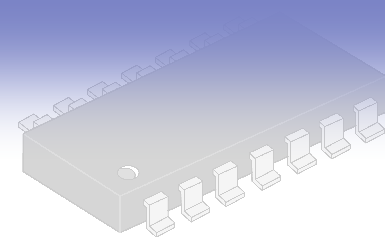
输出同时取决于状态和输入

下一状态 =  $F$ (当前状态, 输入)

输出 =  $G$ (当前状态, 输入)



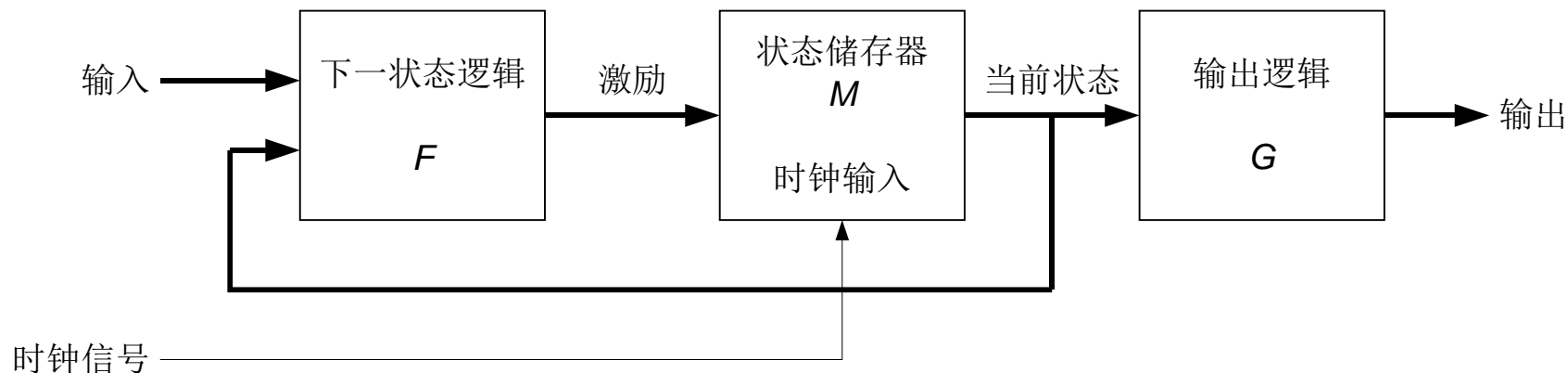
# 时钟同步状态机结构 —— Moore 机



## □ Moore机

### □ 输出只由状态决定

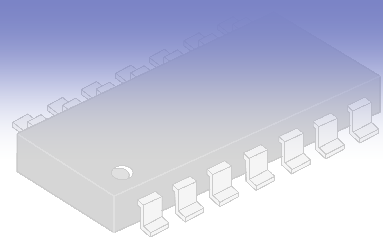
输出 =  $G(\text{当前状态})$



## □ 两种状态机模型可以互相转换

□ 实际设计中，准确地分类并不重要，关键是满足设计目标的需要

# 状态图



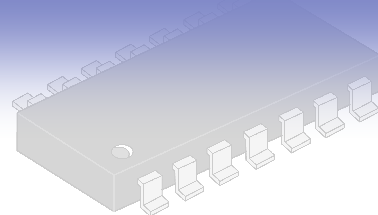
## □ 表示在各种输入条件下的状态转换

- 每个状态用一组状态变量表示
- 每一个状态用于一个圆圈表示
  - ◆ 圈内有标识该状态名字的符号（状态名）
  - ◆ 二进制值——状态变量的值
- 从一个状态到另一个状态的转换用有向弧表示
  - ◆ 引起状态变化的输入和作为结果的输出标注在有向弧旁边

## □ 状态变量 —— 触发器的输出

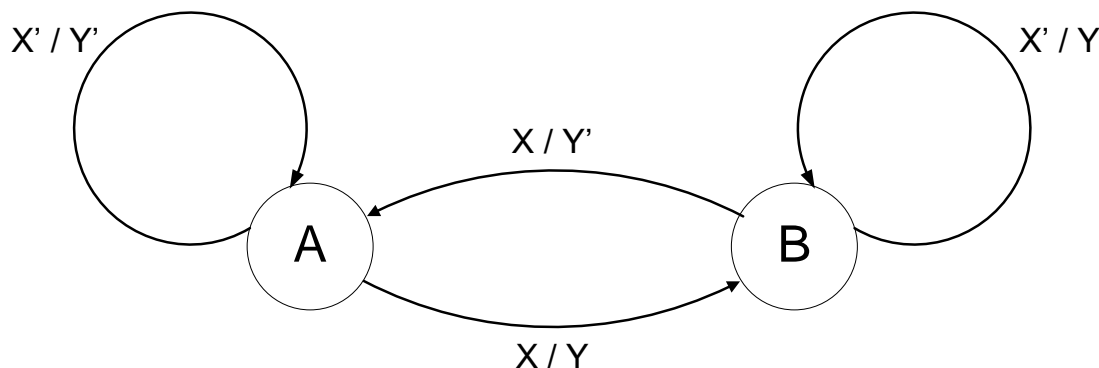
- 现态
  - ◆ 在下一时钟边沿（脉冲）到来之前的某个时刻  $t$ ，一组状态变量改变前的二进制值
- 次态
  - ◆ 在下一个时钟脉冲到来之后的某个时刻  $t+1$ ，所有状态变量呈现的新值

# Mealy 机的状态图表示



## □ 例

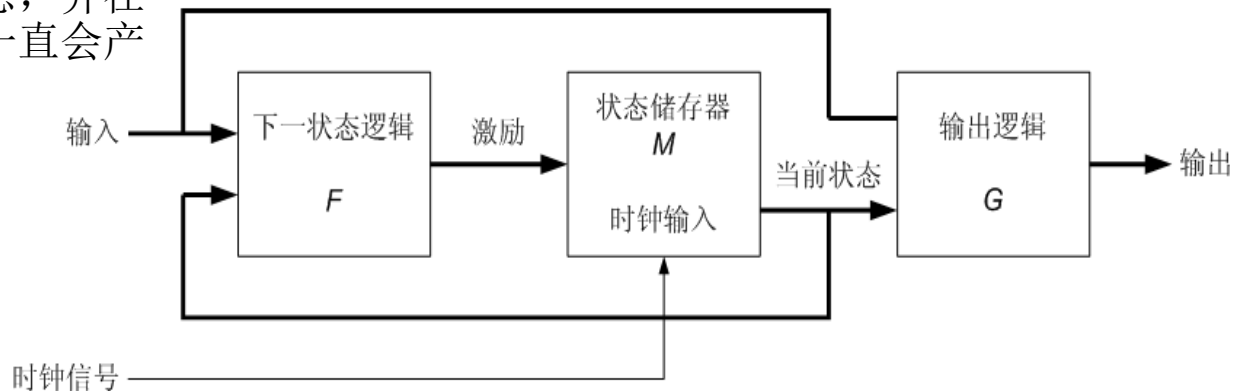
- 单输入、单输出的 Mealy 机
- $X$  —— 输入变量
- $Y$  —— 输出变量
- $A$  和  $B$  —— 表示状态的符号，也就是状态名
- $X/Y$  —— 输入/输出
  - ◆ 左边 —— 输入变量
  - ◆ 右边 —— 输出的结果
- 现态：可以是  $A$ ，也可以是  $B$
- 次态： $A$  或  $B$



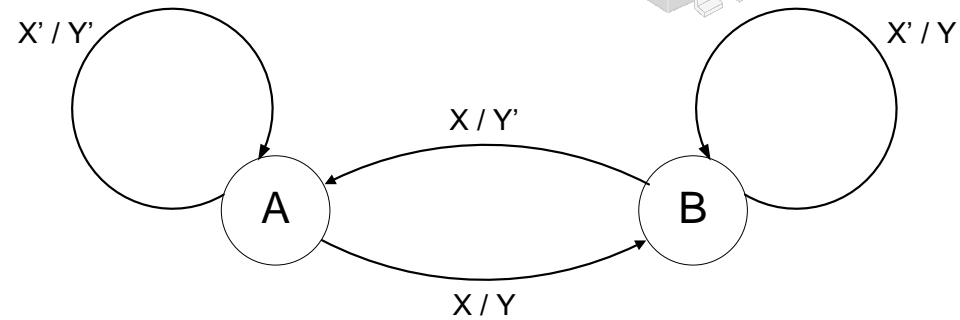
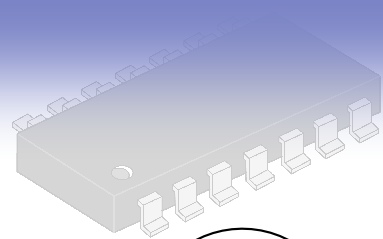
## □ 由于输出既是现态的函数，又是输入变量的函数 —— Mealy机

