# 计算机科学基础 C

# Verilog 设计技巧与实践

# Verilog 设计技巧与实践

- ❋ 阻塞赋值与非阻塞赋值
- ❋ 同步设计及异域信号的处理
- ❋ 再谈有限状态机
- ❋ 块思维
- ❋ 代码规范

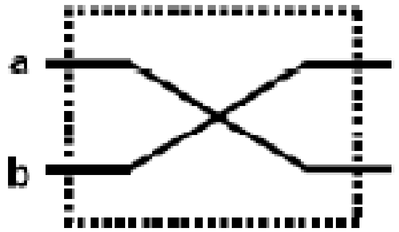
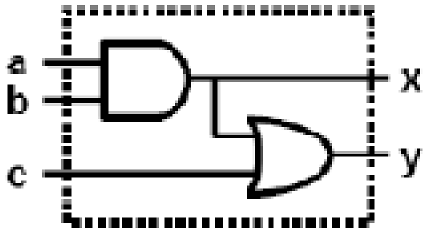
## 阻塞赋值与非阻塞赋值

## 阻塞赋值与非阻塞赋值

- 什么叫阻塞赋值？
  - 见到阻塞赋值立刻赋值，赋值完成后，才可执行下一条语句。
  - 用“＝”表示

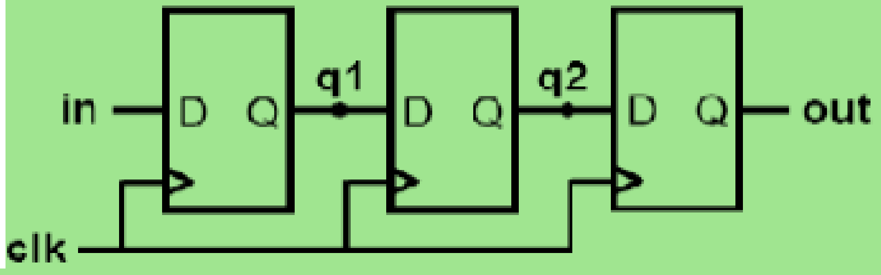
- 什么叫非阻塞赋值？
  - 等待过程块中所有非阻塞赋值统一节拍，在过程块最后时刻赋值。
  - 用“<=”表示

## 如何区分

| | | |
|---|---|---|
| |  |  |
| 阻塞赋值：<br>电路输出实时根据输入变化而变化 | ~~a = b~~<br>~~b = a~~ | x = a & b<br>y = x \| c |
| 非阻塞赋值：<br>赋值先被挂起，等待其他赋值完成后，一同赋值 | a <= b<br>b <= a | ~~x <= a & b~~<br>~~y <= x \| c~~ |
| 何时使用 | 时序电路 | 组合电路 |

## 时序电路

基于触发器的
数字延迟链



使用阻塞赋值和非阻塞赋值实现，电路综合结果？

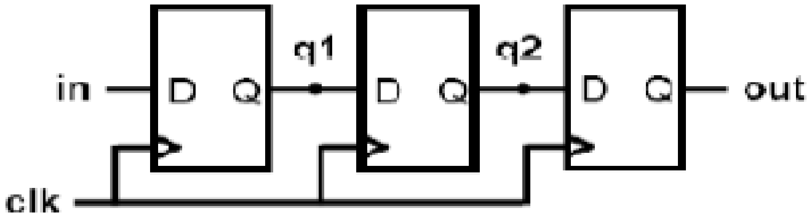
```
module nonblocking(in, clk, out);
  input in, clk;
  output out;
  reg q1, q2, out;
  always @ (posedge clk)
  begin
    q1 <= in;
    q2 <= q1;
    out <= q2;
  end
endmodule
```

```
module blocking(in, clk, out);
  input in, clk;
  output out;
  reg q1, q2, out;
  always @ (posedge clk)
  begin
    q1 = in;
    q2 = q1;
    out = q2;
  end
endmodule
```

# 时序电路

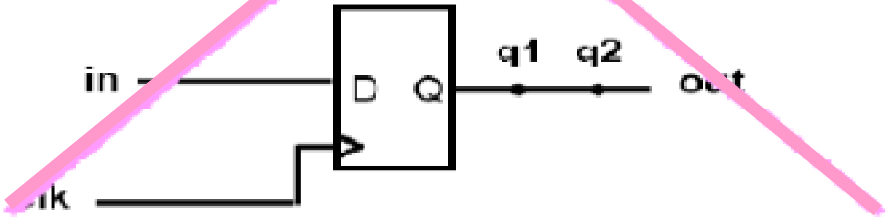
```
always @ (posedge clk)
begin
   q1 <= in;
   q2 <= q1;
   out <= q2;
end
```

在每个时钟上升沿q1,q2,out
分别接收到前一级in,q1,q2的值



~~always @ (posedge clk)~~
~~begin~~
~~q1 = in;~~
~~q2 = q1;~~
~~out = q2;~~
~~end~~

~~在每个时钟上升沿q1=in,q2=q1=in,~~
~~out=q2=q1=in，中间电路别优化掉~~

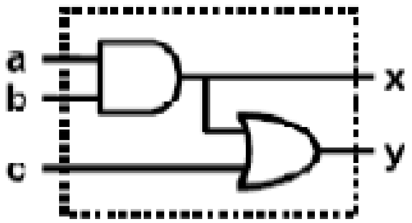


阻塞赋值在时序链上没能起到延时传输的目的，所以说
在时序电路中，永远使用非阻塞赋值 <=

## 组合电路

| *Blocking Behavior* | a b c x y |
|---|---|
| (Given) Initial Condition | 1 1 0 1 1 |
| a changes; always block triggered | 0 1 0 1 1 |
| x = a & b; | 0 1 0 0 1 |
| y = x \| c; | 0 1 0 0 0 |

```
always @ (a or b or c)
begin
   x = a & b;
   y = x | c;
end
```



| *Nonblocking Behavior* | a b c x y | *Deferred* |
|---|---|---|
| (Given) Initial Condition | 1 1 0 1 1 | |
| a changes; always block triggered | 0 1 0 1 1 | |
| x <= a & b; | 0 1 0 1 1 | x<=0 |
| y <= x \| c; | 0 1 0 1 1 | x<=0, y<=1 |
| Assignment completion | 0 1 0 0 1 | |

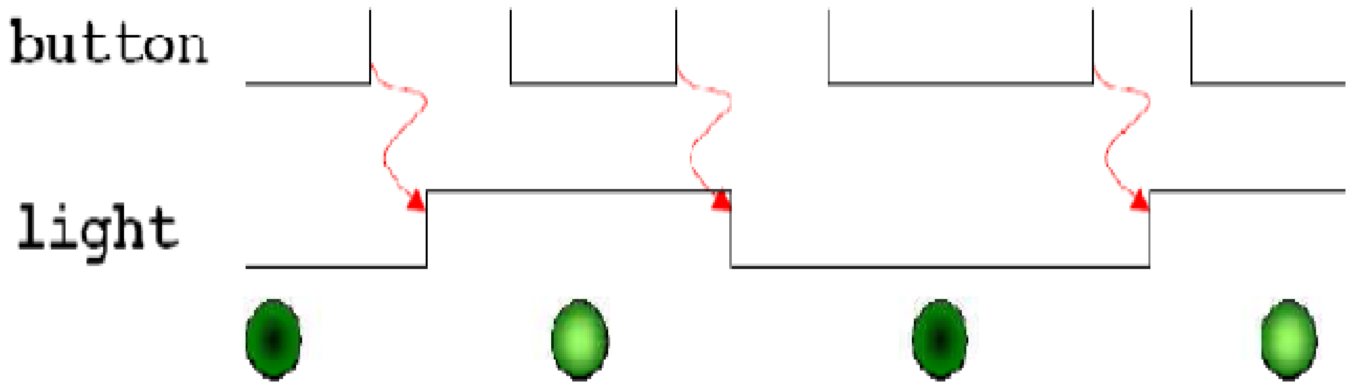
```
always @ (a or b or c)
begin
   x <= a & b;
   y <= x | c;
end
```

非阻塞赋值在组合电路中不可实时影响电路输出值，所以在组合电路中，永远使用阻塞赋值 =

## 阻塞赋值与非阻塞赋值

- ＊ 时序电路一律使用非阻塞赋值 <=
- ＊ 组合电路一律使用阻塞 =

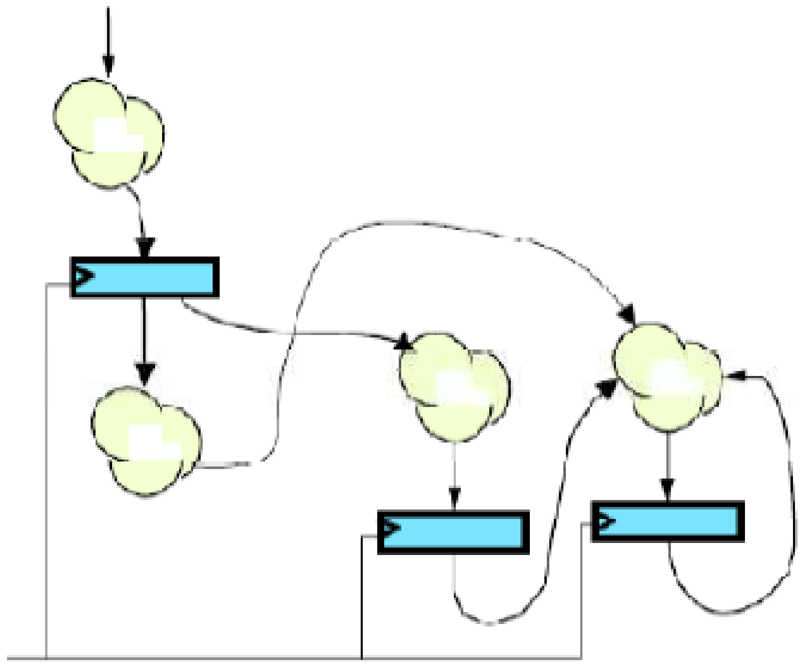
## 实例：LED灯开关



如何实现？

## 同步设计及异域信号处理

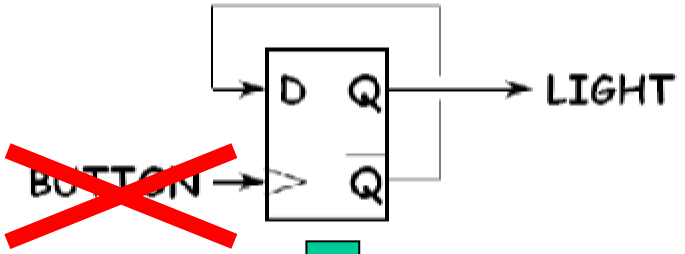
# 同步设计

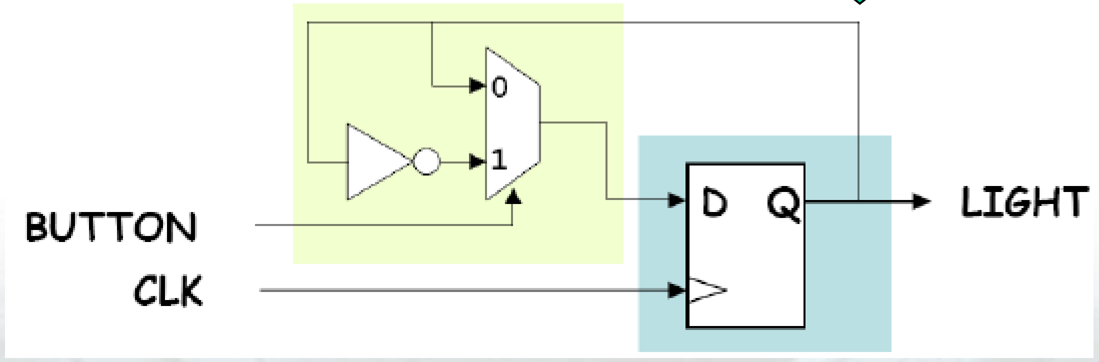

所有电路 ＝ 组合电路模块 ＋时序电路结点

- 我们要求整个电路共用一个时钟；
- 要求时钟应该直接接入时钟输入端；
- 对于组合电路模块，只关心触发沿到来前一时刻，组合电路的输出值；
- 时钟周期应大于每个组合电路块的延时；

# 同步电路

```
module onoff(button,light);
   input button;
   output light;
   reg light;
   always @ (posedge button)
   begin
      light <= ~light;
   end
endmodule
```

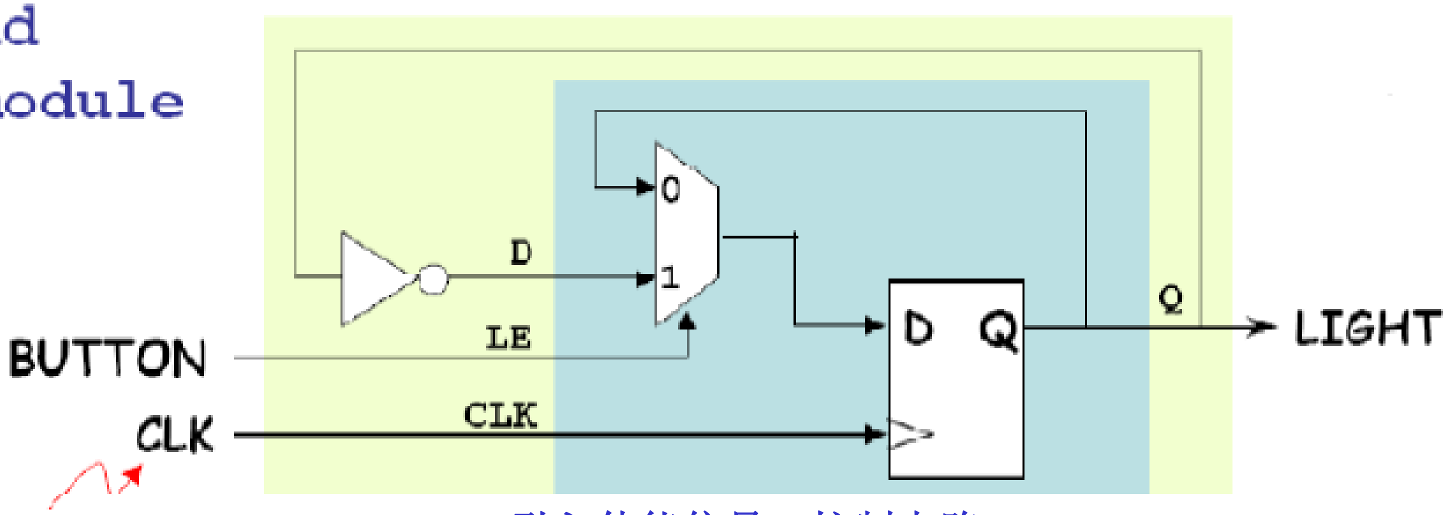




# 同步电路

```
module onoff(clk,button,light);
   input clk,button;
   output light;
   reg light;
   always @ (posedge
   begin
     if (button) ligh
   end
endmodule
```

