Verilog答题与解析

版本日志：

更新于2024/6/25 14:00，这是Verilog答题与解析的第一个版本

更新于2024/6/25 15:28，增加了真题演练3、4、5

更新于2024/6/25 20:16，增加了数电作业5、6、7、8、9、10

更新于2024/6/25 20:39，优化了数电作业10的答案，使其更简洁

更新于2024/6/26 21:12，修正了真题演练1、综合题目：59归0电路设计与显示

更新于2024/6/26 21:24，添加了带有复位逻辑设计的JK触发器、D触发器设计

更新于2024/6/27 22:06，增加了真题演练6、7

点击链接加入群聊【计算机速通之家】：https://qm.qq.com/q/grXg8314uA

带有复位逻辑设计的JK触发器

考点：JK触发器设计、时序逻辑、复位逻辑

```
module JK_FF (
    input wire clk,   // 时钟信号
    input wire rst_n, // 复位信号，低电平有效
    input wire j,     // J输入
    input wire k,     // K输入
    output reg q      // 输出Q
);

    // JK触发器逻辑
    always @(negedge clk or negedge rst_n) begin // 每个时钟周期下降沿和复位信号的下降沿输出
        if (!rst_n) begin
            q <= 1'b0; // 复位时Q置为0
        end else begin
            q <= j & (~q) | (~k) & q;
        end
    end
endmodule
```

核心语法：

1. `always @(negedge clk or negedge rst_n)`：定义一个时序逻辑块，敏感于时钟的下降沿和复位信号的下降沿。
2. 逻辑运算符 `&` 和 `|`：用于实现 JK 触发器的状态转换。

注意事项：

1. **时钟信号的选择**：确保在时钟的下降沿触发逻辑块，以符合JK触发器的设计要求。
2. **复位信号处理**：使用 `negedge rst_n` 使能复位逻辑，确保在复位信号为低电平时将输出 `q` 置为0。
3. **状态转换方程解释**：
   - `j & (~q)`：当 `J` 为高电平且 `q` 当前为低电平时，`q` 应该置为高电平（Set）。
   - `(~k) & q`：当 `K` 为低电平且 `q` 当前为高电平时，`q` 应该保持高电平（保持状态）。
   - `j & (~q) | (~k) & q`：这是将上述两种情况综合起来，当 `J` 和 `K` 都为高电平时，`q` 的状态将翻转（Toggle）。当 `J` 和 `K` 都为低电平时，`q` 保持原状态。
4. **寄存器定义**：输出信号 `q` 必须定义为 `reg` 类型，以便在时序逻辑块中赋值。
5. **一次空翻**：在一个时钟周期内，JK触发器的输出可能会不稳定地改变多次，而不是按预期的仅改变一次。这种现象通常称为“毛刺”或“振荡”。

D触发器设计

考点：D触发器设计、时序逻辑、复位逻辑、避免空翻

```
module D_FF (
    input wire clk,   // 时钟信号
    input wire rst_n, // 复位信号，低电平有效
    input wire d,     // D输入
    output reg q      // 输出Q
);

    // D触发器逻辑
```

```
    always @(posedge clk or negedge rst_n) begin // 每个时钟周期上升沿和复位信号的下降沿输出
        if (!rst_n) begin
            q <= 1'b0; // 复位时Q置为0
        end else begin
            q <= d; // 时钟上升沿时Q跟随D
        end
    end
endmodule
```

核心语法：

1. `always @(posedge clk or negedge rst_n)`：定义一个时序逻辑块，敏感于时钟的上升沿和复位信号的下降沿。
2. 简单的赋值语句 `q <= d`：用于实现 D 触发器的状态转换。

注意事项：

1. **时钟信号的选择**：确保在时钟的上升沿触发逻辑块，以符合D触发器的设计要求。
2. **复位信号处理**：使用 `negedge rst_n` 使能复位逻辑，确保在复位信号为低电平时将输出 `q` 置为0。
3. **输出信号**：输出信号 `q` 必须定义为 `reg` 类型，以便在时序逻辑块中赋值。
4. **避免空翻**：D 触发器通过在时钟的上升沿（或下降沿）锁存输入信号，从而避免了空翻问题。

什么是空翻

空翻（glitch）是指在数字电路中，由于信号传播延迟或竞争现象导致的瞬时错误状态。具体来说，在触发器中，当输入信号在时钟信号边沿附近发生变化时，可能会引发不稳定的输出，这种不稳定状态就是空翻。

D触发器如何避免空翻

D 触发器通过在时钟的上升沿（或下降沿）锁存输入信号，从而避免了空翻问题。其主要机制如下：

1. **采样输入信号**：D 触发器在时钟信号的上升沿（或下降沿）瞬间采样输入信号 `d`，并将其值锁存在输出 `q` 上，直到下一个时钟边沿到来。
2. **同步复位**：使用复位信号（通常为低电平有效）来强制输出 `q` 置为0，从而确保在特定条件下输出状态是可预测的和稳定的。

四位二进制加法器设计

考点：加法器设计、组合逻辑、位运算

```
module binary_adder (
    input [3:0] A, B,
    output reg [3:0] S,
    output reg C_out
);
    // 定义加法器的输出信号
    always @(*) begin
        // 计算四位二进制加法并生成进位
        S = A + B;
        C_out = (A + B > 4'd15) ? 1'b1 : 1'b0;
    end
endmodule
```

核心语法：

- **always块**：使用`always @(*)`实现组合逻辑，根据输入信号`A`和`B`计算输出信号`S`和进位`C_out`。
- **位运算**：使用`>`运算符判断加法结果是否溢出。

注意事项：

- 确保加法器的输入和输出位宽匹配，避免数据截断或溢出。
- 使用三元运算符`?:`来生成进位信号`C_out`，确保对加法结果溢出的正确判断。

十二归一

考点：状态机设计、时序逻辑、边沿触发D触发器

```
module 12to1 (
    input inclk,
    output reg [3:0] oth,
    output reg [3:0] otl
);
    // 定义4位寄存器h和l，用于状态存储
    reg [3:0] h;
    reg [3:0] l;

    // 在时钟上升沿时更新状态
    always @(posedge inclk) begin
        // 如果h为1且l为2，重置h和l
        if (h == 1 && l == 2) begin
            h <= 0;
            l <= 1;
        // 如果l为9，设置h为1，重置l
        end else if (l == 9) begin
            h <= 1;
            l <= 0;
        // 否则，保持h不变，l加1
        end else begin
            h <= h;
            l <= l + 1;
        end
    end
```

```
    // 将寄存器h和l的值赋给输出oth和otl
    always @(*) begin
        oth = h;
        otl = l;
    end
endmodule
```

