组成原理课程第 三 次实报告

实验名称：仿真调试

学号：2310420 姓名：王晶 班次：周二上午

实验目的：

1.复习设计 CPU 必须掌握的数字逻辑电略和 Verilog 描述的知识

2.理解同步 RAM 和并步 RAM 的区别及其仿真行为

3.初步掌握进行数字逻辑电略功能仿真时常见的错误及其调试方法

实验内容说明

1.实践任务 1：本书提供的寄存器堆源码为“两读一写”的结构，也就是有两个读端口（读端口没有使能位控制，表示永远使能）和一个写端口。对工程进行仿真测试，结合波形观察寄存器堆的读写行为。

2.实践任务 2：本实践任务要求为同步 RAM，异步 RAM 各建立一个工程,调用库实例化同步 RAM，异步 RAM，会提供一个设计的顶层文件，将它们封装成相同的模块名和接口。在完成工程的创建后,对它们进行仿真对比读写行为的异同。

在完成工程的仿真后,对它们进行综合和实现,参考 3.3 节介绍的方法查看时序结果和资源利用率,并结合读写时序进行分析。

3.实践任务 3：本实践任务提供了一个有 5 个 bug 的数字逻辑电路设计源码。该设计的正确功能是:

1）获取开发板最右侧 4 个拨码开关的状态（记为“拨上为 1’拨下为 0’,实际开发板上拨码开关的电平是“拨上为低电平’拨下为高电平”）,共有 16 状态（数字编号是 0～15）

2）最左侧数码管实时显示 4 个拨码开关的状态。数码管只支持显示 0～9。如果拨码开关状态是 10～15,则数码管的显示状态不更改（显示上一次的显示值）。

3）最右侧的 4 个单色 LED 灯会显示上一次的拨码开关的状态,支持显示 0～15（拨码开关拨上，对应 LED 灯亮）。

进行仿真，并充分利用仿真的辅助小技巧（分割、分组、颜色变化、标志等）进行调试，找出所有的 bug。仿真完成后,进行综合、实现并生成比特流文件。生成比特流文件后,连接开发板,进行上板验证。

实验步骤

实践任务 1：

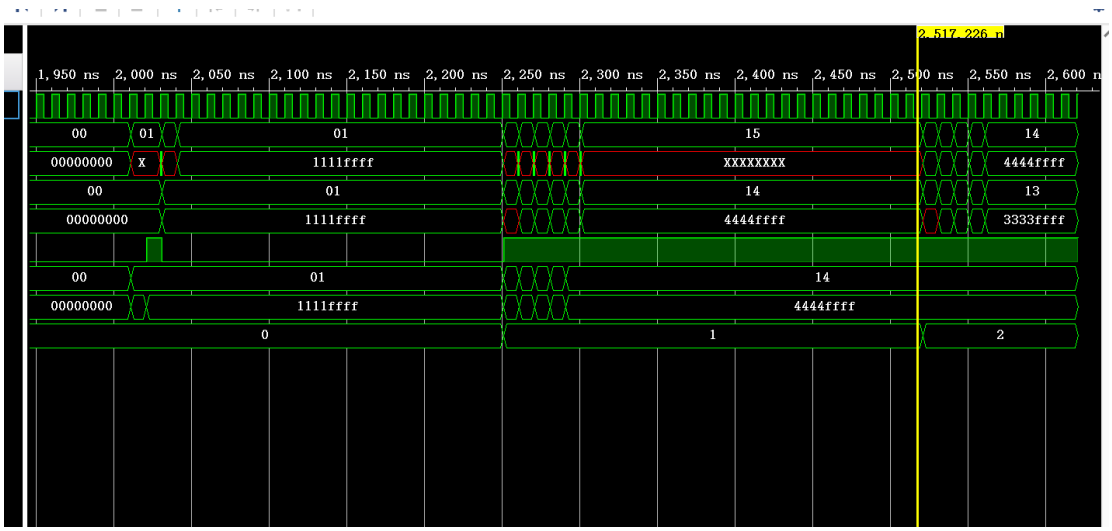
```
module regfile(
  input         clk,
  input  [ 4:0] raddr1,
  output [31:0] rdata1,
  input  [ 4:0] raddr2,
  output [31:0] rdata2,
  input         we,
  input  [ 4:0] waddr,
  input  [31:0] wdata
);
reg [31:0] rf[31:0];
// WRITE
always @(posedge clk) begin
    if (we) rf[waddr] <= wdata;
end
// READ OUT 1
assign rdata1 = (raddr1==5'b0) ? 32'b0 : rf[raddr1];
// READ OUT 2
assign rdata2 = (raddr2==5'b0) ? 32'b0 : rf[raddr2];
endmodule
```

```
`timescale 1ns / 1ps

module tb_regfile();

    // 信号定义
    reg         clk;
    reg  [ 4:0] raddr1;
    wire [31:0] rdata1;
    reg  [ 4:0] raddr2;
    wire [31:0] rdata2;
    reg         we;
    reg  [ 4:0] waddr;
    reg  [31:0] wdata;

    // 实例化寄存器堆
    regfile u_regfile (
        .clk    (clk),
```

```
        .raddr1 (raddr1),
        .rdata1 (rdata1),
        .raddr2 (raddr2),
        .rdata2 (rdata2),
        .we     (we),
        .waddr  (waddr),
        .wdata  (wdata)
    );

    // 时钟生成
    initial clk = 1'b0;
    always #5 clk = ~clk;  // 10ns 时钟周期

    // 初始化寄存器堆
    integer i;
    initial begin
        for (i = 0; i < 32; i = i + 1) begin
            u_regfile.rf[i] = 32'b0;
        end
    end

    // 测试流程
    initial begin
        // 初始化信号
        raddr1 = 0;
        raddr2 = 0;
        waddr  = 0;
        wdata  = 0;
        we     = 0;

        #10;

        $display("================ Test Start ================");

        // 测试写入和读取
        we = 1; waddr = 5'd1; wdata = 32'hAAAA5555; #10;
        $display("Write to addr %d: %h at time %0t", waddr, wdata,
$time);
        we = 1; waddr = 5'd2; wdata = 32'h12345678; #10;
        $display("Write to addr %d: %h at time %0t", waddr, wdata,
$time);
        we = 1; waddr = 5'd3; wdata = 32'hDEADBEEF; #10;
        $display("Write to addr %d: %h at time %0t", waddr, wdata,
$time);
```

```
        we = 0;

        // 读取测试
        raddr1 = 5'd1; raddr2 = 5'd2; #10;
        $display("Read rdata1 = %h, expected = 0xAAAA5555",
rdata1);
        $display("Read rdata2 = %h, expected = 0x12345678",
rdata2);

        raddr1 = 5'd3; raddr2 = 5'd0; #10;
        $display("Read rdata1 = %h, expected = 0xDEADBEEF",
rdata1);
        $display("Read rdata2 = %h, expected = 0x00000000",
rdata2);

        #20;
        $display("================ Test End ================");
        $finish;
    end

endmodule
```

将上述代码在 Verilog 仿真之后，得到的波形输出如下



分析波形结果：

借助 initial 和 always 块生成周期为 10ns 的时钟信号。对所有输入信号进行初始化，等待 2000ns 后开始测试。 在测试过程中，

Part 0：首先将 we 置为 0，尝试读取地址 1 的数据，此时寄存器 1 未写入数据，rdata1 应为 0。接着将 we 置为 1，把 32'h1111ffff 写入地址 1。之后将 we 置为 0，读取地址 1 和 2 的数据，rdata1 应为 0（因为地址 2 未写入数据），rdata2 应为 32'h1111ffff。

