

# 组成原理课程第三次实报告

## 实验名称：CPU 设计实战三次实践

学号：2312141 姓名：张德民 班次：李涛老师班

### 一、 实验目的

根据《CPU 设计实战》书中的第三章讲解，完成 Lab2 的三个实验，深入认识寄存器堆，了解同步 RAM 与异步 RAM，并且学习使用 vivado 进行调试代码，发现 bug。

### 二、 实验内容说明

请根据《CPU 设计实战》书中的第三章讲解，完成 Lab2 的三个实验，并撰写实验报告。

- 1、针对任务一寄存器堆实验，完成仿真，在感想收获中思考并回答问题：为什么寄存器堆要设计成“两读一写”？
- 2、针对任务二同步 ram 和异步 ram 实验，可以参考实验指导手册中的存储器实验，注意同步和异步需要分开建工程，然后仿真，在感想收获中分析同步 ram 和异步 ram 各自的特点和区别。
- 3、针对任务三，重点介绍清楚发现 bug、修改 bug 和验证的过程，在感想收获中总结使用 vivado 调试的经验步骤。

### 三、 实验步骤

#### 任务一：寄存器堆仿真实验

代码分析：由于代码过多，因此主要分析核心的部分。

## regfile 模块的实现:

```
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
```

这个模块定义了两个读地址 `raddr1` 和 `raddr2`，用于规定要读取的寄存器，寄存器是 32 个，所以读地址是 5 位， $2^5=32$ 。还用两个读数据堆，`rdata1` 和 `rdata2`，用于存储从寄存器里读到的数据。`we`(write enable)是写使能信号，只有当其为 1 时，才能向寄存器中写入。`waddr` 用于确定向哪一个寄存器写入，`wdata` 用于确定写入什么数据。然后是 32 个 32 位的寄存器。

之后使用 `always` 语句，使得在时钟上升沿，如果 `we` 为 1，那么就向指定寄存器写入指定数据。

然后读取数据，规定若 `raddr` 为 0，则读取 0，否则读取相应寄存器的值。

## 任务二：同步 RAM 与异步 RAM 仿真

### 同步 RAM 仿真实验

#### block\_ram\_top 模块的实现

```
module ram_top (
    input      clk,
    input  [15:0] ram_addr,
    input  [31:0] ram_wdata,
    input      ram_wen,
    output [31:0] ram_rdata
);

    block_ram block_ram (
        .clka (clk),
        .wea  (ram_wen),
        .addra (ram_addr),
        .dina  (ram_wdata),
        .douta (ram_rdata)
    );
endmodule
```

这个模块有四个输入端口，clk 是时钟端口，ram\_addr 是 ram 寄存器存取的地址，ram\_wdata 是写入的数据，ram\_rdata 用来存储读出的数据，ram\_wen 是使能信号，为 0 时代表读，为 1 时代表写。

并且为了保持一致性，所有的读写操作都是在时钟上升沿进行的。

## 异步 RAM 仿真实验

### distributed\_ram\_top 模块的实现

```
module ram_top (  
    input      clk      ,  
    input  [15:0] ram_addr ,  
    input  [31:0] ram_wdata,  
    input      ram_wen  ,  
    output [31:0] ram_rdata  
);  
  
distributed_ram distributed_ram(  
    .clk (clk      ),  
    .we  (ram_wen  ),  
    .a   (ram_addr ),  
    .d   (ram_wdata ),  
    .spo (ram_rdata )  
);  
  
endmodule
```

和同步 ram 的代码基本一致，只有名称换了。核心的修改点在于使用的 RAM IP 不同，异步 RAM 与同步 RAM 不同，它进行读写操作时不需要等到时钟上升沿，而是立即进行的。同时也没有写后读的机制。

## 任务三：数字逻辑电路的设计与调试

这里有五个 bug，我们根据书中的提示了一个一个寻找。

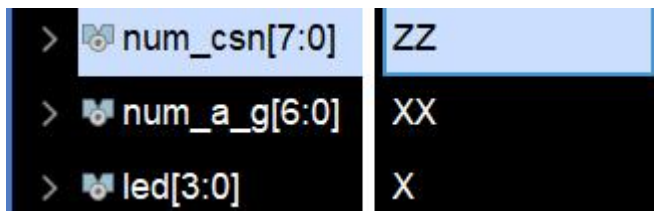
## Bug1: 波形为 Z

```
show_sw u_show_sw(  
    .clk      (clk      ),  
    .resetn   (resetn  ),  
  
    .switch   (switch ),    //input  
  
    .num_csn(),    //new value |  
    .num_a_g(),  
  
    .led      ()    //previous value  
);
```