## □ Mealy机的输出值

- 当状态机处于所示的状态，并在所示的输入作用下，就一直会产生图中所列的输出值



# Mealy 机的Verilog 模块



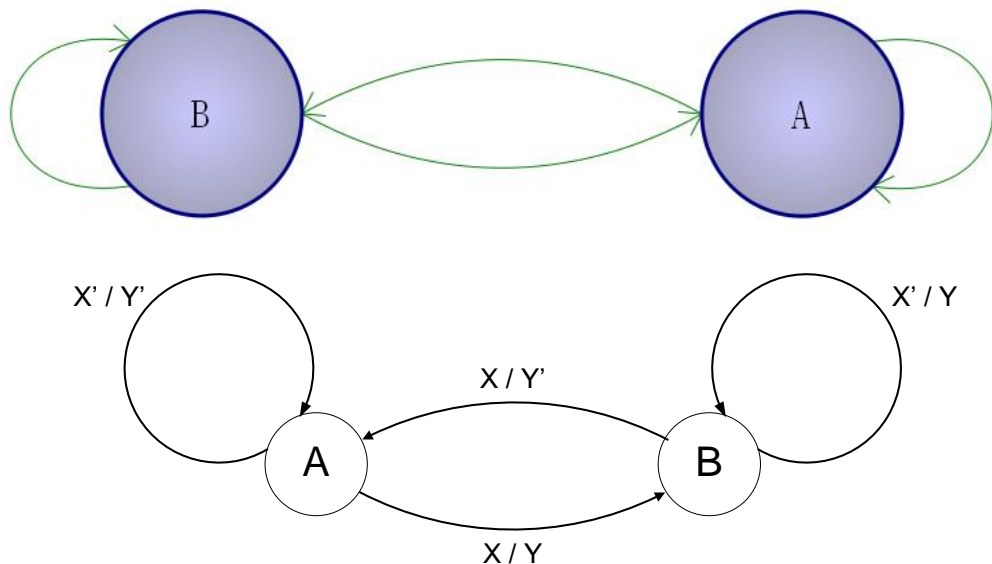
```
module mealy( output reg yout, input clk, rst, xin );
    parameter A=2'b01, B=2'b10;
    reg [1:0] state;

    always @(posedge clk) begin
        if (!rst) begin state <= A; yout <= 1'b0; end
        else
            case(state)
                A:      if (xin) begin yout <= 1'b1; state <= B; end
                        else begin yout <= 1'b0; state <= A; end
                B:      if (xin) begin yout <= 1'b0; state <= A; end
                        else begin yout <= 1'b1; state <= B; end
                default:begin yout <= 1'b0; state <= A; end
            endcase
    end
endmodule
```

# Mealy 机的综合结果

## □ 综合结果

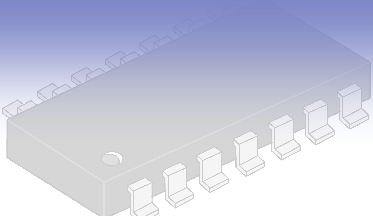
	From State	To State	
1	B	B	rst@!xin
2	A	B	rst@xin
3	B	A	xin
4	B	A	!rst
5	A	A	!xin
6	A	A	!rst



```
module mealy( output reg yout, input clk, rst, xin );
    parameter A=2'b01, B=2'b10;
    reg [1:0] state;

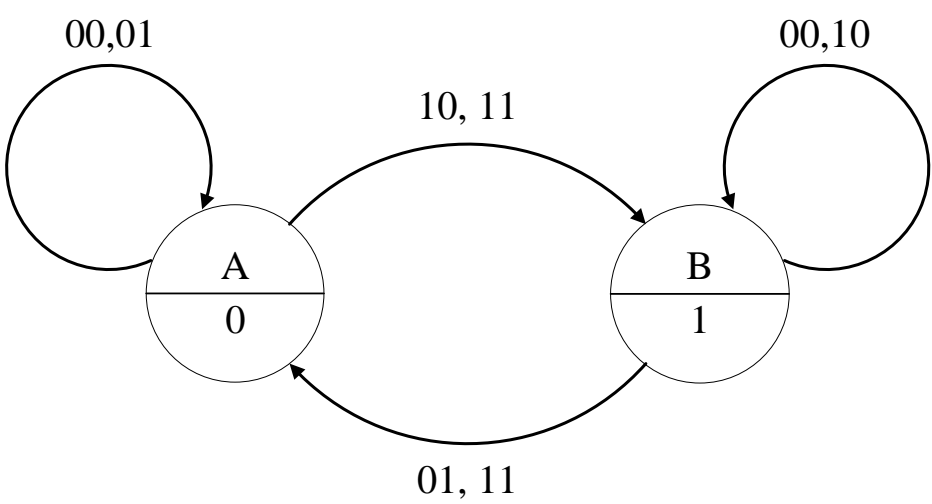
    always @(posedge clk) begin
        if (!rst) begin state <= A; yout <= 1'b0; end
        else
            case(state)
                A:    if (xin) begin yout <= 1'b1; state <= B; end
                     else begin yout <= 1'b0; state <= A; end
                B:    if (xin) begin yout <= 1'b0; state <= A; end
                     else begin yout <= 1'b1; state <= B; end
                default:begin yout <= 1'b0; state <= A; end
            endcase
    end
endmodule
```

# Moore 机的状态图表示



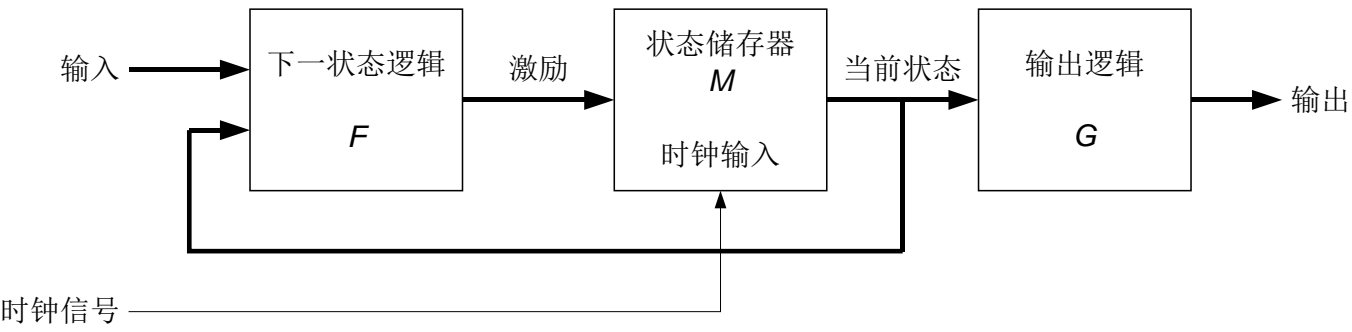
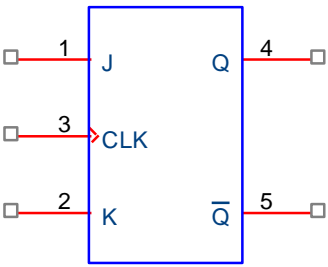
## JK触发器的Moore 电路表示

- 状态 A:  $QQ' = 01$
- 状态 B:  $QQ' = 10$
- 输入  $JK = 00, 01, 10, 11$

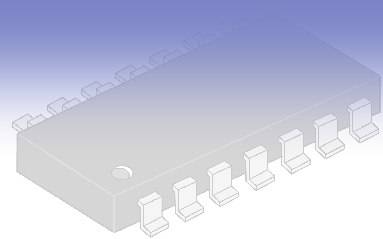


正边沿触发式 J-K' 触发器真值表

J	K	CLK	Q	QN
X	X	0	上一个 Q 值	上一个 QN 值
X	X	1	上一个 Q 值	上一个 QN 值
0	0		上一个 Q 值	上一个 QN 值
0	1		0	1
1	0		1	0
1	1		上一个 QN 值	上一个 Q 值

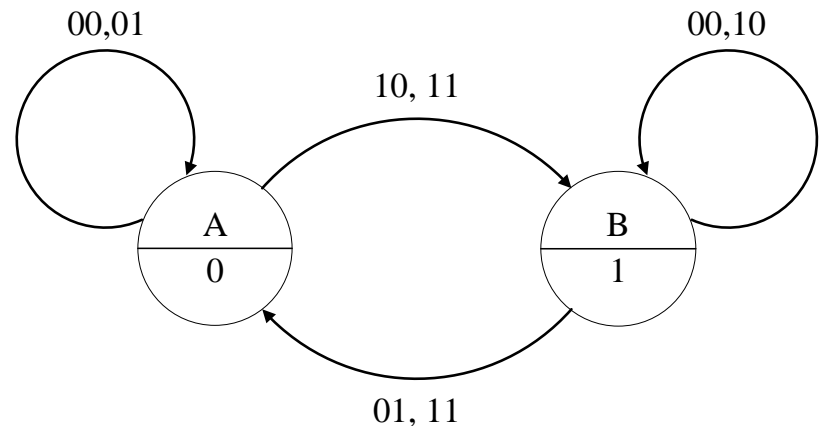


# Moore 机的Verilog 模块



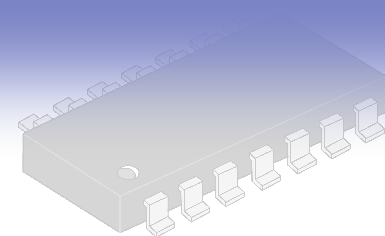
```
module Moore_JK( output reg q, qn, input clk, rst, j, k );
    parameter A=2'b01, B=2'b10;
    reg [1:0] state;

    always @(posedge clk) begin
        if (!rst) begin {q,qn} <= 2'b01; state <= A; end
        else begin
            case(state)
                A:   begin
                    {q,qn} <= 2'b01;
                    state <= (j==1'b1) ? B : A;
                end
                B:   begin
                    {q,qn} <= 2'b10;
                    state <= (k==1'b1) ? A : B;
                end
                default: ;
            endcase
        end
    end
endmodule
```





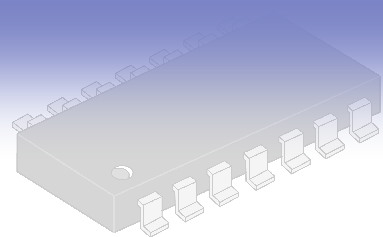
# 利用状态图设计同步状态机的基本步骤



## □ 利用状态图进行设计的基本步骤

- ① 根据功能（问题描述）要求，确定：
  - ◆ 输入变量的个数 —— 选择输入变量
  - ◆ 输出变量的个数 —— 选择输出变量
  - ◆ 状态变量的个数
- ② 对状态进行编号，或用标识符（助记符）给状态命名
- ③ 构造状态图
  - ◆  $n$  个输入变量的状态转换数为  $2^n$
  - ◆ 对于一个有  $s$  位状态及  $k$  位输入的状态机，共有  $2^{s+k}$  状态/输入组合
  - ◆ 对于大量的输入变量，只表示引起状态变换或输出变化的变量组合
    - 未指定的输入变量组合默认为状态机保持现态
- ④ 选择一组状态变量
- ⑤ 状态分配（或，状态编码，state encoding）
  - ◆ 对每个状态赋给一个状态变量的取值组合
  - ◆ 有多种方案可以选择 —— 通常采用独热码（one-hot encoding）
- ⑥ 设计 Verilog 模块