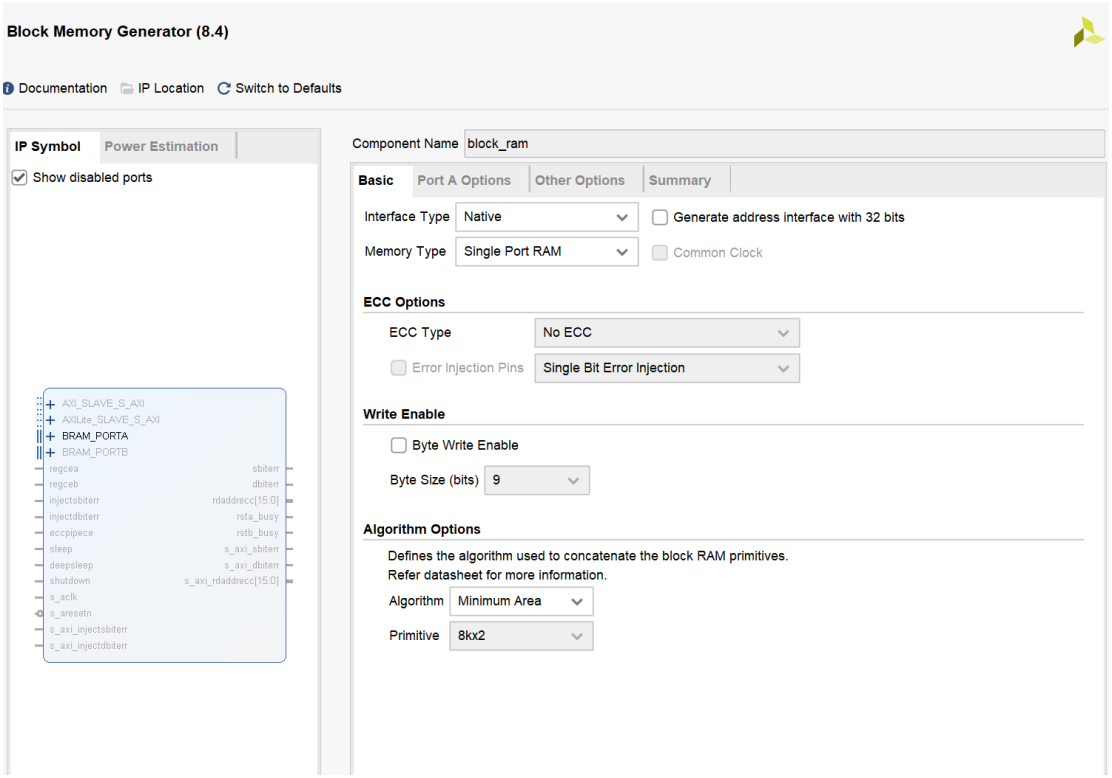
Part 1：把不同的数据依次写入地址 0x10 到 0x14。在写入过程中，读取相应地址的数据，验证写入操作是否正确。

