# 素数循环显示说明文档

**组员：周睿妍、徐昕妍、杨茗羽**

## 1.设计要求：

设计一个素数判断并显示电路。要求：检测 2~999,999 之间的素数，每一秒钟在数码管上显示一个，并可通过按键暂停。

## 2.设计原理及过程：

电路结构如图 1 所示，由 top 模块、按键消抖模块、显示使能模块、素数判断模块、二进制转 BCD 模块和数码管显示驱动模块构成。

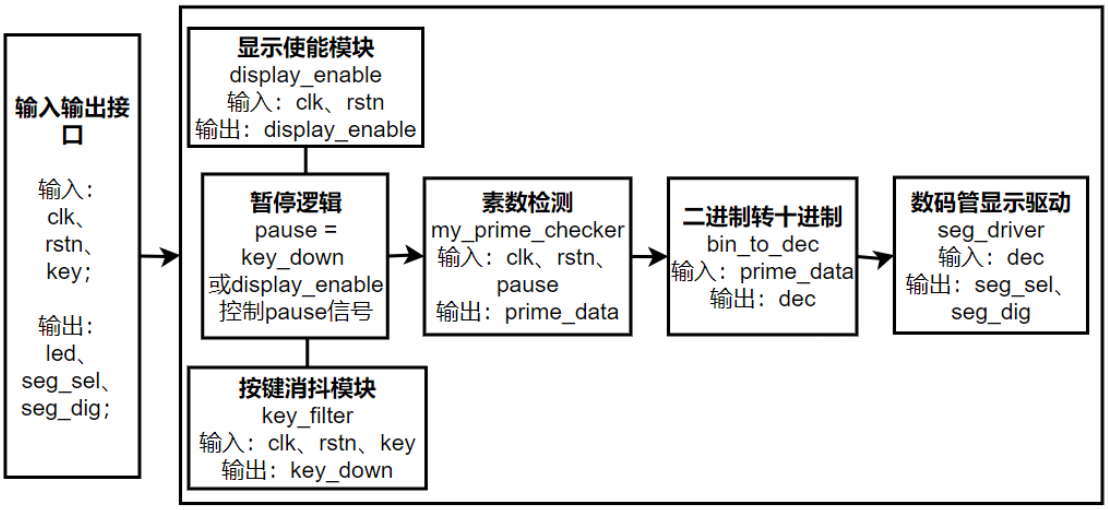


*图 1　素数判断显示电路结构框图*

• 按键去抖模块 (key_filter)：对原始按键信号进行去抖，输出一个稳定的按键按下信号，避免误动作。

• 显示使能模块 (display_enable)：生成一个周期性信号以控制显示，使系统能够在暂停状态下正常运行。

• 暂停控制逻辑：根据按键信号和显示使能信号生成系统的暂停控制信号，决定系统是否继续运行或暂停素数检测。

• 素数检测模块 (my_prime_checker)：计算从 2 开始的所有素数，输出当前检测到的素数，支持在暂停状态下停止计算。

• 二进制转十进制模块 (bin_to_dec)：将输入的二进制素数转换为十进制 BCD 编码，供数码管显示使用。

• 七段数码管解码模块 (led7seg_decode)：将十进制数字（BCD）转换为数码管的段码，以驱动数码管显示具体的数字。
• 数码管驱动模块 (seg_driver)：动态扫描多个数码管，控制显示的数字和状态，实现快速切换以产生同时点亮的效果。

## 2.1 按键消抖模块

按键消抖模块主要用于处理来自机械按键的抖动问题。由于机械按键在开关操作时，触点可能会因弹性作用而产生短暂的多次开启和关闭（即“抖动”），这会导致数字电路对按键信号的误判。为了避免产生误操作，必须实现一个有效的消抖机制。模块的设计通过一定的延时和状态采样来确保输入信号的稳定性。

### 模块设计：

对输入信号进行多次采样，直到确保输入信号稳定为止。设定一个延时（例如 20ms）来判断按键的状态。

• 输入采样：通过状态寄存器存储按键的当前和前几次状态值，生成延时信号。

```
always @(posedge clk or negedge rst_n)begin
    if(!rst_n)begin
        key_r0 <= -1;
        key_r1 <= -1;
        key_r2 <= -1;
    end
    else begin
        key_r0 <= key_in;
        key_r1 <= key_r0;
        key_r2 <= key_r1;
    end
end
```