# 根据状态图设计 Verilog 模块



## ❑ 首先根据要求选择状态机的置位与复位方式

- ❑ 两种方式：异步、同步
- ❑ 两种方式的 Verilog 模块的事件控制方法不同

## ❑ 异步置位与复位

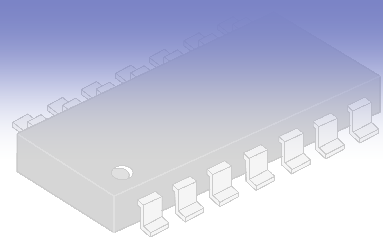
- ❑ 置位与复位与时钟信号无关
- ❑ 当置位与复位脉冲到来时，立即将触发器的输出端置 1 或 0

## ❑ 实现异步置位与复位的事件控制语法：

always @( 边沿关键字 clock or 边沿关键字 reset or 边沿关键字 set )

- ❑ 边沿关键字
  - ◆ 正边沿：posedge
  - ◆ 负边沿：negedge

# 状态机设计方法



## □ 根据状态图

- 未指定的输入变量组合默认为状态机保持现态

## □ Mealy 机

- 输出既依赖于现态，又取决于输入

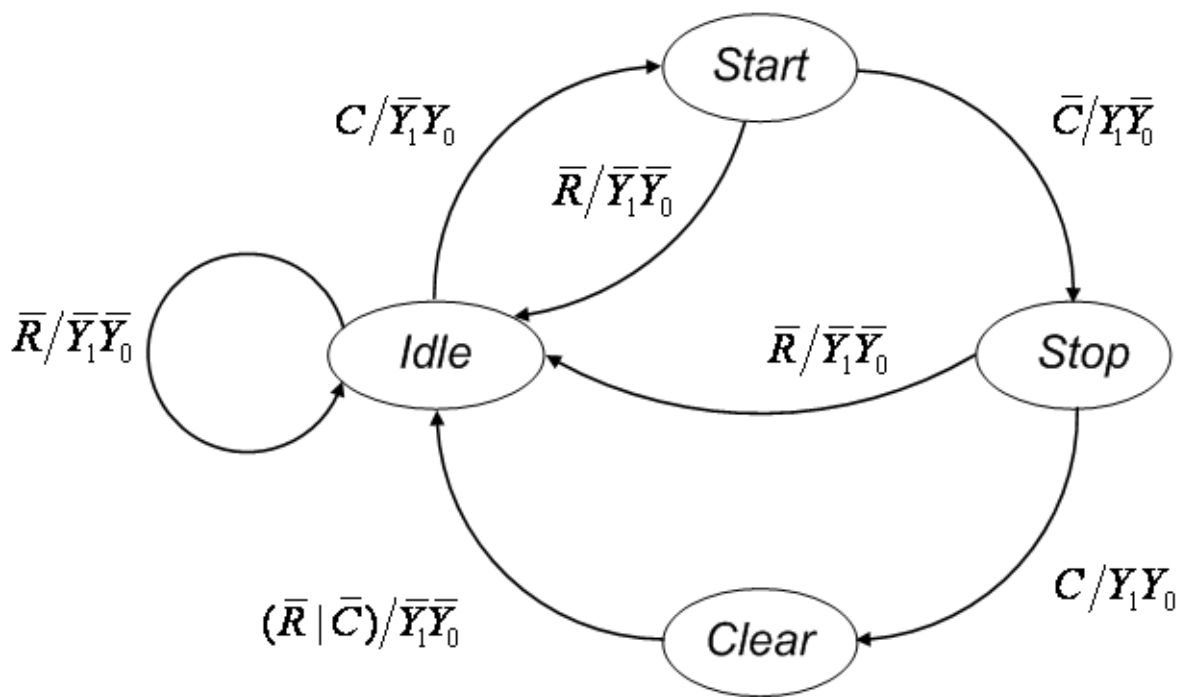
## □ 输入

- 复位 Reset: 记为 R
- 控制 Ctrl: 记为 C

## □ 输出

- $Y_1$ 、 $Y_0$

## □ 在设计时，使用同步复位



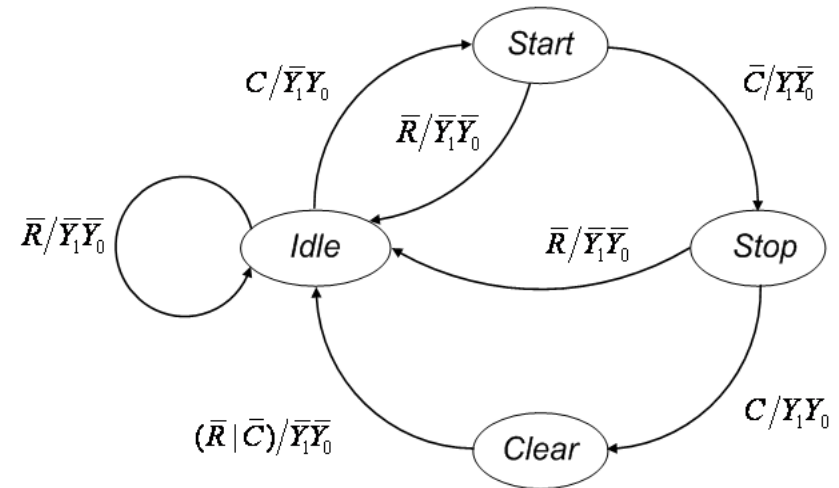
```

module FSM( output reg [1 : 0] Y, input clk, ctrl, reset );
    reg [3 : 0] state;
    parameter IDLE = 4'b0001, START = 4'b0010, STOP = 4'b0100, CLEAR = 4'b1000;

    always @(posedge clk)
        if (!reset) begin
            state <= IDLE; Y <= 2'b00;
        end
        else
            case (state)
                IDLE: begin
                    if (ctrl) begin
                        state <= START; Y <= 2'b01;
                    end
                    else
                        state <= IDLE;
                    end
                START: begin
                    if (!ctrl) begin
                        state <= STOP; Y <= 2'b10;
                    end
                    else state <= START;
                    end
                STOP: begin
                    if (ctrl) begin
                        state <= CLEAR; Y = 2'b11;
                    end
                    else state <= STOP;
                    end
                CLEAR: begin
                    if (!ctrl) begin
                        state <= IDLE; Y = 2'b00;
                    end
                    else state <= CLEAR;
                    end
                default: state <= IDLE;
            endcase
    endmodule

```

## FSM的 Verilog 模块



```
`timescale 1ns / 100ps
```

```
`include "FSM.v"
```

```
module FSM_tb();
```

```
  wire [1 : 0] p_y;
```

```
  reg p_c, p_res, p_clk;
```

```
  reg [7:0] p_data;
```

```
  initial begin
```

```
    p_clk = 0;
```

```
    forever begin
```

```
      p_clk = #6 1; p_clk = #4 0;
```

```
    end
```

```
end
```

```
  initial begin
```

```
    p_c = 0;
```

```
    forever begin
```

```
      #6 p_c = ~p_c;
```

```
      p_data = {$random} % 256;
```

```
      #1 p_res = p_data[3];
```

```
    end
```

```
end
```

```
  initial
```

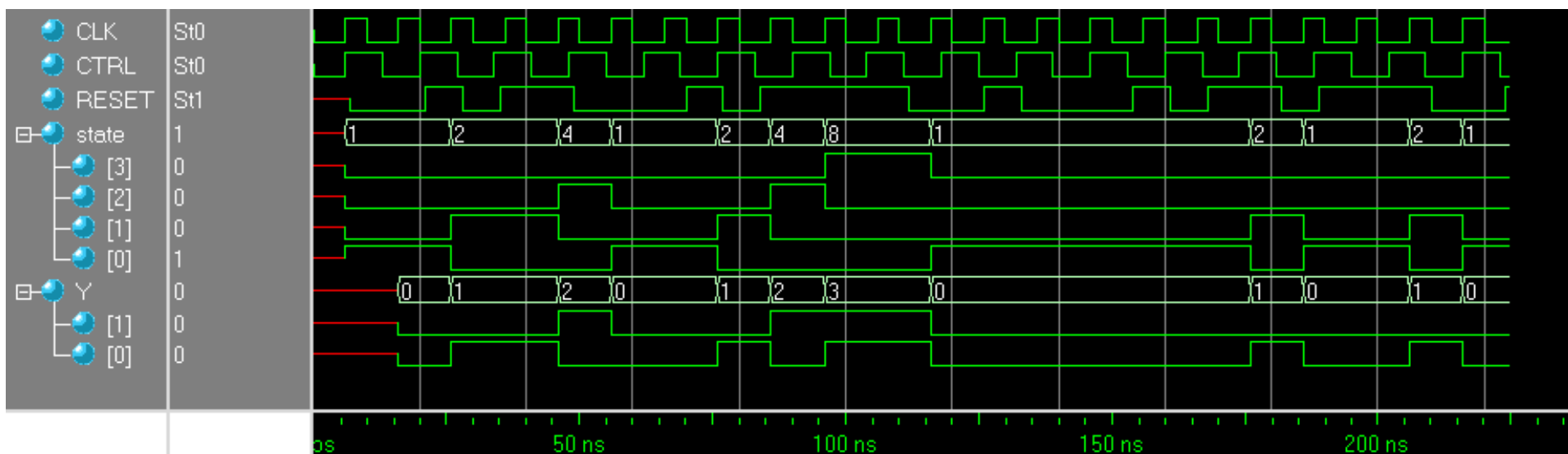
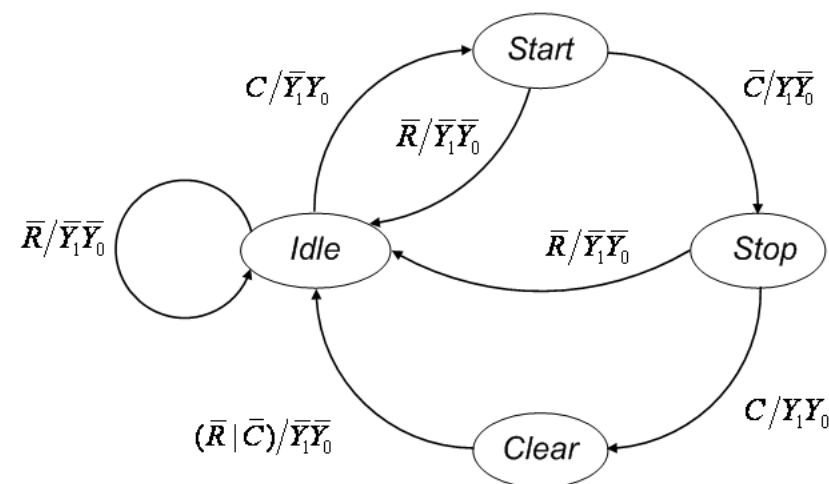
```
    $monitor("At time %t, Y=%b, CLK=%b, CTRL=%b, RESET=%b, state=%b",
```

```
      $time, p_y, p_clk, p_c, p_res, u0.state );
```