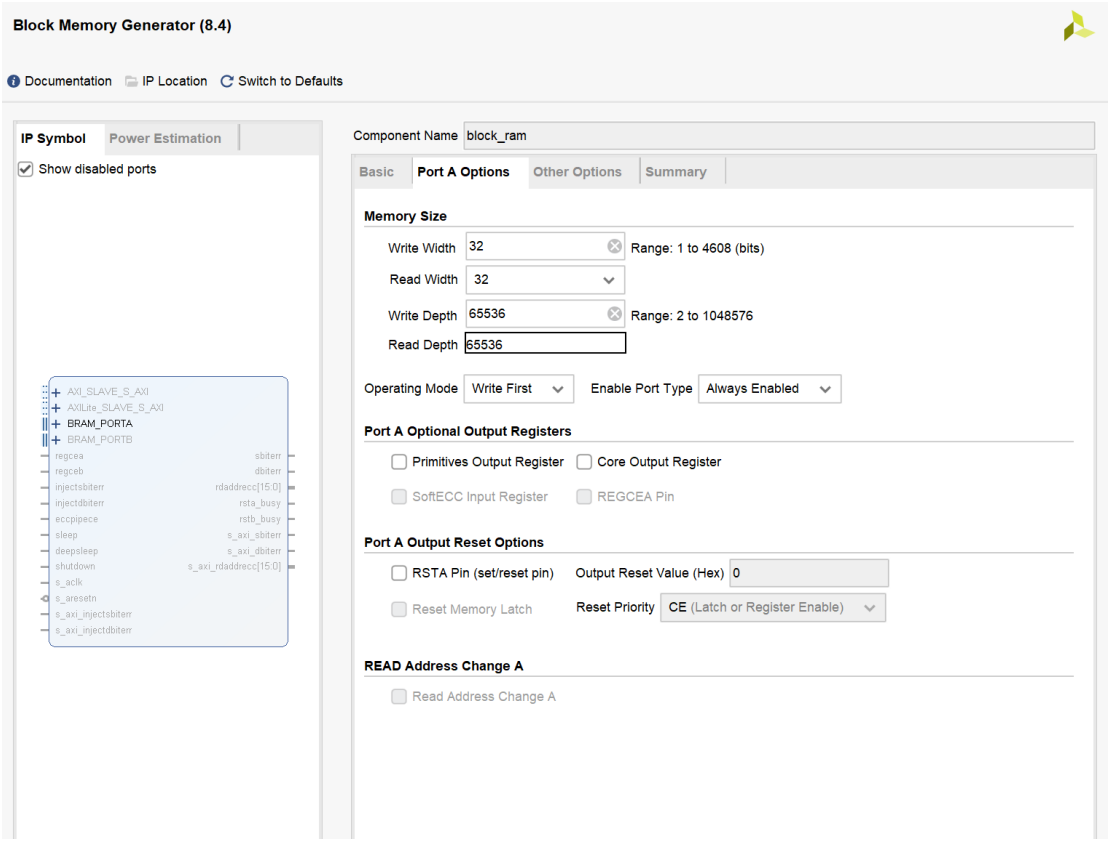
Part 2：持续进行读操作，验证之前写入的数据是否能正确读取。

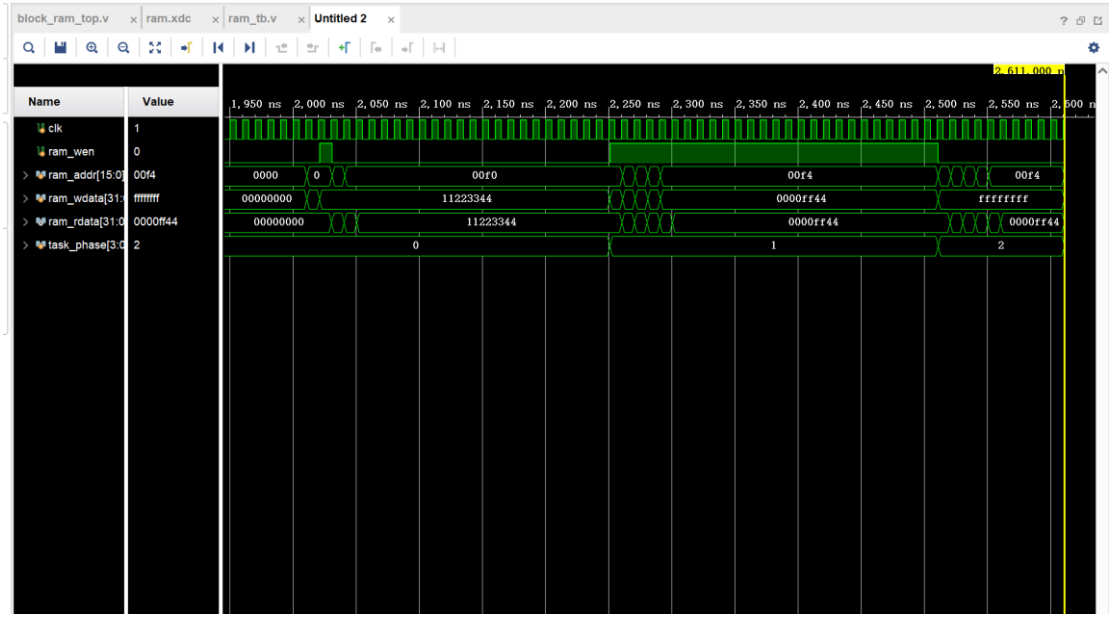
实践任务 2：

（1）同步 IP 核

按照如下图实验手册中所示，设置同步 IP 核的参数







分析波形结果：

在开始测试前，所有输入信号都被初始化为 0，并且等待 2000ns。之后打印测试开始的信息。

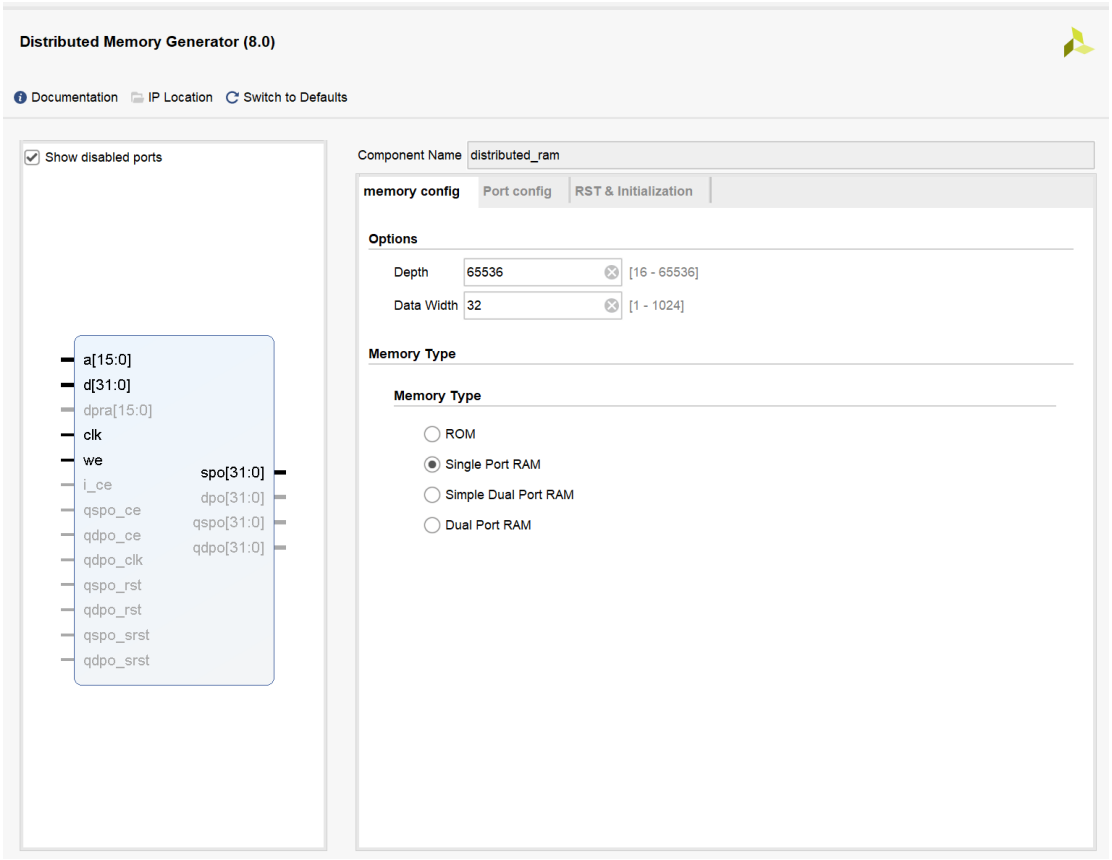
Part0:在第一个 10ns,ram_wen 为 0，不进行写操作，将地址设置为 16'hf0，写数据为 32'hffffffff，此时 ram_rdata 应保持之前的状态（通常为初始值）。第二个 10ns，ram_wen 为 1，将 32'h11223344 写入地址 16'hf0。由于是同步 RAM，数据会在时钟上升沿写入。第三个 10ns，ram_wen 为 0，地址设置为 16'hf1，进行读操作，

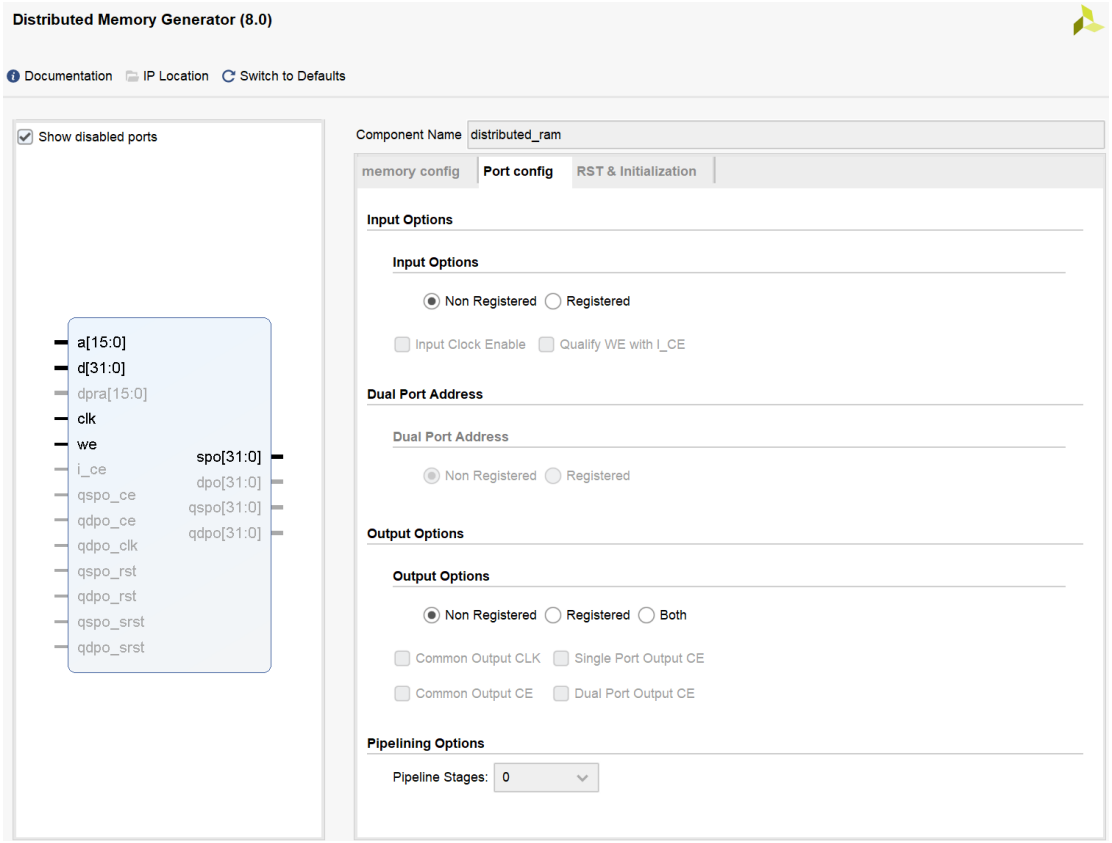
此时 ram_rdata 应是地址 16'hf1 处的值（如果之前未写入，可能是初始值）。第四个 10ns：ram_wen 为 0，地址设置为 16'hf0，进行读操作，ram_rdata 应是之前写入的 32'h11223344。

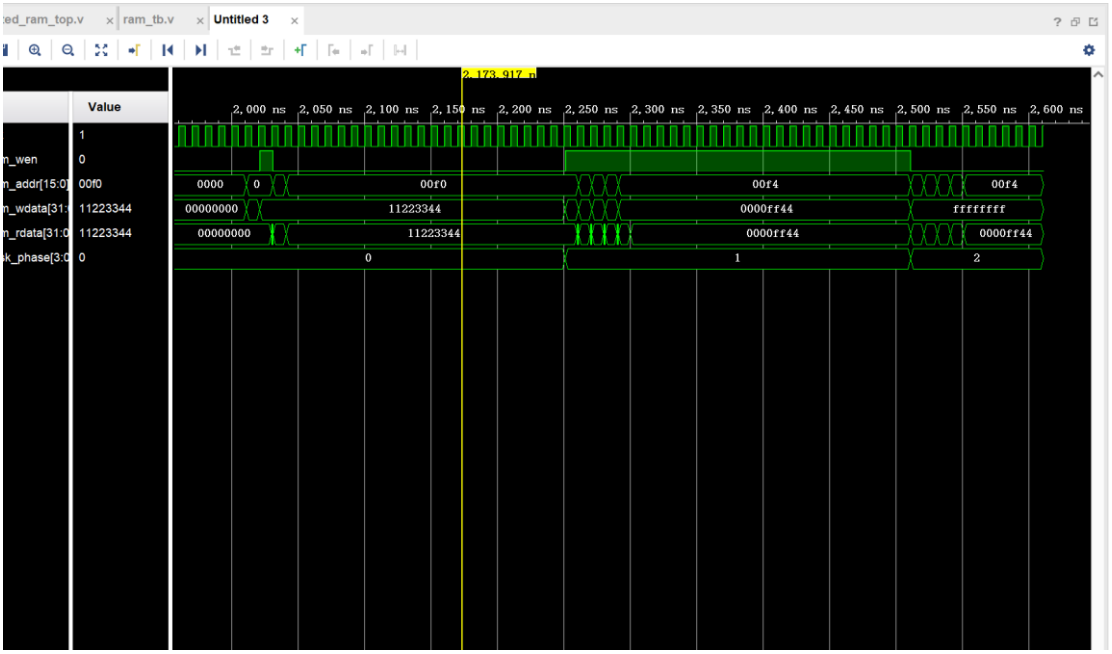
Part1：在这个阶段，ram_wen 始终为 1，依次将 32'hff00、32'hff11、32'hff22、32'hff33 和 32'hff44 写入地址 16'hf0 到 16'hf4。每次写入操作都会在时钟上升沿完成

Part2:在这个阶段，ram_wen 为 0，不进行写操作，依次对地址 16'hf0 到 16'hf4 进行读操作。ram_rdata 应依次输出之前写入的 32'hff00、32'hff11、32'hff22、32'hff33 和 32'hff44。

（2）异步 IP 核







分析波形结果：

初始时，所有信号都被初始化为 0。等待 2000ns 后，测试开始。由于 ram_wen 为 0，此阶段没有写操作，ram_rdata 会保持初始值。

Part0:第一个 10ns：ram_wen 为 0，不进行写操作。将地址设为 16'hf0，ram_rdata 会输出地址 16'hf0 处的值（可能是未定义值）。第二个 10ns：ram_wen 变为 1，将