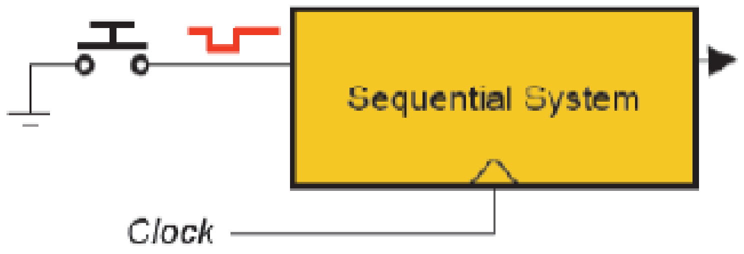
1、如何保证Button单脉冲？
2、如何保证单脉冲Button在触发沿上稳定，在触发的沿激励下同步跳变？



全局单时钟

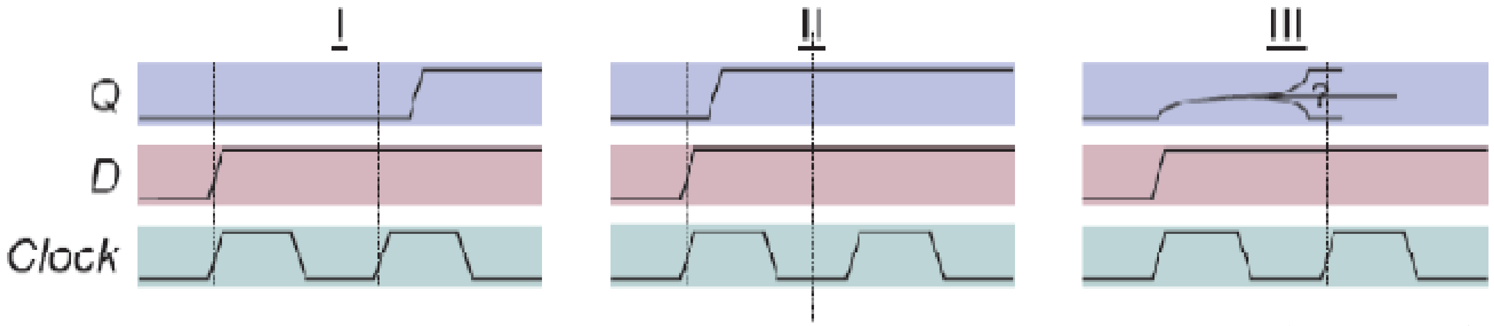
引入使能信号，控制电路

# 异域信号进入的三种情况



如何保证这个异域信号满足建立时间和保持时间？

异域信号进入同步电路



第一个clk沿没抓到信号变化，第二个clk沿抓到信号变化。

第一个clk沿就抓到信号变化了。

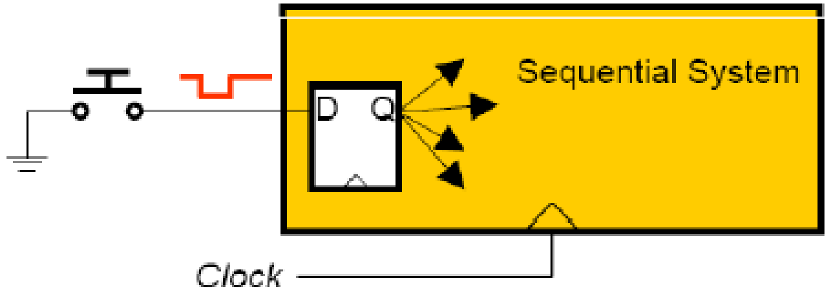
抓到信号变化后，输出不稳定，即亚稳态。

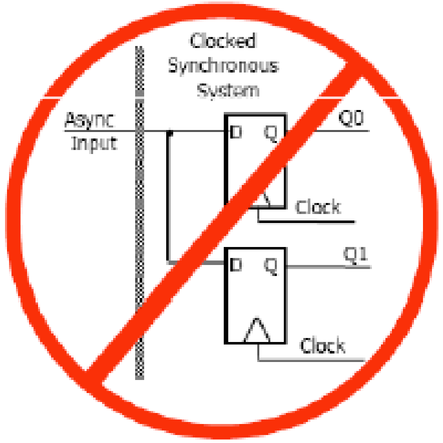
哪个会对电路产生影像？

# 异域信号的处理

在一个电路中，同时发生以上两种情况，则电路将出现错误！

异域信号进门时，先用同步时钟打一拍，则可防止I、II的发生。





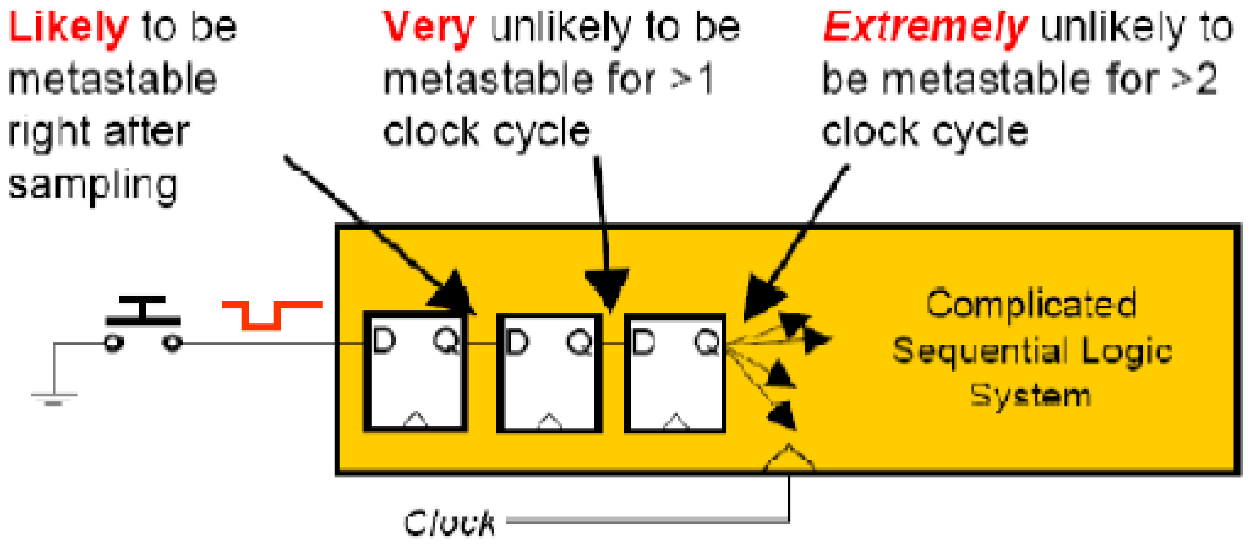
针对第三种情况，亚稳态信号如何处理？

## 如何消除亚稳态

要想消除电路的亚稳态，是不可能的。

使用高增益的器件，可以很快恢复信号的稳定输出。

解决方法：用时间成本逐级减小压稳态产生概率。



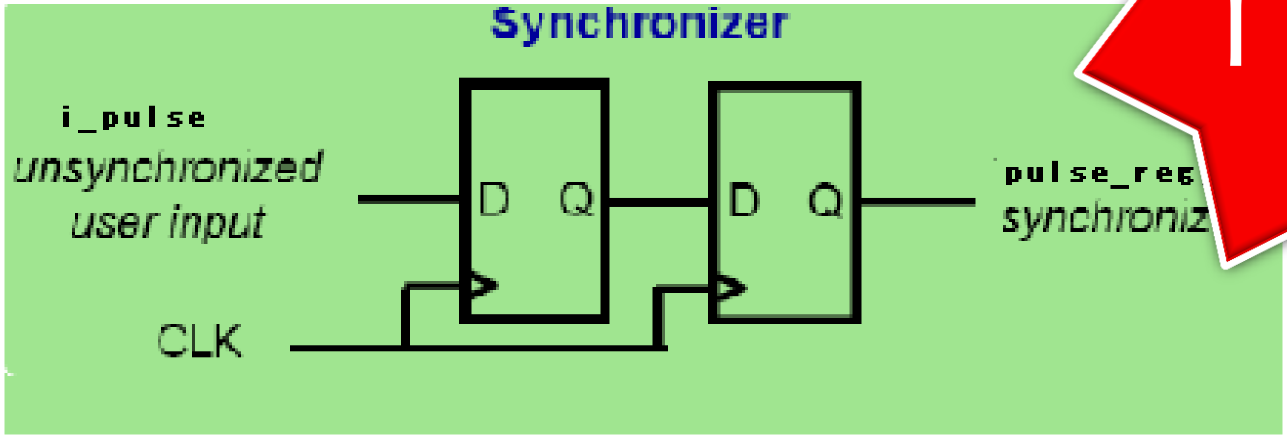
Likely to be metastable right after sampling
Very unlikely to be metastable for >1 clock cycle
Extremely unlikely to be metastable for >2 clock cycle
D Q
D Q
D Q
Complicated Sequential Logic System
Clock

入口处加几级同步触发器合适？

## 如何消除亚稳态

异步信号入门，用同步时钟打两拍。

门 规



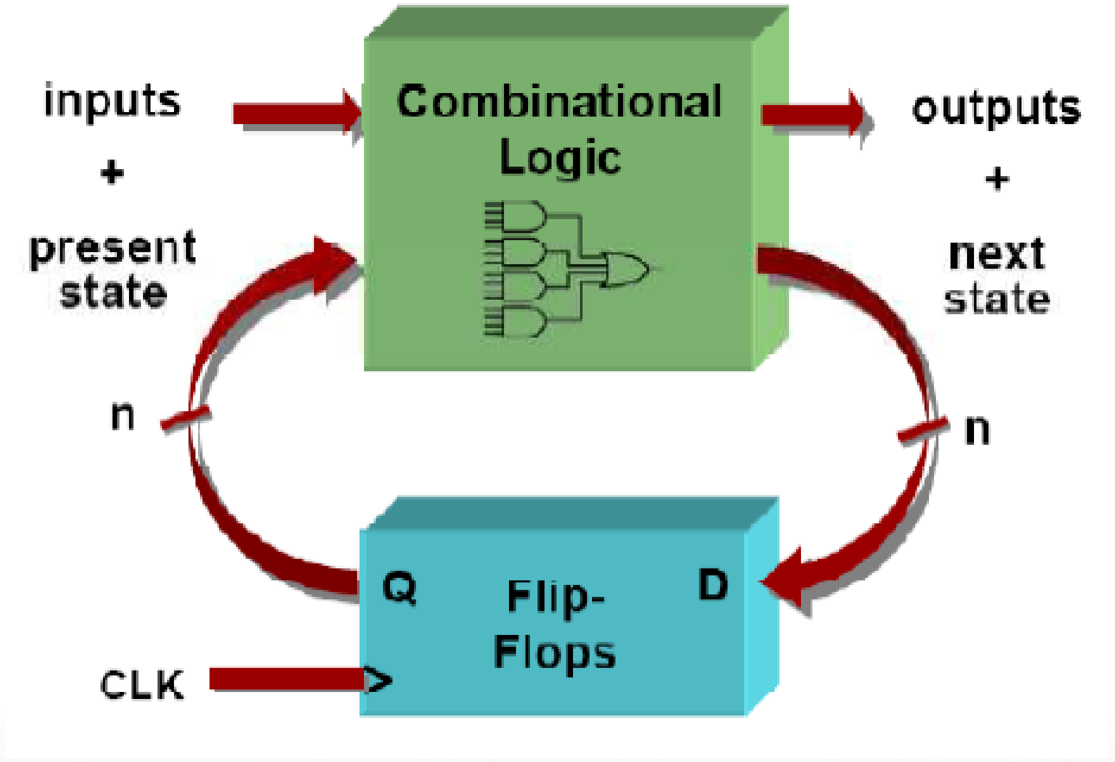
pulse_reg[0] <= **i_pulse**;

pulse_reg[1] <= pulse_reg[0];