核心语法：

1. **always块**：使用`always @(posedge inclk)`实现时序逻辑，即在时钟的上升沿触发状态更新。
2. **寄存器定义**：使用`reg [3:0] h`和`reg [3:0] l`定义4位寄存器，存储状态。
3. **条件语句**：使用`if-else`结构实现状态转移条件判断，注意每个条件的顺序和逻辑。
4. **阻塞赋值和非阻塞赋值**：在时序逻辑块中使用非阻塞赋值`<=`，确保寄存器在时钟沿更新时正确赋值。
5. **组合逻辑块**：使用`always @(*)`实现组合逻辑，将寄存器的值赋给输出端口。

注意事项：

- 确保在时序逻辑块中使用非阻塞赋值，以避免时序竞争问题。
- 在组合逻辑块中要注意赋值的顺序和正确性，防止逻辑错误。
- 检查条件语句的完整性，确保所有可能的状态变化都已考虑。

十分频设计

考点：时序逻辑设计、计数器、边沿触发D触发器

```
module fp10 (
    input inclk,
    output reg fpf
);
    // 定义3位计数器寄存器count和频率分频寄存器fp
    reg [2:0] count;
    reg fp;

    // 在时钟上升沿时更新状态
    always @(posedge inclk) begin
        // 如果计数器count等于4，重置count并翻转fp
        if (count == 3'b100) begin
            count <= 3'b000;
            fp <= ~fp;
        // 否则，计数器count加1，fp保持不变
        end else begin
            count <= count + 1;
            fp <= fp;
        end
    end

    // 将fp的值赋给输出fpf
    always @(*) begin
        fpf = fp;
    end
endmodule
```

核心语法：

1. **always块**：使用`always @(posedge inclk)`实现时序逻辑，即在时钟的上升沿触发状态更新。
2. **寄存器定义**：使用`reg [2:0] count`定义3位计数器寄存器，和使用`reg fp`定义分频寄存器。
3. **条件语句**：使用`if-else`结构实现状态转移条件判断，注意每个条件的顺序和逻辑。
4. **阻塞赋值和非阻塞赋值**：在时序逻辑块中使用非阻塞赋值`<=`，确保寄存器在时钟沿更新时正确赋值。
5. **组合逻辑块**：使用`always @(*)`实现组合逻辑，将寄存器的值赋给输出端口。

注意事项：

- 确保在时序逻辑块中使用非阻塞赋值，以避免时序竞争问题。
- 在组合逻辑块中要注意赋值的顺序和正确性，防止逻辑错误。
- 检查条件语句的完整性，确保所有可能的状态变化都已考虑。

含进位输出的四位BCD计数器设计

考点：BCD计数器、时序逻辑设计、边沿触发D触发器

```
module BCDCNT (
    input inclk,
    output reg [3:0] ot,
    output reg cy
);
    // 定义4位计数器寄存器count
    reg [3:0] count;

    // 在时钟上升沿时更新状态
    always @(posedge inclk) begin
        // 如果计数器count等于9，重置count并设置进位输出cy为高
        if (count == 4'd9) begin
            count <= 4'd0;
            cy <= 1'b1;
        // 否则，计数器count加1，进位输出cy为低
        end else begin
            count <= count + 1;
            cy <= 1'b0;
        end
    end

    // 将计数器count的值赋给输出ot
    always @(*) begin
        ot = count;
    end
endmodule
```

核心语法：

1. **always块**：使用`always @(posedge inclk)`实现时序逻辑，即在时钟的上升沿触发状态更新。
2. **寄存器定义**：使用`reg [3:0] count`定义4位计数器寄存器。
3. **条件语句**：使用`if-else`结构实现状态转移条件判断，注意每个条件的顺序和逻辑。

4. **组合逻辑块**：使用 `always @(*)` 实现组合逻辑，将寄存器的值赋给输出端口。

注意事项：

- 确保在时序逻辑块中使用非阻塞赋值 `<=`，以避免时序竞争问题。
- 检查条件语句的完整性，确保所有可能的状态变化都已考虑。
- 保证进位输出cy在计数达到9时正确设置，在其他情况下为低。

三人表决器

考点：多路选择器设计、组合逻辑、条件语句

```
module VOTE3 (
    input [2:0] vote,
    output reg result
);
    // 使用组合逻辑块来实现三人表决器
    always @(*) begin
        case (vote)
            // 当输入vote为0、1、2或4时，输出result为低
            3'b000, 3'b001, 3'b010, 3'b100: result = 1'b0;
            // 其他情况，输出result为高
            default: result = 1'b1;
        endcase
    end
endmodule
```

核心语法：

1. **case语句**：使用 `case (vote)` 实现多路选择器，根据输入的不同值选择相应的输出。
2. **组合逻辑块**：使用 `always @(*)` 实现组合逻辑，确保所有输入变化时输出都会及时更新。

注意事项：

- 使用 `case` 语句时，要确保覆盖所有可能的输入值，并为未指定的情况提供 `default` 分支，防止不完全匹配的问题。

3/8 译码器

考点：译码器设计、组合逻辑、条件语句

```
module YMQ (
    input [2:0] in,
    output reg [7:0] out
);
    // 使用组合逻辑块来实现3/8译码器
    always @(*) begin
        case (in)
            // 根据输入in的不同值选择相应的输出out
            3'b000: out = 8'b00000001; // H"01"
            3'b001: out = 8'b00000010; // H"02"
            3'b010: out = 8'b00000100; // H"04"
            3'b011: out = 8'b00001000; // H"08"
```

```
            3'b100: out = 8'b00010000; // H"10"
            3'b101: out = 8'b00100000; // H"20"
            3'b110: out = 8'b01000000; // H"40"
            3'b111: out = 8'b10000000; // H"80"
            // 确保其他情况下输出为零，不过在此设计中不会出现
            default: out = 8'b00000000;
        endcase
    end
endmodule
```

核心语法：

1. **case语句**：使用`case (in)`实现译码器，根据输入的不同值选择相应的输出。
2. **组合逻辑块**：使用`always @(*)`实现组合逻辑，确保所有输入变化时输出都会及时更新。

注意事项：

- 使用`case`语句时，确保覆盖所有可能的输入值。
- 定义输出值时确保使用正确的二进制格式对应原代码中的十六进制值。
- 译码器输出应该是单热编码，因此在每个输入条件下，仅有一个输出位为高。

