说明文档

成员：邱明阳 王溥纪 王思源

目录

1 概述

对于 1000000 以内素数的查找和显示，本组采用埃拉托色尼筛法，该筛法需使用 RAM 存储素数。其中，我们使用的 IP 核是 Vivado 中自带的 Block Memory Generator，其具体配置见**文末附录**。我们调用了 simple 并实现了四个模式下的素数输出，效果符合要求。虽然本代码未使用除法模块，但我们也研究并设计出了能够高效使用的流水线除法器。

2 代码解析

本节将对代码的三个重要部分进行解析。**（完整代码另见源码附件）**

2.1 底层优化部分

本节是对底层优化的模块的解析，主要模块有按键除抖模块、边沿检测模块、一秒钟计时器模块。

2.1.1 按键除抖模块

```
module key_debounce(
    input clk, //FPGA 时钟频率 50MHz
    input rstn,
    input [3:0] key,
    output [3:0] key_out //输出处理后的按键信号
);

    reg [31:0] delay_cnt; //准备饱和计时器检测有效低电平时间
    reg [3:0] key_reg = 4'b1111;
    reg [3:0] key_value;

    always @(posedge clk) begin //上升沿触发
        key_reg <= key;
        if(key_reg != key) //检测到按键信号改变
            delay_cnt <= 32'd1000000; //设计饱和计时器时间为 20Ms
        else if(key_reg == key) begin //按键信号改变后开始计时
            if(delay_cnt > 32'd0)
                delay_cnt <= delay_cnt - 32'd1;
            else
                delay_cnt <= delay_cnt;
        end
    end

    always @(posedge clk) begin
        if (~rstn)
            key_value <= 4'b1011;
        else if(delay_cnt == 32'd1) //此时按键低电平信号已稳定保持 20Ms
            key_value <= key;
        else
            key_value <= 4'b1111;
```

```
    end
    assign key_out = key_value; //将 key 的值输出到 key_out
endmodule
```

此板块的设计重点就是如何检测稳定的按键信号，防止手抖、接触不良产生的噪声。于是这里就想到设置 20ms 的时间阈值，只有低电平信号保持时间超过阈值时，才认为是有效的并且输出。时间阈值由计时器来确定，当按键信号改变时"key_reg != key"条件成立，计时器设置初始时间，而由于每个上升沿都用"key_reg <= key"对 key_reg 进行赋值，所以从下一个周期开始"key_reg == key"成立，计时器开始计时。而这过程中如果产生噪声导致 key 信号出现高电平，则又回到"key_reg != key"的情况，计时重置。由于时钟的频率为 50MHz，计时器从 1000000 降到 0 需要 20ms，这样就达成了所说的时间阈值。当计时器时间计数到 0 时，认为 key 的值是有效的并将 key 的值传递到 key_out 输出。

2.1.2 边沿检测模块

```
module edge_detect(
    input clk, //时钟信号
    input rstn, //复位信号
    input [3:0] key_out, //输入的按键信号
    output [3:0] pulse //输出的脉冲信号
);
    reg [3:0] key_last; //上一个时钟周期的按键状态
    reg [3:0] pulse1; //内部存储的脉冲状态

    always @(posedge clk or negedge rstn) begin
        if (!rstn) begin
            key_last <= 4'b1111;  //复位时，将上一个时钟周期的按键状态设为全高（1111）
            pulse1 <= 4'b0000; //复位时，将脉冲状态清零
        end else begin
            if (!key_out[0] && key_last[0]) begin
                pulse1 <= 4'b0001; //当前按键 0 从高变为低时，生成脉冲信号 0001
            end else if (!key_out[1] && key_last[1]) begin
                pulse1 <= 4'b0010; //当前按键 1 从高变为低时，生成脉冲信号 0010
            end else if (!key_out[2] && key_last[2]) begin
                pulse1 <= 4'b0100; //当前按键 2 从高变为低时，生成脉冲信号 0100
            end else if (!key_out[3] && key_last[3]) begin
                pulse1 <= 4'b1000; //当前按键 3 从高变为低时，生成脉冲信号 1000
            end else begin
                pulse1 <= 4'b0000; //其他情况下，脉冲状态清零
            end
            key_last <= key_out;  //更新上一个时钟周期的按键状态为当前状态
        end
    end

    assign pulse = pulse1; //输出脉冲信号
```

```
endmodule
```

这个边沿检测模块监测已经经过除抖后的的按键信号（key_out），当检测到输入按键信号在不同的周期由高变低时，会生成相应的脉冲信号。在这里，我们将输入的 key_out 打一拍（即保存 key_out 前一周期的值），则模块根据当前时钟周期的按键状态和上一个时钟周期的按键状态来判断是否有按键边沿触发，是哪个按键边沿被触发。当有按键按下或释放时，会根据按键的变化情况生成相应的脉冲信号，脉冲信号的位置对应按键号，通过 pulse 输出。

2.1.3 一秒钟计时器模块

```
module pulse_timer(
    input clk, //时钟信号
    input rstn, //复位信号 (低有效)
    input [3:0] pulse, //脉冲信号
    output [31:0] timer, //计时器
    output [3:0] key_state
);

    reg [31:0] timer1; //计时器寄存器
    reg [3:0] key_state1; //键状态寄存器

    always @(posedge clk or negedge rstn) begin
        if (~rstn) begin
            timer1 <= 32'd0; //复位时清零计时器
            key_state1 <= 4'b1111; //复位时设置键状态为全 1
        end
        else begin
            if (pulse != 4'd0) begin
                key_state1 <= ~pulse; //当脉冲信号不为 0 时，更新键状态为脉冲信号的取反
            end
            if ((key_state1 != 4'd1) && (timer1 != 32'd49_999_999)) begin
                timer1 <= timer1 + 1; //如果键状态不为 0001 并且计时器值不等于 49,999,999，则递增计时器
            end
            else if (timer1 == 32'd49_999_999) begin
                timer1 <= 32'd0; //如果计时器值等于 49,999,999，则计时器清零
            end
        end
    end

    assign timer = timer1; //连接计时器值到模块的输出端口
    assign key_state = key_state1; //连接键状态到模块的输出端口
endmodule
```