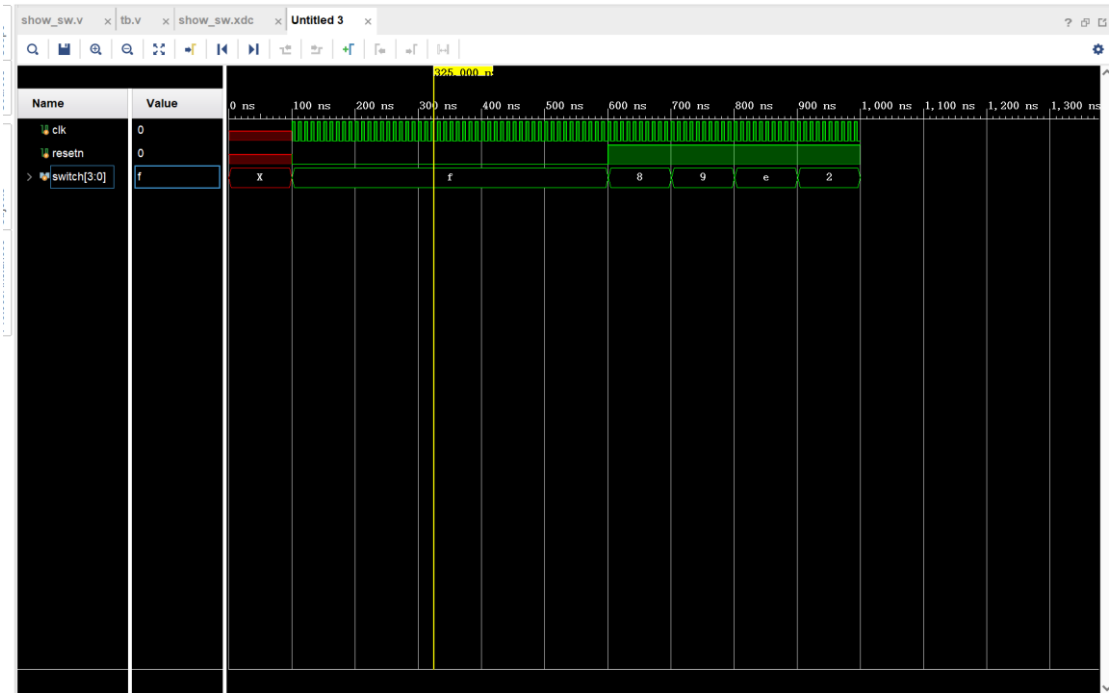
32'h11223344 写入地址 16'hf0。因为是异步 RAM，数据会立即写入该地址。第三个 10ns：ram_wen 变回 0，地址改为 16'hf1，ram_rdata 会立即输出地址 16'hf1 处的值（可能是未定义值）。第四个 10ns：地址再次变为 16'hf0，ram_rdata 会立即输出之前写入的 32'h11223344。

Part1：在这个阶段，ram_wen 始终为 1，依次将 32'hff00、32'hff11、32'hff22、32'hff33 和 32'hff44 写入地址 16'hf0 到 16'hf4。每一次地址和数据的改变，数据都会立即被写入相应地址

Part2：在这个阶段，ram_wen 为 0，不进行写操作。依次对地址 16'hf0 到 16'hf4 进行读操作。ram_rdata 会立即输出对应地址处的值，即依次输出 32'hff00、32'hff11、32'hff22、32'hff33 和 32'hff44。

实践任务 3：

由仿真之后的输出波形为：



任务要求中，指出代码中存在 5 个 bug，其中四个 bug 分别对应波形的 4 种异常情况：波形为 Z，波形为 X，波形停止和越沿采样。

（1）分析波形为 Z：

Z 表示高阻,比如电路断路就会显示为高阻,这种错误往往是以下两个原因导致的:

1）RTL 里声明为 wire 型的变量从未被赋值。

2）模块调用的信号未连接导致信号悬空

观察代码知：num_scn 是**未定义的 wire 变量**，但在 show_num 模块中作为 num_csn 端口输入，导致综合错误

修改后为：

```
show_num u_show_num(
        .clk        (clk      ),
        .resetn     (resetn   ),
        .show_data  (show_data),
        .num_csn    (num_csn  ),
        .num_a_g    (num_a_g  )
);
```

（2）分析波形为 X：

show_data 作为 show_num 模块的输入，但未被赋值，导致 num_a_g 没有正确更新数码管显示

修改后的代码为：

```
always @(posedge clk)
begin
    show_data   <= ~switch;
end
```

（3）分析波形停止：

波形停止是指仿真在某一时刻停止,再也无法前进分毫，而仿真却显示仍然在运行,这种错误往往是 RTL 里存在组合环路导致的。所谓组合环路是指信号 A 的组合逻辑表达式中某个产生因子为 B，而 B 的组合逻辑表达式中又用到了信号 A，组合逻辑循环嵌套会造成仿真器循环计算,导致其无法退出、最终导致波形停止的现象。

此次实验仿真中，波形并未停止。并不是所有的组合环路都会导致波形停止，有些复杂的组合环路（比如跨多个模块形成的组合环路）可能会被工具自动处理,但这种处理是有风险的,可能导致‘仿真通过’上板不过。
keep_a_g **直接由 num_a_g + nxt_a_g 计算**，然后 nxt_a_g 又**依赖 keep_a_g**，形成组合逻辑环路，可能导致综合失败或不稳定的行为

修改后代码为：

```
reg [6:0] prev_num_a_g;

always @(posedge clk)
begin
```

```
    if (show_data < 4'd10)
        prev_num_a_g <= nxt_a_g;
end

assign nxt_a_g = show_data==4'd0 ? 7'b1111110 :
                 show_data==4'd1 ? 7'b0110000 :
                 show_data==4'd2 ? 7'b1101101 :
                 show_data==4'd3 ? 7'b1111001 :
                 show_data==4'd4 ? 7'b0110011 :
                 show_data==4'd5 ? 7'b1011011 :
                 show_data==4'd7 ? 7'b1110000 :
                 show_data==4'd8 ? 7'b1111111 :
                 show_data==4'd9 ? 7'b1111011 :
                                   prev_num_a_g;
```

（3）分析波形为越沿采样：

越沿采样是指一个被采样的信号在上升沿采样到了其在上升沿后的值，一般情况下认为这是一个错误，是 RTL 里阻塞赋值=和非阻塞赋值＜＝使用不当导致的

越沿采样是一个隐藏较深的错误,往往可能和逻辑错误混在一起。初看起来,其波形是很正常的,而且在发生越沿采样后，要再执行很长时间才会出错。因此,大家可以先按照逻辑错误进行调试,如果发现数据采样有异常,就需要甄别是否出现了越沿采样的错误。

修改后代码为：

```
always @(posedge clk)
begin
    show_data_r <= show_data;
end
```

（4）分析功能错误：

代码修改如下：

```
always @(posedge clk)
begin
    if(!resetn)
    begin
        prev_data <= 4'd0;
    end
    else
    begin
        prev_data <= show_data_r;
```

```
    end
end
```

总体修改后代码为：

修改后的 show_sw.v 代码为

```
module show_sw (
    input             clk,
    input             resetn,

    input      [3 :0] switch,    // 拨码开关输入

    output     [7 :0] num_csn,   // 数码管片选
    output     [6 :0] num_a_g,   // 数码管显示

    output     [3 :0] led        // LED 显示上一次拨码开关状态
);

// 变量定义
reg [3:0] show_data;    // 当前拨码开关状态
reg [3:0] prev_data;    // 记录上一次拨码开关状态
reg [3:0] show_data_r;  // 用于保持数码管的稳定显示

// 处理拨码开关状态 (拨上 = 低电平，拨下 = 高电平，需要取反)
wire [3:0] switch_processed = ~switch;

// 获取拨码开关状态
always @(posedge clk or negedge resetn) begin
    if (!resetn)
        show_data <= 4'd0;
    else
        show_data <= switch_processed;
end

// 记录上一次数码管有效显示数据
always @(posedge clk or negedge resetn) begin
    if (!resetn)
        show_data_r <= 4'd0;
    else if (show_data < 4'd10)  // 仅在 0~9 更新
        show_data_r <= show_data;
end
```

```
// 记录上一次拨码开关状态 (用于 LED 显示)
always @(posedge clk or negedge resetn) begin
    if (!resetn)
        prev_data <= 4'd0;
    else if (show_data_r != show_data)  // 仅在数值变化时更新
        prev_data <= show_data_r;
end

// LED 显示上一次拨码开关状态
assign led = ~prev_data;  // 需要取反，以匹配实际硬件电平

// 数码管显示模块
show_num u_show_num(
    .clk       (clk),
    .resetn    (resetn),
    .show_data (show_data_r),  // 确保 10~15 时保持数码管显示
    .num_csn   (num_csn),
    .num_a_g   (num_a_g)
);

endmodule

//----------------------------{数码管显示模块}----------------------
-//
module show_num (
    input             clk,
    input             resetn,

    input      [3:0]  show_data,   // 显示数据
    output     [7:0]  num_csn,     // 数码管片选
    output reg [6:0]  num_a_g      // 数码管段选
);

// 使能数码管的最左侧
assign num_csn = 8'b0111_1111;

// 译码逻辑
reg [6:0] nxt_a_g;

always @(*) begin
    case (show_data)
        4'd0: nxt_a_g = 7'b1111110;  // 0
```

```
        4'd1: nxt_a_g = 7'b0110000;  // 1
        4'd2: nxt_a_g = 7'b1101101;  // 2
        4'd3: nxt_a_g = 7'b1111001;  // 3
        4'd4: nxt_a_g = 7'b0110011;  // 4
        4'd5: nxt_a_g = 7'b1011011;  // 5
        4'd6: nxt_a_g = 7'b1011111;  // 6
        4'd7: nxt_a_g = 7'b1110000;  // 7
        4'd8: nxt_a_g = 7'b1111111;  // 8
        4'd9: nxt_a_g = 7'b1111011;  // 9
        default: nxt_a_g = num_a_g;  // 10~15 时保持原值
    endcase
end

// 时序逻辑更新数码管
always @(posedge clk or negedge resetn) begin
    if (!resetn)
        num_a_g <= 7'b0000000;
    else
        num_a_g <= nxt_a_g;
end

endmodule
```

