基本逻辑门：

1 AND (与)

逻辑符号：&
规则：只有两个输入都为 1 时输出才为 1。

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

2 OR (或)

逻辑符号：|
规则：只要有一个输入为 1，输出就是 1。

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

3 NOT (非)

逻辑符号：~或!
规则：取反，0 → 1，1 → 0。

A	~A
0	1
1	0

4 XOR (异或)

逻辑符号：^
规则：输入相同输出 0，输入不同输出 1。

A	B	A ^ B
0	0	0
0	1	1
1	0	1

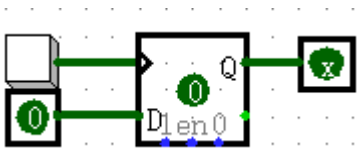
A	B	A ^ B
1	1	0

📌 小结：

- AND = “都要 1 才行”
- OR = “有 1 就行”
- NOT = “翻转”
- XOR = “不同为 1，相同为 0”

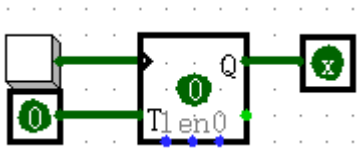
基本门级电路：

1、D触发器



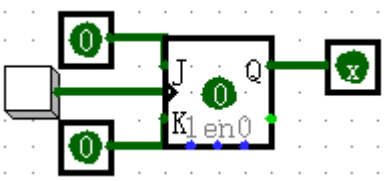

```
D触发器：在时钟边沿把输入 D 的值锁存到输出 Q，最常用，输入即输出。
D   Q(next) 说明
--------------------------
0   0       输出跟随输入，置 0
1   1       输出跟随输入，置 1

module D_flip_flops(input clk,d,output q,not_q);
    always @(posedge clk) begin
        q<=d;
    end
    assign not_q=!q; // 关键！！！
endmodule
```

2、T触发器



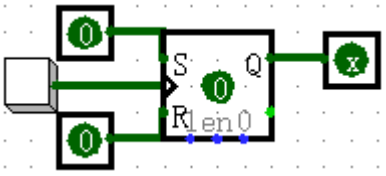

```
T触发器：每次时钟边沿如果 T=1 就翻转输出，否则保持不变，翻转型，常用于分频。
T   Q(next)     说明
--------------------------
0   Q(prev)     保持不变
1   ~Q(prev)    翻转
```

3、JK触发器



```
JK触发器：综合了RS和T触发器功能，J=1,K=0置1，J=0,K=1清0，J=K=1翻转，功能全，能模拟其他触发器。
J   K   Q(next)     说明
-------------------------
0   0   Q(prev)     保持
0   1   0           复位
1   0   1           置位
1   1   ~Q(prev)    翻转

module JK_flip_flops (
    input  wire clk,
    input  wire j,
    input  wire k,
    output wire Q
);
    wire D;
    assign D = (j & ~Q) | (!k & Q);
    D_flip_flops u_dff (.clk(clk), .d(D), .q(Q));
endmodule

module D_flip_flops (
    input  wire clk,
    input  wire d,
    output reg  q
);
    always @(posedge clk) q <= d;
endmodule
```

4、RS锁存器（SR锁存器）




```
RS锁存器：通过 R（复位） 和 S（置位） 控制输出状态，最基础的存储电路，但有禁止态。
S   R   Q(next)     说明
---------------------------------
0   0   Q(prev)     保持
1   0   1           置位
0   1   0           复位
1   1   未定义      禁止态（forbidden）
```

verilog语言知识点：

“ | | ”为逻辑运算符，a| | b当且仅当a与b都为0时，结果才为零

“ {} ”为拼接符，b[15:0]={a[15:1] , 1'b1}

“ ^ ”异或，“ | ”或，“ & ”与，“ ~”按位取反，“！ ”逻辑取反“ ~^、^~、~(a ^ b)、~a ^ b ”异或非

assign 是 **连续赋值**，只能在过程块（always、initial）外面用。而reg类型只能在always、initial里使用。

编译指令default_nettype none，若不声明则未声明的信号会被当作wire导致出错。

在纯Verilog-2001 或之前标准里不支持 int 类型而是integer(还必须在always块外部声明)且不支持i--/++而是i=i-1或i=i+1（仅SystemVerilog支持现代写法）