8位比较器设计

考点：比较器设计、组合逻辑、条件语句

```
module cmp8 (
    input [7:0] a,
    input [7:0] b,
    output reg dy,
    output reg xy,
    output reg eq
);
    // 使用组合逻辑块来实现8位比较器
    always @(*) begin
        // 初始化输出值
        dy = 1'b0;
        xy = 1'b0;
        eq = 1'b0;

        // 比较a和b的值
        if (a > b) begin
            dy = 1'b1;
        end else if (a == b) begin
            eq = 1'b1;
        end else begin
            xy = 1'b1;
        end
    end
endmodule
```

核心语法：

1. **条件语句**：使用`if-else if-else`结构实现比较逻辑，分别判断`a`是否大于、等于或小于`b`。

2. **组合逻辑块**：使用`always @(*)`实现组合逻辑，确保所有输入变化时输出都会及时更新。

注意事项：

- 初始化输出值以确保在任何比较条件下都有一个明确的输出值。
- 确保每个比较条件都是互斥的，以避免多重赋值问题。

六十进制

考点：计数器设计、时序逻辑、边沿触发D触发器

```
module 59to0 (
    input inclk,
    output reg [3:0] oth,
    output reg [3:0] otl
);
    // 定义高位和低位计数器寄存器h和l
    reg [3:0] h;
    reg [3:0] l;

    // 在时钟上升沿时更新状态
    always @(posedge inclk) begin
        // 如果h为5且l为9，重置h和l
        if (h == 4'd5 && l == 4'd9) begin
            h <= 4'd0;
            l <= 4'd0;
        // 如果l为9，重置l并使h加1
        end else if (l == 4'd9) begin
            l <= 4'd0;
            h <= h + 1;
        // 否则，保持h不变，l加1
        end else begin
            h <= h;
            l <= l + 1;
        end
    end

    // 将寄存器h和l的值赋给输出oth和otl
    always @(*) begin
        oth = h;
        otl = l;
    end
endmodule
```

核心语法：

1. **always块**：使用`always @(posedge inclk)`实现时序逻辑，即在时钟的上升沿触发状态更新。
2. **寄存器定义**：使用`reg [3:0] h`和`reg [3:0] l`定义4位寄存器，分别存储高位和低位计数值。
3. **条件语句**：使用`if-else`结构实现状态转移条件判断，注意每个条件的顺序和逻辑。
4. **组合逻辑块**：使用`always @(*)`实现组合逻辑，将寄存器的值赋给输出端口。

注意事项：

- 确保在时序逻辑块中使用非阻塞赋值`<=`，以避免时序竞争问题。

- 检查条件语句的完整性，确保所有可能的状态变化都已考虑。

BCD七段译码电路

考点：七段译码器设计、组合逻辑、条件语句

```
module BCD7 (
    input [3:0] in,
    output reg [6:0] led
);
    // 使用组合逻辑块来实现BCD七段译码器
    always @(*) begin
        // 根据输入in的不同值选择对应的七段LED输出
        case (in)
            4'b0000: led = 7'b0111111; // 0
            4'b0001: led = 7'b0001110; // 1
            4'b0010: led = 7'b1010111; // 2
            4'b0011: led = 7'b1001111; // 3
            4'b0100: led = 7'b1100110; // 4
            4'b0101: led = 7'b1101011; // 5
            4'b0110: led = 7'b1111011; // 6
            4'b0111: led = 7'b0001111; // 7
            4'b1000: led = 7'b1111111; // 8
            4'b1001: led = 7'b1101111; // 9
            4'b1010: led = 7'b1110111; // A
            4'b1011: led = 7'b1111100; // b
            4'b1100: led = 7'b0111001; // c
            4'b1101: led = 7'b1011110; // d
            4'b1110: led = 7'b1111001; // E
            4'b1111: led = 7'b1110001; // F
            default: led = 7'b0000000; // 默认情况，显示空
        endcase
    end
endmodule
```

核心语法：

- `module`：定义Verilog模块的开始。
- `input` 和 `output reg`：定义模块的输入输出端口，其中 `output reg` 表示输出是寄存器类型。
- `always @(*)`：定义组合逻辑块，对所有输入信号的变化做出响应。
- `case` 语句：用于多路选择，实现BCD到七段LED的译码。

注意事项：

- 此代码段是组合逻辑，不需要时钟信号触发，因此使用 `always @(*)` 而非 `always @(posedge clk)`。
- 译码显示部分，根据BCD码值设置LED状态，注意BCD码与LED状态的对应关系。这里使用的是共阴极接法，即LED亮对应 `1`，灭对应 `0`。
- 使用 `default` 语句来处理未定义的输入情况，关闭所有LED，避免不确定的输出。

带BCD显示译码输出的24进制计数电路

考点：计数器设计、BCD编码、译码显示、状态机实现

```
module SUBDESIGN_23to0(
    input inclk,
    output reg [13:0] led
);

// 定义计数器变量
reg [1:0] hw;
reg [3:0] hl;

// 计数器时钟信号
always @(posedge inclk) begin
    if (hw == 2 && hl == 3) begin
        hw <= 0;
        hl <= 0;
    end else if (hl == 9) begin
        hl <= 0;
        hw <= hw + 1;
    end else begin
        hl <= hl + 1;
    end
end

// 显示译码逻辑
always @(hl or hw) begin
    // 低位显示译码
    case(hl)
        0: led[6:0] = 7'b0111111; // 0
        1: led[6:0] = 7'b0000110; // 1
        2: led[6:0] = 7'b1011011; // 2
        3: led[6:0] = 7'b1001111; // 3
        4: led[6:0] = 7'b1100110; // 4
        5: led[6:0] = 7'b1101101; // 5
        6: led[6:0] = 7'b1111101; // 6
        7: led[6:0] = 7'b0000111; // 7
        8: led[6:0] = 7'b1111111; // 8
        9: led[6:0] = 7'b1101111; // 9
        default: led[6:0] = 7'b0000000; // 其他情况，关闭所有LED
    endcase

    // 高位显示译码
    case(hw)
        0: led[13:7] = 7'b011_1111; // 0 加横线能增强数字的可读性，加不加都行
        1: led[13:7] = 7'b000_0110; // 1
        2: led[13:7] = 7'b101_1011; // 2
        default: led[13:7] = 7'b000_0000; // 其他情况，关闭所有LED
    endcase
end

endmodule
```