该模块的作用是实现一个简单的**脉冲计时器**，用于计算持续时间。当接收到脉冲信号时，开始计时，并在达到 1 秒时清零计时器，重新计时。这是模式 1，2 所实现的基本条件。另外，本模块还实现了将按下按键的脉冲信号 pulse 转变为按键当前状态的 key_state，从而实现使用者按下按键后 FPGA 板**可保持该模式运行**。

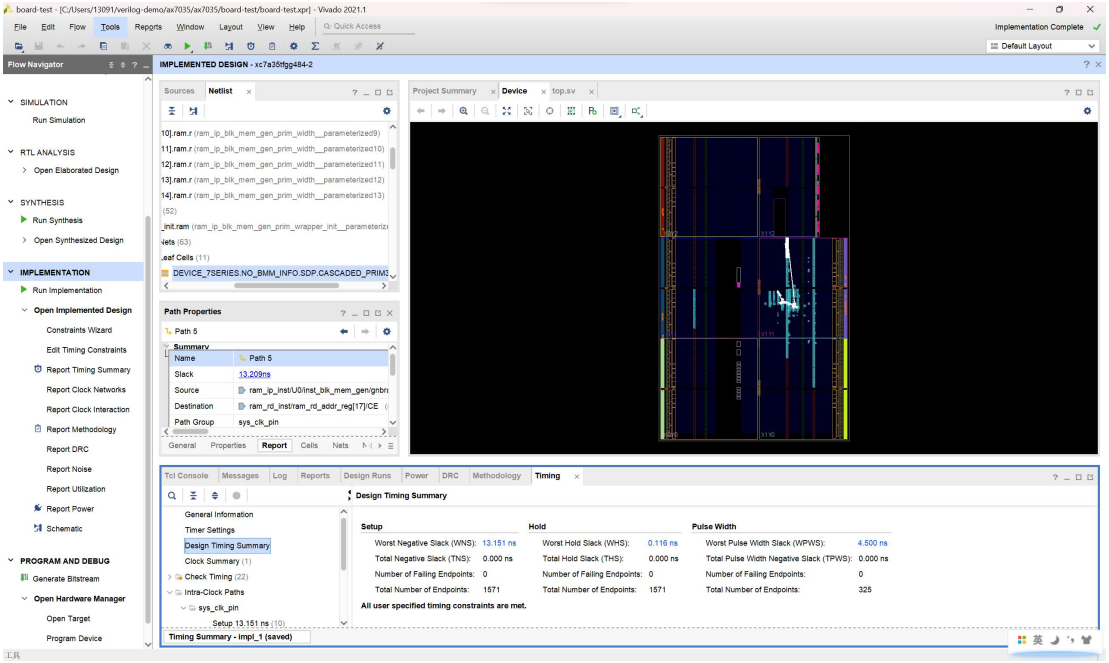
2.2 素数算法部分

2.2.1 算法研究

根据我们的初步讨论，计算 1000000 以内的素数大致有两种方式：

一种是使用**埃拉托色尼筛法**，也就是把 2 到 999999 间的所有数字中，为 2 的倍数、3 的倍数、5 的倍数……直到小于 1000 的最大的素数的倍数全部“删除”，这样剩下的数字即为素数。理论上，每次用某个数 n 筛完后，我们至少能保证 n 方内的数字为素数，所以说当我们用 n 筛完后，总能在已经筛好的数字中找到我们要用的下一个素数。

该算法的时间复杂度为 **O(n log log n)**，对于我们所需的 1000000 范围内素数，该算法可认为是线性复杂度，整个程序的计算时长应当在 0.1~1s 间。最差时序裕度为 13.151ns。



另一种是从数字 2 开始，对所有数字进行遍历，做**质因数分解**，确认他们能否被 1 和自身以外的数字整除。这种方法也是可行的，2 显然为素数；3 不被 2 整除，为素数；4 被 2 整除，为合数；5 不被 2、3 整除，为素数……直到最后一个素数。但是此方法需要用到除法运算，而一个除法器要计算完一个 20 位的二进制数，需要不止 1 个时钟周期，如果一个周期只进行一个被除数和一个除数的运算的话。

该算法的时间复杂度为 **O(n^2 * log n)**，由粗略计算和仿真结果可知，算完所有素数所需总时长约为 3s，最差时序裕度为-35ns（上限为 20ns）。这个时间说明时序“**爆炸**”了，很有可能使时序产生问题，同时该算法在处理数据方面有些难度，这里就不赘述。

综上述，我们可以看到**第一种方法**的时间复杂度小于第二种，且在实际实现过程中可以**不用编写除法器**，只需要用到乘法（或者说累加）即可实现对合数的筛除。因此，**我们选择第一种方法**，并且使用 RAM 存储已经算好的素数，然后使用这些素数的倍数继续筛除更多的合数。