## 再谈有限状态机

## 再谈有限状态机

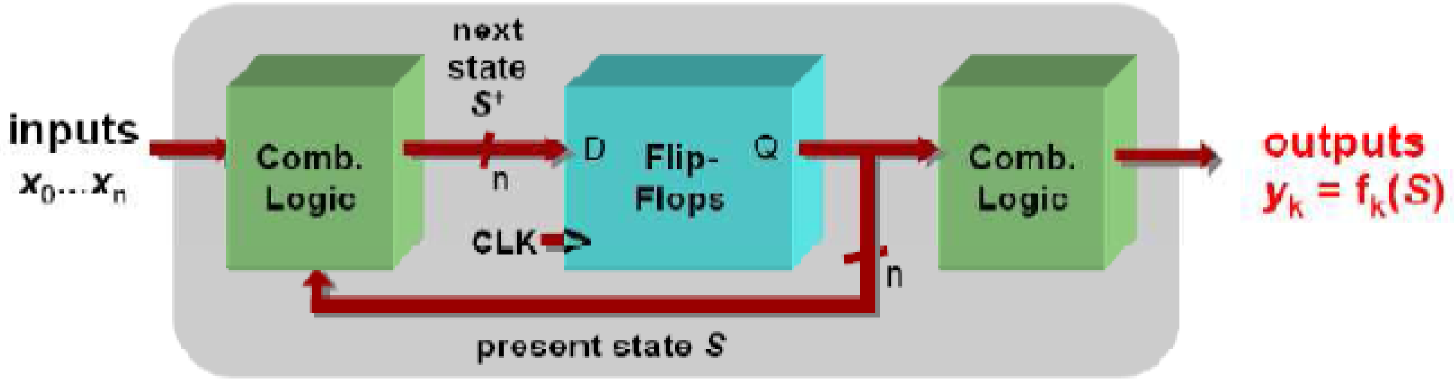
什么是有限状态机？



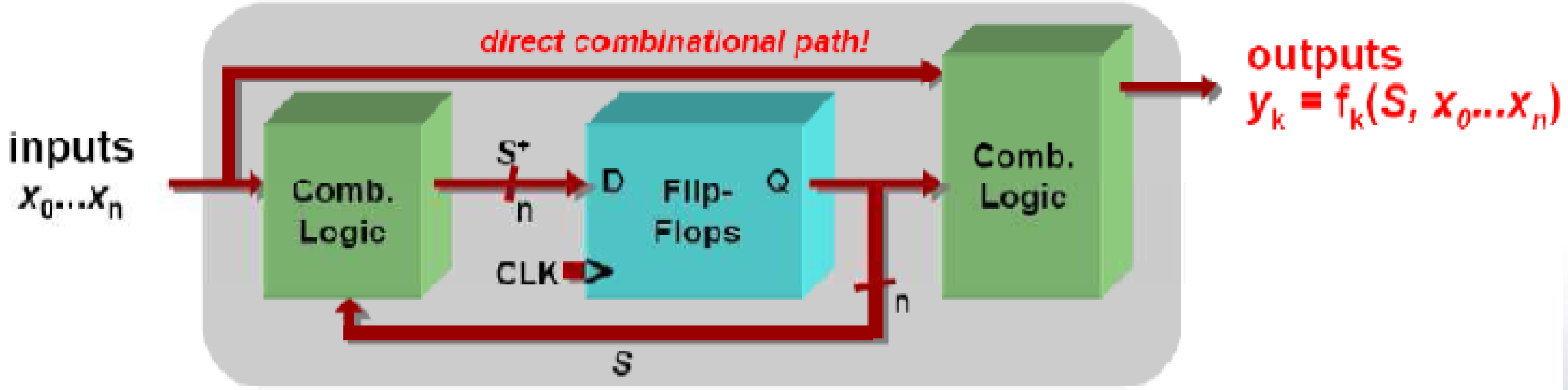
# 再谈有限状态机

两种有限状态机：

- Moore FSM:

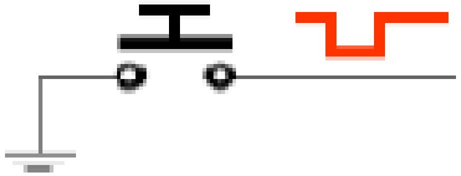


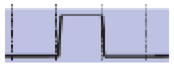
- Mealy FSM:



# 再谈有限状态机

点灯问题





## 再谈有限状态机

**Level To Pulse Converter:**

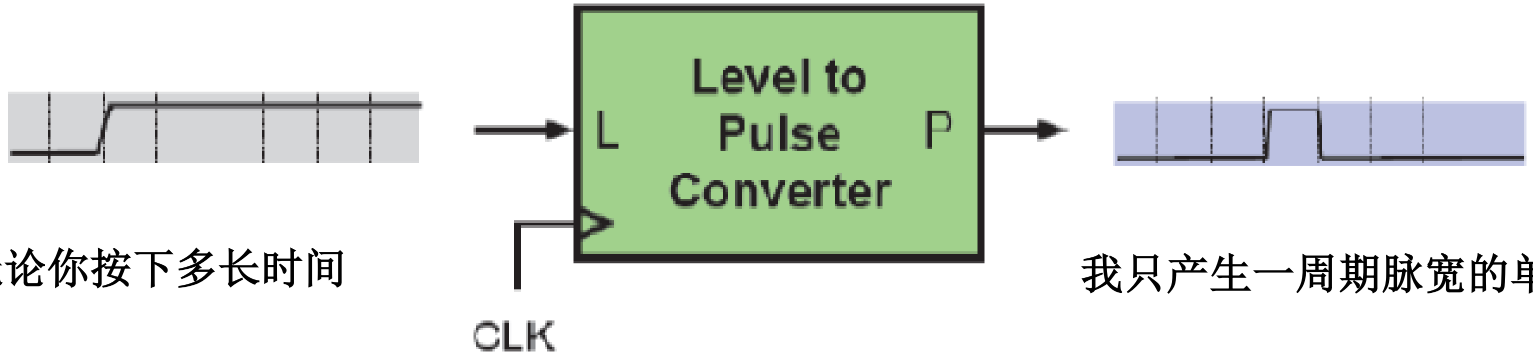
当按键按下，产生一个周期高脉冲。

应用：

A、按键开关

B、counter的使能信号

……

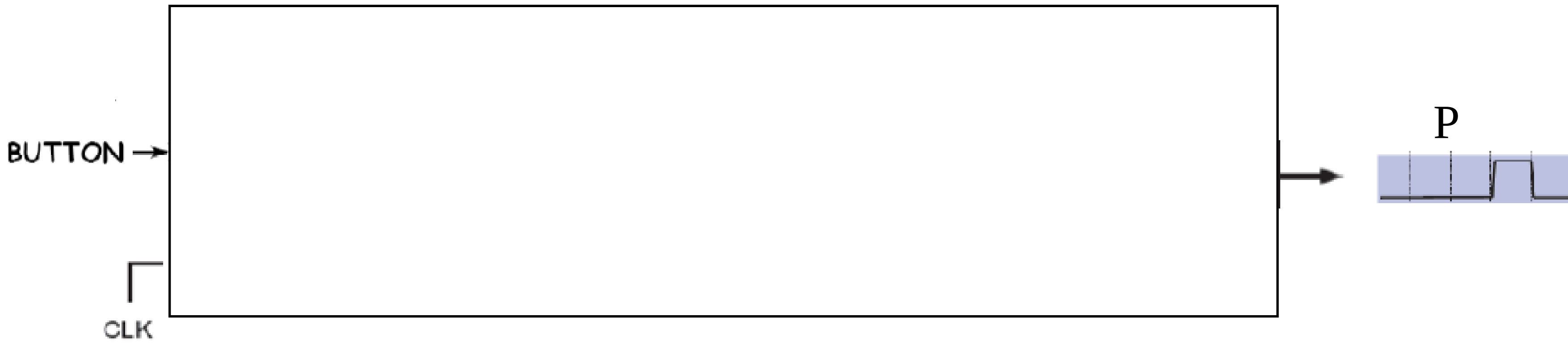


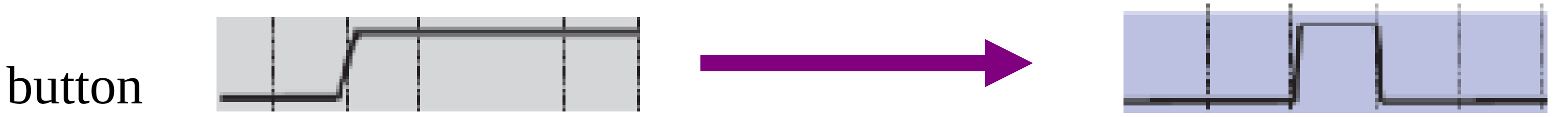
无论你按下多长时间

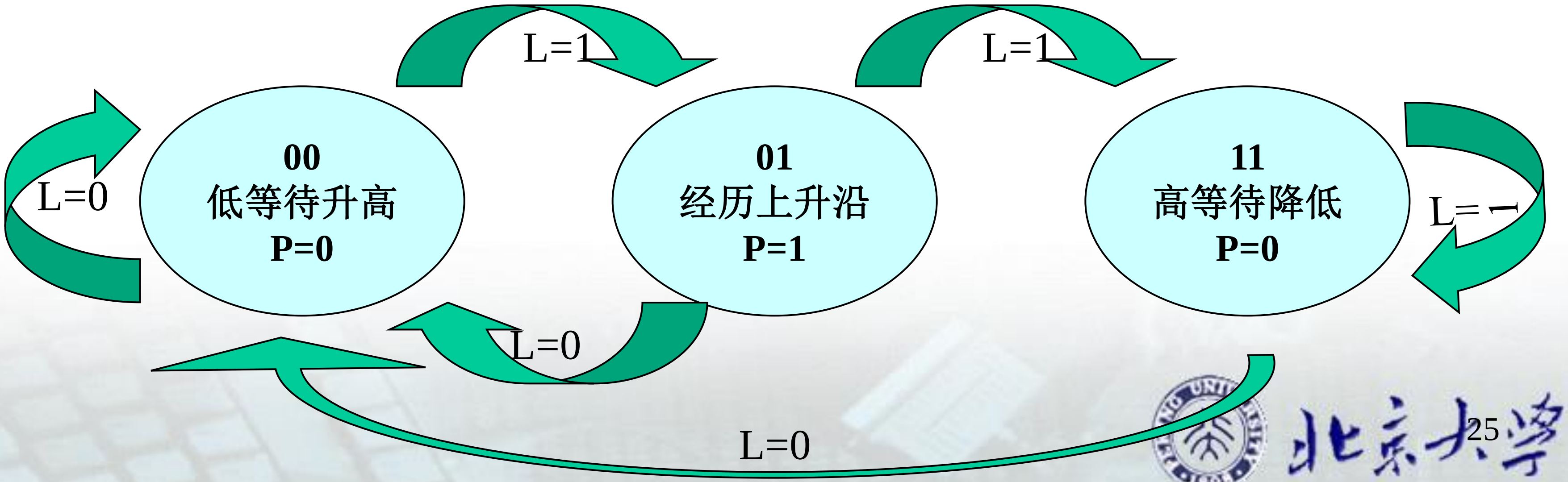
我只产生一周期脉宽的单脉冲

# 再谈有限状态机

电路设计：



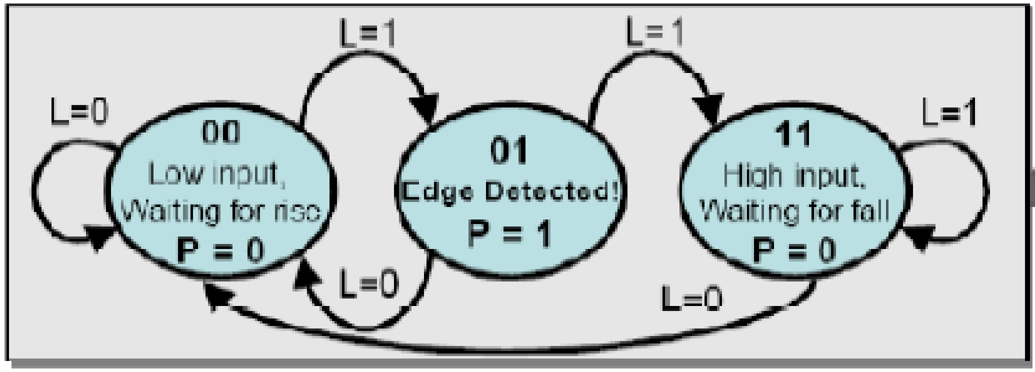




# 再谈有限状态机

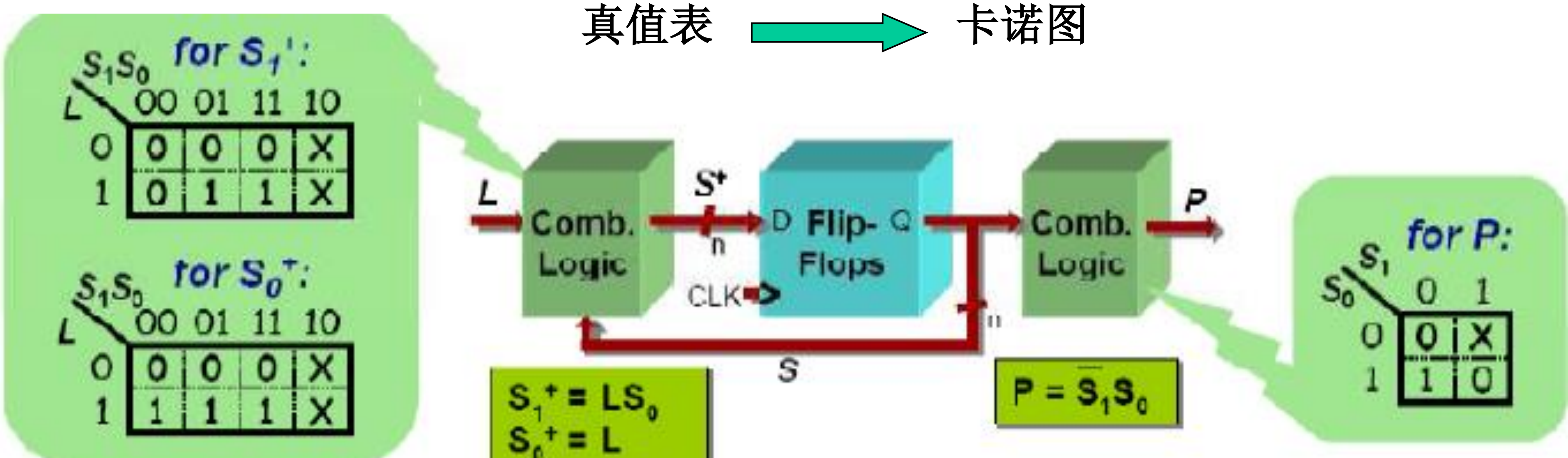
电路实现：：

状态转移图 $\longrightarrow$ 真值表



| Current State | | In | Next State | | Out |
|---|---|---|---|---|---|
| $S_1$ | $S_0$ | $L$ | $S_1^+$ | $S_0^+$ | $P$ |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 |