核心语法：

- `module`：定义Verilog模块的开始。
- `input` 和 `output`：定义模块的输入输出端口。
- `reg` 和 `always`：定义寄存器变量和时序逻辑。
- `case` 语句：用于多路选择。

注意事项：

- 确保时钟信号 `inclk` 是上升沿触发。
- 译码显示部分，根据BCD码值设置LED状态，注意BCD码与LED状态的对应关系。
- Verilog中使用2位寄存器表示`hw`，而两位寄存器的值范围是0到3。所以当 `hw` 达到2时，应该重置为0。
- 当 `hl` 达到9时，应该重置为0，并使 `hw` 加1。

数电作业

5.用Verilog HDL定义以下线网、变量或者常数

考点：变量定义、参数定义、存储器定义
（1）8位寄存器变量qtmp，并初值为-2；
（2）16位整数变量xdata；
（3）内部参数S1、S2、S3和S4，取值分别为4'b0001、4'b0010、4'b0100和4'b1000；
（4）容量为1024×10位的存储器sin_rom；
（5）值为16的参数DATA_BUS_SIZE。

```
module definitions;

  // (1) 8位寄存器变量qtmp，并初值为-2
  reg [7:0] qtmp = -2;
  // qtmp的二进制表示为：11111110，对应十进制-2

  // (2) 16位整数变量xdata
  integer xdata;

  // (3) 内部参数S1、S2、S3和S4，取值分别为4'b0001、4'b0010、4'b0100和4'b1000
  localparam S1 = 4'b0001;
  localparam S2 = 4'b0010;
  localparam S3 = 4'b0100;
  localparam S4 = 4'b1000;

  // (4) 容量为1024×10位的存储器sin_rom
  reg [9:0] sin_rom [1023:0];

  // (5) 值为16的参数DATA_BUS_SIZE
  parameter DATA_BUS_SIZE = 16;

endmodule
```

核心语法：

- `reg [7:0] qtmp = -2;`：定义一个8位寄存器变量并赋初值。
- `integer xdata;`：定义一个16位整数变量。
- `localparam S1 = 4'b0001;`：定义局部参数，使用localparam避免外部修改。
- `reg [9:0] sin_rom [1023:0];`：定义一个容量为1024x10位的存储器。
- `parameter DATA_BUS_SIZE = 16;`：定义一个参数。

注意事项：

- 变量的初始值赋值要符合位宽的定义，特别是对于负数，二进制表示应为补码形式。
- 局部参数 `localparam` 是常量，不能在模块外部修改。
- 存储器定义中，注意位宽和容量的设置。
- 参数 `parameter` 是常量，可以在模块实例化时被重新定义。

6.阅读下述Verilog HDL代码，分析运算结果，并将答案填入相应的括号中。

考点：位运算、逻辑运算、连接操作、移位操作

```
reg [3:0] dat_a;
reg [5:0] dat_b;
assign dat_a=4'b1101;
assign dat_b=6'b110100;
dat_a &  dat_b=(              );
dat_a && dat_b=(              );
~dat_a=(                 );
&dat_b=(                 );
{dat_a,dat_b}=(               );
dat_a >> 1=(               );
dat_b << 1=(               );
```

```
// 运算结果分析
dat_a & dat_b;  // 位与操作
dat_a && dat_b; // 逻辑与操作
~dat_a;         // 位取反操作
&dat_b;         // 归约与操作
{dat_a, dat_b}; // 连接操作
dat_a >> 1;     // 右移操作
dat_b << 1;     // 左移操作
```

- `dat_a = 4'b1101`：二进制 `1101` 对应十进制 `13`。
- `dat_b = 6'b110100`：二进制 `110100` 对应十进制 `52`。

运算结果

- `dat_a & dat_b`：(按位与)

  ```
  1101 & 110100 = 1100  // 仅取dat_a的4位部分
  ```

  结果是 `4'b1100`，对应十进制 `12`。

- `dat_a && dat_b`：(逻辑与)

  ```
  1101（非零）&& 110100（非零）= 1
  ```

  结果是 `1`。

- `~dat_a`：(按位取反)

  ```
  ~1101 = 0010
  ```

  结果是 `4'b0010`，对应十进制 `2`。

- `&dat_b`：(归约与操作)

  ```
  110100 -> 1 & 1 & 0 & 1 & 0 & 0 = 0
  ```

  结果是 `0`。

- `{dat_a, dat_b}`：(拼接操作)

  ```
  {1101, 110100} = 1101110100
  ```

  结果是 `10'b1101110100`，对应十进制 `884`。

- `dat_a >> 1`：(右移操作)

  ```
  1101 >> 1 = 0110
  ```

  结果是 `4'b0110`，对应十进制 `6`。

- `dat_b << 1`：(左移操作)

  ```
  110100 << 1 = 1101000
  ```

  结果是 `7'b1101000`，对应十进制 `104`。

填写结果

- `dat_a & dat_b = (12)`
- `dat_a && dat_b = (1)`
- `~dat_a = (2)`
- `&dat_b = (0)`
- `{dat_a, dat_b} = (884)`
- `dat_a >> 1 = (6)`
- `dat_b << 1 = (104)`

7.描述具有同步复位功能的4位二进制计数器

考点：同步复位、4位计数器

```
module counter_4bit_sync_reset (
```

```
    input clk,        // 时钟信号
    input reset,      // 同步复位信号
    output reg [3:0] q  // 4位计数器输出
);

always @(posedge clk) begin
    if (reset)
        q <= 4'b0000;  // 同步复位，计数器清零
    else
        q <= q + 1;  // 计数器加1
end

endmodule
```

核心语法：

- `always @(posedge clk)`：定义在时钟上升沿触发的进程。
- `if (reset)`：判断复位信号。
- `q <= 4'b0000`：使用非阻塞赋值进行复位操作。
- `q <= q + 1`：非阻塞赋值计数器加1。

注意事项：

- 同步复位在时钟上升沿判断复位信号，应注意复位信号和时钟信号的同步性。
- 非阻塞赋值符号`<=`，适用于时序逻辑，确保逻辑正确执行。
- 在实现4位计数器时，要保证计数器溢出后回到0，4位计数器的范围是0到15（0000到1111）。

8.应用条件语句描述8线-3线优先编码器

考点：条件语句、优先编码器

```
module priority_encoder_8to3 (
    input [7:0] in,     // 8位输入
    output reg [2:0] out // 3位输出
);

always @(*) begin
    if (in[7])
        out = 3'b111;
    else if (in[6])
        out = 3'b110;
    else if (in[5])
        out = 3'b101;
    else if (in[4])
        out = 3'b100;
    else if (in[3])
        out = 3'b011;
    else if (in[2])
        out = 3'b010;
    else if (in[1])
        out = 3'b001;
    else if (in[0])
        out = 3'b000;
```

```
    else
        out = 3'b000; // 默认输出
end

endmodule
```