BCD码（Binary-Coded Decimal，二进制编码的十进制）：

十进制	BCD（4-bit）
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

```
module bcd (
    input clk,
    input reset,
    input enable,
    output [3:0] q);
    always @(posedge clk) begin
        if(reset) q<=0;
        else q<=(!enable)?q:
                (q<4'd9)?(q+1):0;
    end
endmodule
```

逆反字符串方法：

```
always @(*) begin
        for (int i=0; i<8; i++)
            out[i] = in[8-i-1];
end
```

字符串自复制方法：

```
{num{向量}}这将向量复制num次。num必须是常量。两组括号都是必需的
{5{1'b1}} // 5'b11111（或 5'd31 或 5'h1f）
{2{a,b,c}} // 与 {a,b,c,a,b,c} 相同
{3'd5, {2{3'd6}}} // 9'b101_110_110
```

模块实例化：

```
module mod_a ( input in1, input in2, output out );
    // Module body
endmodule

方法一：mod_a instance1(a, b, c);
方法二：mod_a instance2(.out(c), .in1(a), .in2(b));
```

阻塞赋值和非阻塞赋值：

> 阻塞赋值（=）：顺序执行，后面的语句要等前面的执行完才能执行，像 C 语言那样一步一步算，先算完 x，再算 y。
> 非阻塞赋值（<=）：并行执行，右边的值在时钟边沿“同时”更新，在时钟来临时，大家一起更新，好比多个寄存器同时锁存。

case：

```
case(sel)
    2'b00: q = d;
    2'b01: q = w1;
    2'b10: q = w2;
    2'b11: q = w3;
    default q = d;
endcase
```

if / else：

```
方法一：
assign out_assign=((sel_b1==1)&&(sel_b2==1)) ? b : a;
————————————————————————————————————————————————————————————————————————————————
方法二：
always @(*) begin
    if((sel_b1==1)&&(sel_b2==1))    out_always=b;
    else    out_always=a;
end
```

在实例化的时候可以空开一个参数不填吗？

```
my_module u1 (a, b, , d);   // ❌ 不允许空着

my_module u2 (
    .a (sig_a),
    .b (sig_b),
    // .c ()   // ✅ 可以省略整个端口
    .d (sig_d)
);
```

减法实现：

```
out_b = b^{32{sub}} + sub;//当sub=1时是减法，sub=0时是加法
```

"在组合式Always 块中，使用阻塞赋值。在时钟控制式Always 块中，使用非阻塞赋值"：

```
一、组合逻辑 always 块
always @(*) begin
    y = a & b;
```

```
    z = y | c;
end
```
用阻塞赋值=表示语句顺序执行，后面的语句能用到前面刚算好的值

二、时序逻辑 always 块

```
always @(posedge clk or posedge rst) begin
    if (rst)
        q <= 0;
    else
        q <= d;
end
```
用非阻塞赋值<=在同一个时钟沿，所有非阻塞赋值同时更新

casez：

```
casez 是 Verilog / SystemVerilog 里 带通配符的 case 语句，主要用在 组合逻辑匹配和优先级判断，casez 会把 z 和 ? 视作通配符，在匹配时忽略对应位的 0/1。

casez (in[3:0])  // in[3:1] can be anything
  4'bzzz1: out = 0;
  4'bzz1z: out = 1;
  4'bz1zz: out = 2;
  4'b1zzz: out = 3;
  default: out = 0;
endcase
```

一元缩减运算符：

```
& a[3:0] // AND: a[3]&a[2]&a[1]&a[0]. 等价于 (a[3:0] == 4'hf)
| b[3:0] // OR: b[3]|b[2]|b[1]|b[0]. 等价于 (b[3:0] != 4'h0)
^ c[2:0] // XOR: c[2]^c[1]^c[0]
```