• 边沿检测：利用位运算来检测按键的下降沿和上升沿，以判断是否有按键操作。

```
assign n_edge = ~key_r1 & key_r2 ? 1'b1 : 1'b0; //  下降沿检测
assign p_edge = key_r1 & ~key_r2 ? 1'b1 : 1'b0; //  上升沿检测
```

• 状态标志控制：在检测到下降沿时，设置 filter_flag 为高状态，表示准备进行消抖计数。

```
always @(posedge clk or negedge rst_n)begin
    if(!rst_n)begin
        filter_flag <= 1'b0;
    end
    else if(n_edge)begin
```

```
            filter_flag <= 1'b1; //  下降沿触发
        end
        else if(end_cnt)begin
            filter_flag <= 1'b0; //  计数结束，设置为低
        end
    end
```

• 计数器实现：在 filter_flag 处于高状态时启动计数器，如果 end_cnt 信号触发（计数到设定值），则认为按键状态稳定。

```
always @(posedge clk or negedge rst_n)begin
    if(!rst_n)begin
        cnt <= 0;
    end
    else if(add_cnt)begin
        if(end_cnt || p_edge)begin
            cnt <= 0; //  出现上升沿或计数结束，重置计数器
        end
        else begin
            cnt <= cnt + 1; //  正常情况下计数
        end
    end
end
```

• 输出稳定的按键信号：在 end_cnt 触发时，更新 key_down 输出为稳定状态，即按键的最终状态。

```
always @(posedge clk or negedge rst_n)begin
    if(!rst_n)begin
        key_down <= 0;
    end
    else if(end_cnt)begin
        key_down <= ~key_r2; //  取反输出稳定的按键信号
    end
end
```

## 2.2 使能模块设计

使能模块(display_enable)，用于控制输出信号 display_enable 的翻转，以达到在一定频率下使能显示器或其他设备的效果。

### 模块设计：

模块内部采用了一个计数器 display_cnt 来跟踪时钟脉冲的数量。由于输入的时钟频率为 50MHz，因此每秒钟有 50,000,000 个时钟脉冲。设计中设置了一个阈值为 32'd49999，计数器每达到 49999 时便翻转一次 display_enable 信号，从而实现每 1ms 翻转一次。

```
always @(posedge clk or negedge rstn) begin
```

```
    if (!rstn) begin
        display_cnt <= 0;
        display_enable <= 0;
    end else if (display_cnt == 32'd49999) begin //  每 50,000 个时钟周期翻转一次
        display_cnt <= 0; //  重新计数
        display_enable <= 1 ;
    end else begin
        display_cnt <= display_cnt + 1; //  增加计数
        display_enable <= 0;
    end
end
```

## 2.3 素数判断模块设计

素数检测模块的设计目的是通过逐步筛查的方式计算从 2 开始的素数。由于素数的特性以及对计算资源的考虑，该模块实现了带状态机的逐步运算，确保在高频时钟环境下的准确性和可靠性。同时，模块还实现了暂停功能，允许在任何时候中断计算。

**模块设计：**

素数检测模块的输入包括时钟信号（sclk）、复位信号（rst_n）和暂停信号（pause）。其中时钟信号用于驱动模块的状态机，复位信号用于初始化模块状态，暂停信号允许用户在任意时刻中断计算。模块通过逐步筛查的方式判断从 2 到指定最大值（如 999999）之间的素数。输出结果为当前检测到的素数（prime_data）。

该模块使用状态机设计，主要包含多个状态（如 idle、div_init、div_shift、div_sub 和 div_end），以控制数的检测流程。在每轮检测结束后，模块会标记当前数是否为素数，并将其输出。

• 状态更新逻辑：更新状态机的当前状态，依赖于时钟的上升沿或复位信号。

```
always@(posedge clk or negedge rst_n) begin
    if (!rst_n)
        pstate <= idle;
    else
        pstate <= nstate;
end
```