核心语法：

- `always @(*)`：组合逻辑块，敏感列表为所有输入信号。
- 条件语句 `if-else`：用于优先级判断，从高位到低位依次检查输入信号。
- 直接赋值 `=`：适用于组合逻辑，用于更新输出值。

注意事项：

- 条件语句的顺序很重要，必须从最高优先级信号开始依次检查，以确保正确的优先级编码。
- 组合逻辑使用阻塞赋值符号 `=`，不同于时序逻辑的非阻塞赋值 `<=`。
- 确保在没有输入信号时，输出有默认值（如代码中的 `else out = 3'b000;`）。

9.用Verilog HDL设计七人表决电路

考点：计数、逻辑运算、条件判断

```
module voting_circuit (
    input [6:0] votes,  // 7位输入，每位表示一个人的投票，1表示同意，0表示反对
    output reg decision // 表决结果，1表示通过，0表示否决
);

integer i;
integer count;

always @(*) begin
    count = 0;
    // 统计同意票的数量
    for (i = 0; i < 7; i = i + 1) begin
        if (votes[i] == 1)
            count = count + 1;
    end

    // 判断同意票是否过半
    if (count >= 4)
        decision = 1; // 通过
    else
        decision = 0; // 否决
end

endmodule
```

核心语法：

- `always @(*)`：组合逻辑块，敏感列表为所有输入信号。
- `for` 循环：用于遍历所有投票信号。
- `if` 条件语句：用于判断每个投票信号及最终的表决结果。

- 整数变量 `integer`：用于计数同意票数。

注意事项：

- 确保 `for` 循环正确遍历所有7位输入信号。
- 在组合逻辑中使用阻塞赋值符号 `=`，确保在同一个 `always` 块中按顺序执行赋值操作。
- 注意对 `count` 变量进行初始化，以防止不确定的值影响结果。
- 逻辑判断的阈值 `count >= 4`，需要根据表决规则设定，确保多数票同意事件通过。

10.设计序列信号检测器，能够从输入的串行数据X中检测出“1011”序列，输出检测结果Y

考点：时序逻辑、移位寄存器、条件判断

```
module sequence_shift (
    input wire X,         // 输入串行数据
    input wire clk,       // 时钟信号
    input wire rst,       // 复位信号，低电平有效
    output wire Y,        // 检测结果输出
    output reg [3:0] q    // 移位寄存器
);

always @(posedge clk or negedge rst) begin
    if (!rst)
        q <= 4'd0; // 复位时清零
    else
        q <= {q[2:0], X}; // 移位操作，将新输入的X加到寄存器中
end

assign Y = (q == 4'b1011) ? 1'b1 : 1'b0; // 判断移位寄存器中的值是否为"1011"

endmodule
```

核心语法：

- `always @(posedge clk or negedge rst)`：定义在时钟上升沿或复位信号下降沿触发的进程。
- 移位操作 `q <= {q[2:0], X};`：将当前寄存器内容左移一位，并将新输入的X赋给最低位。
- 三元运算符 `? :`：用于判断条件是否满足，如果满足则输出1，否则输出0。

注意事项：

- 复位信号为低电平有效，所以在`always`块中使用`negedge rst`。
- 使用非阻塞赋值 `<=` 进行时序逻辑操作。
- 在组合逻辑中，通过 `assign` 语句进行条件判断并赋值给输出 `Y`。
- 移位寄存器 `q` 应正确初始化，特别是在复位时清零。

综合题目：59归0电路设计与显示

要求：

- 59 归 0 电路， 提供的脉冲频率为 40 MHz， 计数脉冲计数 1 Hz
- 先将 40 MHz 分频到 500 Hz， 再分频到 1 Hz
- 需要设计显示驱动电路，七段译码管为共阴接法
- 模块化思想

考点：分频器设计、可逆计数器设计、状态机实现

1. 第一次分频模块

```
module SUBDESIGN_fp500(
    input inclk,
    output reg fpf
);

// 40MHz分频到500Hz
reg [15:0] count;
reg fp;

always @(posedge inclk) begin
    if (count == 39999) begin
        count <= 0;
        fp <= ~fp;
    end else begin
        count <= count + 1;
    end
end

assign fpf = fp;

endmodule
```

2. 第二次分频模块

```
module SUBDESIGN_fp1(
    input inclk,
    output reg fpf
);

// 500Hz分频到1Hz
reg [7:0] count;
reg fp;

always @(posedge inclk) begin
    if (count == 249) begin
        count <= 0;
        fp <= ~fp;
    end else begin
        count <= count + 1;
    end
end

assign fpf = fp;
```

```
endmodule
```

3. 计数模块

```
module 59to0 (
    input inclk,
    output reg [3:0] oth,
    output reg [3:0] otl
);
    // 定义高位和低位计数器寄存器h和l
    reg [3:0] h;
    reg [3:0] l;

    // 在时钟上升沿时更新状态
    always @(posedge inclk) begin
        // 如果h为5且l为9，重置h和l
        if (h == 4'd5 && l == 4'd9) begin
            h <= 4'd0;
            l <= 4'd0;
        // 如果l为9，重置l并使h加1
        end else if (l == 4'd9) begin
            l <= 4'd0;
            h <= h + 1;
        // 否则，保持h不变，l加1
        end else begin
            h <= h;
            l <= l + 1;
        end
    end

    // 将寄存器h和l的值赋给输出oth和otl
    always @(*) begin
        oth = h;
        otl = l;
    end
endmodule
```