返回信号宽度的系统函数$bits()：

```
reg [99:0] out
$bits(out) is 100
example:    for (int i=0;i<$bits(out);i++)
```

生成语句generate+genvar，可实现批量实例化

```
基本语法：
genvar i;  // 声明一个 generate 变量
generate
    for (i = 0; i < N; i = i + 1) begin : block_name
        // 在这里写你要重复的东西
        // 例如bcd_fadd fa(
        //         a[i*4+3:i*4],
        //         b[i*4+3:i*4],
        //         c[i],
        //         c[i+1],
        //         sum[i*4+3:i*4]
        //     );
    end
endgenerate
```

真值表 → 卡诺图（K-map）化简 → 最简逻辑式 → 门级电路 → Verilog

一、真值表

```
x3  x2  x1  f
0       0       0       0
0       0       1       0
0       1       0       1
0       1       1       1
1       0       0       0
1       0       1       1
1       1       0       0
1       1       1       1
```

二、卡诺图（K-map）

```
        x2 x1
         00   01   11   10
x3=0     0    0    1    1
x3=1     0    1    1    0
```

三、化简

找出 1 的格子：从上表中 f=1 的行有：

Row 2: 0 1 0 → (x3=0, col=10)

Row 3: 0 1 1 → (x3=0, col=11)

Row 5: 1 0 1 → (x3=1, col=01)

Row 7: 1 1 1 → (x3=1, col=11)

横向组合：x3=0, col=11 & col=10 → 两个相邻 → x3=0, x2=1 → minterm = ~x3 & x2

纵向组合：col=11, x3=0 & x3=1 → 两个相邻 → x2=1, x1=1 → minterm = x2 & x1

四、最简逻辑式

最简逻辑表达式：f=(x3&x1)|(x2&x1)|(!x3&x2)

如何利用“积之和（SOP，最小项化简）”和“和之积（POS，最大项化简）”两种形式简化“卡诺图”？

例子：

```
  BC   00  01  11  10
————————————————————
A=0    0   0   1   1
A=1    0   1   1   0
```

SOP：从 1 出发 → 用“积项相加”；F=!AB+AC

m2 & m3（横向相邻，A=0, B=1 不变） ⇒ 项： $!AB$

m5 & m7（纵向可合并为 A C，因为 m5(101) 与 m7(111) 都有 C=1, A=1） ⇒ 项： AC

POS：从 0 出发 → 用“和项相乘”；F=(A+B)(!A+C)

零点为 m0,m1,m4,m6。先写 $F'=\Sigma m(0,1,4,6)$的 SOP，再取补得到 POS

F’=m0+m1+m4+m6=!A!B+A!C，然后取反得F=(!A!B+A!C)'=(A+B)(!A+C)

规则：

1、♥要充分利用无关项可以当 0 也可以当 1 的特点，尽量扩大卡诺圈♥

2、每个圈应至少包含一个新的 1格，否则这个圈是多余的

3、用卡诺图化简所得到的最简与或式不是唯一的

4、将卡诺图中的 1格画圈，一个也不能漏圈，否则最后得到的表达式就会与所给函数不等；1格允许被一个以上的圈所包围

SystemVerilog 的动态切片：

```
out = in[sel*4 +: 4]; // +: 表示从 sel*4 开始的 4 位

若不用动态切片也可使用out = {in[sel*4+3], in[sel*4+2], in[sel*4+1], in[sel*4+0]};//语法禁止位宽含有变量，但允许位宽为1的含变量形式存在，可用多个这种形式组成合并。
```

（有/无符号数）溢出判断：

```
{cout,sum} = a + b
有符号加法溢出只与 最高位 和 结果最高位 有关//overflow = (sum[最高位]!=a[最高位]) & (a[最高位]==b[最高位]);
无符号溢出才用 cout//overflow = cout;
```