• 状态判断与状态转移：每个状态被明确情况控制，决定进入下一个状态的条件和准备的操作。例如在 div_init 中，准备除数和被除数并更新状态至 div_sub。

```
always @(*) begin
    case (pstate)
        idle: begin
            temp_B = 20'd0;
            temp_A = 20'd0;
            mod = 10'd0;
            D = 20'd0;
            done = 1'b0;
            if (cnt_data <= 20'd3)
                nstate = div_end; // 若当前值小于等于 3，直接进入结束状态
            else if (start == 1'b0)
                nstate = div_init; // 若准备开始新检测，则初始化
            else
                nstate = idle; // 保持当前状态
        end
        div_init: begin
            temp_B = {cnt_cnt_data, 10'b0}; // 初始化除数和被除数
            temp_A = cnt_data;
            i_cnt = 5'd11; // 设置循环计数
            temp_D = 20'd0;
            nstate = div_sub; // 进入除法运算状态
        end
        ...
    endcase
end
```

• 除法与判断逻辑：在 div_sub 状态中，执行核心的除法逻辑，判断被除数是否大于或等于除数，如果条件满足，则减去除数并更新结果。这种方法通过移位和减法模拟除法运算。

```
div_sub: begin
    if (i_cnt > 5'd0) begin
        i_cnt = i_cnt - 1'b1; // 计数递减
```

```
            nstate = div_shift; //  准备进入右移状态
            if (temp_A >= temp_B) begin
                temp_A = temp_A - temp_B; //  若被除数大于或等于除数，执行减法
                temp_D[0] = 1'b1; //  该位为 1，表示对应的二进制位
            end
            else begin
                temp_D[0] = 1'b0; //  该位为 0
            end
        end
        else begin
            i_cnt = 5'd11; //  重置计数
            nstate = div_end; //  进入结束状态
        end
    end
```

• 结束状态和输出：在 div_end 状态中将结果保存和标记计算完成，同时根据情况输出当前是否为素数。

```
    div_end: begin
        D = temp_D; //  将结果保存
        done = 1'b1; //  标记为完成
        nstate = idle; //  回到初始化状态
        if (cnt_data <= 20'd3)
            mod = 10'd1; //  对于特殊情况，设定模值
        else
            mod = temp_A[9:0]; //  按位从被除数中提取
    end
```

• 主循环和计数的实现

使用定时器和状态机控制素数的检测范围，确保每次运行都能准确判断当前的数，同时根据需要调整状态以便更新被检测的数值。

```
    always @(posedge clk or negedge rst_n) begin
        if (rst_n == 1'b0)
            cnt_data <= 20'd2; //  重置计数
        else
            if (cnt_data == 20'd999999 && (cnt_cnt_data == 10'd1000))
```

```
            cnt_data <= 20'd2; //  达到上限后重置
        else if (cnt_data == 20'd2)
            cnt_data <= cnt_data + 1'b1; //  从 2 开始
        else ...
end
```

这段逻辑确保了从 2 开始逐步检索直至最大值 999999，每完成一次检测都将计数器递增。

• 标志信号 flag 的使用：对素数的最终判断，使得模块能在每次计算后更新当前的素数状态。

```
always @* begin
    if (mod == 20'd0 && done == 1'b1)
        flag = 1'b1; //  若模为 0 且已计算完成，则该数不是素数
    else
        flag = 1'b0;
end
```

## 2.4 二进制转 BCD 模块设计

进制转换模块( bin_to_dec) ，模块接收 20 位二进制输入信号 bin，通过内部 44 位寄存器 shift_reg 存储及中间计算，利用整数变量 i, j 控制循环，最终输出 24 位 BCD 编码的十进制信号 dec，能表示最多 6 位十进制数。

**模块设计：**

• 初始化移位寄存器: 将输入的 20 位二进制数放入移位寄存器的低位，并初始化高 24 位为 0，以便存储 BCD 结果。

• BD 转换循环: 使用双重循环进行 20 次迭代，每次迭代对当前的移位寄存器进行处理：

• 修正步骤: 从高位向低位检查每 4 位 BCD 组是否大于或等于 5。若是，则需将其加上 3，以进行 BCD 的修正。

• 位移操作: 在修正完成后，将整个寄存器左移一位。

• 提取 BCD 结果: 在完成 20 次迭代后，从移位寄存器中提取高 24 位作为 BCD 输出。

**相关代码如下：**

```
always @(*) begin
    shift_reg = {24'b0, bin};   // 高 24 位为 BCD 结果，低 20 位为二进制输入
    for (i = 0; i < 20; i = i + 1) begin
        for (j = 43; j >= 20; j = j - 4) begin
            if (shift_reg[j-:4] >= 5) begin
                shift_reg[j-:4] = shift_reg[j-:4] + 3;   // 修正加 3
            end
        end
        shift_reg = shift_reg << 1;// 左移一位
    end
    dec = shift_reg[43:20];
end
```