真值表 $\longrightarrow$ 卡诺图

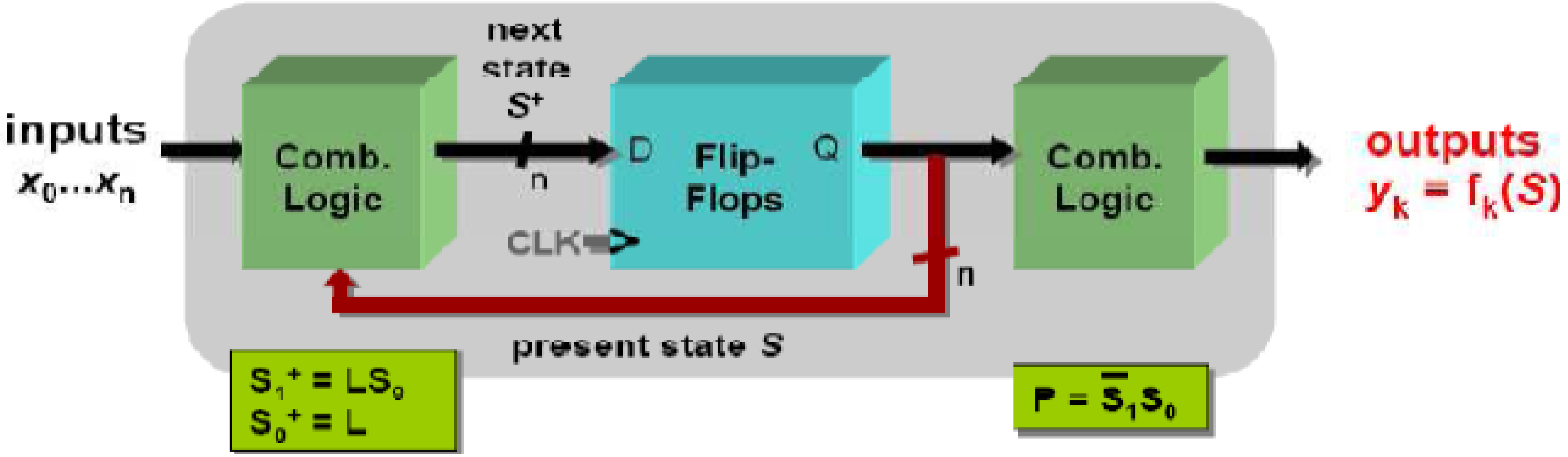


$S_1^+ = LS_0$
$S_0^+ = L$

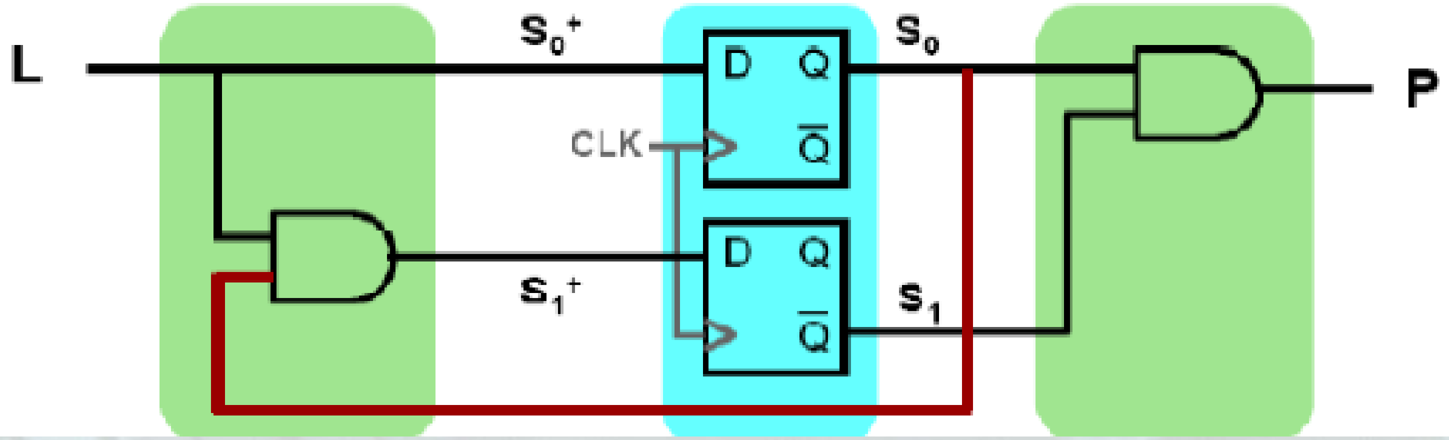
$P = \overline{S_1}S_0$

## 再谈有限状态机

电路实现：



用moore状态机实现level to pulse电路：

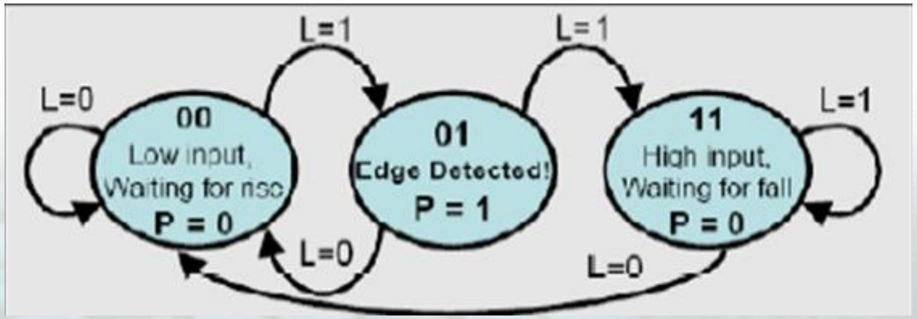


# 再谈有限状态机

代码实现：

```
module pulse_FSM (Clock, Resetn, i_pulse, o_pulse);
    input Clock, Resetn, i_pulse;
    output reg o_pulse;
    reg [1:0] cs, ns;   // present and next state variable
    reg [1:0] pulse_reg;
    parameter S0 = 0, S1 = 1, S2 = 2;
```

```
  always @(posedge Clock)
  begin
      pulse_reg[1] <= pulse_reg[0];
      pulse_reg[0] <= i_pulse;
      if (Resetn == 1'b0) // synchronous clear
          cs <= S0;
      else
          cs <= ns;
  end
```



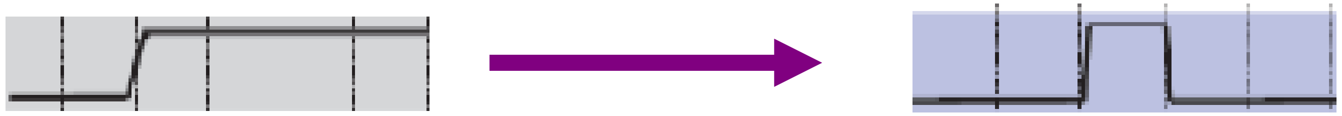
```
    always @(cs, pulse_reg[1])
    begin: state_table
        case (cs)
            S0:     if (pulse_reg[1]) begin
                        ns = S1;
                        o_pulse = 1'b1 end
                    else begin
                        ns = S0;
                        o_pulse = 1'b0; end
            S1:      if (pulse_reg[1]) begin
                        ns = S2;
                        o_pulse = 1'b0 end
                    else begin
                        ns = S0;
                        o_pulse = 1'b0;
            S2:     iif (pulse_reg[1]) begin
                        ns = S2;
                        o_pulse = 1'b0 end
                    else begin
                        ns = S0;
                        o_pulse = 1'b0; end
            default:    ns = 2'bxx;
        endcase
    end // state_table
endmodule
```

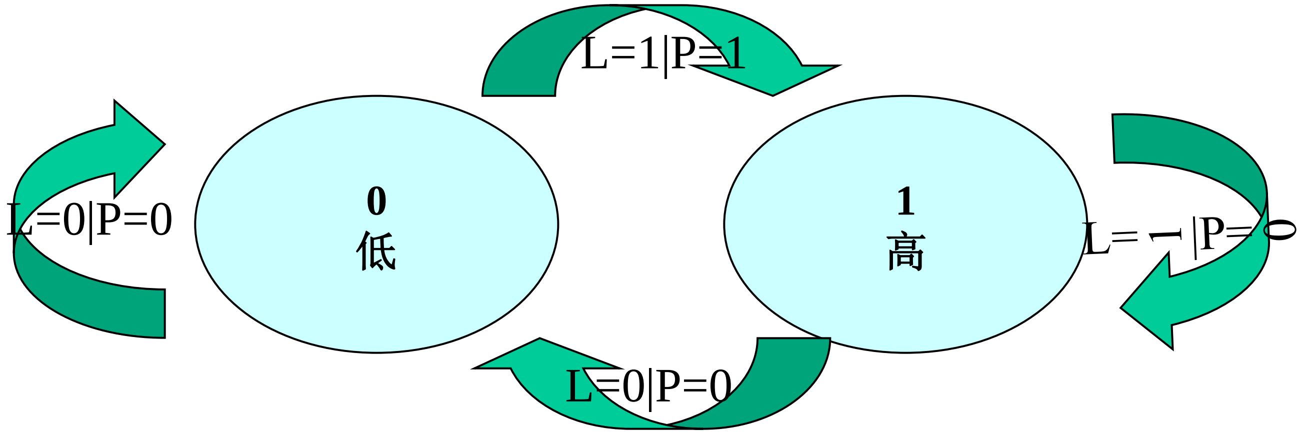
# 再谈有限状态机

电路实现：：

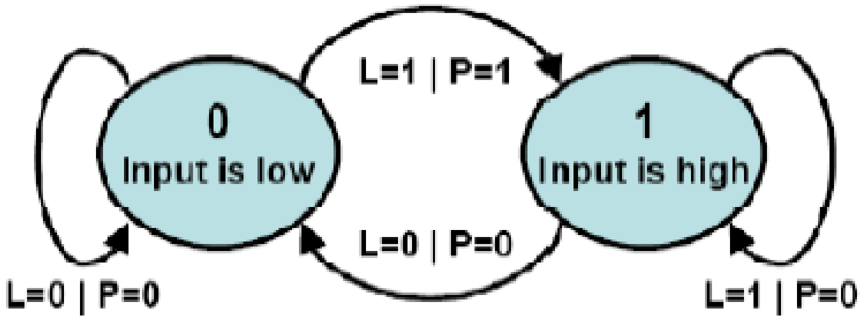
当前状态的输出由当前状态和输入共同决定的状态机成为mealy状态机

button

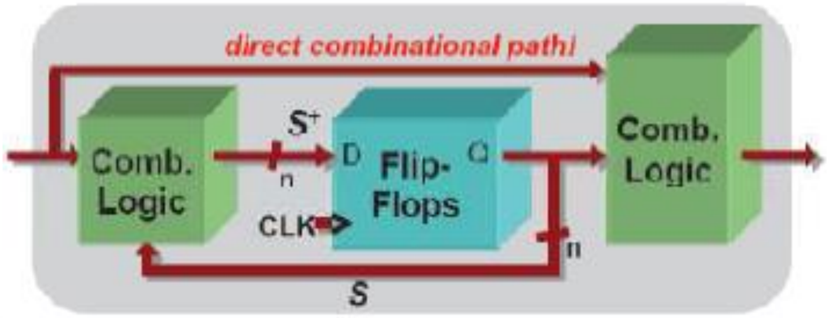


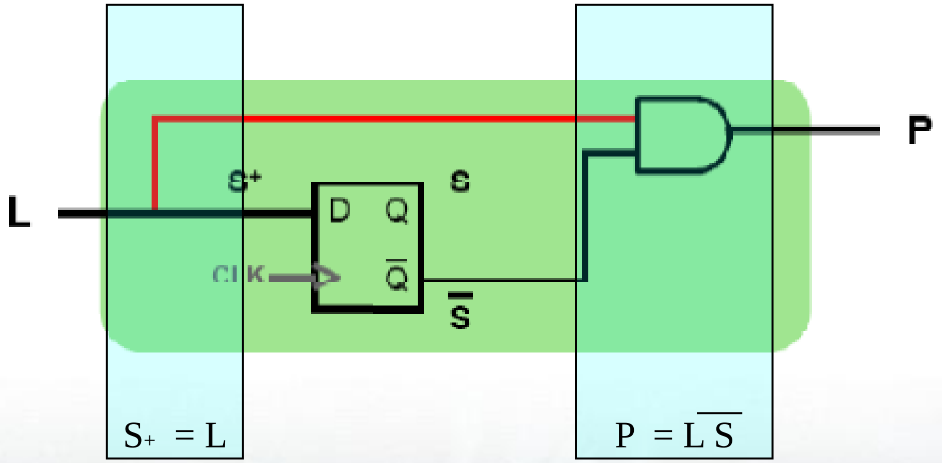


# 再谈有限状态机



| Pres. State | In | Next State | Out |
|---|---|---|---|
| $S$ | $L$ | $S^+$ | $P$ |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 |