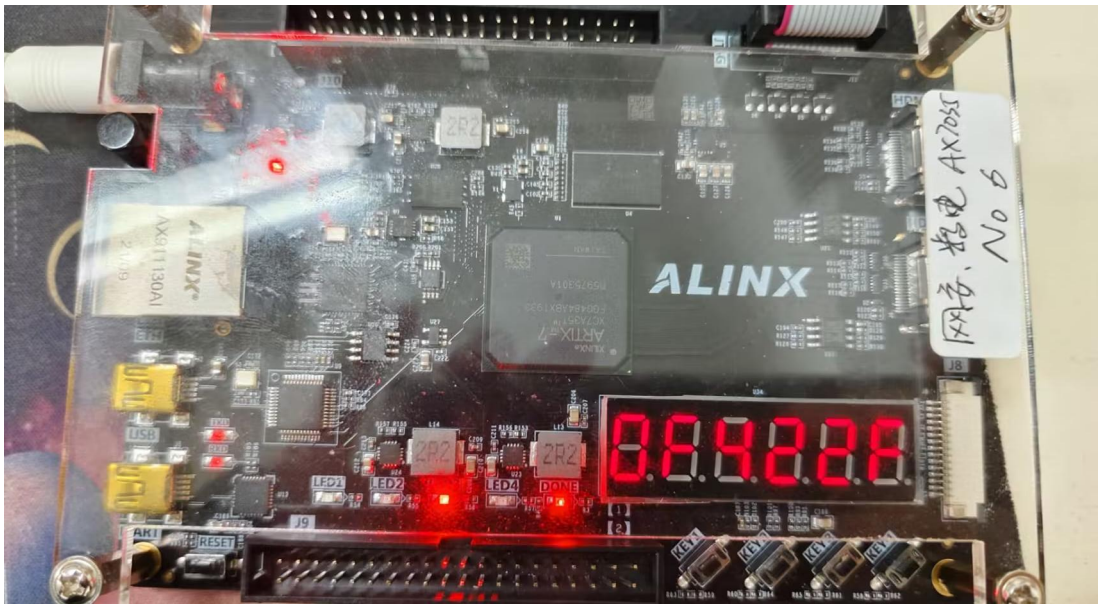
2.2.2 计算素数模块

```
module find(//筛数模块
    input clk,
    input rstn,
    input [19:0] num,
    output [19:0] sum);

    reg [19:0] num_r; //存储用于筛数的素数的寄存器
    reg [19:0] sum_r; //存储被筛出的合数（素数的倍数）

    assign sum=sum_r;

    always @(posedge clk or negedge rstn) begin
        if(~rstn) begin  //由于复位后无输入，即无法收到 num，所以需要自己给寄存器赋值
            num_r <= 2; //第一个素数
            sum_r <= 4; //第一个素数的第一个倍数（2 乘 2）
        end else begin
            if(num_r != num) begin //当 read 模块找到新的素数时，num 和寄存器 num_r 不一致时，更新 num_r 的值为下一个素数
                num_r <= num;
                sum_r <= num+num;
            end else begin
                if(sum_r < 1000000 && sum_r > 1) begin //当倍数在范围内时，继续增加倍数
                    sum_r <= sum_r + num; //乘法的本质是累加
                /*当倍数超出范围时，下一个周期将其改为 1，一方面 1 对其他模块没有影响（未定义该条件下操作），
                另一方面使得“大于等于一百万”成为一个优良的脉冲信号*/
                end else if(sum_r >= 1000000) begin
                    sum_r <= 1;
                end
            end
        end
    end
endmodule
```

该模块的**输入**为时钟信号、复位信号和从 RAM 中给出的素数 num，而**输出** sum 为 num 的倍数，该数据会返回给 RAM，从而使该合数被筛除。在每个时钟周期，该模块会通过累加将输入素数的所有倍数输出，当输出的合数大于所求范围时，将输出 sum 改为 1，这个数既不会产生逻辑上的冲突，也会使得“大于等于一百万”成为一个优良的脉冲信号。其作用后面会阐述。当 RAM 读取并输出一个新的不同于寄存器存储的素数时，寄存器内的素数会更新，于是开启新一轮的筛数。

比如在模式 3 中，我们的开发板会在约 0.2s 内计算出一百万内的所有素数，并在数码

管上顺序显示，停留在一百万内最大的素数：**0F422F**，转换为十进制即为 **999983**。



2.2.3 除法器（拓展研究）

关于本节开头所述的第二种方法，质因数分解，其核心部分在于写出一个能在一周期算出结果的除法器，我们设计了一个流水线除法器，其能在输入被除数和除数后立刻得出余数。从而判断出被除数是否会被除数整除。

```
module div #(parameter N=20,L=40)
    (input clk,rstn,
    input [L-1:0] addnumber,
    output [1:0] done,//00 表示还未算完，01 表示算出合数，10 表示算出素数
    output [L-1:0] numgap,//记录当算出素数时，该素数与上一个素数的差（memory）
    output [L-1:0] memory
    );

    wire [L-1:0] ram_rd_addr;
    wire [L-1:0] dividend [N-1:0];//被除数端口
    wire [L-1:0] divisor [N-1:0];//除数端口
    wire [L-1:0] remainder [N-1:0];//余端口
    genvar i;
    reg [L-1:0] now;//记录这一个素数
    reg [L-1:0] memory1;//记录上一个素数
    reg [1:0]done_r;

    assign done = done_r;
    assign numgap = now - memory;
    assign memory = memory1;

    always @(posedge clk or negedge rstn) begin
        if(~rstn) begin//1.复位信号低电平
```

```
            done_r <= 2'b00;
            memory1 <= 2;
            now <= 3;
        end else begin//2.时钟上升沿
            if(remainder[0]==0 && divisor[0] !=0 && now > 999983) begin//
出现余数为 0，且除数不为 0 的情况（复位初始状态），说明为合数
                done_r <= 2'b01;
                now <= now + 2;//+2 直接跳过偶数
            end else if(addnumber == memory1) begin//(已经模完所有已有素数，
均无整除，说明该数是素数)
                done_r <= 2'b10;
                memory1 <= now;
                now <= now + 2;//+2 直接跳过偶数
            end
            if(done!=2'b00) begin
                done_r <= 2'b00;
            end
        end
    end

    generate for(i=N-1;i>=0;i=i-1) begin//流水线上各子模块的输出传递
        assign remainder[i] = (dividend[i]>=divisor[i]) ?
dividend[i]-divisor[i] : dividend[i];
        if(i==N-1) begin
            assign dividend[N-1] = now;
            assign divisor[N-1] = (ram_rd_addr << 19);
        end else begin
            assign dividend[i] = remainder[i+1];
            assign divisor[i] = (divisor[i+1] >> 1);
        end
    end endgenerate
endmodule
```

该模块要比上一个算法的模块长得多。流水线除法器本质上是并联的，它**将除数与每一位对齐**，每一位的**被除数为上一位的被除数**或**上一位的余。**最后在同周期每一位都能得出余数的一部分，拼接后得到正确的余数。