## 2.5 数码管显示驱动模块设计

2.5.1 LED 7 段显示解码

该模块将 4 位二进制数转换为用于 7 段显示器的 8 位输入信号。当 valid 信号为高时，使用 case 语句将输入的数字 digit 映射到相应的 7 段数码管显示编码。

2.5.2 驱动模块(seg_driver)

该模块用于驱动多个 7 段显示器，通过时钟信号控制哪一个 7 段显示器被激活。模块实际上是一个多路复用器，根据时钟信号来选择激活的 7 段显示器：

计数器: 使用 15 位计数器 cnt 来进行计数。每当 cnt 达到最大值（例如 32767）时，更新选择的显示器。选择闪烁频率取决于时钟频率。

选择逻辑: 通过 sel 寄存器来记录当前被选择的显示器。当计数器重新开始时， sel 会在 0 到 NPorts-1 之间循环。

有效性输出: valid_o 信号由当前选择的显示器控制，表示哪个显示器是有效的。

输出信号: 最后，seg_o 将当前选择的 7 段显示器的输入信号押出，注意在此之前进行必要的操作以确保显示内容的正确性。

两个模块相结合实现了一个完整的 7 段显示驱动和解码系统。led7seg_decode 模块负责将输入的数字转换成适合 7 段显示器显示的格式，而 seg_driver 模块负责多个 7 段显示器的选择与输出。

## 2.6 顶层模块设计

### 顶层模块代码：

```
module top(
    input clk,
```

```
    input rstn,
    output reg [3:0] led,
    input [3:0] key,
    output [5:0] seg_sel,
    output [7:0] seg_dig
);
    reg pause;
    reg calculate_enable;
    reg [19:0] delay_prime;
    reg [19:0] display_prime;
    wire [19:0] current_prime;
    wire display_enable;
    always @(posedge clk or negedge rstn) begin
        if (!rstn) begin
            delay_prime <= 2;
            display_prime <= 2;
        end else begin
            delay_prime <= current_prime;
            if (display_enable) display_prime <= current_prime;
        end
    end

    assign prime_change = (delay_prime != current_prime) ? 1 : 0;

    always @(posedge clk or negedge rstn) begin
        if (!rstn) begin
            calculate_enable <= 0;
        end else begin
            if(display_enable) calculate_enable <= 1;
            else if(prime_change)    calculate_enable <= 0;
        end
    end

    wire [3:0] key_press;
    wire [3:0] key_down;

    key_filter #(
        .KEY_W(4),
        .DELAY_TIME(1000000)
    ) key_filter_instance (
        .clk(clk),
        .rst_n(rstn),
        .key_in(key),
        .key_down(key_down)
    );

    display_enable display_enable_instance(
        .clk(clk),
        .rstn(rstn),
```

```
        .display_enable(display_enable)
);

always @(posedge clk or negedge rstn) begin
        if (!rstn) begin
                pause <= 1'b1;
        end else begin
                pause <= key_down[0] | calculate_enable;
        end
end

my_prime_checker checker_instance (
        .sclk(clk),
        .rst_n(rstn),
        .pause(pause),
        .prime_data(current_prime)
);

wire [23:0] dec_value;
bin_to_dec b2d_instance (
        .bin(display_prime),
        .dec(dec_value)
);

reg [3:0] digits [5:0];
integer i;

always @(posedge clk or negedge rstn) begin
        if (!rstn) begin
                for (i = 0; i < 6; i = i + 1) begin
                        digits[i] <= 0;
                end
        end else begin
                digits[5] <= dec_value[23:20];
                digits[4] <= dec_value[19:16];
                digits[3] <= dec_value[15:12];
                digits[2] <= dec_value[11:8];
                digits[1] <= dec_value[7:4];
                digits[0] <= dec_value[3:0];
        end
end

wire [7:0] seg_out [5:0];
generate
        genvar j;
        for (j = 0; j < 6; j = j + 1) begin : seg_decoder
                led7seg_decode d(.digit(digits[j]), .valid(1'b1), .seg(seg_out[j]));
        end
endgenerate
```

```
    reg [47:0] prime_digits;
    always @(*) begin
        for (i = 0; i < 6; i = i + 1) begin
            prime_digits[i*8 +: 8] = seg_out[i];
        end
    end

    seg_driver #(6) driver(clk, rstn, 6'b111111, prime_digits, seg_sel, seg_dig);

endmodule
```