$S_+ = L$

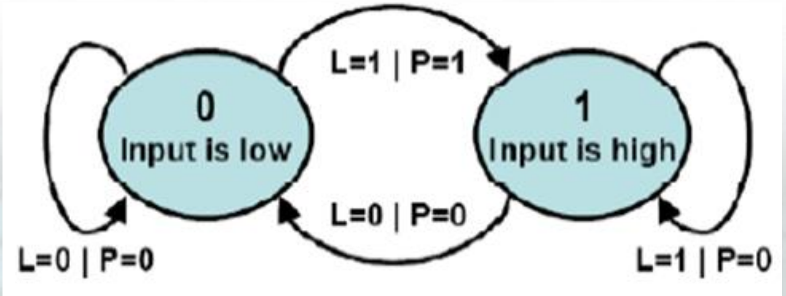
$P = L\overline{S}$

# 再谈有限状态机

代码实现：

```
module pulse_FSM (Clock, Resetn, i_pulse, o_pulse);
    input Clock, Resetn, i_pulse;
    output reg o_pulse;
    reg  cs, ns;     // present and next state variable
    reg [1:0] pulse_reg;
    parameter S0 = 0, S1 = 1;


    always @(posedge Clock)
    begin
        pulse_reg[1] <= pulse_reg[0];
        pulse_reg[0] <= i_pulse;
        if (Resetn == 1'b0) // synchronous clear
            cs <= S0;
        else
            cs <= ns;
    end
```

```
    always @(cs, pulse_reg[1])
    begin: state_table
        case (cs)
            S0:     if (pulse_reg[1]) begin
                        ns = S1;
                        o_pulse = 1'b1 end
                    else begin
                        ns = S0;
                        o_pulse = 1'b0; end/
            S1:      if (pulse_reg[1]) begin
                        ns = S1;
                        o_pulse = 1'b0 end
                    else begin
                        ns = S0;
                        o_pulse = 1'b0;
            default:    ns = 2'bxx;
        endcase
    end // state_table
endmodule
```

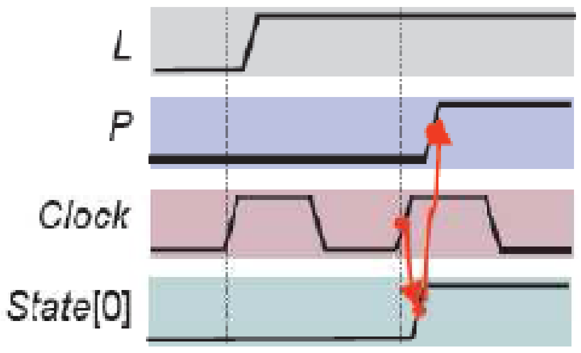


# 再谈有限状态机

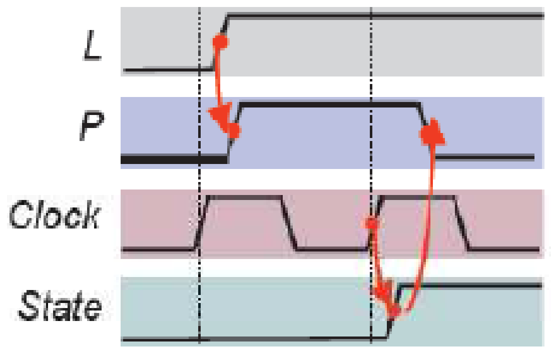
## ❋Moore Vs  Mealy

- ☞Moore: outputs = f（只当前状态）
- ☞Mealy： outputs = f（当前状态，当前输入）
- ☞Mealy状态机的输出比Moore快一个周期
- ☞Mealy状态机比Moore设计复杂，但可用较少的状态实现同样的功能。

**Moore:** delayed assertion of P



**Mealy:** immediate assertion of P

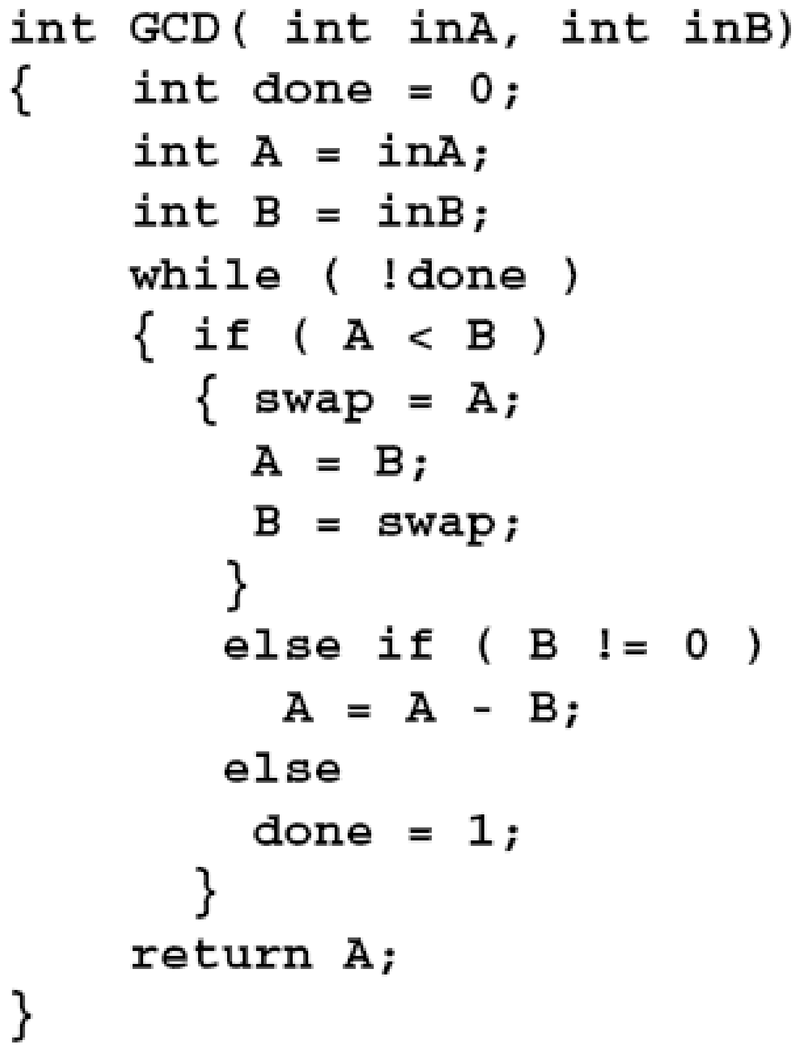


## 小 结

- **阻塞赋值与非阻塞赋值**
  - Always块中阻塞赋值用于组合逻辑；
  - Always块中非阻塞赋值用于时序逻辑；
- **同步设计及异域时钟的处理**
  - 电路使用同步时钟
  - 异域时钟进入先打两拍
- **有限状态机**
  - 多用其输出作为电路的控制信号

## 块　思　维

# 计算最大公约数



```
int GCD( int inA, int inB)
{   int done = 0;
    int A = inA;
    int B = inB;
    while ( !done )
    { if ( A < B )
      { swap = A;
        A = B;
        B = swap;
       }
       else if ( B != 0 )
         A = A - B;
       else
        done = 1;
      }
    return A;
}
```

GCD in C

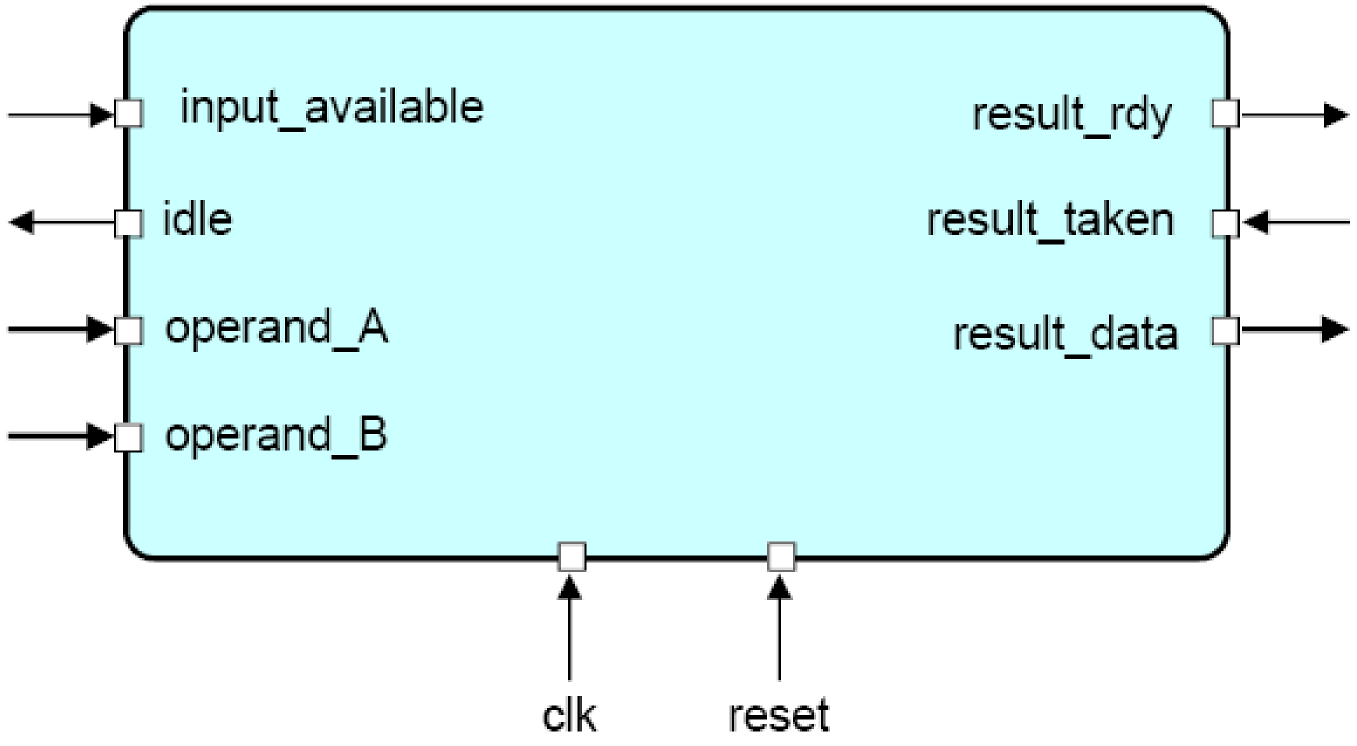
# 计算最大公约数

```
module gcdGCDUnit_behav#( parameter W = 16 )
( input   [W-1:0] inA, inB,
  output [W-1:0] out );
  reg [W-1:0] A, B, out, swap;
  integer done;
  always @(*)
  begin
    done = 0; A = inA; B = inB;
    while ( !done )
    begin
      if ( A < B )
        swap = A;
        A = B; B = swap;
      else if ( B != 0 )
        A = A - B;
      else
        done = 1;
    end
    out = A; end endmodule
```

用行为级verilog描述GCD

## 电路设计第一步

**0、画出方框图外接口：**



```
module gcdGCDUnit_behav#( parameter W = 16 )
( input  [W-1:0] inA, inB,
  output [W-1:0] out );
  reg [W-1:0] A, B, out, swap;
  integer done;
  always @(*)
  begin
    done = 0; A = inA; B = inB;
    while ( !done )
    begin
      if ( A < B )
        swap = A;
        A = B; B = swap;
      else if ( B != 0 )
        A = A - B;
      else
        done = 1;
    end
    out = A; end endmodule
```

What does the RTL implementation need?