延迟周期信号

```
reg sig_last;// 延迟寄存器
always @(posedge clk) begin
     sig_last <= sig;   // 延迟一个时钟周期输出
end

reg sig_last, sig_last1;// 延迟寄存器
always @(posedge clk) begin
     sig_last <= sig;
     sig_last1 <= sig_last; // 延迟两个时钟周期输出
end
```

双边沿触发器

```
module detff_gate  (input clk,input d,output q);
    reg q_rise, q_fall;
    // 上升沿触发的触发器
    always @(posedge clk) begin
        q_rise <= d;
    end
    // 下降沿触发的触发器
    always @(negedge clk) begin
        q_fall <= d;
    end
    // 根据时钟选择输出
    assign q = clk ? q_rise : q_fall;
endmodule
```

不能在条件运算符里用 q++，应该改为 q + 1，Verilog 里的 ++/-- 只在过程块里单独一行使用，不能嵌套在表达式中

```
q<=(q<4'd9)?(q++):0;//错误
q<=(q<4'd9)?(q+1):0;//正确
```

三个BCD计数器组成能将1000Hz转1Hz

```
module top_module (
    input clk,
    input reset,
    output OneHertz,
    output [2:0] c_enable
);
```

```
    wire [3:0] q0, q1, q2;
    // 级联使能逻辑
    assign c_enable[0] = 1'b1;              // 最低位计数器始终运行
    assign c_enable[1] = (q0 == 4'd9);      // q0到9时，允许q1进位
    assign c_enable[2] = (q0 == 4'd9) & (q1 == 4'd9); // q0=9且q1=9时，允许q2进位
    // 三级BCD计数器
    bcdcount counter0 (
        .clk(clk),
        .reset(reset),
        .enable(c_enable[0]),
        .Q(q0)
    );
    bcdcount counter1 (
        .clk(clk),
        .reset(reset),
        .enable(c_enable[1]),
        .Q(q1)
    );
    bcdcount counter2 (
        .clk(clk),
        .reset(reset),
        .enable(c_enable[2]),
        .Q(q2)
    );
    // 当 q2q1q0 = 999 时，下一个周期归零 => 产生OneHertz
    assign OneHertz = (q0 == 4'd9) & (q1 == 4'd9) & (q2 == 4'd9);
endmodule
```

不能在同一模块内部被**自己赋值**

```
output pm;
assign pm = ~pm;// 错误！！！这种写法只能在always块里使用
```

q[3:0] + 1'b1和q[3:0] + 1区别：

> 当个位 `q[3:0]` 与 `1'b1` 相加时，`Verilog` 会把 `1'b1` 自动扩展到 `4` 位（或者最少足够的位宽），然后执行加法。
>
> 当 `q[3:0]`（`4` 位）与 `32` 位 `1` 相加时，`Verilog` 会 先扩展 `q[3:0]` 到 `32` 位再计算。

算术右移和逻辑右移的区别： 只有当操作数为signed时候算术运算>>>或<<<才起作用否则和逻辑移位没区别.

移位类型	运算符	高位填充	适用类型
逻辑右移	>>	0	无符号数
算术右移	>>>	符号位	有符号数

reg [3:0] arr的索引范围是0~3，且值大小范围0~15。reg [位宽] arr [向量个数]

- `reg [15:0] arr [15:0];` → 16 个元素，每个是 16 bit → **16 行，每行 16 bit 向量**
- `reg arr [15:0][15:0];` → 16×16 个元素，每个是 1 bit → **16 行 × 16 列的单 bit 矩阵**

子模块命名不能和官方库重复，否则会报错

```
dff ins3(KEY[0],(LEDR[1]^LEDR[2]),SW[2],KEY[1],LEDR[2]);

module dff(input clk,d0,d1,L,output q);//不允许！！！
    always @(posedge clk) begin
        q<=(L)?d1:d0;
    end
endmodule
```