4. 译码显示模块

```
module display(
    input [3:0] a, b,
    output reg [6:0] outa, outb
);

// 七段译码显示，共阴接法
always @(*) begin
    // 输入a的译码
    case(a)
        4'd0: outa = 7'b1111110; // 显示 "0"
        4'd1: outa = 7'b0110000; // 显示 "1"
        4'd2: outa = 7'b1101101; // 显示 "2"
        4'd3: outa = 7'b1111001; // 显示 "3"
        4'd4: outa = 7'b0110011; // 显示 "4"
        4'd5: outa = 7'b1011010; // 显示 "5"
```

```
            4'd6: outa = 7'b1011111; // 显示 "6"
            4'd7: outa = 7'b1110000; // 显示 "7"
            4'd8: outa = 7'b1111111; // 显示 "8"
            4'd9: outa = 7'b1111011; // 显示 "9"
            default: outa = 7'b0000000; // 其他值，关闭所有段
        endcase

        // 输入b的译码，与输入a相同
        case(b)
            4'd0: outb = 7'b1111110;
            4'd1: outb = 7'b0110000;
            4'd2: outb = 7'b1101101;
            4'd3: outb = 7'b1111001;
            4'd4: outb = 7'b0110011;
            4'd5: outb = 7'b1011010;
            4'd6: outb = 7'b1011111;
            4'd7: outb = 7'b1110000;
            4'd8: outb = 7'b1111111;
            4'd9: outb = 7'b1111011;
            default: outb = 7'b0000000;
        endcase
    end

    endmodule
```

核心语法：

- 使用 `always @(*)` 块来实现组合逻辑，这意味着每当输入 `a` 或 `b` 变化时，`outa` 和 `outb` 将立即更新。
- `case` 语句用于将4位二进制数（BCD）转换为七段显示器的段控制信号。每个 `case` 项都对应一个BCD数字，并输出相应的段控制信号，以点亮七段显示器的正确段。

注意事项：

- 分频模块需要正确计算计数器的值，以实现准确的分频。
- 计数模块需要根据分频模块的输出进行计数，当计数到59时归0。
- 七段显示器使用的是共阴接法，意味着当输出为 `0` 时，对应的LED段是关闭的；为 `1` 时，LED段是亮的。
- 默认情况下，如果输入不是有效的BCD数字（0-9），所有段都关闭，显示为熄灭状态。
- 模块化设计允许将复杂系统分解为更小、更易于管理的部分。

真题演练1：INK控制的模60可逆计数器

题目来源：信控学院2020年《电路与数字系统》秋季期末考试

八、用模块化、HDL语言设计一个由INK控制的模为60的可逆计数器。（12分）

要求：

1. INK = 1 加计数， INK = 0 减计数；
2. 外部脉冲 Clock=1000Hz；
3. 模为60的计数器只能对 1Hz 信号进行计数。

考点：模块化设计、可逆计数器、时钟分频

```
// 时钟分频模块
module clk_divider (
    input wire clk,           // 外部时钟信号
    input wire rst_n,         // 复位信号，低电平有效
    output reg clk_1hz        // 分频后的1Hz时钟信号
);
    reg [9:0] clk_div; // 需要一个10位的计数器来分频

    // 时钟分频逻辑
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            clk_div <= 10'd0;
            clk_1hz <= 1'b0;
        end else if (clk_div == 10'd499) begin
            clk_div <= 10'd0;
            clk_1hz <= ~clk_1hz;
        end else begin
            clk_div <= clk_div + 1;
        end
    end
endmodule
```

```
// 可逆计数器模块
module reversible_counter (
    input wire clk_1hz,       // 1Hz时钟信号
    input wire rst_n,         // 复位信号，低电平有效
    input wire ink,           // 控制信号，1为加计数，0为减计数
    output reg [5:0] count    // 6位计数器输出
);

    // 计数器逻辑
    always @(posedge clk_1hz or negedge rst_n) begin
        if (!rst_n) begin
            count <= 6'd0;
        end else if (ink) begin
            if (count == 6'd59) begin
                count <= 6'd0;
            end else begin
                count <= count + 1;
            end
        end else begin
            if (count == 6'd0) begin
                count <= 6'd59;
            end else begin
                count <= count - 1;
            end
        end
    end
endmodule
```

```
// 顶层模块
```

```
module top_module (
    input wire clk,            // 外部时钟信号
    input wire rst_n,          // 复位信号，低电平有效
    input wire ink,            // 控制信号，1为加计数，0为减计数
    output wire [5:0] count    // 6位计数器输出
);
    wire clk_1hz;

    // 实例化时钟分频模块
    clk_divider u_clk_divider (
        .clk(clk),
        .rst_n(rst_n),
        .clk_1hz(clk_1hz)
    );

    // 实例化可逆计数器模块
    reversible_counter u_reversible_counter (
        .clk_1hz(clk_1hz),
        .rst_n(rst_n),
        .ink(ink),
        .count(count)
    );
endmodule
```

核心语法：

- `module` 和 `endmodule`：使用 定义各个功能模块，并在顶层模块中实例化它们。
- `input` 和 `output reg`：定义模块的输入输出端口，其中 `output reg` 表示输出是寄存器类型。
- `always @(posedge signal)`：定义时序逻辑块，对特定信号的上升沿做出响应。
- `if-else` 语句：用于条件判断。

注意事项：

- 分频器部分，使用 `count` 变量来实现1000Hz到1Hz的分频，当计数到499时，输出 `fp` 取反，实现1Hz信号。
- 计数器部分，使用 `fp` 作为时钟信号，根据 `INK` 的值决定是加计数还是减计数。
- 当计数器达到59时，加计数应回绕到0；当计数器为0时，减计数应设置为59。
- 确保复位信号为低电平有效，并在所有时序逻辑块中正确使用。

真题演练2：56进制计数器的层次化模块化设计

题目来源：计算机学院2023年《电路与数字系统》秋季期末考试

请你使用层次化、模块化的思想，设计实现56进制计数器

1. 输入 INK 为 1000Hz 的时钟
2. 56 进制计数器只接受占空比为 50%的秒脉冲
3. 设计一个顶层模块，将各个模块连接起来

考点：层次化设计、模块化设计、时钟分频、计数器设计、信号同步

1. 时钟分频模块

```
module fp(
    input INK,          // 1000Hz的时钟信号
    output reg CLK_1Hz   // 1Hz的输出时钟
);

// 将1000Hz的时钟信号分频至1Hz
always @(posedge INK) begin
    reg [15:0] counter = 0;
    counter <= counter + 1;
    if(counter == 499) begin  // 每1000个周期分频一次得到1Hz
        counter <= 0;
        CLK_1Hz <= ~CLK_1Hz;
    end
end

endmodule
```

2. 56进制计数器模块

```
module base_56_counter(
    input CLK_1Hz,     // 1Hz的时钟信号
    output reg [5:0] count // 56进制计数器的输出
);

// 56进制计数器逻辑
always @(posedge CLK_1Hz) begin
    if(count == 55) begin
        count <= 0; // 56进制计数到55后归零
    end else begin
        count <= count + 1;
    end
end

endmodule
```