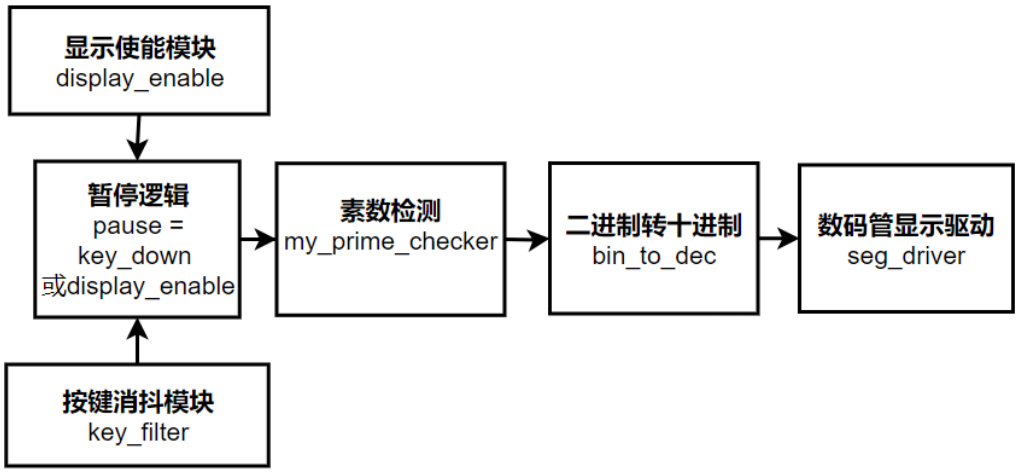


*图 2　顶层模块图*

**模块设计：**

top 顶层模块实现了一个集成的系统，结合了按键处理、质数计算以及 7 段显示驱动的功能。该模块首先接收时钟信号和复位信号，并通过 4 个按键输入允许用户与系统交互。其主要功能是在接收到有效的按键输入的情况下，计算质数并将结果以十进制形式显示在多个 7 段显示器上。

模块中通过去抖动机制处理按键输入，确保按键反馈的稳定性。此外，设计了一个质数检查模块，能够在用户按键信号激活时持续计算质数，并输出结果。在计算完成后，转换模块将质数的二进制表示转换为 BCD 格式，以便在 7 段显示器上显示。

为了驱动 7 段显示器，该模块将处理后的数字通过解码模块进行转换，确保每个显示器展示正确的数字。最后，通过显示驱动逻辑，控制具体哪个显示器被激活，从而实现数字的动态显示。