State

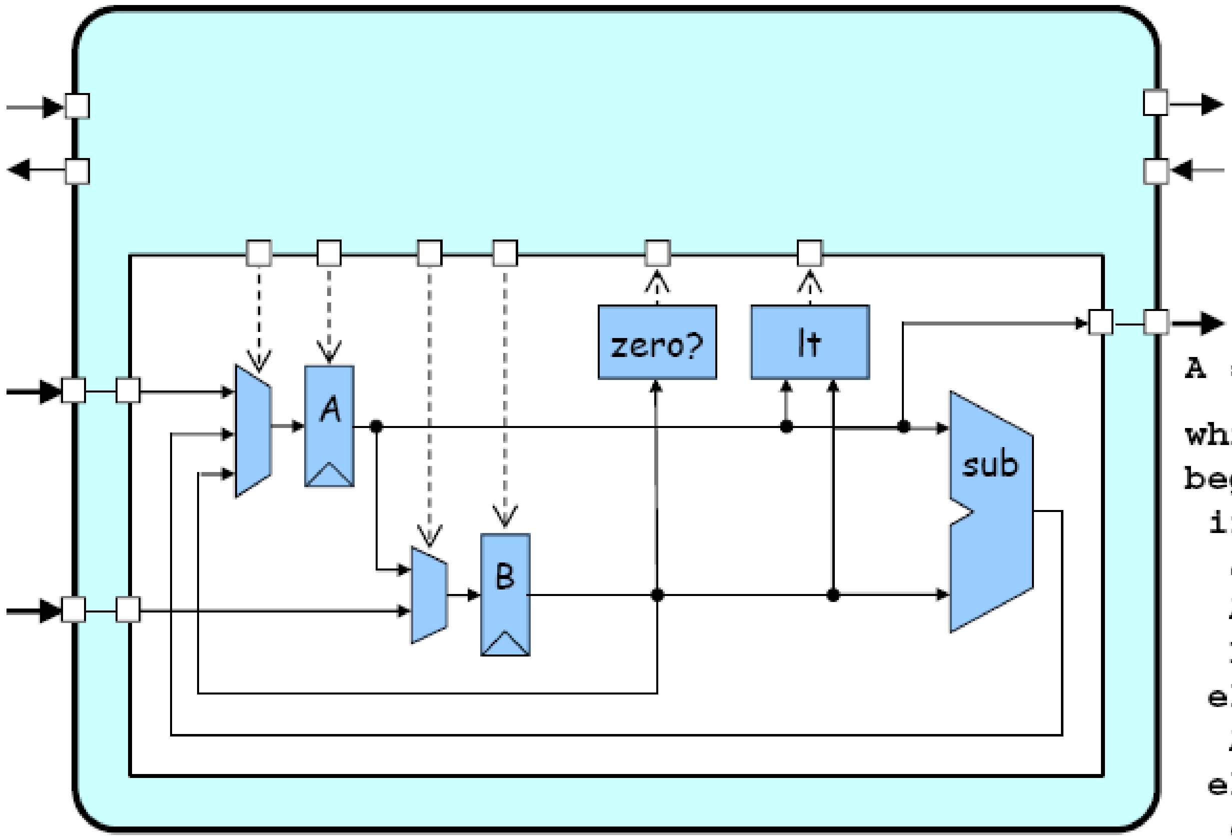
Less-Than Comparator

Equal Comparator

Subtractor

## 明晰功能，画出方框图内电路

1、画出方框图内功能电路：



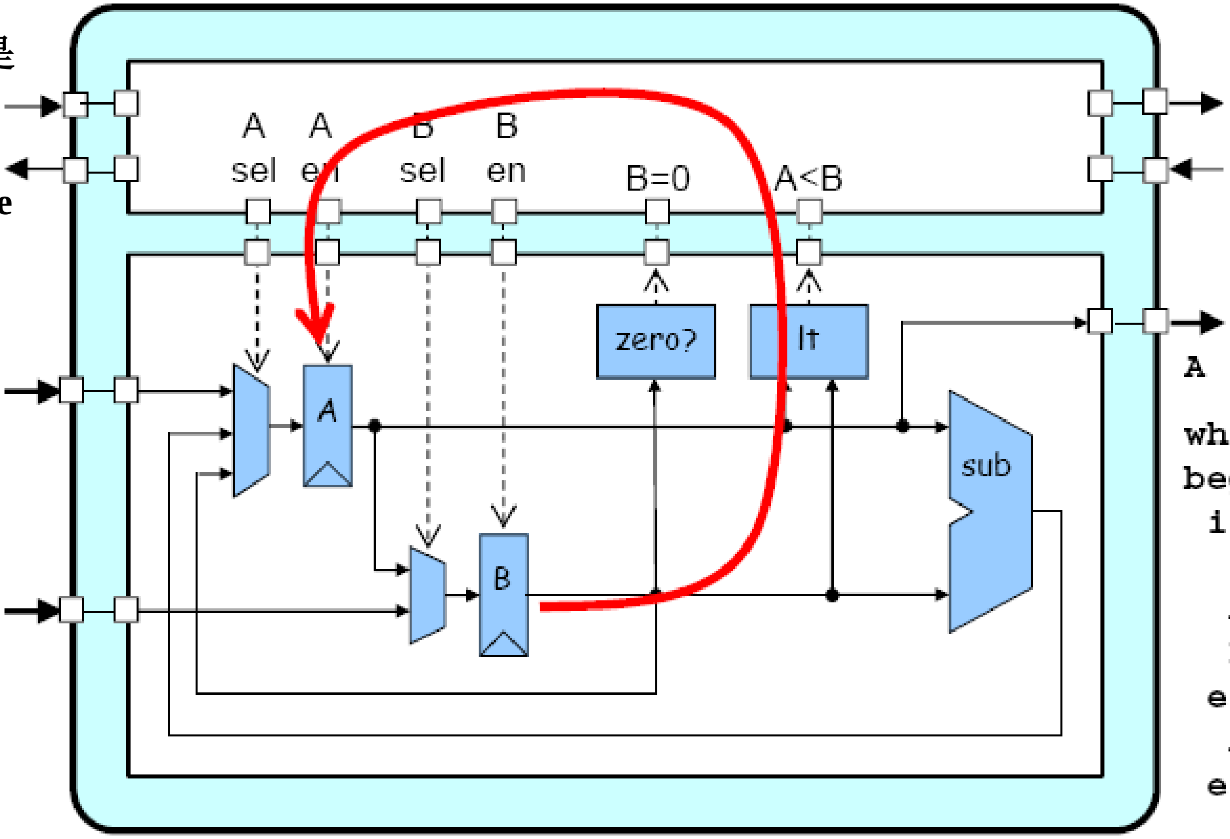
```
A = inA; B = inB;

while ( !done )
begin
 if ( A < B )
  swap = A;
  A = B;
  B = swap;
 else if (B != 0)
  A = A - B;
 else
  done = 1;
End
Y = A;
```

确定使用哪个模块，布局布线

# 明晰功能，画出方框图内电路

## 2、画出方框图内控制电路：

模块的工作是否完成，可接活儿了
Available/Idle

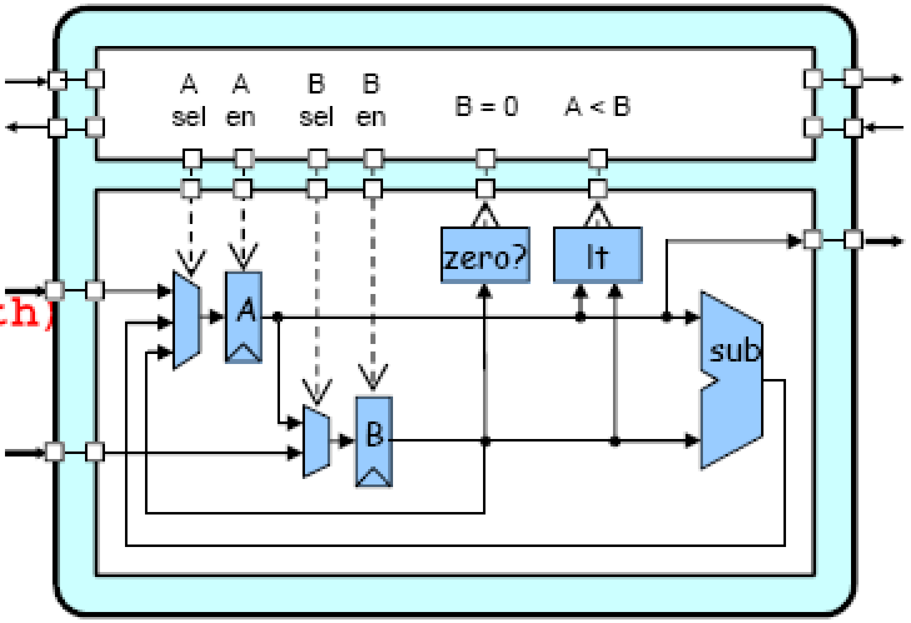


模块的工作是否完成，可交活儿了
Ready/taken

```
A = inA; B = inB;

while ( !done )
begin
 if ( A < B )
  swap = A;
  A = B;
  B = swap;
 else if (B != 0)
  A = A - B;
 else
  done = 1;
End
Y = A;
```

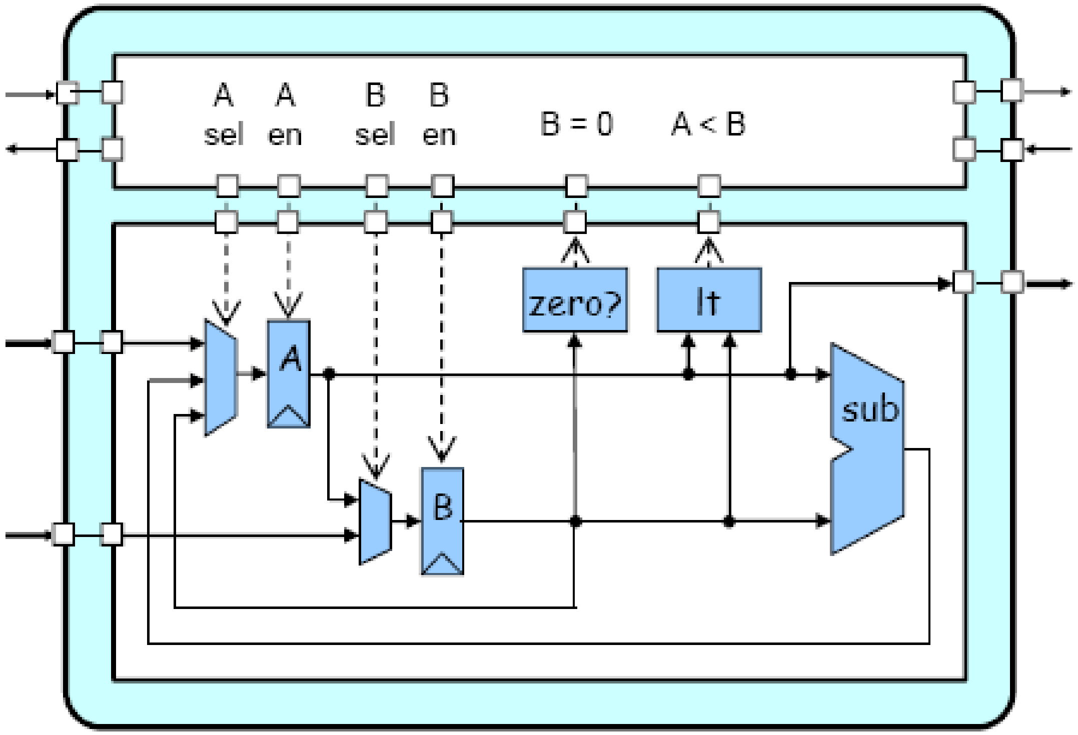
## 功能电路的实现

```
module gcdGCDUnitDpath_sstr#( parameter W = 16 )
( input        clk,

  // Data signals
  input  [W-1:0] operand_A,
  input  [W-1:0] operand_B,
  output [W-1:0] result_data,

  // Control signals (ctrl->dpath)
  input          A_en,
  input          B_en,
  input    [1:0] A_sel,
  input          B_sel,

  // Control signals (dpath->ctrl)
  output         B_zero,
  output         A_lt_B
);
```



## 功能电路的实现

```
wire  [W-1:0]  B;
wire  [W-1:0]  sub_out;
wire  [W-1:0]  A_out;

vcMux3#(W)  A_mux
(  .in0  (operand_A),
   .in1  (B),
   .in2  (sub_out),
   .sel  (A_sel),
   .out  (A_out)    );

wire  [W-1:0]  A;

vcEDFF_pf#(W)  A_pf
(  .clk   (clk),
   .en_p  (A_en),
   .d_p   (A_out),
   .q_np  (A)        );
```



# 功能电路的实现

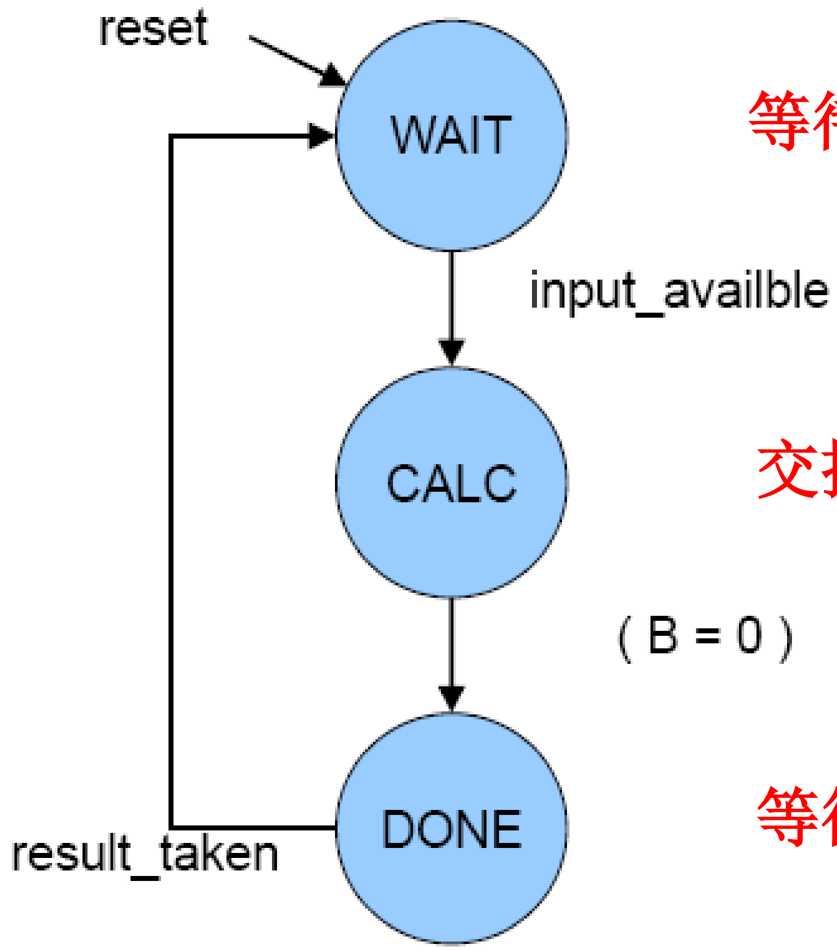
```
wire  [W-1:0]  B;
wire  [W-1:0]  sub_out;
wire  [W-1:0]  A_out;

vcMux3#(W)  A_mux
(  .in0  (operand_A),
   .in1  (B),
   .in2  (sub_out),
   .sel  (A_sel),
   .out  (A_out)       );

wire  [W-1:0]  A;
vcEDFF_pf#(W)  A_pf
(  .clk   (clk),
   .en_p  (A_en),
   .d_p   (A_out),
   .q_np  (A)          );
```

```
wire  [W-1:0]  B_out;

vcMux2#(W)  B_mux
(  .in0  (operand_B),
   .in1  (A),
   .sel  (B_sel),
   .out  (B_out)       );

vcEDFF_pf#(W)  B_pf
(  .clk   (clk),
   .en_p  (B_en),
   .d_p   (B_out),
   .q_np  (B)          );

assign B_zero   =  (B==0);
assign A_lt_B   =  (A < B);
assign sub_out = A - B;
assign result_data = A;
```

需要存储的重要数据，实例化FF，使数据在enable信号到来时更新FF输出

使用持续赋值语句，描述实时变化的组合电路

## 控制电路的实现



等待新的操作数

交换 相减 操作

等待结果被取走

## GCD的控制单元

```
localparam WAIT = 2'd0;
localparam CALC = 2'd1;
localparam DONE = 2'd2;

reg  [1:0] state_next;
wire [1:0] state;

vcRDFF_pf#(2,WAIT)
state_pf
( .clk     (clk),
  .reset_p (reset),
  .d_p     (state_next),
  .q_np    (state)   );
```