组合逻辑反馈环路（例: `Q <= ... Q`）

```
方法一：中间变量
wire D; // 声明一个中间连线
// 组合逻辑：根据控制信号选择下一状态D的值
assign D = L ? R : (E ? w : Q);
// 时序逻辑：在时钟上升沿将D的值寄存到Q
always @(posedge clk) begin
     Q <= D;
end
方法二：
always @(posedge clk) begin
     if (L)
         Q <= R;     // 加载模式 (Load)
     else if (E)
         Q <= w;     // 使能模式 (Enable)
     // 否则保持Q不变，无需写 else Q <= Q;
end
```

wire类型与物理连线不同，Verilog 中的连线具有*方向性*。这意味着信息仅沿一个方向流动，从右往左。

parameter是 Verilog 中用于定义**常量**的关键字，它定义的标识符代表一个在编译时就已经确定的固定值，在代码运行时不能被修改，这类似于 C 语言中的 `#define`宏定义。

如果你的复位是异步的，就需要在敏感列表中加入 `posedge reset`；如果是同步的，则不需要

有限状态机FSM基本框架

```
module top_module(
    input clk,
    input areset,
    input bump_left,
    input bump_right,
    output walk_left,
    output walk_right);

    parameter LEFT=0, RIGHT=1;
    /*
    状态定义有两种方式：
    一、数字：
    parameter A=3'd0,B=3'd1,C=3'd2,D=3'd3,E=3'd4,F=3'd5;
    二、独热码：
    parameter A=6'b000001;B=6'b000010;C=6'b000100;D=6'b001000;E=6'b010000; F=6'b100000;
    */
    reg state, next_state;

    always @(*) begin
        // State transition logic
        case(state)
```

```
            LEFT:  next_state=(bump_left)?RIGHT:LEFT;
            RIGHT: next_state=(bump_right)?LEFT:RIGHT;
        endcase
    end

    always @(posedge clk, posedge areset) begin
        // State flip-flops with asynchronous reset
        if(areset) state<=LEFT;
        else state<=next_state;
    end

    // Output logic
    assign walk_left  = (state == LEFT);
    assign walk_right = (state == RIGHT);

endmodule
```

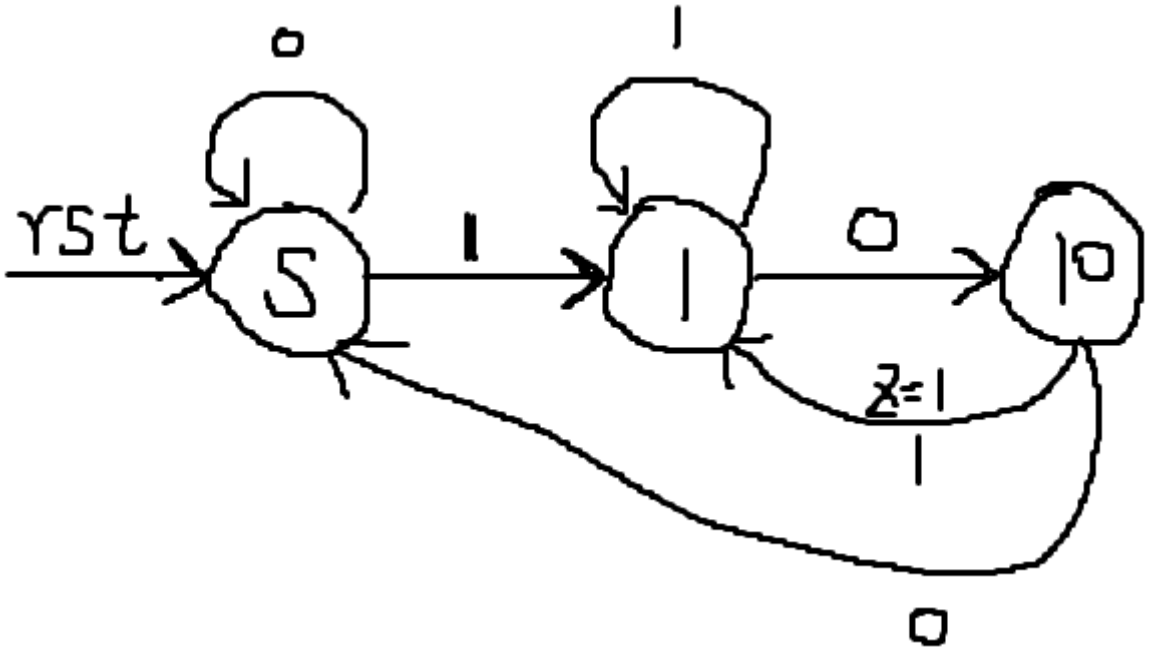
延迟周期信号

```
reg sig_last;// 延迟寄存器
always @(posedge clk) begin
     sig_last <= sig;   // 延迟一个时钟周期输出
     out <= sig_last;
end

reg sig_last, sig_last1;// 延迟寄存器
always @(posedge clk) begin
     sig_last <= sig;
     sig_last1 <= sig_last; // 延迟两个时钟周期输出
     out <= sig_last1;
end
```