需要注意余数为 0 不代表一定为合数，特别是在复位后的前几个周期除数可能为 0 的特殊情况，这种情况下流水线得到的最终余数也为 0。所以该算法在复位时存在较多需要考虑的情况，容易出错。

另外，该算法运行时长方面有着天然劣势，因为我们需要频繁读取 RAM 中数据，而读取并非即时的，在时序方面很容易出错，我们最后难以消除所有时序 bug，于是浅尝辄止。

2.3 RAM 读写、数字显示部分

本实验使用的 RAM 为伪双端口 RAM，其有一个写端口和一个读端口，两端口相互独立。下面将分写部分和读部分进行分析。

2.3.1 RAM 写部分

```
module ram_wr(//写入模块
    input clk,
    input rstn,
    input [19:0] sum,
    output reg ram_wr_we,
    output reg ram_wr_en,
    output [19:0] ram_wr_addr,
    output reg [0:0] ram_wr_data
);
    reg init_state;

    assign ram_wr_addr = {20{rstn}} & sum;//每得到一个合数，就将写指针移动到对应的地址

    always @(posedge clk or negedge rstn) begin
    if (~rstn) begin
            ram_wr_we <= 1;
            ram_wr_en <= 1;
        end
        else if ((sum < 1000000) && (sum > 1)) begin
            ram_wr_data <= 1;//在现在的地址内写入 1，这样就可以实现在合数地址内写入 1 作为标记
        end
    end
```

这个模块是与读取模块共同作用实现对合数的**标记与筛去**。这里的主要思想就是利用不断更新的 sum 值**遍历**范围内的所有**合数**，同时将这些**合数**与寄存器的**地址**对应并**写入 1** 作为标记，这样就可以在读取模块中分辨出合数并且筛去。

2.3.2 RAM 读部分

```
module ram_rd(
    input clk,
    input rstn,
    input [19:0] sum,
    input ram_rd_data,
    input [3:0] key_state,
    input [3:0] pulse,
    output [19:0] printtoscreen, //准备输出到数码管上的素数
    output [19:0] num, //需要的素数
    output reg ram_rd_en,
    output reg [19:0] ram_rd_addr
);
```

```
    reg [19:0] num1; //需要的素数
    reg [19:0] printtoscreen1;
    reg [3:0] delay;
    reg [31:0] count;
    reg [0:0] waitkey;
    reg [0:0] initstate;
    reg [0:0] launch;
    always @(posedge clk or negedge rstn )begin
        if(~rstn)begin
            num1 <= 2;
            ram_rd_addr <= 2;
            printtoscreen1 <= 0;
            delay <= 0;
            count <= 0;
            ram_rd_en <= 1;
            waitkey <= 0;
            initstate <= 1;
            launch <= 1;
        end

        else if ((num1 == 997) && (sum > 999998) && initstate) begin
            waitkey <= 1; //说明已经计算完毕，准备开始向屏幕上输出
            initstate <= 0;
        end

        else if (waitkey && (pulse[0] || launch))begin //接受到按下按键 1
的脉冲，此时应实现模式 1
            ram_rd_addr <= 2; //地址回到 2，准备从头输出。
            launch <= 0;
            delay <= 3;
            count <= 0;
        end

// ****************************************
//*****此处略去接受到按下按键 2，3，4 的脉冲*****
//****************************************

        else if ((printtoscreen1 == 999983) && ((key_state[2] == 0) ||
(key_state[0] == 0))) begin
            printtoscreen1 <= 999983;
        end

        else if((printtoscreen1 == 2) && ((key_state[1] == 0) || (key_state[3]
== 0)))begin
```

```
            printtoscreen1 <= 2;
        end

        else if (delay == 3) begin //延时三个时钟周期
            if (count < 32'd49_999_997) begin
                count <= count + 1;
            end else begin
                delay <= 4;
            end
        end

        else if (delay == 4) begin
            if (ram_rd_data != 0) begin //检测到地址是合数
                ram_rd_addr <= ram_rd_addr + 1; //继续查找下一个地址
                count <= 32'd49_999_992;
                delay <= 3;
            end
            else if (ram_rd_data == 0) begin //检测到地址是素数
                if (ram_rd_addr != 4) begin
                printtoscreen1 <= ram_rd_addr; //输出当前地址的素数到屏幕上
                count <= 0;
                delay <= 5;
                end else begin
                ram_rd_addr <= ram_rd_addr + 1;
                count <= 32'd49_999_992;
                delay <= 3;
                end
            end
        end

        else if (delay == 5) begin
            ram_rd_addr <= ram_rd_addr + 1;
            delay <= 3;
        End

//*********************************************
//*******此处略去 delay 值为 6-14 的判断条件********
//*********************************************

        else if ((num1 < 997) && (sum > 999998)) begin //该查找下一个地址
了
            ram_rd_addr <= ram_rd_addr + 1; //地址加一
            delay <= 2; //进入延时状态
            count <= 0;
```

```
        end
        else if (delay == 2) begin //延时三个时钟周期
            if (count < 2) begin
                count <= count + 1;
            end else begin
                delay <= 1;
            end
        end
        else if (delay == 1) begin
            if (ram_rd_data != 0) begin //检测到地址是合数
                ram_rd_addr <= ram_rd_addr + 1; //继续查找下一个地址
                count <= 0;
                delay <= 2;
            end
            else if (ram_rd_data == 0) begin //检测到地址是素数
                num1 <= ram_rd_addr; //输出当前地址的素数
                count <= 0;
                delay <= 0;
            end
        end
        else
            count <= 0;
    end
    assign num = num1;
    assign printtoscreen = printtoscreen1;
endmodule
```

以上代码主要实现了两个功能：首先，当每次埃氏筛法的返回值大于地址最大值时，在内存范围内，该素数的倍数便已全部被标记为合数，这时需将读地址**自增 1**，并读出自增后的地址存放的数据以判断其是否已被标记为合数，若为合数则地址继续自增，**直至寻找到下一个素数**并将其地址输入到埃氏筛法模块继续计算。