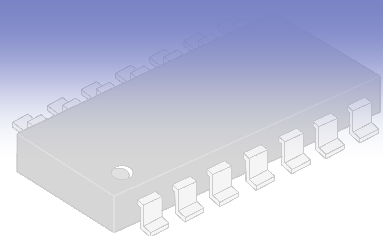
```
  FSM u0( .Y(p_y), .clk(p_clk), .ctrl(p_c), .reset(p_res) );
```

```
endmodule
```

## FSM的仿真波形



# 序列检测器（1）



## □ 将一个指定的序列从输入数字序列中检测出来

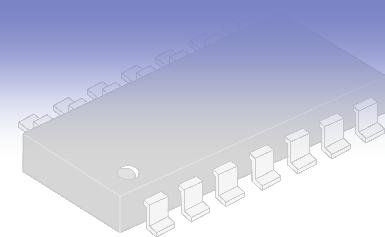
### ① 功能描述

- 检测串行输入数字序列中的指定序列：10010
- 如果在输入序列中检测到一个指定序列，则设置发现标志为 1，否则，置 0
- 指定序列以重叠形式出现时，也认为是有效的被检测序列
  - ◆ 如：10010010

### ② 输入/输出

- 3 个输入变量：
  - ◆ CLK —— 时钟信号
  - ◆ RESET —— 异步复位信号
  - ◆ S\_IN —— 表示输入序列，如：110010010000100101...
- 1 个输出变量
  - ◆ FLAG —— 检出标志
    - 如果检测到一个指定序列，则置 1，否则，置 0

## 序列检测器 (2)



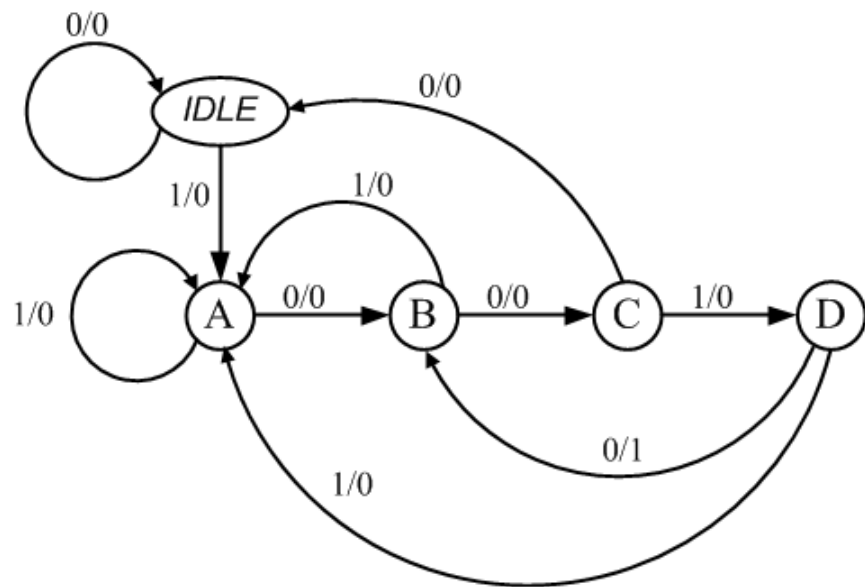
### ③ 采用Mealy型状态机，构造状态图

- 如果无法先确定状态的个数，先构造状态图
  - ◆ 然后，确定状态个数、状态变量的个数
- 构造过程
  - ◆ 状态 Idle: 等待第一个 1, ...

### ④ 状态个数: 5

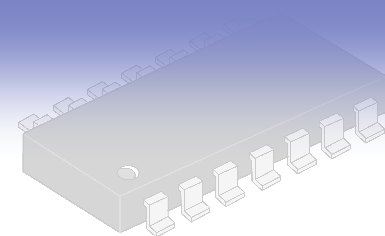
### ⑤ 状态命名 (编号): IDLE、A、B、C、D

10010010



000110010001000010010010100010...

# 序列检测器（3）



## ⑥ 选择一组状态变量

- 使用 5 个状态变量：state[4:0]

## ⑦ 状态分配（编码）

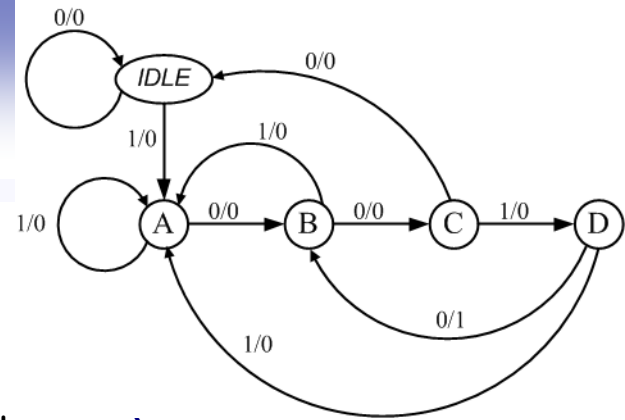
- 采用独热码
  - ◆ 0\_0001、0\_0010、0\_0100、0\_1000、1\_0000

## ⑧ Verilog 模块设计：

- 使用异步复位
  - ◆ 非阻塞性赋值
- 分成两部分设计
  - ① 状态转换部分 —— always 和 case 语句结构
    - 正边沿触发方式
    - 每一个状态转换用一个 case 选项
  - ② 状态、输入/输出组合逻辑变换部分

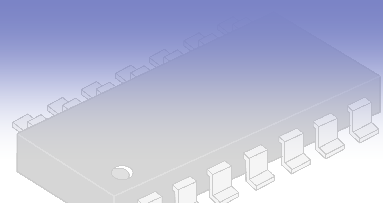


# 序列检测器 Verilog 模块设计



```
module seq_recognize ( output reg flag, input clk, reset, seq);
    parameter IDLE = 5'b0_0001, A = 5'b0_0010, B = 5'b0_0100,
               C = 5'b0_1000, D = 5'b1_0000;
    reg [4 : 0] state;
    always @(posedge clk, negedge reset)
        if (!reset) begin flag <= 1'b0; state <= IDLE; end
        else begin
            case (state)
                IDLE:    if (seq) begin flag <= 1'b0; state <= A; end
                           else begin flag <= 1'b0; state <= IDLE; end
                A:      if (seq) begin flag <= 1'b0; state <= A; end
                           else begin flag <= 1'b0; state <= B; end
                B:      if (seq) begin flag <= 1'b0; state <= A; end
                           else begin flag <= 1'b0; state <= C; end
                C:      if (seq) begin flag <= 1'b0; state <= D; end
                           else begin flag <= 1'b0; state <= IDLE; end
                D:      if (seq) begin flag <= 1'b0; state <= A; end
                           else begin flag <= 1'b1; state <= B; end
                default: begin flag <= 1'b0; state <= IDLE; end
            endcase
        end
end
endmodule
```