## 3.调试结果

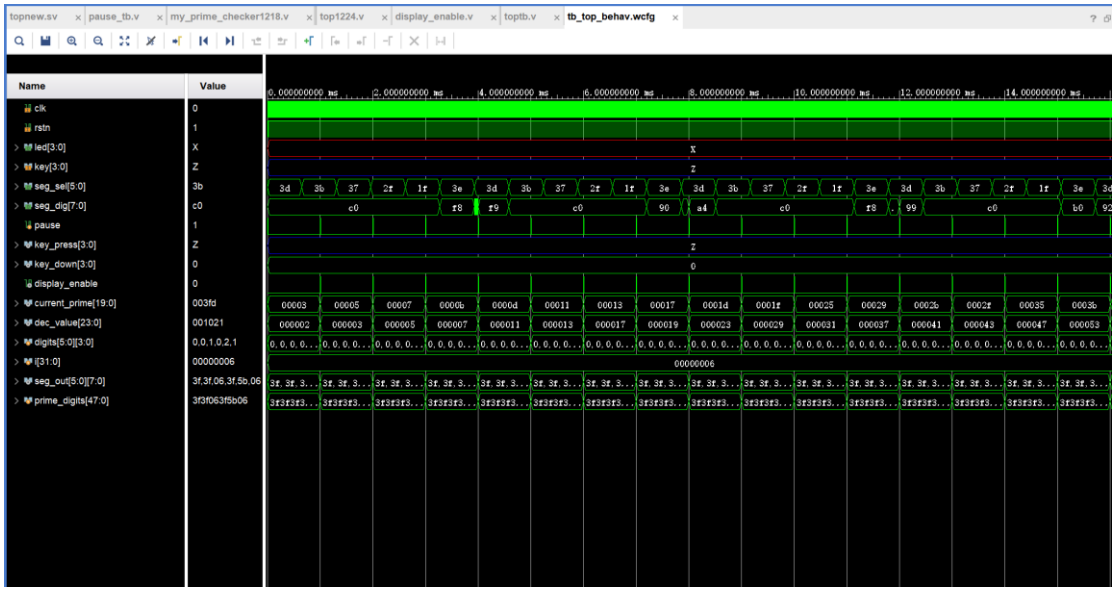


*图 3　素数判断显示电路仿真波形图*

从图中可见，使能信号长期保持 0，每到 1ms 迅速翻转为 1 然后继续保持 0。current_prime 可以周期性的计算出素数，dec_value 也成功将计算出的十六进制素数转化为十进制素数。

从仿真波形图可以看出基本实现 1s 显示一个素数，最后的问题是素数判断只能显示到 1021。

我们获得适当的波形图之后在电路板上进行了实验，实现了从 3 开始不间断输出数字至 1025。然而，在此基础上，我们进一步进行了代码优化，从而进一步提升了性能。

# 4.设计经验总结

## 难点 1：素数计算

素数判断模块的设计是从简单到复杂，逐步优化的过程。以下详细介绍整个设计过程以及设计思路。

### 4.1.1 初始设计：基于组合逻辑的素数判断

最初的设计使用组合逻辑移位除法器来实现素数判断。主要逻辑是利用候选数 cnt_data 和除数 cnt_cnt_data 的模运算结果 mod 来判断是否为素数。实现简单，直观，但对于每个候选数，需要完整遍历其所有可能的除数。即使只判断到 cnt_data/2，也会导致大量的重复运算。组合逻辑模运算在 FPGA 上实现需要占用较多硬件资源，且时钟频率过高时，可能导致无法完成运算。

### 4.1.2 改进设计：优化除法器模块

由于初始设计使用组合逻辑进行模运算，时钟频率过高时运算无法完成，因此进行了改进，将除法器设计为时序逻辑版本，利用状态机分阶段完成模运算。

将除法运算拆分为多个时钟周期完成，避免一次性计算过大的模运算。

总共分为 5 个状态：idle、div_init、div_shift、div_sub、div_end。通过状态机逐步完成模运算（即 R = A % B），大幅减少了时钟频率对运算的限制。

利用时序逻辑分阶段完成计算，避免了组合逻辑在高频时钟下的延迟问题。

### 4.1.3 最终设计：完整的素数判断模块

在引入时序逻辑除法器后，将其与素数判断逻辑结合，设计了最终的素数判断模块。利用 cnt_data 依次生成从 2 到 999999 的所有候选数。跳过偶数的判断（因为偶数一定不是素数，2 除外），每次候选数加 2。如果所有候选数判断完成（达到 999999），重新回到起点。

cnt_cnt_data 从 3 开始，依次生成小于等于 sqrt(cnt_data)的所有奇数。除数只需判断到候选数的平方根（cnt_data >> 1），减少不必要的模运算。引入基于状态机的除法器完成模运算。在 div_end 状态判断模运算结果：

如果 mod == 0 且 cnt_cnt_data 小于 cnt_data，设置 flag=1 表示当前候选数不是素数。如果除数遍历完成且 mod 始终不为 0，则候选数为素数。

### 心得：

模块采用时序逻辑除法器，确保能够在较高的时钟频率下稳定运行，同时降低了对 FPGA 硬件资源的占用。从简单的组合逻辑实现开始，逐步拆分复杂运算，最终形成了基于状态机的模块化设计。每一步的改进都围绕提升效率、降低资源占用展开，通过多次验证和测试确保设计的正确性。

## 难点 2：控制 1s 输出一个素数

要实现每秒输出一个素数，需要在 FPGA 上设计、实现素数生成与显示功能。

### 1.初始设计：RAM 存储素数

最初设计通过 RAM 存储素数，将素数预先计算并写入 RAM 中。每秒从 RAM 中读取下一个素数并显示。

```
module ram #(parameter DW=32, AW=10) (
    input clk,
    input we,
    input [DW-1:0] din,
    input [AW-1:0] write_addr,
```

```
    input [AW-1:0] read_addr,
    output reg [DW-1:0] dout
);
    reg [DW-1:0] mem [0:2**AW-1];

    always @(posedge clk) begin
        if (we)
            mem[write_addr] <= din;
    end

    always @(posedge clk) begin
        dout <= mem[read_addr];
    end
endmodule
```