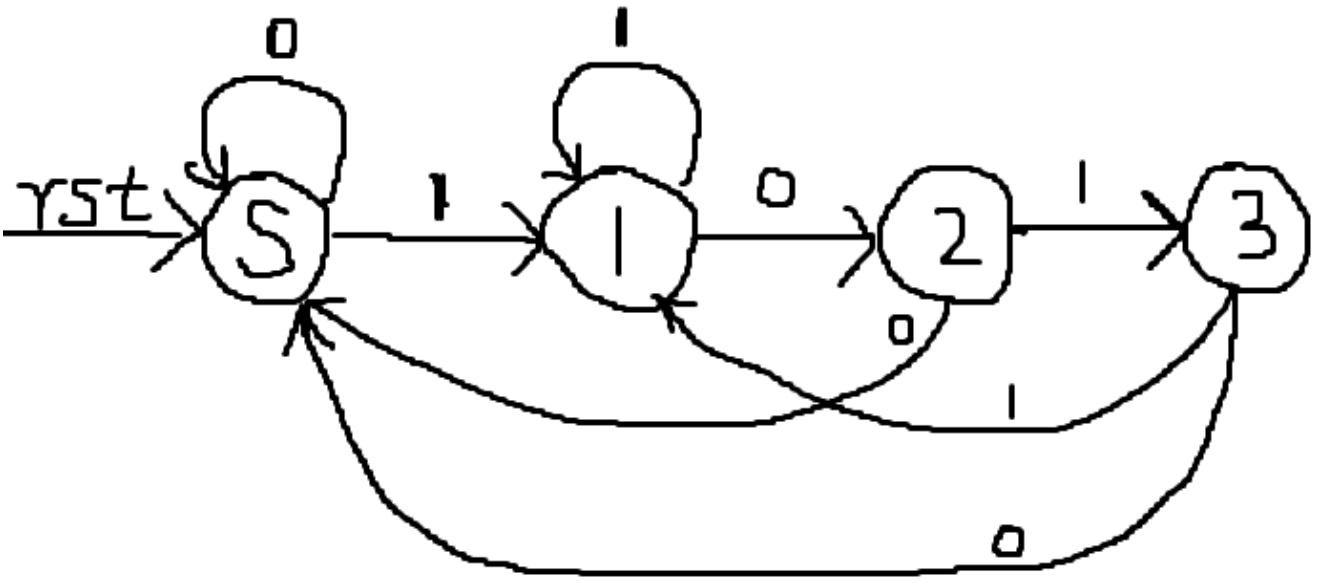
Moore FSM 的输出仅由当前状态决定，而 Mealy FSM 的输出则由当前状态和当前输入共同决定。

题目：用于识别输入信号x上的序列“101”。您的有限状态机应该有一个输出信号z，当检测到“101”序列时，该信号被置位为逻辑 1。

Mealy型：



Moore型：



'x 是 Verilog 的一种**字面值（literal）表示法**，意思是“**未知值（unknown）**

```
case (state)
    s0: next = s1;
    s1: next = s2;
    s2: next = s0;
    default: next = 'x;   // <- 这里
endcase
```