需要注意的是，我们使用的 RAM 在读地址时有**两周期的延时**。也就是说，我们某一周期读到的数据，其实是**前两周期对应地址存放的数据**。这样来，我们便须在更改地址时等待两个周期，使读到的数据与当前地址存放的数据一致。

其次，当所有计算业已完毕，内存地址中所有的合数已全部被标记。此时便可根据按键确定的不同模式，分别按照**不同顺序**，**不同时间间隔**将标记的素数的地址（即素数本身）输出到数码管上，而这一过程是通过 **delay 值的连续变化**，每周期执行**不同的判断语句**来实现的。

3 难点分析

3.1 时序分析混乱

在我们最开始写各个模块时，我们偏向于设置大量的寄存器，在 always 块中做非阻塞赋值，这往往意味着 always 语块中会存在大量的条件判断语句。一旦这些条件间存在交集，便会出现寄存器重复赋值的现象。而如果条件考虑有缺失，代码在板上实际运行时便很可能

陷入停滞。

最关键的是，寄存器的非阻塞赋值意味着赋值是没有顺序的，而我们在自己推演波形时，有时会忽略这一点，导致某个波形提前或推后一周期，从而使整个时序分析混乱，实际运行结果与预期不相符。

3.2 模块间关系不清晰

由于每个模块都分工给不同的人来写，即使我们提前进行了整体架构的分析，并说明了每个模块需要的输入和输出，但是在实际写好后，仍然会出现模块间不协调的情况。这常常表现为模块在错误的时间给出了错误的数据。除了少数语法错误，我们大量的 debug 任务都集中在对模块间时序的协调上。

3.3 Debug 难以准确找到实际问题

由于 Vivado 的报错比较笼统抽象，导致我们在知道存在错误后难以正确的改错。而且即使是编译时未出现错误，在实际运行时也有可能出现错误，此时我们更是无法准确知道错误，只能反复研究运行逻辑，复查代码。这样的 Debug 效率是非常低的。

3.4 RAM 读写存在难度

我们认为几乎所有的素数算法都需要用到已算出的素数，而这些素数需要 RAM 来存储。值得提到的是，RAM 的数据写入虽然是即时的，但是读取却会延迟两个周期，所以在涉及到 RAM 中数据读取时需要考虑数据传出的延迟，否则代码就会出现故障无法顺利运行。

4 心得体会

4.1 加强基础知识的储备

事实上，过去的 Verilog 代码作业，我们习惯于在课堂 PPT 上寻找相似代码，再稍作修改，然后根据仿真波形图不断 Debug 得到正确代码。这使得我们对 Verilog 的很多基础语法缺乏对本质的理解，所以在实现这样的更加复杂的大型代码时，我们很容易犯大量的低级语法错误。对 reg、wire 型变量的理解，对阻塞赋值、非阻塞赋值的理解，对 always 块，generate for 块的理解，对模块实例化的理解等等……这些都是我们在写这个代码时才逐渐深化的。

4.2 厘清思路、注重协作

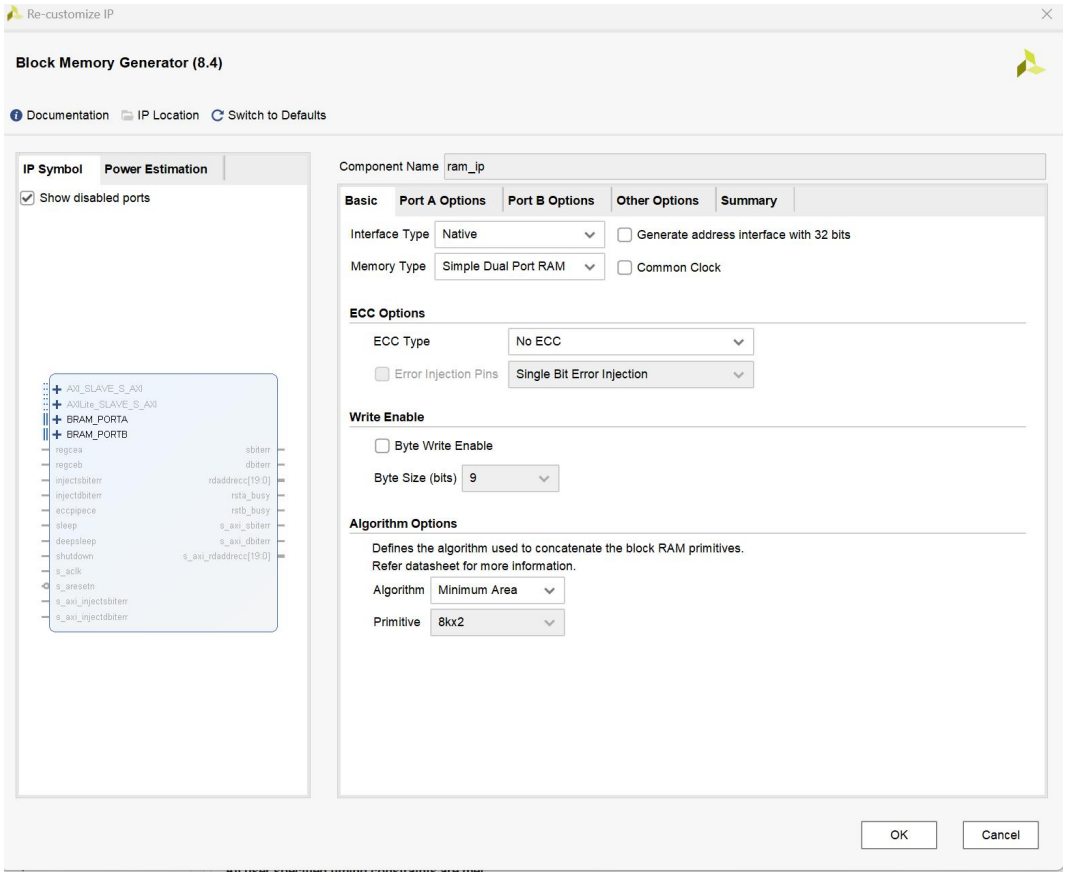
该代码的完成离不开所有组员的分工和合作，所以无论是在讨论思路、设计代码还是修改代码时，都应当注重成员间的信息交流。虽然每个人的任务相对独立，但也要一定程度地理解他人所写模块。互相理解也是个互查互促的过程。这样才能更加高效地完成任务。同时，写代码前一定要做充足的思考，比如对算法优化的可能性分析，对时间复杂度的分析等等。做好详尽的准备有利于任务的顺利推进。