我发现，问题是在 tb 代码里调用模块时，部分端口没有定义，也没有连接。要进行修改。

```
wire [7:0] num_csn;  
wire [6:0] num_a_g;  
wire [3:0] led; |
```

我们在 tb 里加入这些定义，并连接。



> num_csn[7:0]	ZZ
> num_a_g[6:0]	XX
> led[3:0]	X

波形图显示了这三个值，但是 num\_csn 的值是 Z,应该没有连接，而 num\_a\_g 和 led 是 X 不定值，应该没有进行赋值。

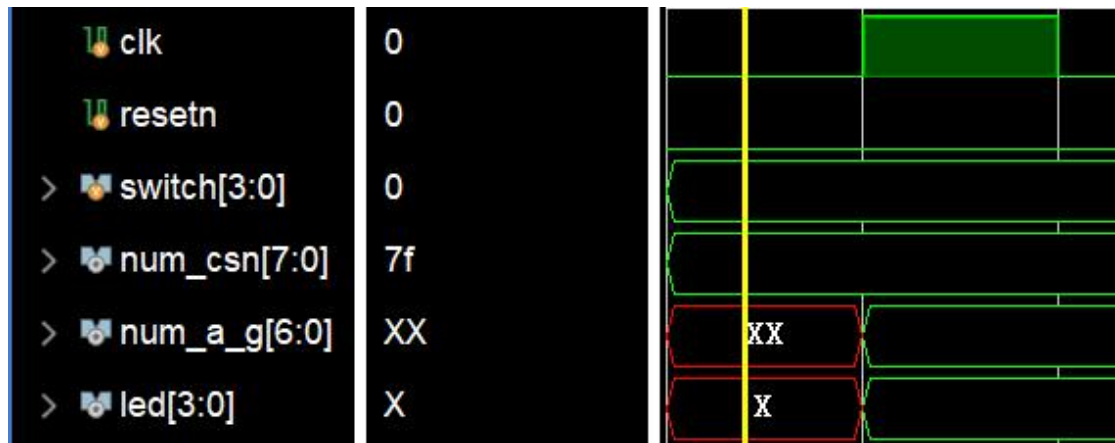
```
show_num u_show_num(  
    .clk      (clk      ),  
    .resetn   (resetn  ),  
  
    .show_data (show_data),  
    .num_csn   (num_scn ),  
    .num_a_g   (num_a_g )
```

这里我们发现，num\_csn 在连接的时候拼错了，这应该是导致 Z 值的原因，我们修改回

来。

> num_csn[7:0]	7f
> num_a_g[6:0]	XX
> led[3:0]	X

可以看见，这样 num\_csn 就正常了。但是 num\_a\_g 和 led 还是不定值，这是因为它们最初还没有被赋值。



但是可以看见，在第一个时钟上升沿后，二者就赋值了，因此这不算 bug，没有必要进行赋值修改。

## Bug2: 波形为 X

```
reg      clk      ;
reg      resetn;
reg [3 :0] switch;  //input

initial
begin
    #100;
    clk      = 1'b0;
    resetn = 1'b0;
```

产生的原因是 reg 类型的端口 clk, resetn 以及 switch 最初都没有赋值，clk 和 resetn 在 100ns 后赋值为 0，但是 switch 一直没有，这就导致了这三个值一直是 x 不定值。

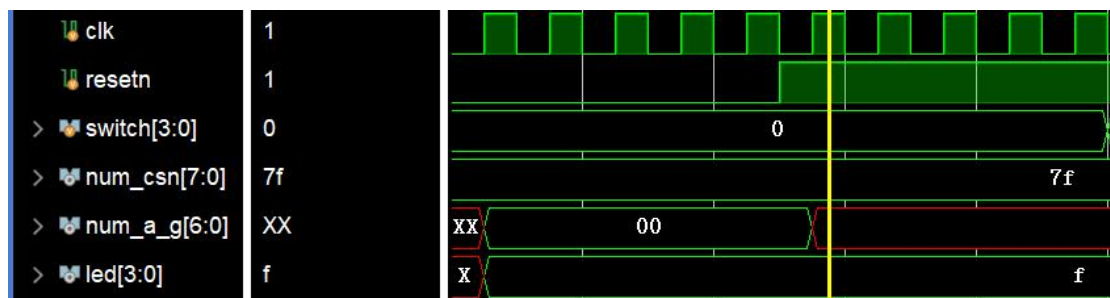
```
begin
    clk    = 1'b0;
    resetn = 1'b0;
    switch = 4'h0;
```