# 序列检测器模块仿真



```
module seq_recognize_tb;
    wire p_flag;
    reg  p_clk, p_rst, p_s;

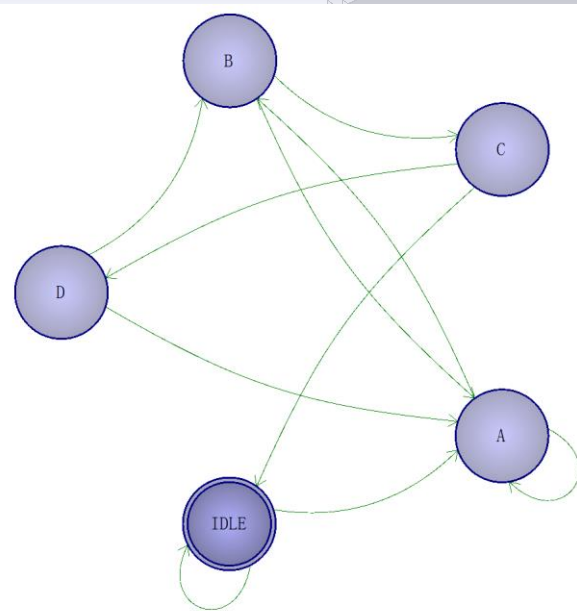
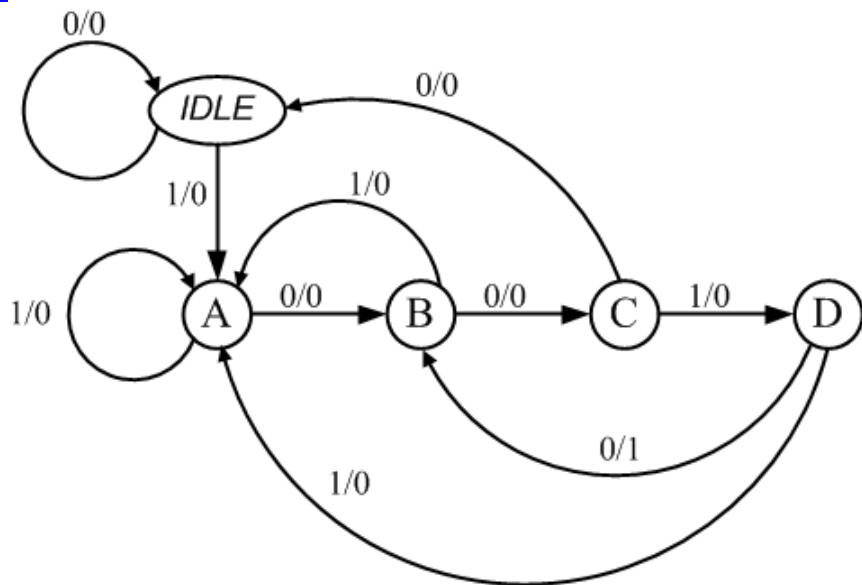
    initial begin p_rst = 1'b0; #25 p_rst = 1'b1; end
    initial begin p_clk = 0;  forever #10 p_clk = ~p_clk; end

    parameter SIZE = 32;
    reg [SIZE-1 : 0] data = 32'b0110_0100_1000_0100_1010_1011_0001_0100

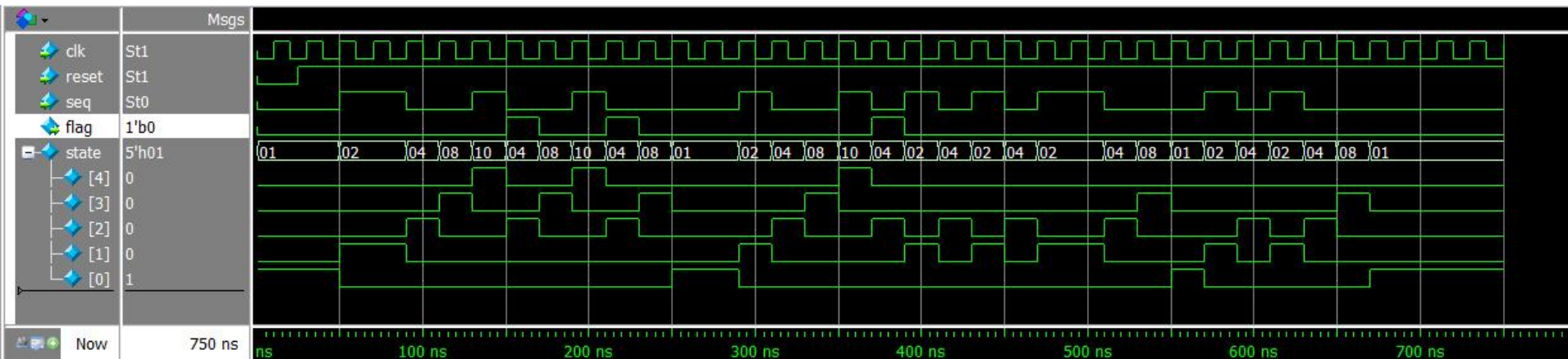
    initial begin: SERIES
        integer i;
        p_s = 0;
        #30;
        for ( i = 0; i < SIZE; i= i+1) begin
            p_s = data[SIZE -1];
            data = data << 1;
            #20;
        end
    end

    seq_recognize u( .flag(p_flag), .clk(p_clk), .reset(p_rst), .seq(p_s) );
endmodule
```

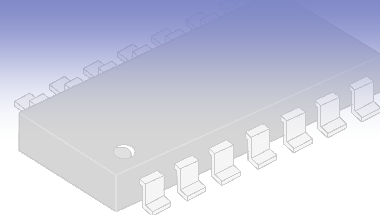
# 序列检测器模块测试



□ 输入序列：0110\_0100\_1000\_0100\_1010\_1011\_0001\_0100

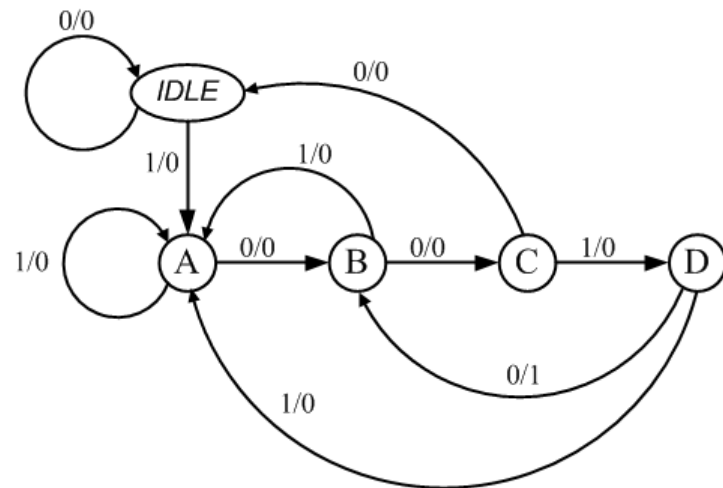
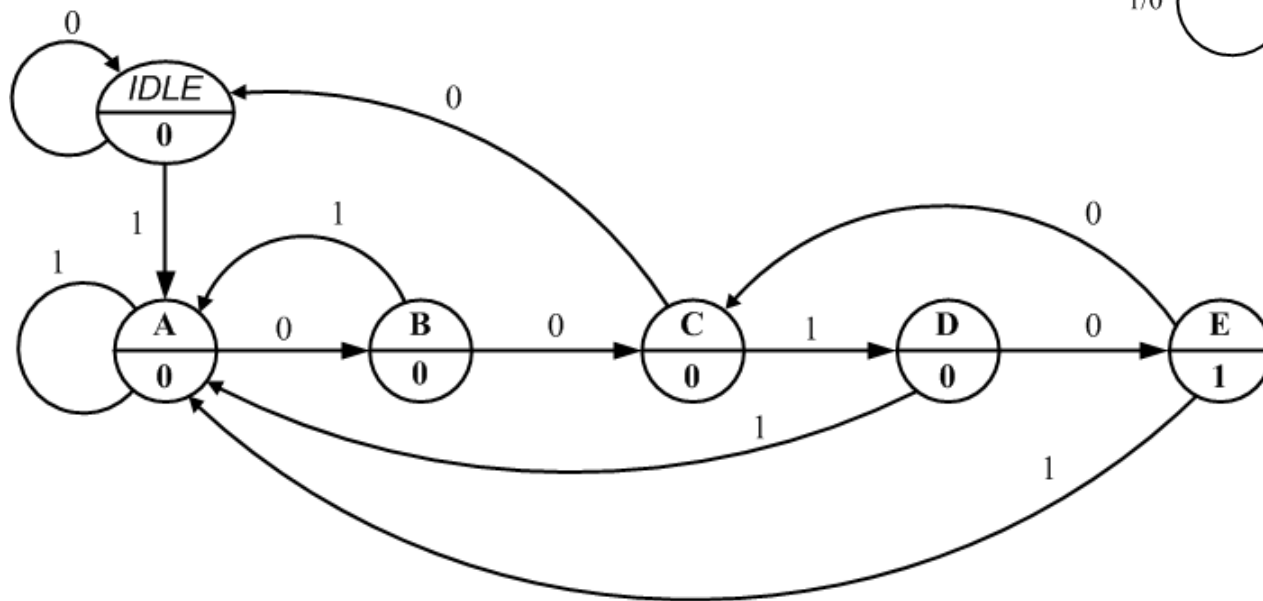


# 用Moore型状态机设计序列检测器（1）



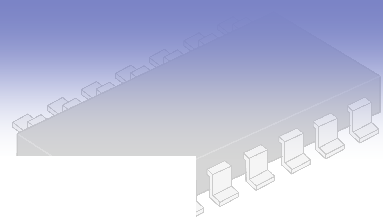
- 序列检测器的Moore型状态机的状态个数：6
  - Mealy型状态机的状态个数：5
- 状态命名（编号）：IDLE、A、B、C、D、E

00010010001000010010010100010...