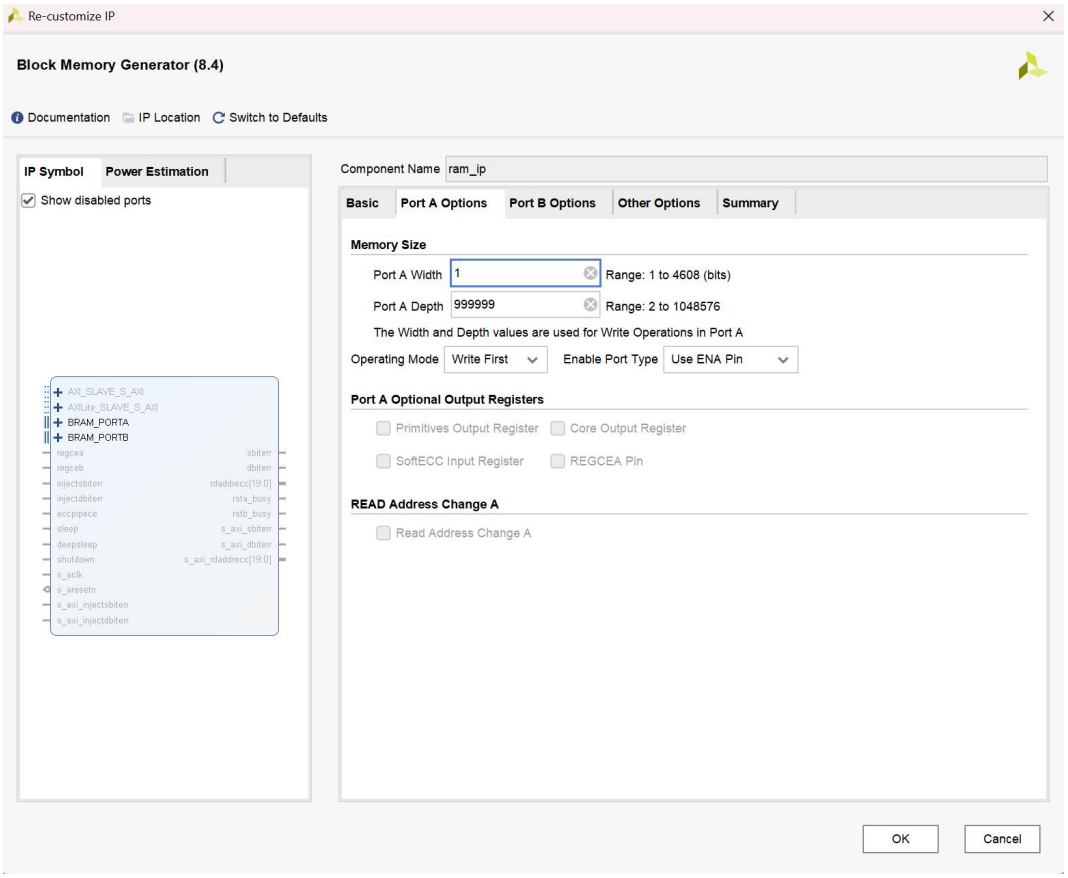
5 致谢

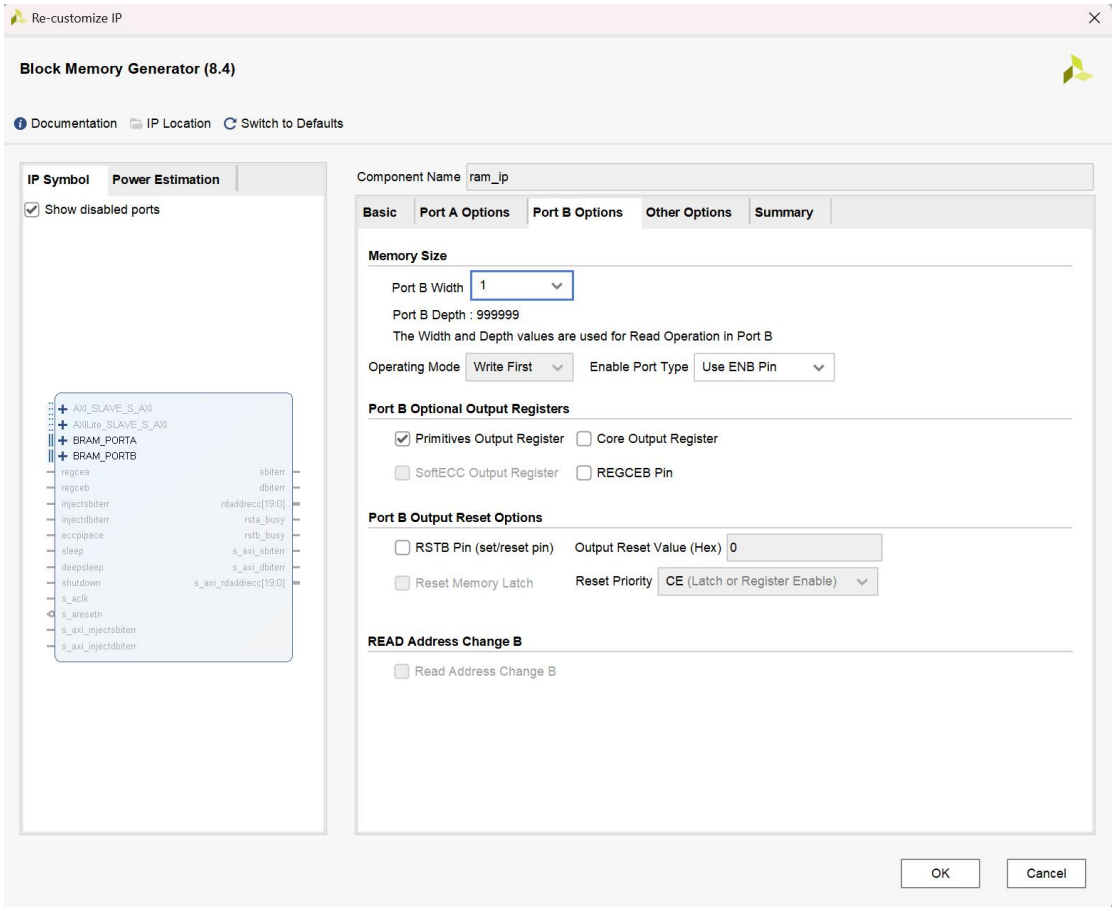
本次大作业的顺利完成，离不开每位组员的通力协作，更离不开平时宋威老师的讲解和马浩助教的指导。这次大作业极大地锻炼了我们的 verilog 代码的设计编写能力和 FPGA 的开发能力，为我们后续课程垒下了坚实的基础。