修改后的 tb.v 代码为：

```
module tb;
    reg        clk;
    reg        resetn;
    reg [3:0]  switch;    // 拨码开关输入
    wire [7:0] num_csn;   // 数码管片选
    wire [6:0] num_a_g;   // 数码管显示
    wire [3:0] led;       // LED 显示上一次拨码开关状态

    // 生成时钟信号
    initial begin
        clk = 1'b0;
        forever #5 clk = ~clk;  // 10ns 时钟周期
    end

    // 复位和测试过程
    initial begin
        resetn = 1'b0;
```

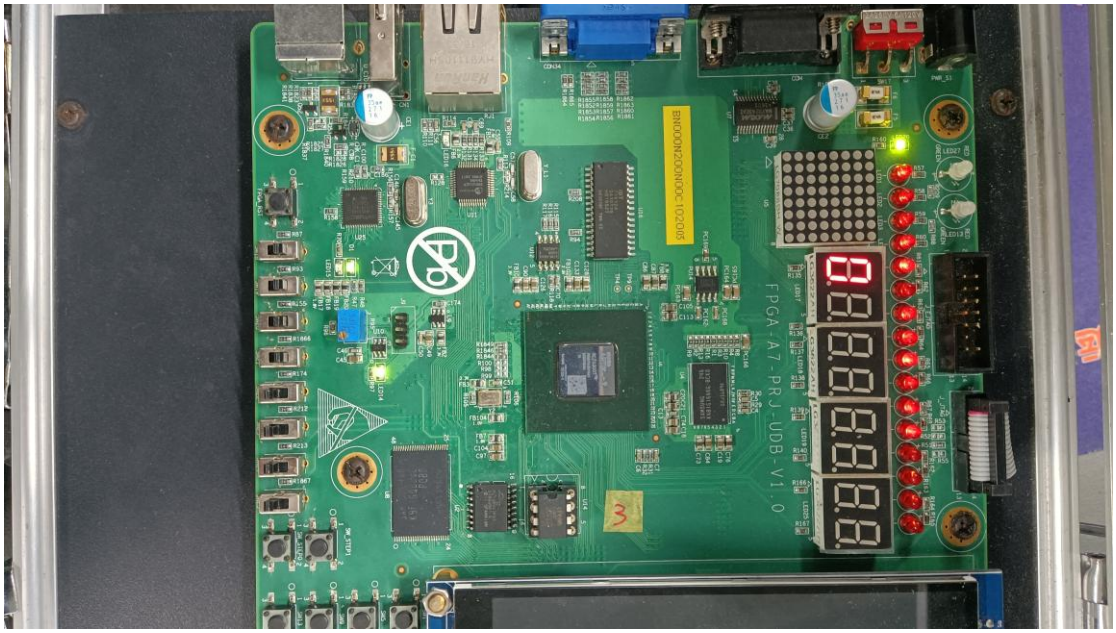
```
        switch = 4'b0000; // 初始值

        #20 resetn = 1'b1;  // 释放复位

        // 依次测试不同开关状态，注意开发板实际电平是拨上=低，拨下=高
        #10 switch = 4'b0000; // 期望数码管显示 0
        #10 switch = 4'b0001; // 期望数码管显示 1
        #10 switch = 4'b0010; // 期望数码管显示 2
        #10 switch = 4'b1010; // 10，数码管应保持不变
        #10 switch = 4'b1111; // 15，数码管应保持不变
        #10 switch = 4'b0111; // 7，数码管应更新
        #10 switch = 4'b1001; // 9，数码管应更新
        #10 switch = 4'b1110; // 14，数码管应保持不变
        #10 switch = 4'b0011; // 3，数码管应更新
        #10 switch = 4'b0000; // 0，数码管应更新

        #50 $finish; // 结束仿真
    end

    // 实例化待测模块
    show_sw u_show_sw (
        .clk    (clk),
        .resetn (resetn),
        .switch (switch),
        .num_csn(num_csn),
        .num_a_g(num_a_g),
        .led    (led)
    );

    // 监视信号变化
    initial begin
        $monitor("Time: %0t | switch: %b | num_a_g: %b | led: %b",
$time, switch, num_a_g, led);
    end

endmodule
```