'x这表示 “宽度自动匹配，被赋值变量的宽度”，并且所有位都是 X。

在always @(posedge clk) 块里混用阻塞和非阻塞，**执行顺序**是先阻塞最后非阻塞在时钟沿结束后统一执行，而阻塞语句之间按照书写顺序从上到下依次顺序执行。

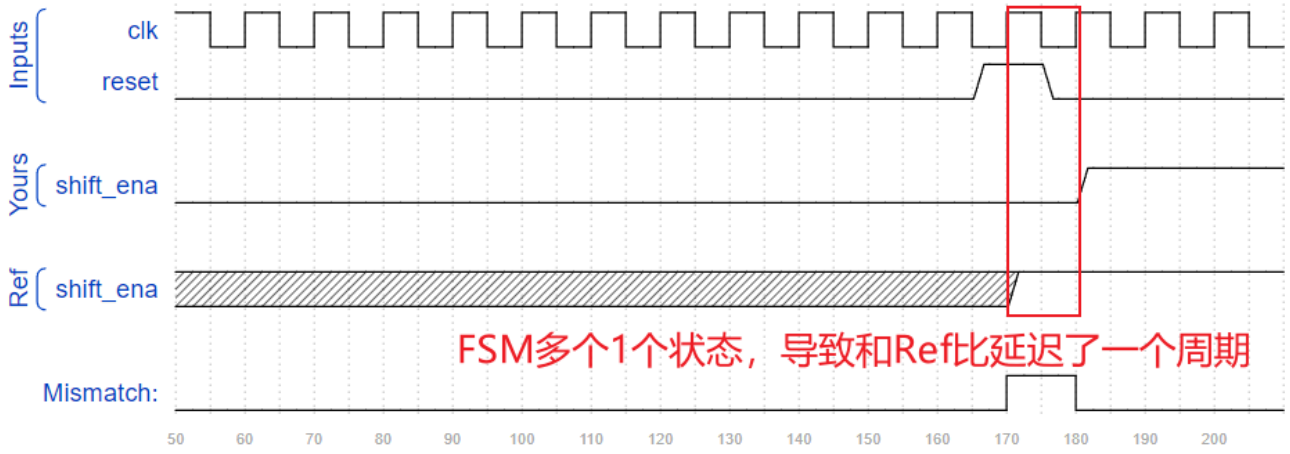
always块相当于一个触发器。**确保每个寄存器信号只在一个时序 always块中被赋值（若出现两个 always @(posedge clk)块都对同一个寄存器信号 out进行了赋值，则会Can't resolve multiple constant drivers for net “信号”；解决办法：合并always块）**

```
always @(posedge clk) begin  // 触发条件：在时钟上升沿触发
    xxx
end
```

易错点：**illegal name "B" used in expression File**。原因：忘记加case语句了

```
always @(*) begin
    // case()
        A: next_state=(data)?B:A;
        B: next_state=(data)?C:A;
        C: next_state=(data)?C:D;
        D: next_state=(data)?B:A;
    // endcase
end
```

FSM易错点：状态数量的多与少：



在always块里不能对非reg对象赋值；可以在assign语句里使用reg对象对其他非reg对象赋值。

'#'是什么？在 Verilog 里，'#'表示延迟（delay）。它告诉仿真器：等指定的时间后再执行下一步操作。

```
#5 clk = ~clk; // 意思是：“延迟 5 个时间单位，然后执行 clk = ~clk;

initial begin
    A=0;B=0;
    #10 A=1; // 程序卡在这里10ps,然后再往下顺序执行
    #5  B=1; // 程序卡在这里5ps,然后再往下顺序执行
    #5  A=0; // 程序卡在这里5ps,然后再往下顺序执行
    #20 B=0; // 程序卡在这里20ps,然后再往下顺序执行
end
```

forever只能在always块或initial块内使用

```
initial begin
    forever #5 clk=!clk; //每隔5ps震荡一次，周期是10ps 或者这样写always #5 clk=!clk;
end
```