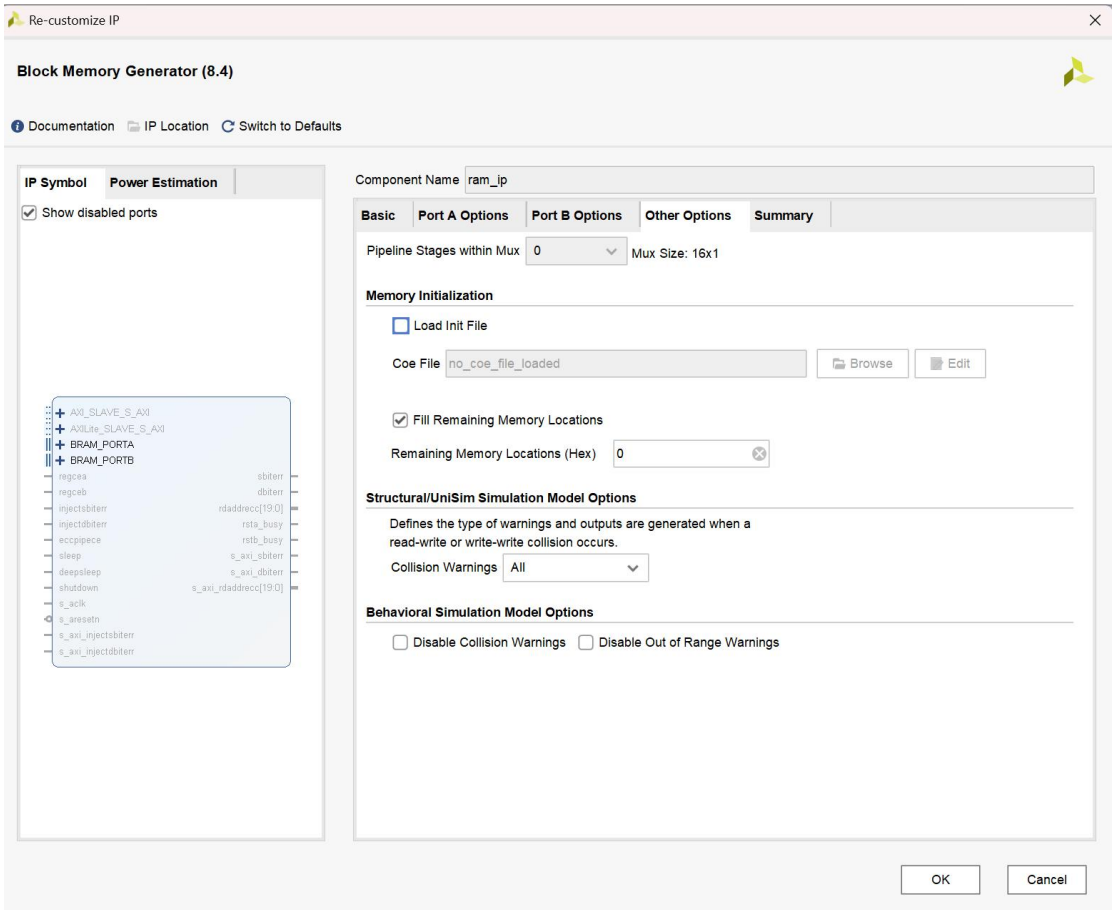
6 附录

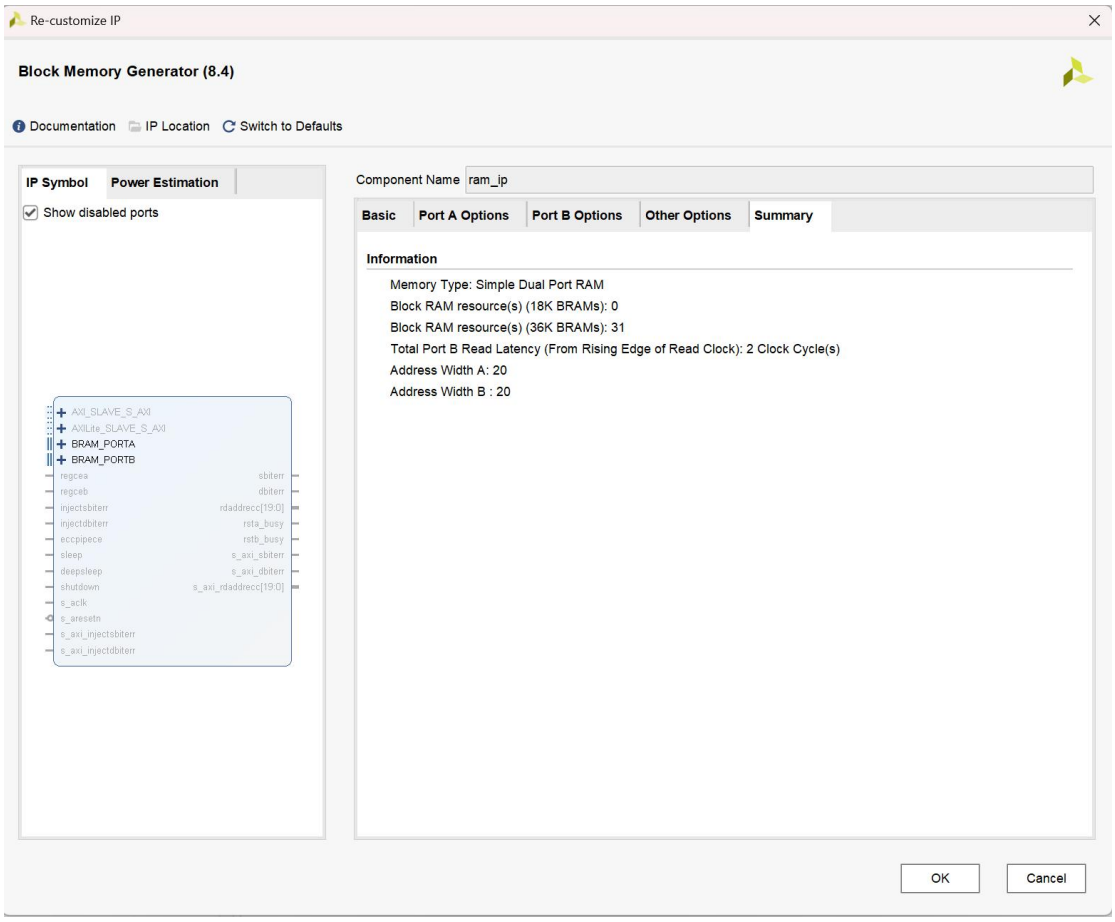
（IP 核配置参数）


Re-customize IP
Block Memory Generator (8.4)
Documentation
IP Location
Switch to Defaults
IP Symbol
Power Estimation
Show disabled ports
Component Name
ram_ip
Basic
Port A Options
Port B Options
Other Options
Summary
Interface Type
Native
Generate address interface with 32 bits
Memory Type
Simple Dual Port RAM
Common Clock
ECC Options
ECC Type
No ECC
Error Injection Pins
Single Bit Error Injection
Write Enable
Byte Write Enable
Byte Size (bits)
9
Algorithm Options
Defines the algorithm used to concatenate the block RAM primitives.
Refer datasheet for more information.
Algorithm
Minimum Area
Primitive
8kx2
OK
Cancel


Re-customize IP
Block Memory Generator (8.4)
Documentation
IP Location
Switch to Defaults
IP Symbol
Power Estimation
Show disabled ports
Component Name
ram_ip
Basic
Port A Options
Port B Options
Other Options
Summary
Memory Size
Port A Width
1
Range: 1 to 4608 (bits)
Port A Depth
999999
Range: 2 to 1048576
The Width and Depth values are used for Write Operations in Port A
Operating Mode
Write First
Enable Port Type
Use ENA Pin
Port A Optional Output Registers
Primitives Output Register
Core Output Register
SoftECC Input Register
REGCEA Pin
READ Address Change A
Read Address Change A
OK
Cancel


Re-customize IP
Block Memory Generator (8.4)
Documentation
IP Location
Switch to Defaults
IP Symbol
Power Estimation
Show disabled ports
Component Name
ram_ip
Basic
Port A Options
Port B Options
Other Options
Summary
Memory Size
Port B Width
1
Port B Depth : 999999
The Width and Depth values are used for Read Operation in Port B
Operating Mode
Write First
Enable Port Type
Use ENB Pin
Port B Optional Output Registers