用Localparam
定义了三个状态

存储当前状态的FF实现

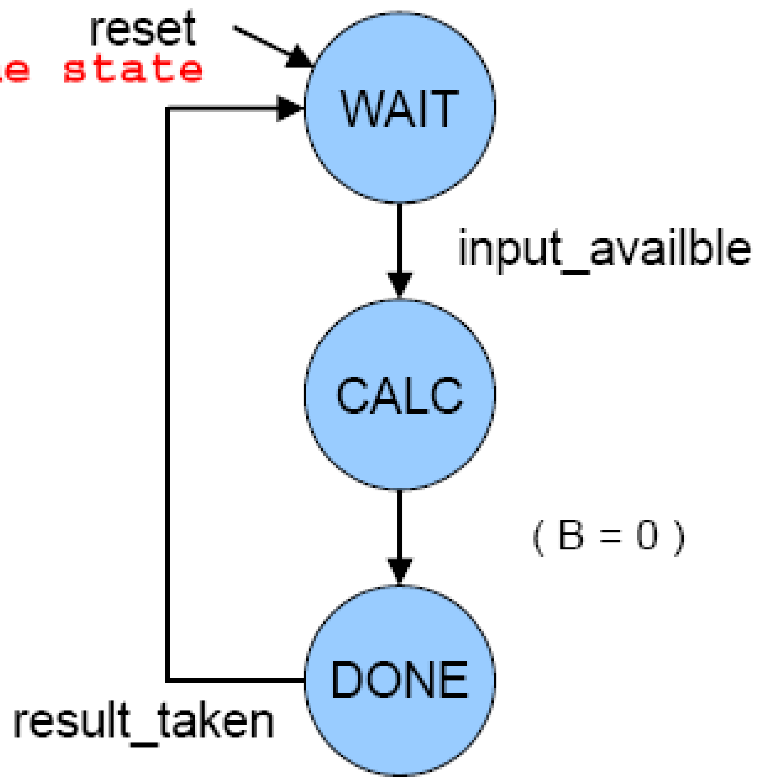
# GCD的控制单元

```
reg [6:0] cs;
always @(*)
begin
  //Default control signals
  A_sel      = A_SEL_X;
  A_en       = 1'b0;
  B_sel      = B_SEL_X;
  B_en       = 1'b0;
  Idle       = 1'b0;
  result_rdy    = 1'b0;
  case ( state )
    WAIT :
      ...
    CALC :
      ...
    DONE :
      ...
  endcase
end
```

```
WAIT: begin
        A_sel      = A_SEL_IN;
        A_en       = 1'b1;
        B_sel      = B_SEL_IN;
        B_en       = 1'b1;
        Idle       = 1'b1;
      end
CALC: if ( A_lt_B )
        A_sel = A_SEL_B;
        A_en       = 1'b1;
        B_sel = B_SEL_A;
        B_en       = 1'b1;
      else if ( !B_zero )
        A_sel = A_SEL_SUB;
        A_en       = 1'b1;
      end
DONE: result_rdy = 1'b1;
```

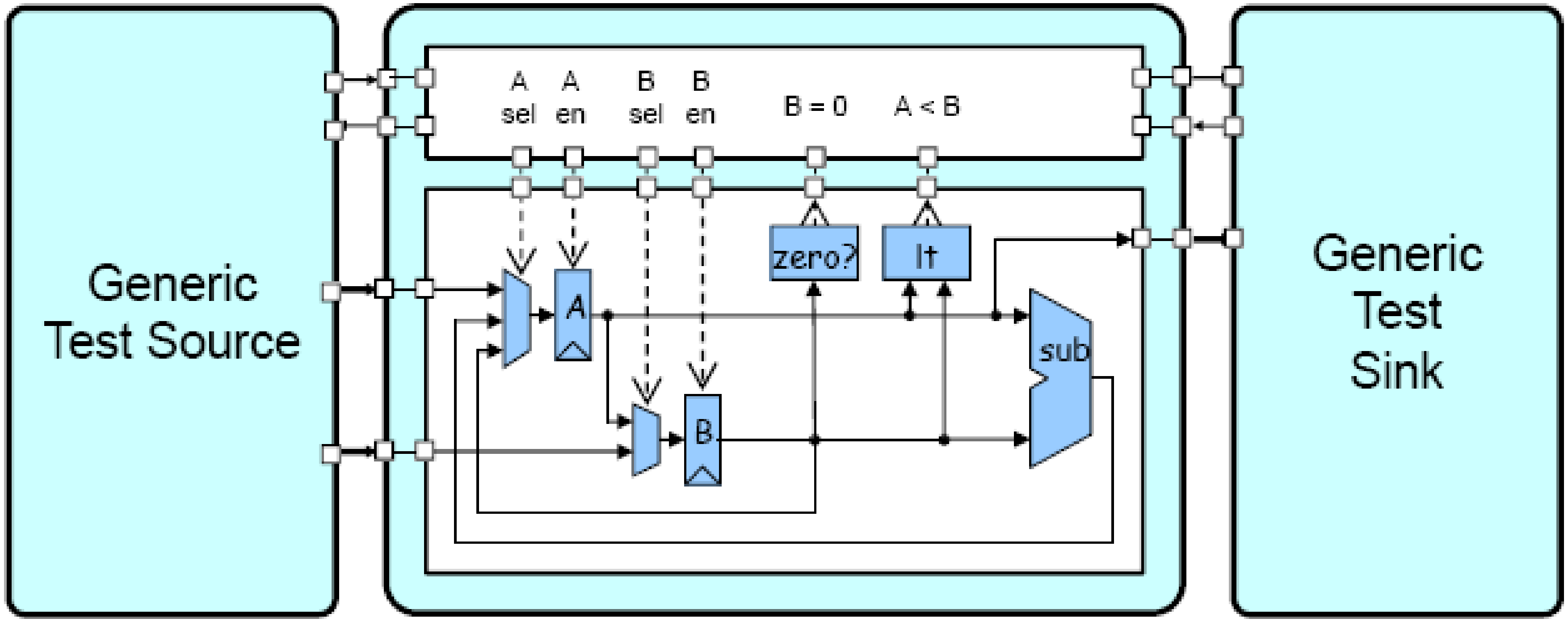
# GCD的控制单元

```
always @(*)
begin
  // Default is to stay in the same state
  state_next = state;

  case ( state )
    WAIT :
      if ( input_available )
        state_next = CALC;
    CALC :
      if ( B_zero )
        state_next = DONE;
    DONE :
      if ( result_taken )
        state_next = WAIT;
  endcase
end
```

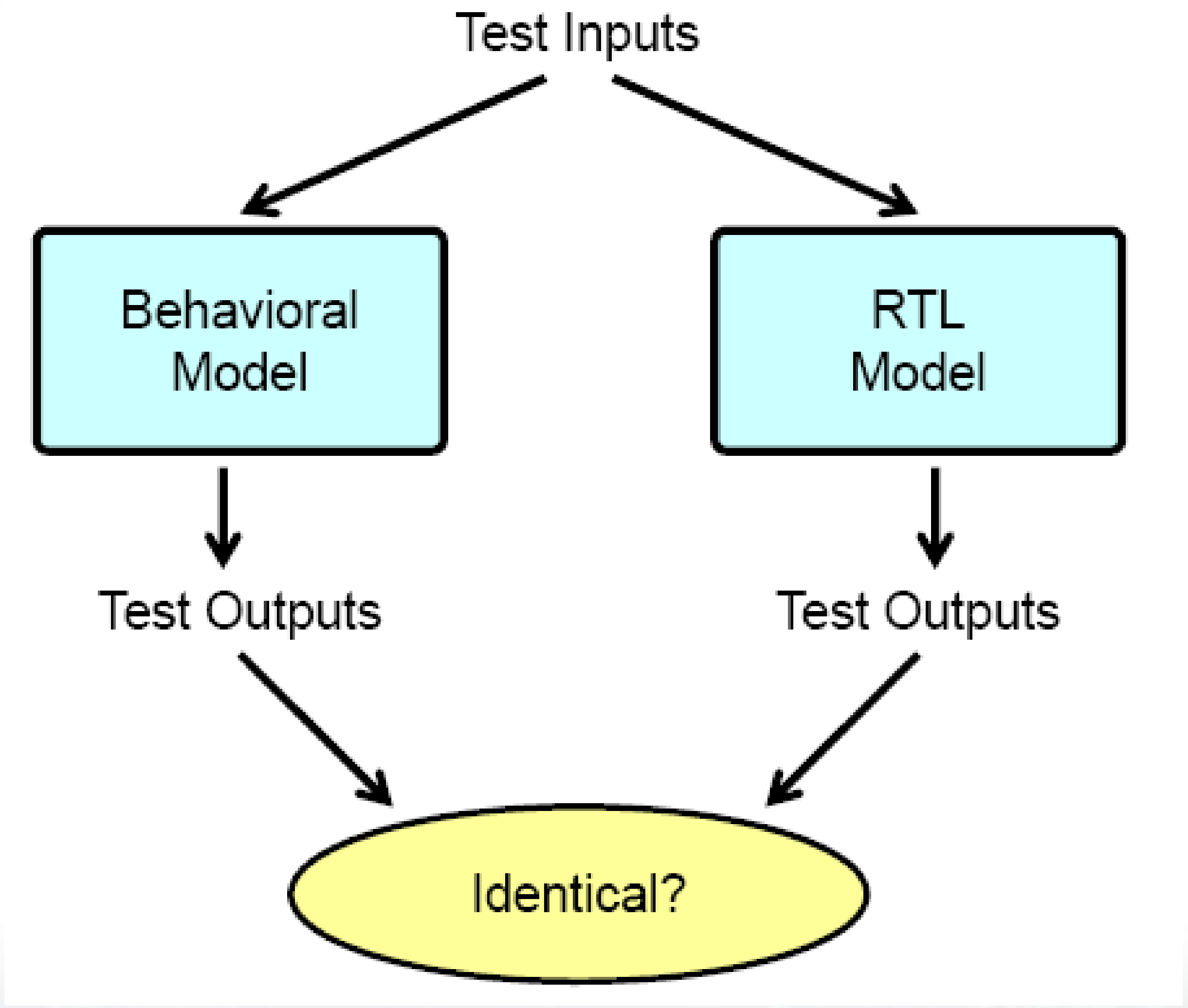


**状态跳变**

## GCD模块的测试


Generic
Test Source
A
sel
A
en
B
sel
B
en
B = 0
A < B
zero?
lt
A
B
sub
Generic
Test
Sink

## GCD模块的测试


Test Inputs
Behavioral Model
RTL Model
Test Outputs
Test Outputs
Identical?

## 代码规范

# 代码规范

| [NC-1-R] | A file must contain only one module |
|---|---|

- Simplifies source code management

| [NC-2-R] | File name has to be identical to module name |
|---|---|

- *<module_name>.v*

```
../counter.v
```

```
module counter (
  clk,
  a_reset_l,
  inc_h,
  count
);
...
```

# 代码规范

| [NC-3-R] | Only [A-Z,a-z,0-9,_] are allowed for identifier names |
|---|---|

| [NC-4-R] | Use lower case for port, signal, module and instance names |
|---|---|

- Verilog is case sensitiv!

```
module counter(
  clk,
  ...
  );

  input  clk;
  output [3:0] count;

  reg    [3:0] count;
  ...
```

# 代码规范

| [NC-5-R] | Use upper case for parameters |
|---|---|

```
parameter DATA_WIDTH = 32;
parameter ADDR_WIDTH = 8;
```

| [NC-6-G] | Instance name should include module name |
|---|---|

- Single module instantiation

```
counter counter_i( ... );
```

- Multiple module instantiations

```
counter counter_i0( ... );
counter counter_i1( ... );
counter counter_i2( ... );
```

# 代码规范

| [NC-7-R] | Use named association port map |
|---|---|

```
counter counter_i(
          clk,
          a_reset_l,
          inc_h,
          count
          );
```

```
counter counter_i(
          .clk(clk),
          .a_reset_l(a_reset_l),
          .inc_h(inc_h),
          .count(count)
          );
```

| [NC-8-G] | Use consistent signal names |
|---|---|

```
counter counter_i(
          .clk(clk2),
          .a_reset_l(reset),
          .inc_h(en),
          .count(data)
          );
```

# 代码规范

| | |
|---|---|
| [NC-9-G] | Use sufficies to differentiate active low/high signals |

- `_l` for active low and `_h` for active high

| | |
|---|---|
| [NC-10-R] | Define only one port per line and use comments |

```
...
input  clk;          //system clock
input  a_reset_l;    //asynchronous reset, activ low
...
```

| | |
|---|---|
| [NC-11-G] | Port order has to be all inputs, inouts, outputs |

# 代码规范

| [NC-12-G] | Use consistent ordering of bus bits |
|---|---|

- Recommended order: MSB to LSB

```
wire [63:0] bus;
reg  [15:0] data;
reg  [7:0] addr;
```

| [NC-13-R] | Explicitly define bit width for static signal assignments |
|---|---|

# 代码规范

[NC-14-R] Avoid operand size mismatch

- Automatic type cast by zero-extention! Do you really want this?

```
wire [63:0] bus;
reg  [15:0] data;
reg  [7:0] addr;

assign bus = data & addr;
```

```
wire [63:0] bus;
reg  [15:0] data;
reg  [7:0] addr;

assign bus = {48'b0, data} & {48'b0, 8'b1, addr};
```

# 代码规范

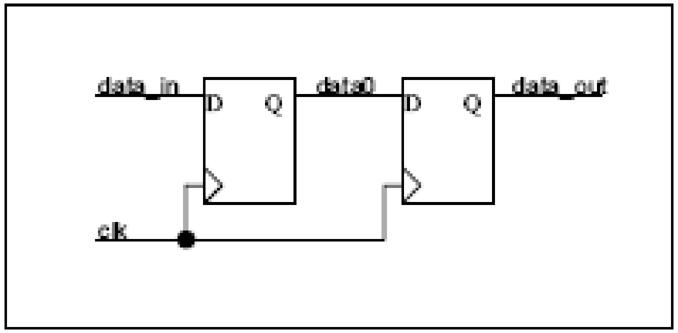
| [SIM-1-R] | Use non-blocking assignments for sequential always blocks |
|---|---|

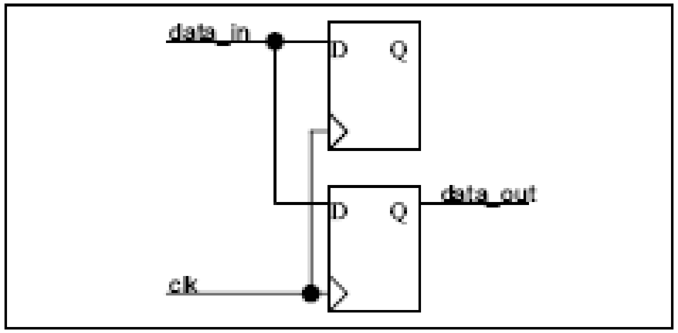
- Verilog race conditions! Order of block execution not defined

```
always @(posedge clk)
begin
  data0 <= data_in;
end

always @(posedge clk)
begin
  data_out <= data0;
end
```

```
always @(posedge clk)
begin
  data0 = data_in;
end

always @(posedge clk)
begin
  data_out = data0;
end
```

?