修改方法是删除等待 100ns，并给 switch 赋值为 0。



这样处理过后，最初就没有 x 的问题了。

## Bug3: 波形停止



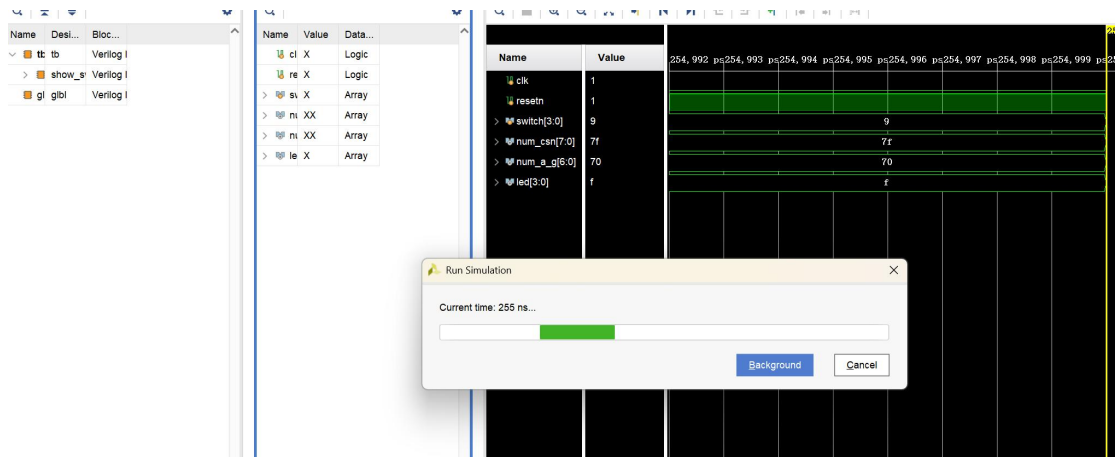
我们观察到当 resetn 变成 1 时，num\_a\_g 一直是 X，这出现了问题。因此我们去查看 num\_a\_g 的代码。

```
begin
    if ( !resetn )
    begin
        num_a_g <= 7'b0000000;
    end
    else
    begin
        num_a_g <= nxt_a_g;
    end
end

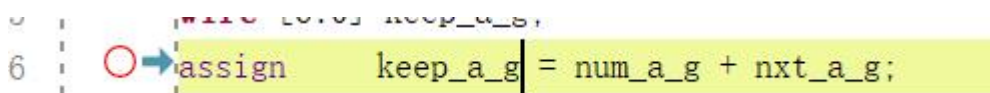
assign nxt_a_g = show_data==4'd0 ? 7'b1111110 : //0
                 show_data==4'd1 ? 7'b0110000 : //1
                 show_data==4'd2 ? 7'b1101101 : //2
                 show_data==4'd3 ? 7'b1111001 : //3
                 show_data==4'd4 ? 7'b0110011 : //4
                 show_data==4'd5 ? 7'b1011011 : //5
                 show_data==4'd6 ? 7'b1110000 : //6
                 show_data==4'd7 ? 7'b1110000 : //7
                 show_data==4'd8 ? 7'b1111111 : //8
                 show_data==4'd9 ? 7'b1111011 : //9
                 keep_a_g      ;

// show_data <= ~switch;
```

我们发现 num\_a\_g 的值由 nxt\_a\_g 决定, 而 nxt\_a\_g 的值由 show\_data 决定, 而 show\_data 的赋值语句被注释掉了, 我们去除注释再仿真。