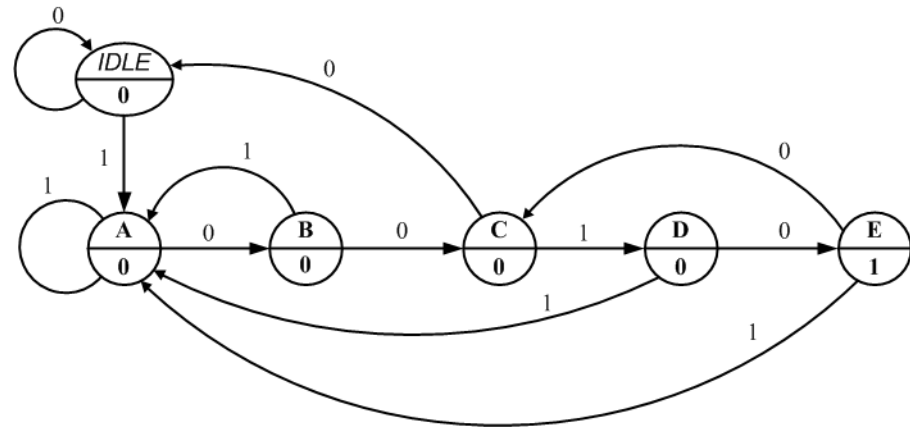
10010010

# 用Moore型状态机设计序列检测器（2）

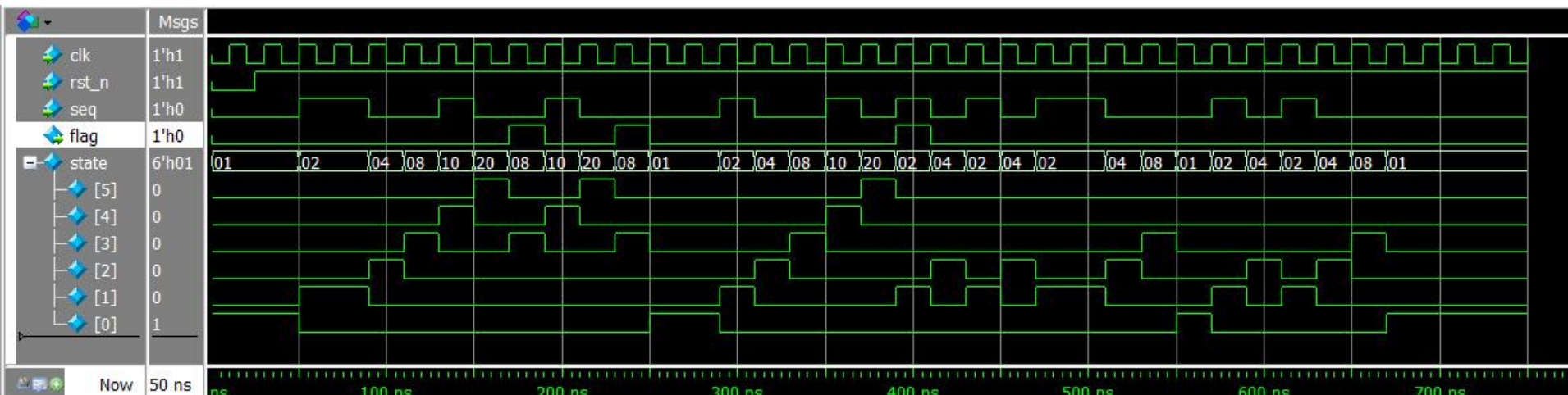
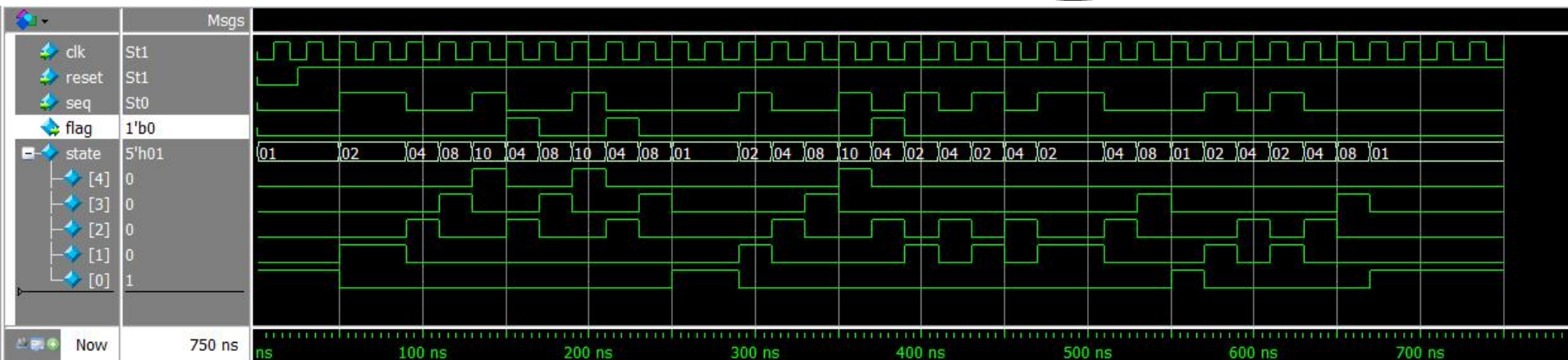
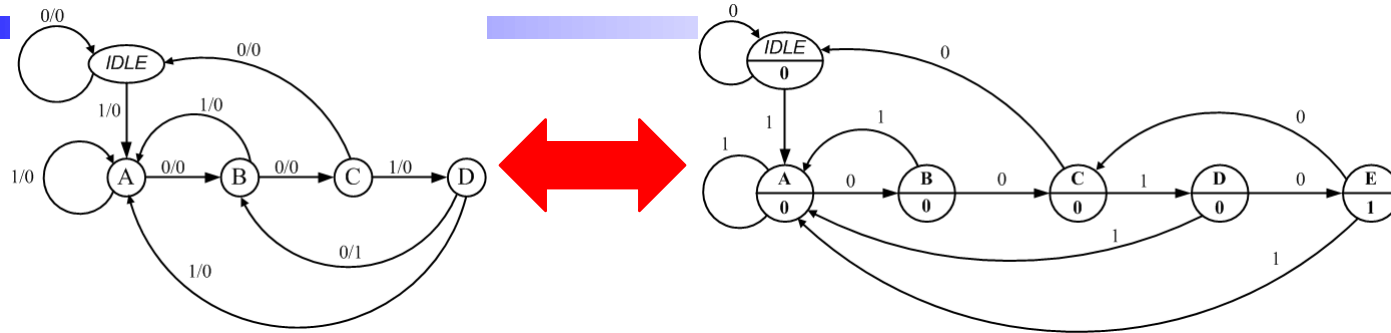
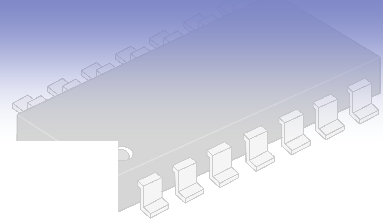


```
module seq_recognize ( output reg flag, input clk, rst_n, seq);
    localparam IDLE = 6'b00_0001, A = 6'b00_0010, B = 6'b00_0100,
                C = 6'b00_1000, D = 6'b01_0000, E = 6'b10_0000;

    reg [5:0] state;
    always @(posedge clk, negedge rst_n) begin
        if (!rst_n) begin
            flag <= 1'b0;
            state <= IDLE;
        end
        else begin
            flag <= (state == E) ? 1'b1 : 1'b0;
            case (state)
                IDLE: state <= (seq) ? A : IDLE;
                A: state <= (seq) ? A : B;
                B: state <= (seq) ? A : C;
                C: state <= (seq) ? D : IDLE;
                D: state <= (seq) ? A : E;
                E: state <= (seq) ? A : C;
                default: state <= IDLE;
            endcase
        end
    end
endmodule
```



# Mealy型与Moore型状态机比较

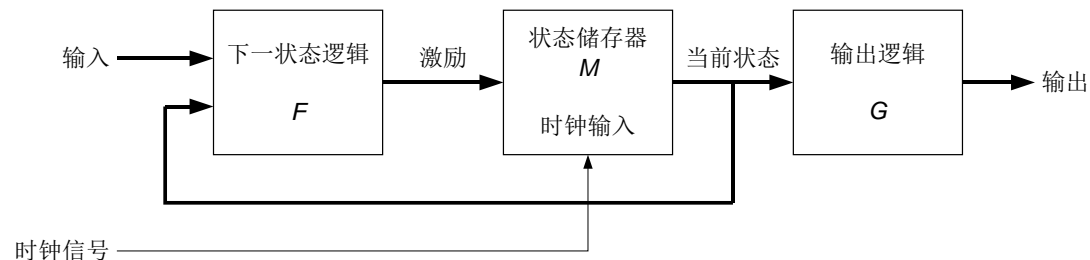
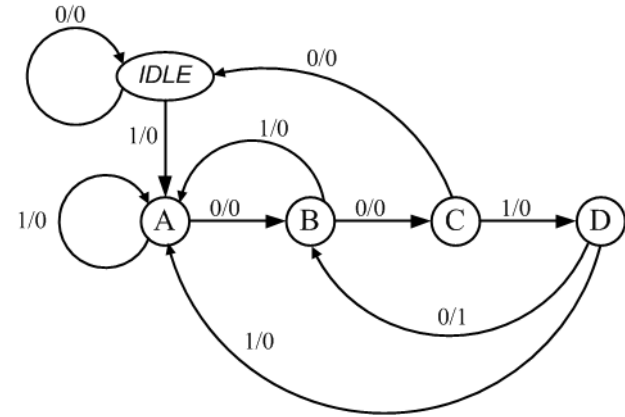


# 序列检测器使用Mealy型状态机（1）

```
module seq_recognize ( output reg flag, input clk, reset, seq);
    parameter IDLE = 5'b0_0001, A = 5'b0_0010, B = 5'b0_0100,
               C = 5'b0_1000, D = 5'b1_0000;
    reg [4 : 0] p_state, n_state;

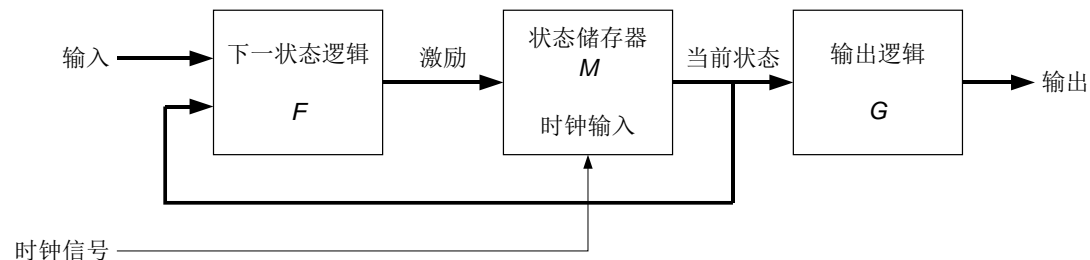
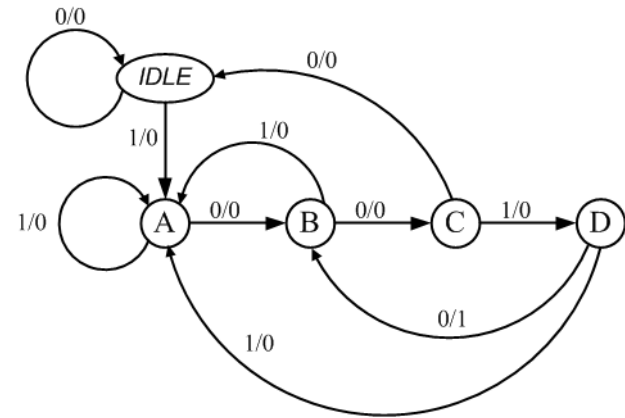
    always @(posedge clk, negedge reset)
        if (!reset) p_state <= IDLE;
        else p_state <= n_state;

    always @(*) begin
        case (p_state)
            IDLE: n_state = (seq) ? A : IDLE;
            A: n_state = (seq) ? A : B;
            B: n_state = (seq) ? A : C;
            C: n_state = (seq) ? D : IDLE;
            D: n_state = (seq) ? A : B;
            default: n_state = IDLE;
        endcase
        flag = ((p_state == D) && (seq == 1'b0)) ? 1'b1 : 1'b0;
    end
endmodule
```