Primitives Output Register
Core Output Register
SoftECC Output Register
REGCEB Pin
Port B Output Reset Options
RSTB Pin (set/reset pin)
Output Reset Value (Hex)
0
Reset Memory Latch
Reset Priority
CE (Latch or Register Enable)
READ Address Change B
Read Address Change B
OK
Cancel


Re-customize IP
Block Memory Generator (8.4)
Documentation
IP Location
Switch to Defaults
IP Symbol
Power Estimation
Show disabled ports
Component Name
ram_ip
Basic
Port A Options
Port B Options
Other Options
Summary
Pipeline Stages within Mux
0
Mux Size: 16x1
Memory Initialization
Load Init File
Coe File
no_coe_file_loaded
Browse
Edit
Fill Remaining Memory Locations
Remaining Memory Locations (Hex)
0
Structural/UniSim Simulation Model Options
Defines the type of warnings and outputs are generated when a read-write or write-write collision occurs.
Collision Warnings
All
Behavioral Simulation Model Options
Disable Collision Warnings
Disable Out of Range Warnings
OK
Cancel


Re-customize IP
Block Memory Generator (8.4)
Documentation
IP Location
Switch to Defaults
IP Symbol
Power Estimation
Show disabled ports
AXI_SLAVE_S_AXI
AXILite_SLAVE_S_AXI
BRAM_PORTA
BRAM_PORTB
regcea
regceb
injectsbiterr
injectdbiterr
eccpipece
sleep
deepsleep
shutdown
s_aclk
s_aresetn
s_axi_injectsbiterr
s_axi_injectdbiterr
sbiterr
dbiterr
rdaddrecc[19:0]
rsta_busy
rstb_busy
s_axi_sbiterr
s_axi_dbiterr
s_axi_rdaddrecc[19:0]
Component Name
ram_ip
Basic
Port A Options
Port B Options
Other Options
Summary
Information
Memory Type: Simple Dual Port RAM
Block RAM resource(s) (18K BRAMs): 0
Block RAM resource(s) (36K BRAMs): 31
Total Port B Read Latency (From Rising Edge of Read Clock): 2 Clock Cycle(s)
Address Width A: 20
Address Width B : 20
OK
Cancel