# 代码规范

| [SIM-2-R] | Use blocking assignments for combinational always blocks |
|---|---|

- Using of non-blocking assignments for description of combinational logic requires more simulator memory and an extended sensitivity list could be necessary

```
always @(a or b or c)
begin
  d = a & b;
  e = d | c;
end
```

```
always @(a or b or c or d)
begin
  d <= a & b;
  e <= d | c;
end
```

| [SIM-3-R] | Don't mix blocking/non-blocking assignments in one always block |
|---|---|

- Don't write compact verilog code at the expense of readability
- Clear separation of combinational and sequential logic avoids verilog race conditions and simulation-synthesis mismatch

## 代码规范

| [SIM-4-R] | Don't make assignments to a signal from different always blocks |
|---|---|

- Order of block execution is not defined, potential Verilog race condition/simulation-synthesis mismatch

```
always @(a or b)
begin
  e = a | b;
end

always @(c or d)
begin
  e = c & d;
end
```

# 代码规范

| | |
|---|---|
| [SIM-5-R] | Sensitivity list of combinational always block has to be complete |

- Incomplete sensitivity list could lead to simulation-synthesis mismatch

```
always @(a)
begin
  c = a | b;
end
```

| | |
|---|---|
| [SIM-6-R] | Avoid unnecessary signals in sensitivity list |

- Additional signals could result in a reduced simulation performance

```
always @(a or b or x)
begin
  c = a | b;
end
```

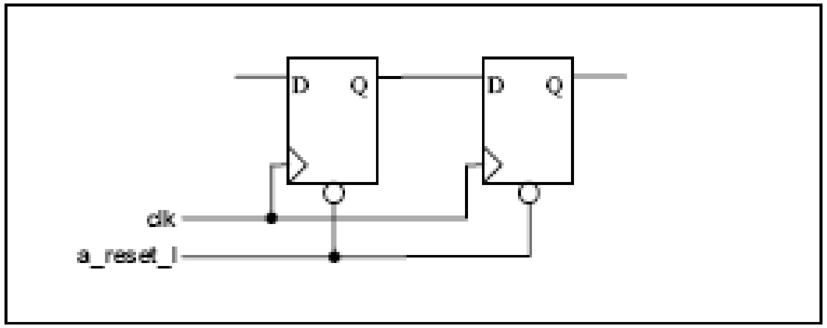
# 代码规范

| [SIM-7-R] | Flip-Flops have to have either a set or a reset |
|---|---|

- To transfer the design to a defined state for simulation

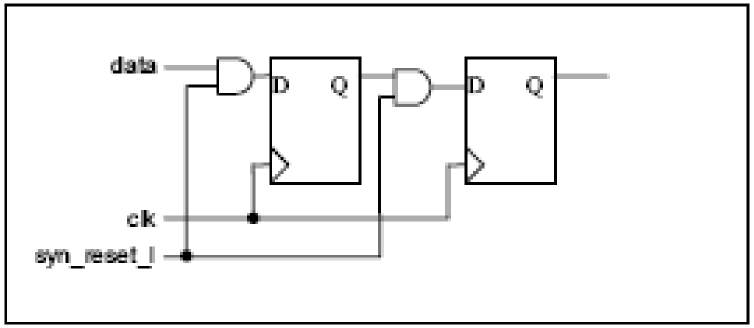
*Asynchronous reset/set*

```
always @(posedge clk or negedge a_reset_l)
begin
  if (a_reset_l == 1'b0) begin
    a <= 1'b0;
    b <= 1'b1;
  end
  else begin
  ...
  end
end
```



*Synchronous reset/set*

```
always @(posedge clk)
begin
  if (syn_reset_l == 1'b0) begin
    a <= 1'b0;
    b <= 1'b1;
  end
  else begin
  ...
  end
end
```



# 代码规范

| [SYN-1-R] | Incomplete case statements have to have default case |
|---|---|

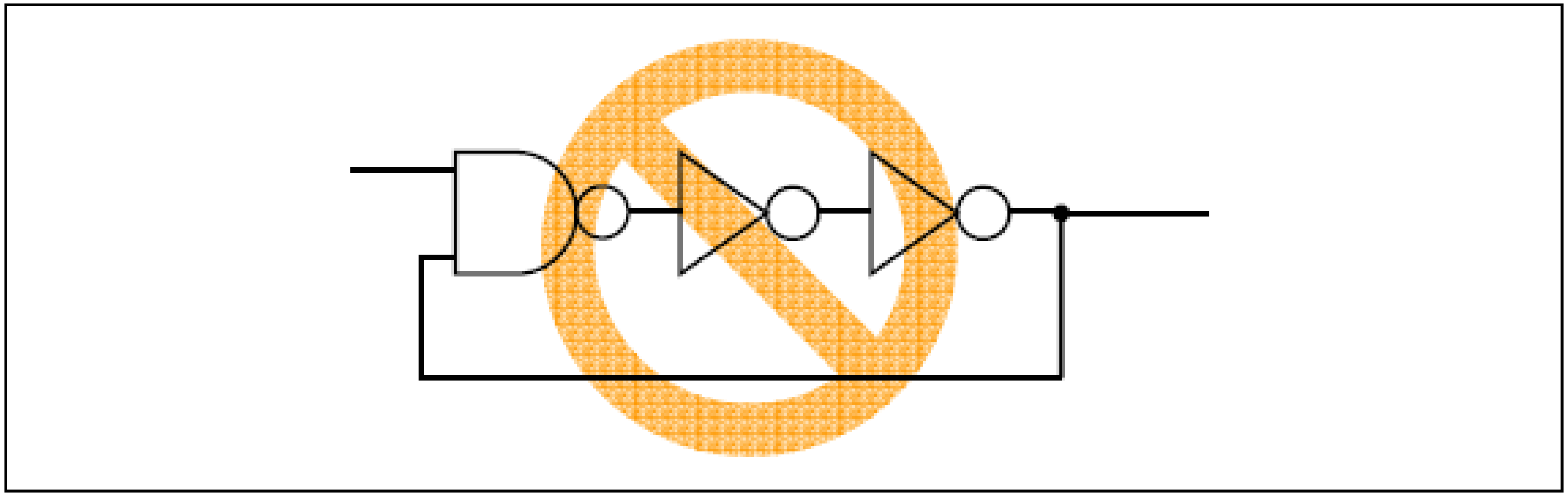
- No unexpected latches are inferred

```
always @(condition)
begin
  case (condition)
    2'b00   : ...
    2'b01   : ...
    2'b10   : ...
    default : ...
  end case
end
```

# 代码规范

| [SYN-2-R] | Combinational feedback loops are forbidden |
|---|---|

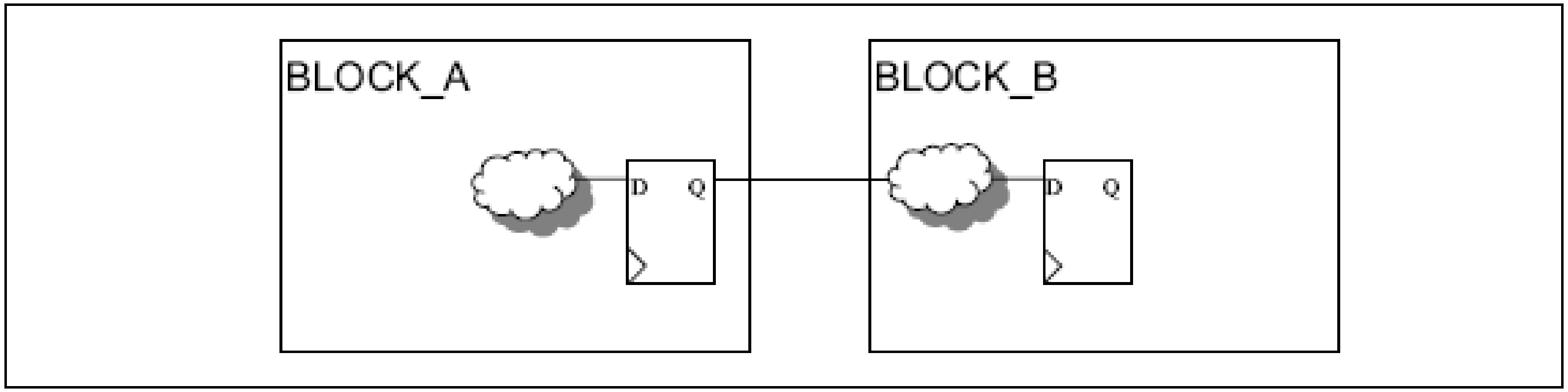
- Combinational feedback loop will cause problems during timing analysis

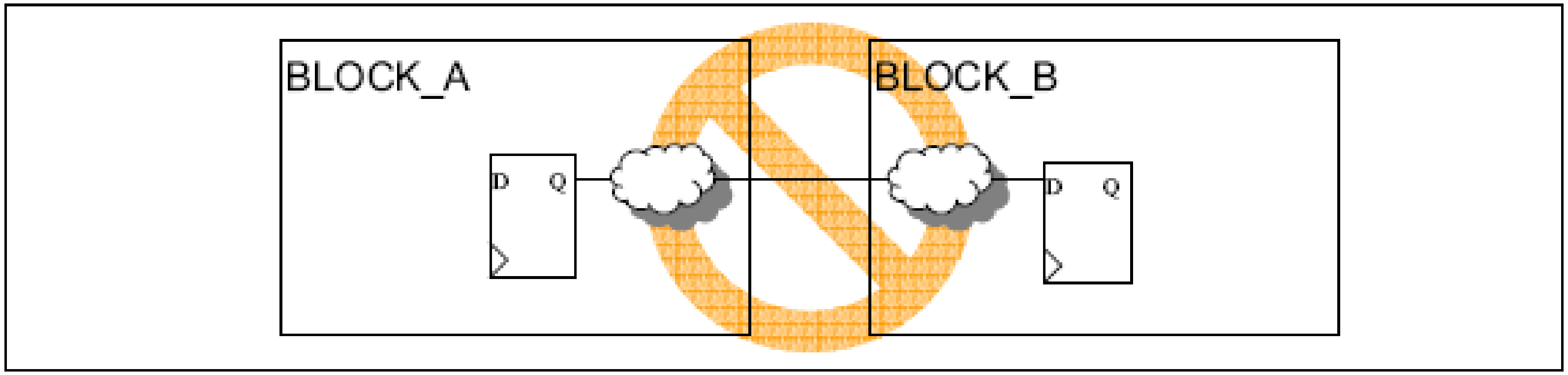


# 代码规范

| [SYN-3-G] | All module outputs should be registered |
|---|---|

- Simplifies synthesis constraints (*set_input_delay, set_output_delay*)

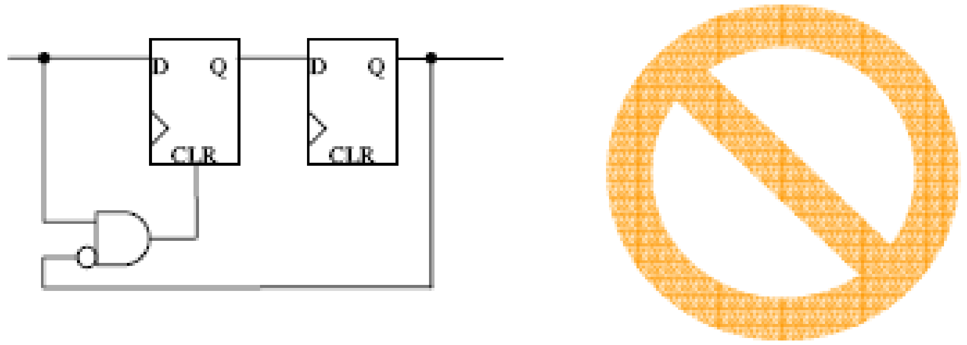




# 代码规范

| [DFT-1-R] | Avoid internally generated set/reset signals |
|---|---|

- Full set/reset control is required for scan operation



| [DFT-2-R] | Avoid internally generated clocks |
|---|---|

- Full clock control is required for scan operation

## 知识点小结

## 知识点小结

- ✱ 阻塞赋值与非阻塞赋值
- ✱ 同步设计及异域时钟的处理
- ✱ 再谈有限状态机
- ✱ 块思维
- ✱ 代码规范