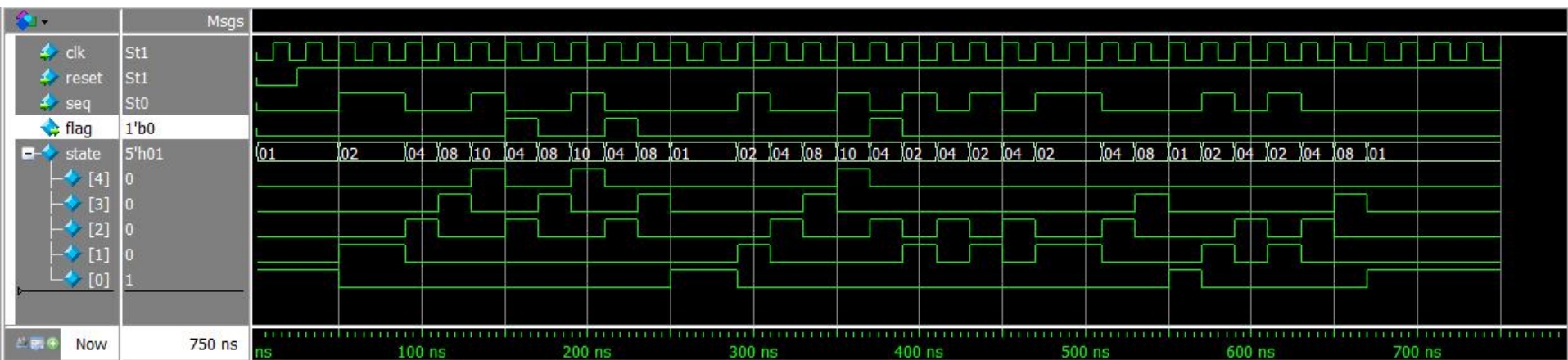
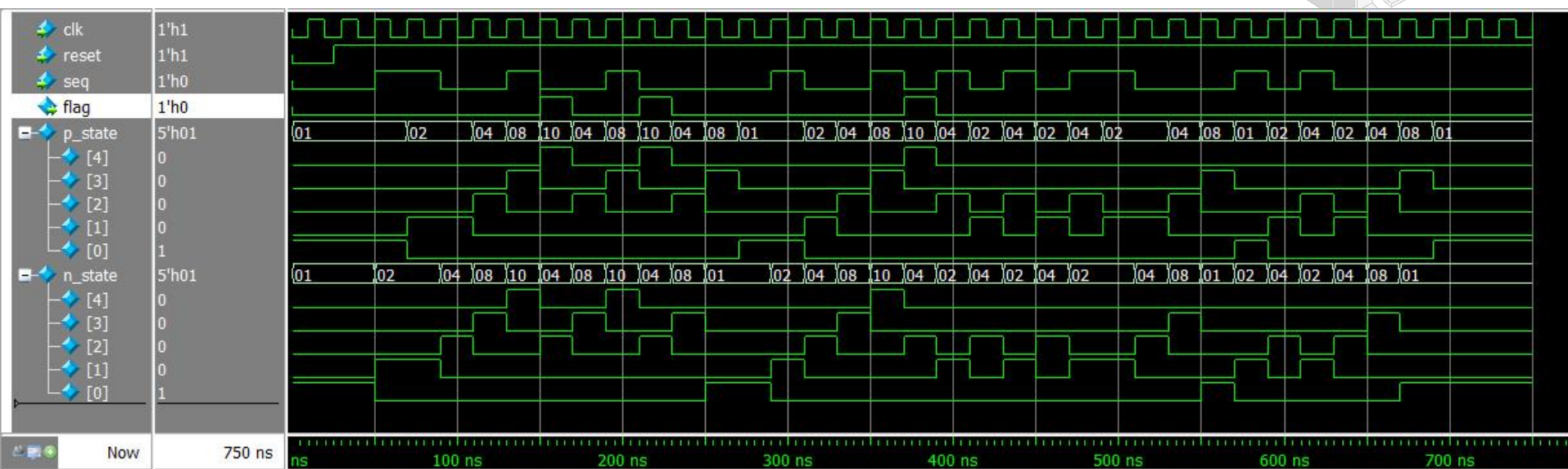


# 序列检测器使用Mealy型状态机 (2)

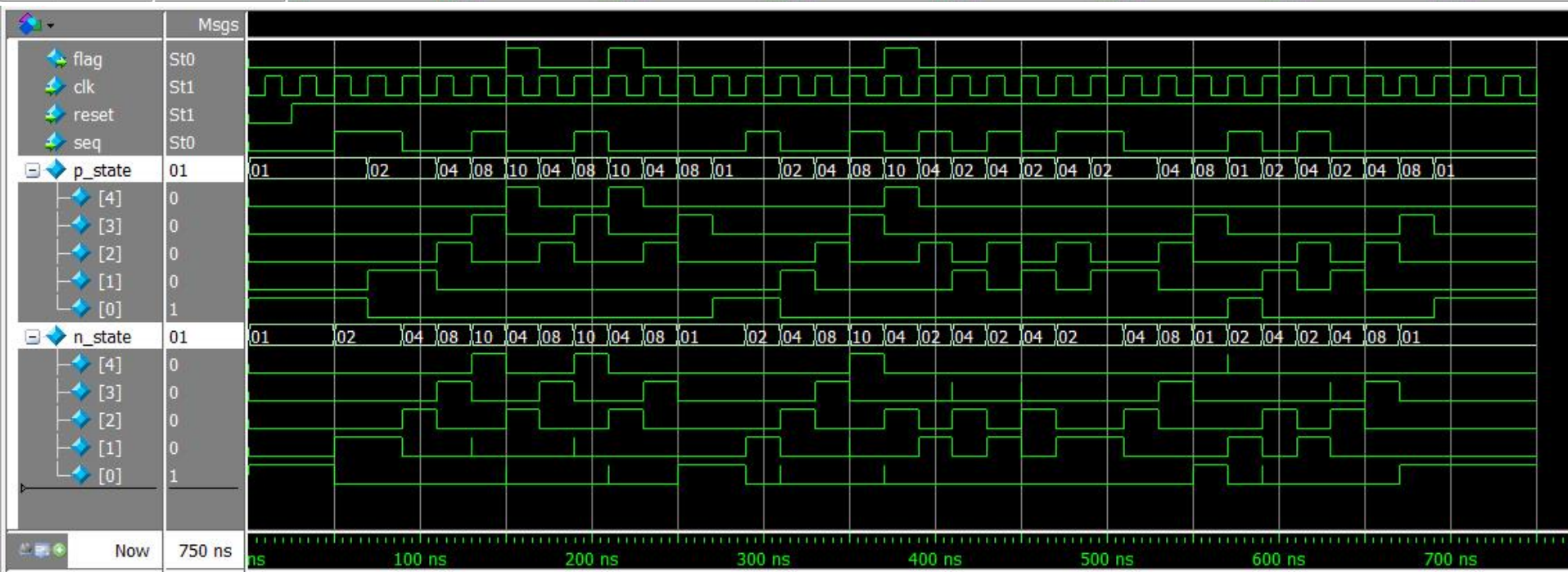
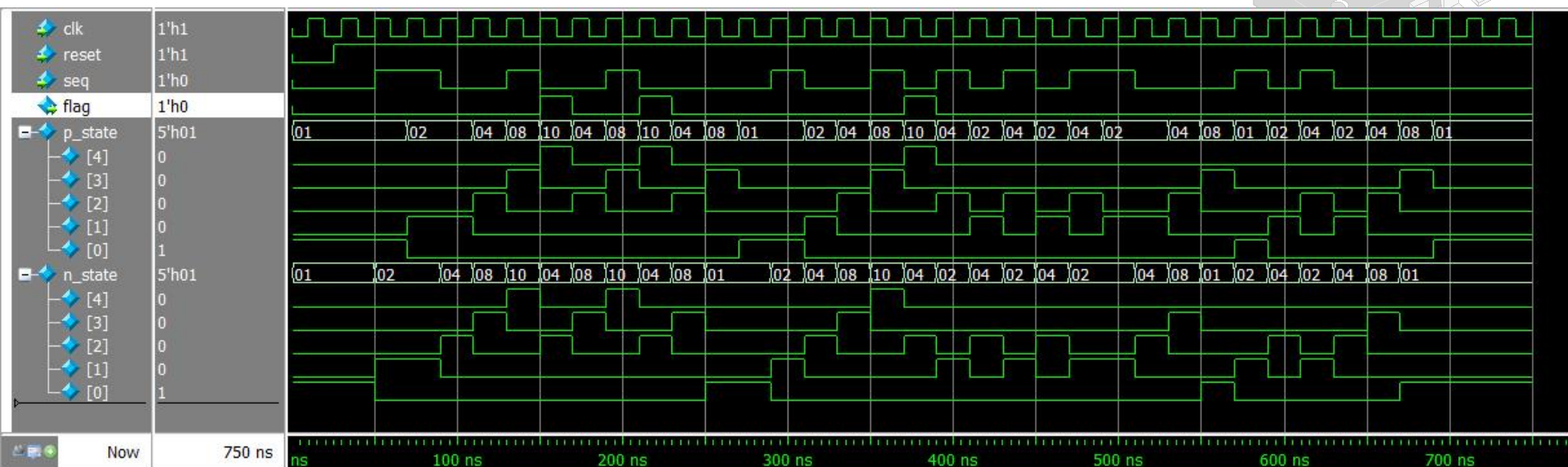
```
module seq_recognize ( output flag, input clk, reset, seq);  
    parameter IDLE = 5'b0_0001, A = 5'b0_0010, B = 5'b0_0100,  
              C = 5'b0_1000, D = 5'b1_0000;  
    reg [4 : 0] p_state, n_state;  
  
    always @(posedge clk, negedge reset)  
        if (!reset) p_state <= IDLE;  
        else p_state <= n_state;  
  
    always @(*) begin  
        case (p_state)  
            IDLE:    n_state = (seq) ? A : IDLE;  
            A:      n_state = (seq) ? A : B;  
            B:      n_state = (seq) ? A : C;  
            C:      n_state = (seq) ? D : IDLE;  
            D:      n_state = (seq) ? A : B;  
            default: n_state = IDLE;  
        endcase  
    end  
    assign flag = ((p_state == D) && (seq == 1'b0)) ? 1'b1 : 1'b0;  
endmodule
```