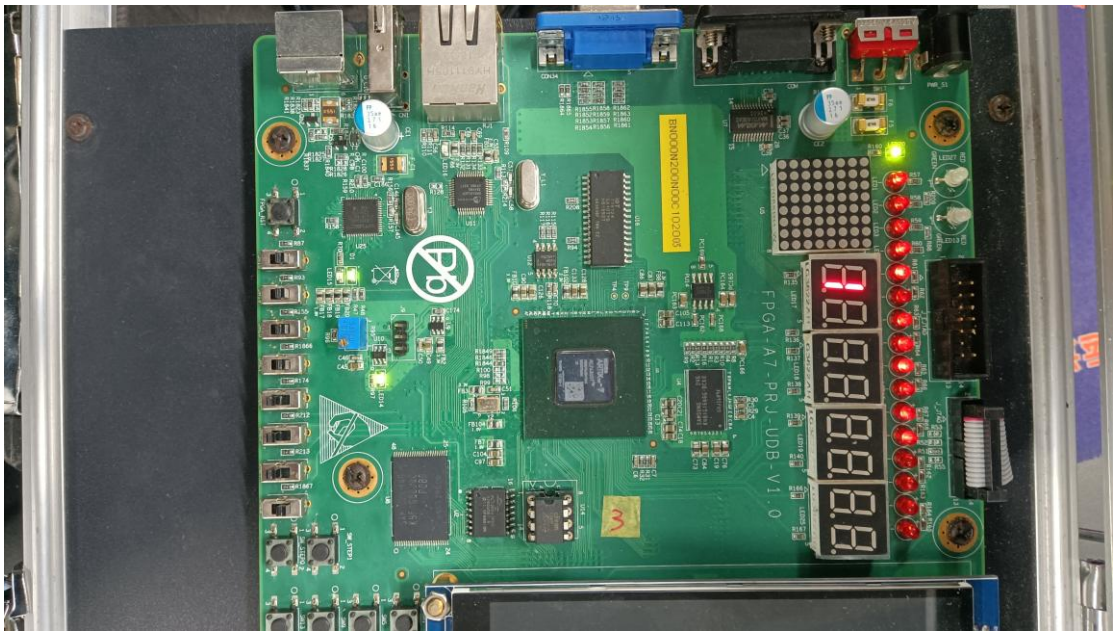
实验结果分析

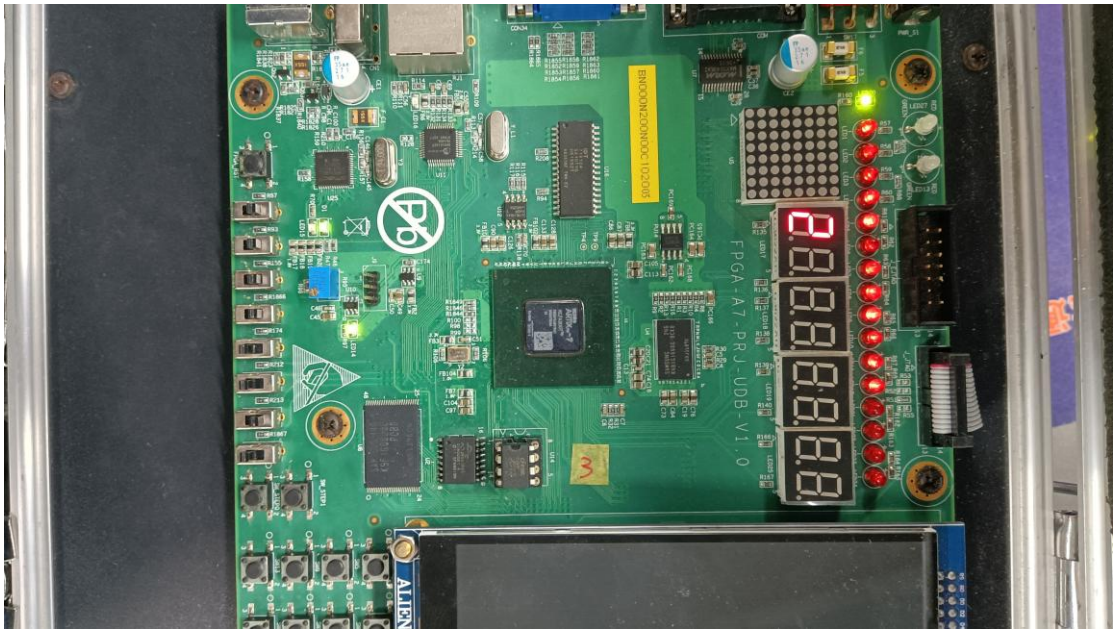
最后的上箱结果为：
最左侧数码管实时显示 4 个拨码开关的状态。数码管只支持显示 0～9。如果拨码开关状态是 10～15,则数码管的显示状态不更改（显示上一次的显示值）。

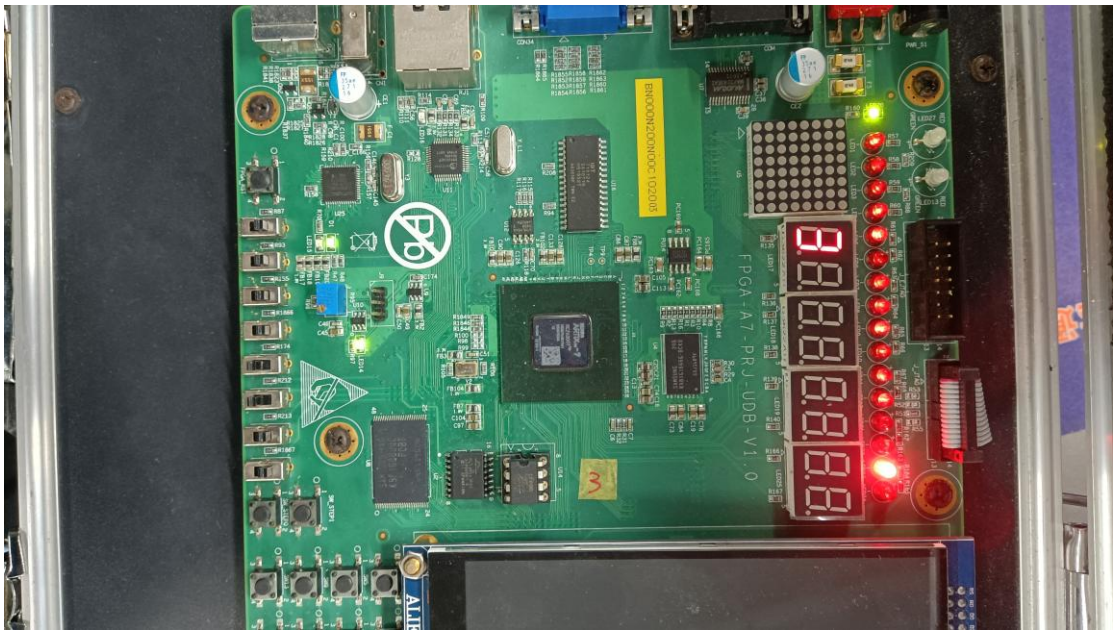
最右侧的 4 个单色 LED 灯会显示上一次的拨码开关的状态,支持显示 0～15（拨码开关拨上，对应 LED 灯亮）。