由代码可见，素数写入 RAM 时，用 write_addr 管理写入地址。每秒从 RAM 的 read_addr 地址读取一个素数，并通过七段数码管显示。用 RAM 存储保证了显示素数的稳定性，同时减少实时计算的复杂性。但素数数量增长较快，RAM 容量会限制存储的素数数量（特别是高位素数）。

### 4.2.2 时钟分频设计

在初始设计中发现 FPGA 时钟频率过高，导致素数生成计算时间不足。因此，引入时钟分频器降低系统时钟频率，同时控制素数生成与显示的同步。

```
module clock_divider(
    input wire clk_in,          // 输入时钟 (50 MHz)
    input wire rstn,            // 复位信号，低电平有效
    output reg clk_out          // 输出时钟
);

    reg [5:0] count;            // 6-bit 计数器可以从 0 计数到 49
    always @(posedge clk_in or negedge rstn) begin
        if (!rstn) begin
            count <= 0;
            clk_out <= 0;
        end else begin
            if (count == 49) begin
                clk_out <= ~clk_out;   // 切换输出时钟状态
                count <= 0;
            end else begin
                count <= count + 1;
            end
        end
    end
endmodule
```

使用 clock_divider 模块，将输入时钟（如 50 MHz）分频到适合素数计算的较低频率。保证分频后的时钟信号既可以驱动素数生成模块，也可以同步显示模

块的计数更新。

### 4.2.3 最终设计：直接使用使能信号控制素数生成与暂停

直接通过时钟频率和 RAM 存储控制生成与显示的逻辑较为复杂，硬件资源占用较大。并且 RAM 写入与读取频繁交替，存在竞争问题，可能导致素数生成与显示不同步。

最终，通过使能信号 pause 控制素数生成的暂停与恢复，取消 RAM 存储，改为实时计算当前素数并输出显示。增加一个暂停信号 pause，用于控制素数生成模块的启停。当显示模块完成素数显示后，通过 display_enable 信号激活下一次素数生成。若生成过程中素数数据发生改变，则暂停当前计算，保证显示稳定性。（详细解释见 *2.设计原理与过程*）

同时在素数判断模块添加了延边检测，cnt_change 和 cnt_cnt_change 用于检测计数器 cnt_data 和 cnt_cnt_data 的变化，通过异或操作来实现。当计数器变化时，start 信号被拉低，触发状态机重新开始除法判断过程。（详细解释见 *2.设计原理与过程*）

**心得：**

本次实验对所有组员来说都是一次考验。从最初的一窍不通，面对任务无从下手，到最终写出完整的代码，我们共同努力，攻坚克难。任务庞大无从下手，那便将整体任务拆分成多个模块，逐一击破；最初的组合逻辑运行时间周期过长，我们便换用时序逻辑；预想的利用 RAM 进行素数存储和输出过于繁琐，
我们便化繁为简，直接通过使能信号控制素数判断模块运行和暂停……在一次又一次的尝试与修改中，我们对各模块的功能和配合有了更深的理解。

动手实践是最好的导师。在本次实验中，我们通过实操逐渐了解了 Vivado 等工具软件的使用，学会了通过观察仿真波形寻找代码逻辑错误、排查代码问题，这些技能的习得对我们来说都是弥足珍贵的经验，是在课堂上仅凭听讲无法习得的宝贵知识。同时，这次实验也让我们彻底认识到了“网络空间安全是一门实践学科”的真正含义，这段经历也将鼓励我们发扬实践精神，在网安的学习道路上愈行愈远。

作为一项小组作业，在这次小组作业的过程中，我们深刻体会到了团队协作对于项目的成功至关重要。团队的合作不是一个人的任务，只有大家各尽其责、共同参与、集思广益，承担起自己应负的责任，小组作业才能顺利推进。

感谢这次实验，为我们提供了宝贵的实践机会，也让我们明白了，知识不应仅存在于书本之中，更应存在于实践之中。