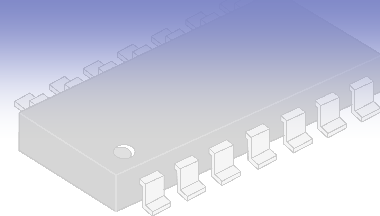




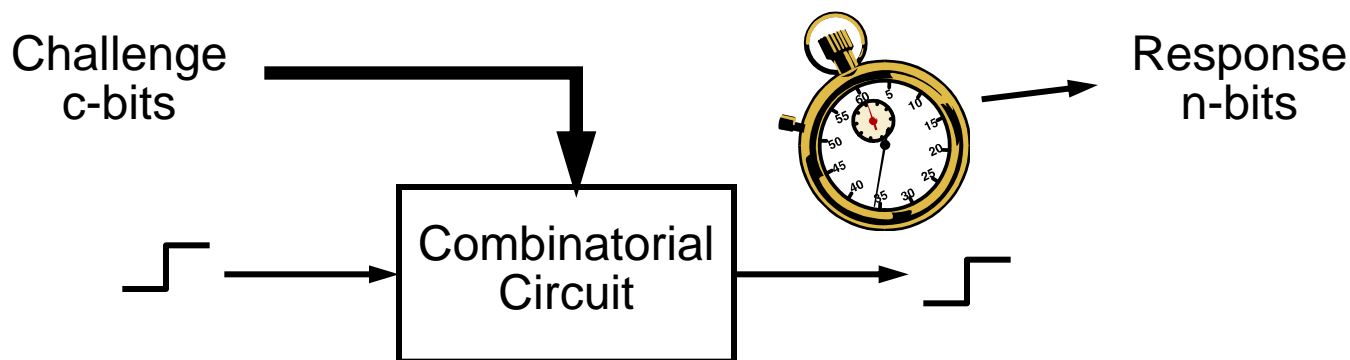
# ModelSim 32bit（上）与 64bit（下） 仿真结果



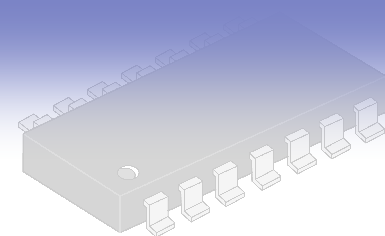
# 物理不可克隆函数



- ❑ 物理不可克隆函数（Physical Unclonable Function, PUF）
  - ❑ 用于从复杂的物理系统特性中提取秘密信息
- ❑ 由于生产工艺的随机偏差，没有任何两个集成电路相同
  - ❑ 即使它们使用相同版图（layout）之中
  - ❑ 制造过程中产生的偏差是固有
  - ❑ 难以消除或预测
  - ❑ 随着制造工艺的越来越复杂，其相应的偏差不断增加
- ❑ 基于延时的硅基 PUF 的概念
  - ❑ 从每个处理器芯片的唯一延迟特性产生密钥



# 什么是 PUFs?



## □ PUF 能用于生成唯一的密钥或 ID

- 高度安全，无需专门的设计进行处理
- 成本低廉，无需特殊制造技术

## □ PUF无需利用加密就可实现安全，低成本的认证

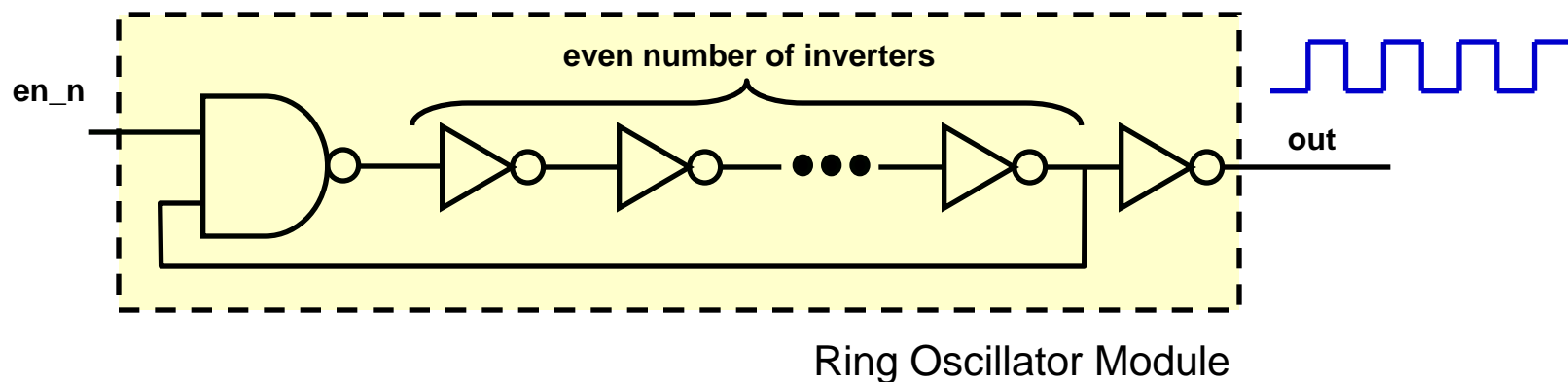
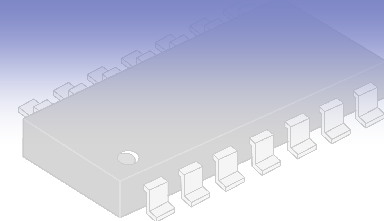
- PUF 作为一个函数实现映射： challenge  $\Rightarrow$  response
- 只有真实的IC才能生成对应于挑战（输入数据）的正确响应（输出数据）

## □ 问题

- 如何设计一个 PUF 电路？
- PUF想法是否可行（可靠性和安全性）？
- 如何使用PUF进行密钥生成和认证？

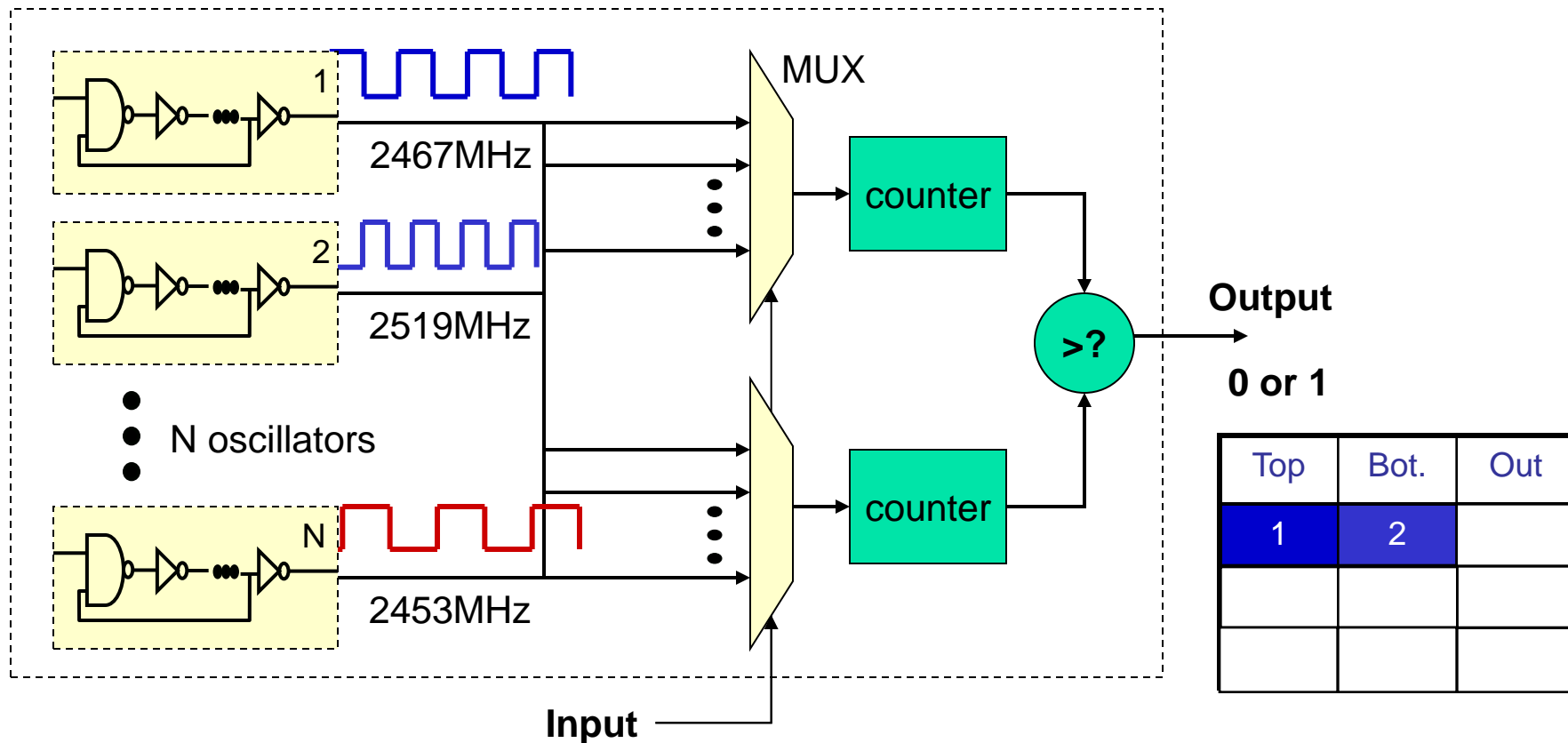
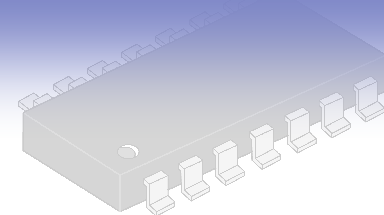


# 环形振荡器（Ring Oscillator）



- ❑ 在IC中广泛使用环形振荡器产生时钟或特征性能
- ❑ 即使采用相同的掩模制造的多个振荡器，每个环形振荡器都具有唯一的频率

# 使用环形振荡器的PUF 电路



比较两个振荡器的频率 → 哪个振荡器的频率更快是由制造工艺的偏差确定随机的