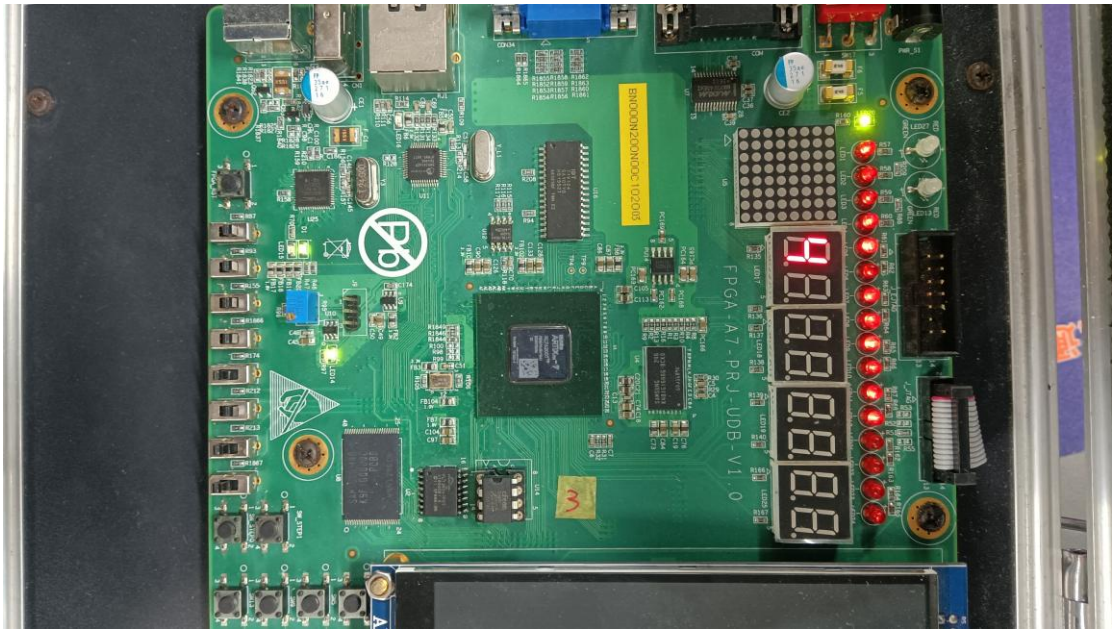
下面为数码管的 9 个状态，说明实验代码正确

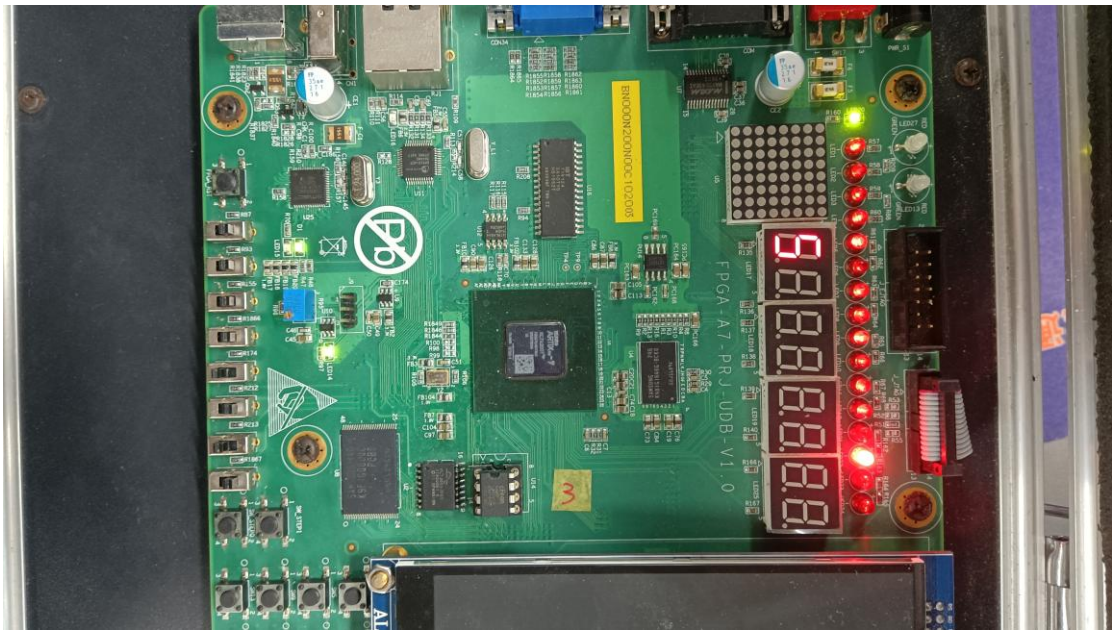


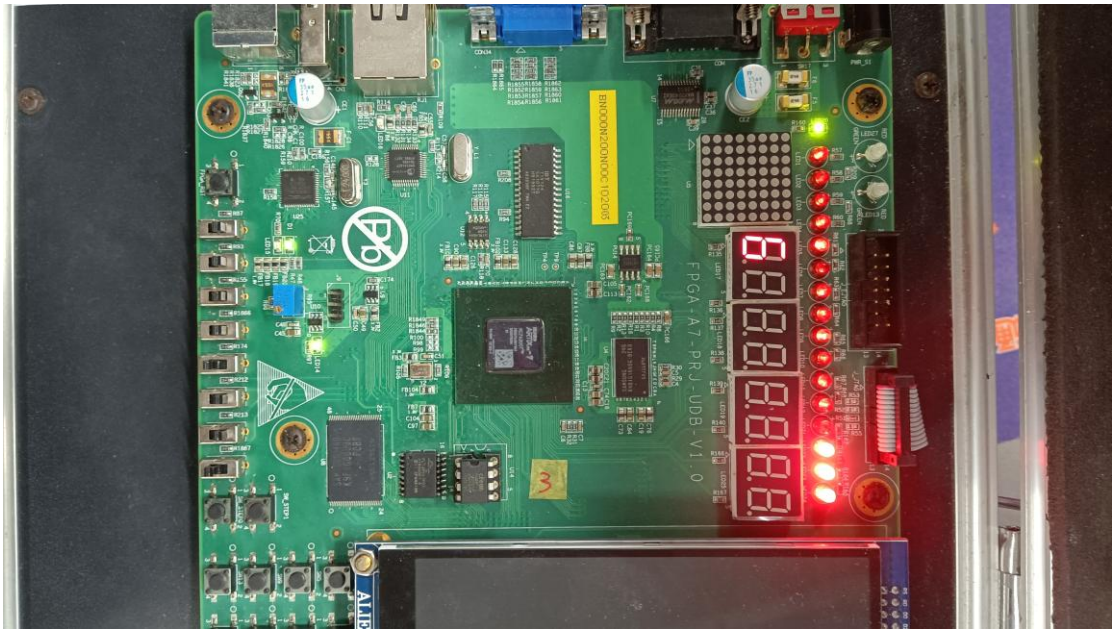


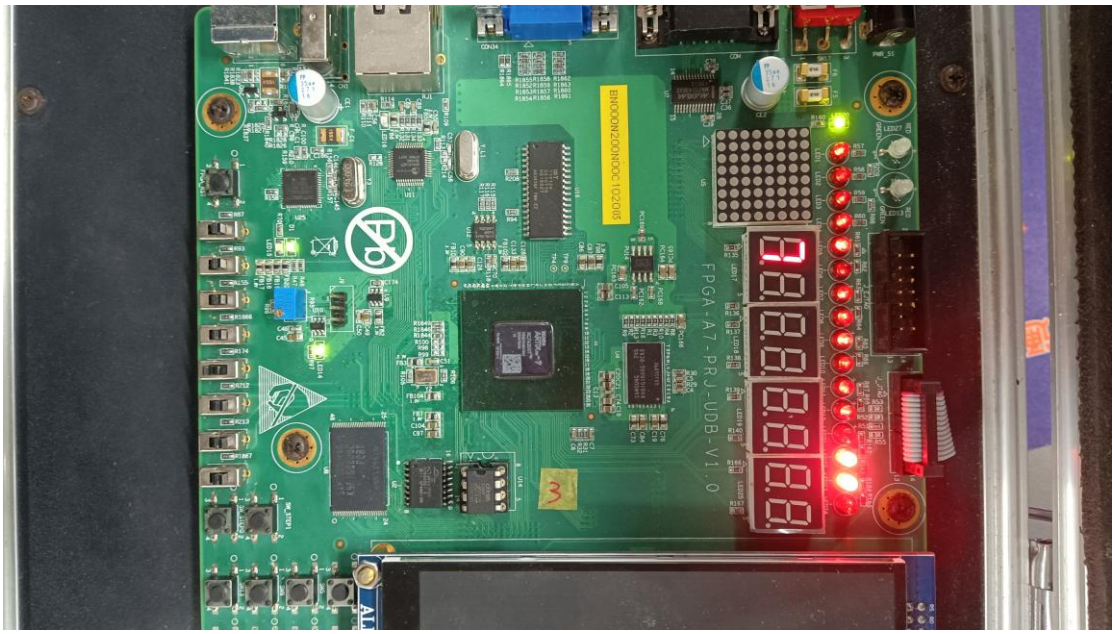

FPGA-A7-PRJ-UDB-V1.0

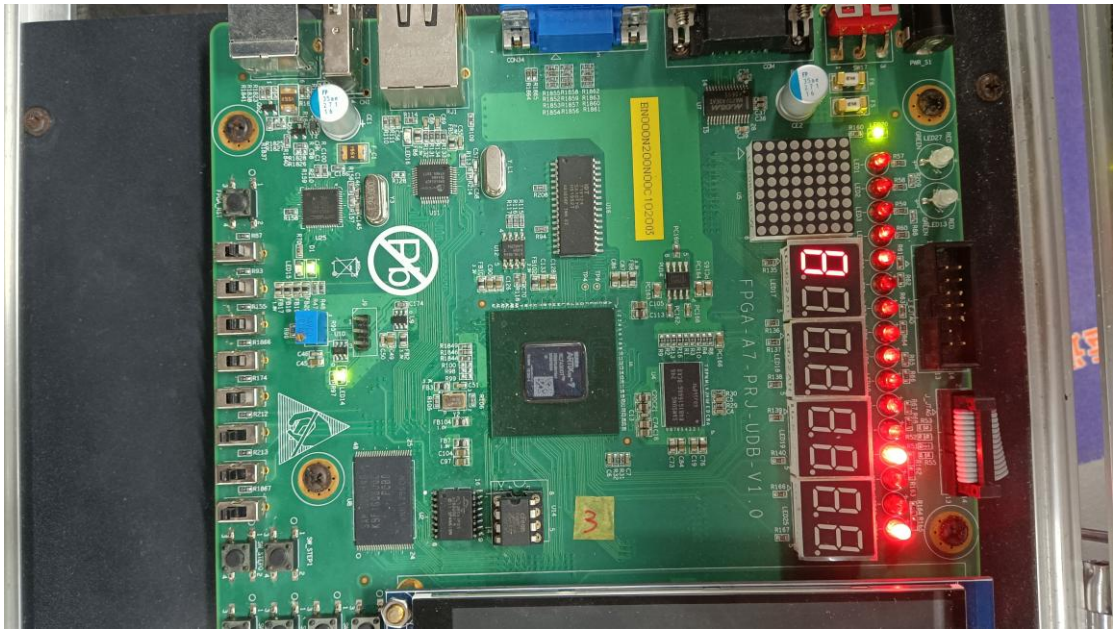

FPGA-A7-PRJ-UDB-V1.0

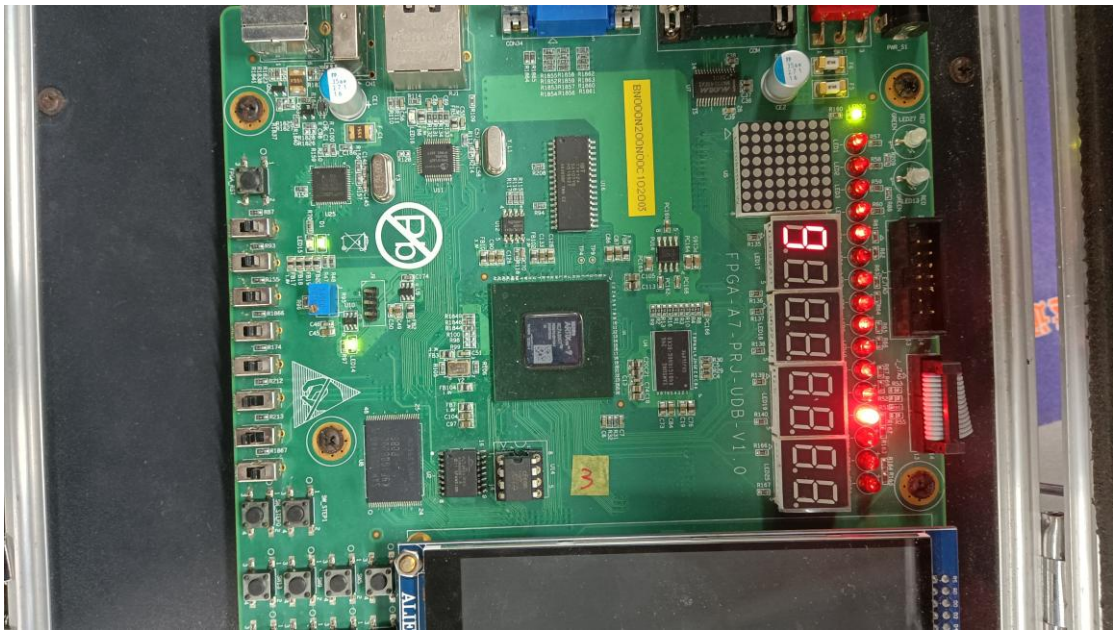

FPGA-A7-PRJ-UDB-V1.0


FPGA-A7-PRJ-UDB-V1.0


FPGA-A7-PRJ-UDB-V1.0


FPGA-A7-PRJ-UDB-V1.0





总结感想

1. 为什么寄存器堆要设置成两读一写

实践任务一中寄存器堆接口信号列表如下：

表 3-1　寄存器堆接口信号列表

名称	宽度	方向	描述
时钟			
clk	1	input	时钟信号
读端口一			
raddr1	5	input	寄存器堆读地址 1
rdata1	32	output	寄存器堆读返回数据 1
读端口二			
raddr2	5	input	寄存器堆读地址 2
rdata2	32	output	寄存器堆读返回数据 2
写端口			
we	1	input	寄存器堆写使能
waddr	5	input	寄存器堆写地址
wdata	32	input	寄存器堆写数据