3. 占空比控制模块（如果需要）

```
module duty_cycle_controller(
    input CLK_1Hz,     // 1Hz的时钟信号
    output reg PULSE   // 50%占空比的秒脉冲
);

// 控制输出脉冲的占空比
always @(posedge CLK_1Hz) begin
    reg [1:0] state = 0;
    always @(state) begin
        case(state)
            0: PULSE = 1'b1; // 高电平
            1: PULSE = 1'b0; // 低电平
```

```
        endcase
    end
    state <= state + 1;
end

endmodule
```

4. 顶层模块

```
module top_module(
    input INK,          // 1000Hz的原始时钟信号
    output [5:0] COUNT  // 56进制计数器的最终输出
);

// 模块间的连线
wire CLK_1Hz;
wire PULSE; // 如果需要50%占空比的脉冲

// 实例化模块
fp divider(.INK(INK), .CLK_1Hz(CLK_1Hz));
base_56_counter counter(.CLK_1Hz(CLK_1Hz), .count(COUNT));

// 50%占空比的脉冲
duty_cycle_controller controller(.CLK_1Hz(CLK_1Hz), .PULSE(PULSE));

endmodule
```

核心语法：

- `module` 和 `endmodule`：定义Verilog模块的开始和结束。
- `input` 和 `output`：定义模块的输入输出端口。
- `always @(posedge signal)`：响应特定信号上升沿的时序逻辑块。
- `reg`：定义寄存器，用于存储状态。
- `wire`：定义连线，用于模块间的信号传递。

注意事项：

- 确保时钟分频模块正确地将1000Hz的时钟信号分频至1Hz。
- 56进制计数器模块应正确计数并在达到55时归零。
- 考虑占空比需要设计额外的占空比控制模块。
- 顶层模块用于连接所有子模块，并提供外部接口。
- 在层次化和模块化设计中，注意信号命名的唯一性和清晰性，避免命名冲突。
- 使用模块实例化来建立模块间的连接，并通过端口映射进行连接。

真题演练3：优先编码器设计

题目来源：计算机学院2021年《电路与数字系统》秋季期末考试

七、用 HDL 设计优先编码器
1.输入端用 H、M、L 表示，高电平有效。
2.要求输出端个数最少
3.输入端的优先级 H 最高、M 其次、L 最低

考点：优先编码器、信号编码、HDL设计

```
module priority_encoder(
    input H,  // 高优先级输入
    input M,  // 中优先级输入
    input L,  // 低优先级输入
    output reg [1:0] code  // 最少2位输出，足以表示3个输入的优先编码
);

// 优先编码逻辑
always @(H, M, L) begin
    if (H) begin
        code = 2'b11;  // H优先级最高，编码为11
    end else if (M) begin
        code = 2'b10;  // M次之，编码为10
    end else if (L) begin
        code = 2'b01;  // L最低，编码为01
    end else begin
        code = 2'b00;  // 所有输入都未激活，编码为00
    end
end

endmodule
```

核心语法：

- `module` 和 `endmodule`：定义Verilog模块的开始和结束。
- `input` 和 `output reg`：定义模块的输入输出端口，其中 `output reg` 表示输出是寄存器类型。
- `always @(signal)`：定义过程块，该块将在输入信号H、M、L变化时触发。
- `if-else if-else` 语句：用于条件判断，实现优先级逻辑。

注意事项：

- 优先编码器的输出应根据输入信号的优先级来确定，最高优先级的输入信号将决定输出编码。
- 输出编码需要能够唯一表示每个激活的输入信号，本例中使用2位输出足以表示3个输入信号的优先编码。
- 当多个输入信号同时激活时，只有最高优先级的信号会影响输出。
- 当没有任何输入信号激活时，输出编码应该有一个明确的表示，本例中为`2'b00`。
- 确保在所有可能的输入条件下，优先编码器都能正确地输出相应的编码。

真题演练4：60进制计数器的模块化设计

题目来源：计算机学院2021年《电路与数字系统》秋季期末考试

八、请用模块化思想设计一个 60 进制计数器。

要求如下：

1. ink 为 1khz 信号输入端
2. 60 进制计数器只对秒信号进行计数
3. 计数器对译码器有 N 位的输出，译码器连接的显示器显示个位与十位。请你自行 确定计数器的输出位数，无需设计译码器。

考点：模块化设计、计数器设计、信号同步、时序逻辑

1. 时钟分频模块

```
module fp(
    input Clk,           // 1kHz信号输入端
    output reg SecTick   // 1Hz秒信号
);

// 利用计数器将1kHz信号分频到1Hz
reg [9:0] counter;
always @(posedge Clk) begin
    counter <= counter + 1;
    if (counter == 500) begin  // 每1000个周期分频一次得到1Hz
        counter <= 0;
        SecTick <= ~SecTick;  // 取反实现50%占空比的秒信号
    end
end

endmodule
```

2. 60进制计数器模块

```
module base_60_counter(
    input SecTick,       // 1Hz秒信号
    output reg [N-1:0] CountOut  // N位输出，根据需要自行确定N的值
);

// 60进制计数器逻辑
parameter N = 6;  // 假设我们选择6位足以表示60进制计数
always @(posedge SecTick) begin
    if (CountOut == (60 - 1)) begin
        CountOut <= 0;
    end else begin
        CountOut <= CountOut + 1;
    end
end

endmodule
```

3. 顶层模块

```
module top_module_60_counter(
    input INK,          // 1kHz信号输入端
    output reg [N-1:0] CountDisp  // 显示器显示的个位与十位
);

// 模块间的连线
wire SecTick;
// 根据需要自行确定N的值
parameter N = 4;  // 4位足以显示00到59

// 实例化模块
fp fp(.Clk(INK), .SecTick(SecTick));
base_60_counter counter(.SecTick(SecTick), .CountOut(CountDisp));

endmodule
```

核心语法：

- `module` 和 `endmodule`：定义Verilog模块的开始和结束。
- `input`、`output reg` 和 `wire`：定义模块的输入输出端口和内部连线。
- `always @(posedge signal)`：响应特定信号上升沿的时序逻辑块。
- `parameter`：定义模块的参数，用于指定计数器的位数。

注意事项：

- 时钟分频模块需要正确地将1kHz的时钟信号分频至1Hz的秒信号。
- 60进制计数器模块应正确计数并在达到59时归零，因为60用二进制表示为111101。
- 计数器的输出位数N应根据需要自行确定，但至少需要6位来表示0到59的计数。
- 顶层模块将所有子模块连接起来，并提供外部接口INK和CountDisp。
- 译码器和显示器设计未包含在内，因为题目要求只确定计数器的输出位数。
- 确保设计好顶层模块，将所有模块之间的连线连好，实现整个系统的功能。