（1）满足指令执行需求

在大多数指令集架构中，一条指令往往需要读取两个操作数，然后将运算结果写回。设置两读一写端口能够让处理器在一个时钟周期内同时获取两条操作数，提高指令执行的并行度和效率，减少指令执行的时钟周期数，从而提升整个系统的性能。

（2）配合数据通路设计

设置两个读端口可以方便地将寄存器中的数据同时送往不同的运算部件，如算术逻辑单元（ALU）、乘法器等，以支持各种复杂的运算操作。而一个写端口则用于将运算结果或其他需要更新的数据写回寄存器堆，保证数据的正确存储和更新。这样的设计能够使数据在寄存器堆与其他部件之间高效地流动，满足不同指令对数据读写的要求。

（3）平衡硬件成本和性能

从硬件实现的角度来看，增加读端口和写端口的数量会增加寄存器堆的复杂度和硬件成本。如果设置更多的读端口和写端口，虽然可以进一步提高数据读写的并行性，但会导致硬件面积增大、功耗增加以及设计复杂度提高；而设置更少的读端口和写端口，则无法满足指令执行的并行性要求，会降低处理器的性能。

（4）兼容性和扩展性

这种设计具有较好的兼容性和扩展性。对于现有的指令集架构，两读一写的寄存器堆能够很好地支持其指令的执行。同时，在对处理器进行升级或扩展时，也可以在不改变基本架构的前提下，通过增加寄存器数量或优化读写端口的控制逻辑等方式来满足新的性能需求，有利于处理器架构的长期发展和演进。

2.同步 ram 和异步 ram 的特点和区别

RAM（随机存取内存）主要分为两种类型：SRAM（静态随机存取内存）和 DRAM（动态随机存取内存），DRAM 又可细分为同步和异步两种类型：同步 RAM 与时钟信号同步运作，速度更快，但价格较高。异步 RAM 与时钟信号不同步，速度较慢，但成本低。

特点：

（1）同步 RAM

1.1 时钟同步：同步 RAM 的工作与时钟信号同步，所有的操作都在时钟信号的上升沿或下降沿进行。

1.2 高速性能：由于操作与时钟同步，数据的传输和处理可以在特定的时钟周期内完成，能够实现较高的数据传输速率。

1.3 易于控制：同步 RAM 的时序逻辑相对简单，通过时钟信号可以精确控制数据的输入、输出和存储，减少了数据竞争和不确定性，使得电路设计和调试更加容易。

（2）异步 RAM

1.1 无时钟同步：异步 RAM 的工作不依赖于时钟信号，数据的读写操作是由地址和控制信号的变化来触发的。它可以在任何时候进行数据的访问，具有较高的灵活性。

1.2 低功耗：由于不需要时钟信号来驱动，异步 RAM 在空闲状态下可以处于低功耗模式，只有在有数据读写请求时才会消耗能量，因此在一些对功耗要求较高的设备中得到广泛应用，如便携式电子设备。

1.3 简单接口：异步 RAM 的接口相对简单，只需要地址线、数据线和一些控制信号线即可完成数据的读写操作，不需要复杂的时钟同步电路，这使得它与一些简单的微控制器或其他异步设备连接时更加方便。

区别：

（1）时序控制：同步 RAM 依赖时钟信号进行同步控制，数据的读写在时钟的特定边沿进行；而异步 RAM 则根据地址和控制信号的变化来异步地进行数据操作，没有固定的时钟节拍。

（2）速度：一般来说，同步 RAM 的速度相对较快，能够在短时间内完成多次数据传输，适用于高速数据处理的场景；异步 RAM 的速度相对较慢，但其读写操作的延迟相对固定，适用于对速度要求不高，但对灵活性和功耗要求较高的场合。

（3）电路复杂度：同步 RAM 需要时钟产生电路和复杂的时序逻辑电路来确保数据的正确读写，因此电路复杂度较高；异步 RAM 不需要时钟电路，其电路结构相对简单，成本也较低。

3.使用 vivado 调试的过程

1.观察仿真波形

1.1 熟悉待调试的设计

1.2 找到一个能明确的错误点

1.3 沿着设计额逻辑链条逆向逐级查看信号，直至找到出错源头

在追查含有时序逻辑的电路波形时，首先要把需要观察的电路所用的时钟信号抓取出来，再明确所观察的时序器件（如触发器、同步 RAM）是用时钟上升沿还是下降沿来触发，从而找到错误值写入的真正时刻。

2.波形分析技巧

2.1 一次仿真记录所有信号的数据

2.2 给重要的时刻做标记

2.3 熟练使用波形缩小和放大功能

2.4 对关联信号进行分割、分组

2.5 用值查找快速定位多位宽的信号

3.波形异常的错误的调试

——3.1 信号为 Z

Z 表示高阻,比如电路断路就会显示为高阻,这种错误往往是以下两个原因导致的:RTL 里声明为 wire 型的变量从未被赋值；模块调用的信号未连接导致信号悬空

（1）编写 RTL 时要注意代码规范，特别是模块调用时，要按照接口顺序一一对应

（2）所有 input 类接口被调用时不允许悬空

（3）一旦发现一个信号为 Z 应向前追溯产生该信号的因子信号，看是哪个信号为 Z,一直追踪到该模块里的 input 接口，随后进行修正

（4）可能 Z 只出现在向量信号里的某几位上，这时也采用同样的追溯方式。调用时某个接口存在宽度不匹配,也会造成该接口上某些位为 Z

——3.2 信号为 X

X 表示不定值,这种错误往往是以下两个原因之一导致的:

1)RTL 里声明为 reg 型的变量从未被赋值。

2)RTL 里多驱动的代码有时候也可能导致这种类型的错误，有些多驱动的代码不会导致 X 因为有些多驱动代码可能会被 Vivado 自动处理，但这种情况其实是有风险的;有些多驱动代码则会导致综合时失败,并且会明确报出多驱动的错误。

针对信号为 X 情况,我们有以下几点建议:

1）一旦发现仿真错误来自某个出现“X”的信号,则向前追溯产生该信号的因子信号,看是哪个信号为 X 一直追溯到某个信号未赋值，随后修正

2）如果因子信号都没有为 X 的,则可能是多驱动导致的。此时先进行综合，然后排查 Error 和 Criticalwarning

3）寄存器信号如果没有复位值,在复位阶段其值可能也为 X 但这种情况可能不会带来错误

4）X 和 1 进行或运算结果为 1,X 和 0 进行与运算结果为 0。

——3.3 波形停止

波形停止是指仿真再某一个时刻停止，再也无法前进分毫，而仿真却显示仍然在运行，这种错误是 RTL 里存在组合环路导致的

并不是所有的组合环路都会导致波形停止，有些复杂的组合环路（比如跨多个模块形成的组合环路）可能会被工具自动处理,但这种处理是有风险的,可能导致仿真通过，上板不过。

所谓组合环路，是指信号 A 的组合逻辑表达式中某个产生因子为 B，而 B 的组合逻辑表达式中又用到了信号 A，出现波形停止时，排查哪部分代码出现组合环路并不容易,我们建议按以下步骤处理:

1）一旦发现波形停止，就先对设计进行综合。

2）查看综合产生的 errorr 和 Criticalwarning 提示，并尝试修正组合环路

——3.4 越沿采样

即上升沿采样到被采样数据在上升沿后的值。造成这一现象更深层的原因是 Verilog 里阻塞赋值=和非阻塞赋值<=混用

针对越沿采样,我们有以下几点建议:

1)编写 RTL 时注意代码规范,所有 always 写的时序逻辑只允许采用非阻塞赋值

2）一旦发现越沿采样的情况,追溯被采样信号，直到追溯到某一个阻塞赋值的信号，随后进行修正

——3.5 波形怪异

即仿真波形图显示怪异，这是与设计的电路功能无关的错。当出现波形怪异类的错误时，需要区分是仿真工具出错还是 RTL 代码出错。