真题演练5：2/4译码器 分频器 填空题

要求程序实现 2/4 译码器功能
（1）译码器输出端低电平有效（2）译码器输入端为 code[1,0],输出 out[3..0]

```
SUBDESIGN A
        code[1..0]:INPUT
        out[3..0]:OUTPUT
BEGIN
        LASE code[]is
WHEN      0－－>out[]=B“ 1 ”;
        1－－>out[]=B“ 10 ”;
        2－－>out[]=B“ 100 ”;
        3－－>out[]=B“ 1000 ”;
        end case;
END;
```

要求程序实现分频器功能
（1） 分频输入端为 clk,输出端 fp（2）输出信号的频率是输入信号频率的十分之一。

```
SUBDESIGN B
(       inclk: input;
        fp:     output;

)
Variable
a[2..0]: dff ;
fp: dff;
begin
a[].clk=inclk;
fp.clk=inclk;
if a[]==4 then
        a[ ]=0;
        fp=!fp;
        else
        a[]=a[]+1;
        fp=fp;
        endif;
fpf =fp;
end;
```

真题演练6：实现16位二进制全减器功能

七、编写程序，实现相关功能（每题 5 分，共 10 分）

1、要求程序实现两组 16 位二进制全减器功能，具体设计要求：

（1）利用数组概念定义减数、被减数和全减差；文件名用 Qjianqi 表示。

（2）定义一个使能输入端 EN，确保实现全减器，输入使能端 EN 高电平有效；

（3）定义一个全减器借位输出端 Co，来自低位的借位输入端 Ci。

考点：全减器、使能输入、借位输出、数组定义

```
module Qjianqi (
    input [15:0] A,       // 被减数
    input [15:0] B,       // 减数
    input Ci,             // 低位借位输入
    input EN,             // 使能信号
    output [15:0] Diff,   // 差
    output Co             // 借位输出
);
    reg [15:0] Diff_reg;
    reg Co_reg;

    // 定义全减器逻辑
    always @(*) begin
        if (EN) begin
            {Co_reg, Diff_reg} = A - B - Ci;  // 执行减法操作，包含借位
        end else begin
            {Co_reg, Diff_reg} = 16'b0;       // 使能信号无效时输出0
        end
    end

    assign Diff = Diff_reg;
    assign Co = Co_reg;

endmodule
```

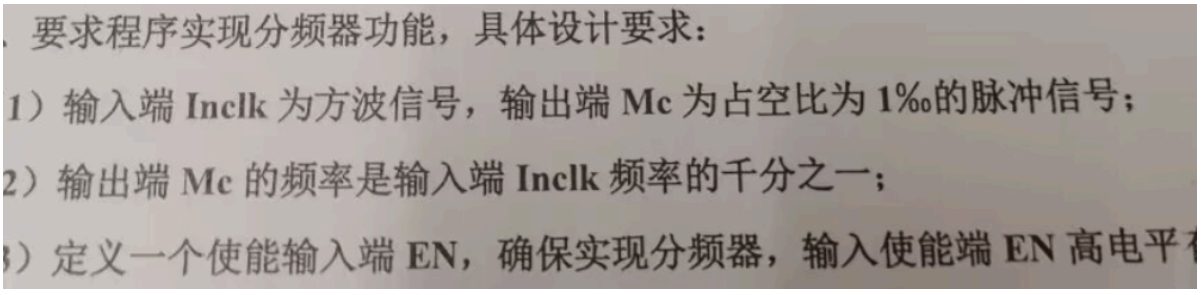
核心语法：

1. `input [15:0] A` / `input [15:0] B`：使用数组定义16位的减数和被减数
2. `input EN`：定义使能输入
3. `always @(*)`：定义组合逻辑块，用于描述全减器逻辑
4. `if (EN)`：条件判断语句，用于判断使能信号 `EN` 的高电平有效
5. `assign`：连续赋值语句，用于输出差 `Diff` 和借位 `Co`

注意事项：

1. 使能信号 `EN` 需要正确处理，确保在 `EN` 高电平时进行减法操作。
2. 借位逻辑的实现要考虑到低位借位输入 `Ci` 的影响。
3. `always` 块内使用阻塞赋值 `=` 而不是非阻塞赋值 `<=`。
4. 确保所有的信号赋值在使能信号无效时归零，避免不必要的输出干扰。

真题演练7：实现分频器功能



考点：分频器、使能输入、占空比控制

```
module Divider (
    input Inclk,     // 输入方波信号
    input EN,        // 使能信号，高电平有效
    output reg Mc    // 输出占空比为1‰的脉冲信号
);
    reg [9:0] counter; // 10位计数器，用于分频

    always @(posedge Inclk) begin
        if (EN) begin
            if (counter < 999) begin
                counter <= counter + 1;
            end else begin
                counter <= 0;
            end
        end else begin
            counter <= 0;
        end
    end

    always @(posedge Inclk) begin
        if (EN) begin
            if (counter == 0) begin
                Mc <= 1;
            end else begin
                Mc <= 0;
            end
        end else begin
            Mc <= 0;
        end
    end
endmodule
```

核心语法：

1. `input Inclk`：定义输入方波信号
2. `input EN`：定义使能输入
3. `output reg Mc`：定义输出脉冲信号
4. `reg [9:0] counter`：使用10位寄存器定义计数器
5. `always @(posedge Inclk)`：在输入时钟上升沿触发的组合逻辑块
6. `if (EN)`：条件判断语句，用于判断使能信号 `EN` 的高电平有效

注意事项：

1. **计数器复位与计数**：确保计数器 `counter` 在使能信号 `EN` 高电平时正常工作，并在计数到1000时复位。
2. **输出脉冲的产生**：输出信号 `Mc` 在每次计数器 `counter` 到达0时产生一个脉冲，确保占空比为1‰。即在每1000个输入时钟周期中，`Mc` 仅有1个周期为高电平，其余999个周期为低电平。
3. **使能信号的处理**：使能信号无效时，计数器和输出信号 `Mc` 都应复位为0，避免不必要的输出。
4. **同步运行**：两个 `always` 块使用同一个时钟信号 `Inclk`，确保同步运行。

5. **占空比解释**：占空比是指信号在一个周期内高电平时间与总时间的比值。在此设计中，占空比为1‰意味着在每1000个输入时钟周期中，输出信号 `Mc` 仅有1个周期为高电平，其他999个周期为低电平。