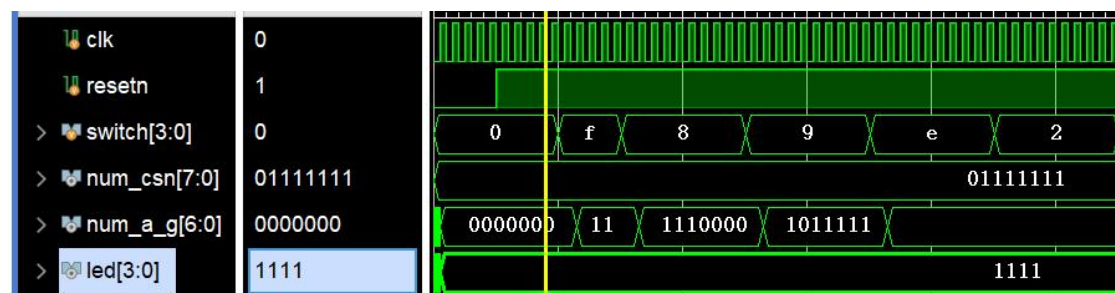


这时我发现 num\_a\_g 不再是 X, 但是出现了波形停止的 bug。我进行综合来 debug。这种错误往往是 RTL 里存在组合环路导致的。



系统显示这里有问题, 我分析后发现, 这一步的目的当 show\_data 大于 9 的时候, 显示之前的数据不变, 因此这句的逻辑就不正确, 只需要保留一个 num\_a\_g 就可以了。原来的代码里 nxt\_a\_g 表达式里有 keep\_a\_g, 二者相互包含, 因此引发了波形停止的 bug。修改之后, 仿真恢复正常。

## Bug4: 越前采样



我发现, led 的值全程都是 1111 没有改变。因此我去反向追溯。

```
assign led = ~prev_data;
```

我发现 led 由 prev\_data 决定,

```
prev_data <= show_data_r;
```

prev\_data 由 show\_data\_r 决定,

```
show_data_r = show_data;
```

show\_data\_r 由 show\_data 决定, 这里我发现了问题, 没有使用非阻塞延迟。修改后 led 正常变化。



## Bug5: 功能 bug

对于 `nxt_a_g` 的赋值，缺少了 6，我们将其添加上去。

```
show_data==4'd6 ? 7'b1011111 :
```

这样程序就正常了。

## 四、 实验结果分析

### 任务一：寄存器堆仿真实验

本次结果分析是基于 `testbench` 代码设定的数据。

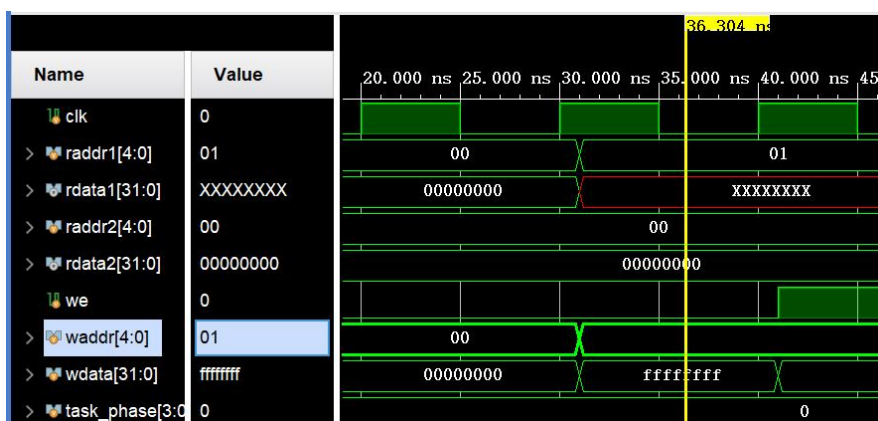
```
initial
begin
    raddr1 = 5'd0;
    raddr2 = 5'd0;
    waddr  = 5'd0;
    wdata  = 32'd0;
    we     = 1'd0;
    task_phase = 4'd0;
```

首先，所有值先初始化为 0。

```
// Part 0 Begin
#10;
task_phase = 4'd0;
we         = 1'b0;
waddr      = 5'd1;
wdata      = 32'hffffffff;
raddr1     = 5'd1;
#10;
we         = 1'b1;
waddr      = 5'd1;
wdata      = 32'h1111ffff;
#10;
we         = 1'b0;
raddr1     = 5'd2;
raddr2     = 5'd1;
#10;
raddr1     = 5'd1;
```

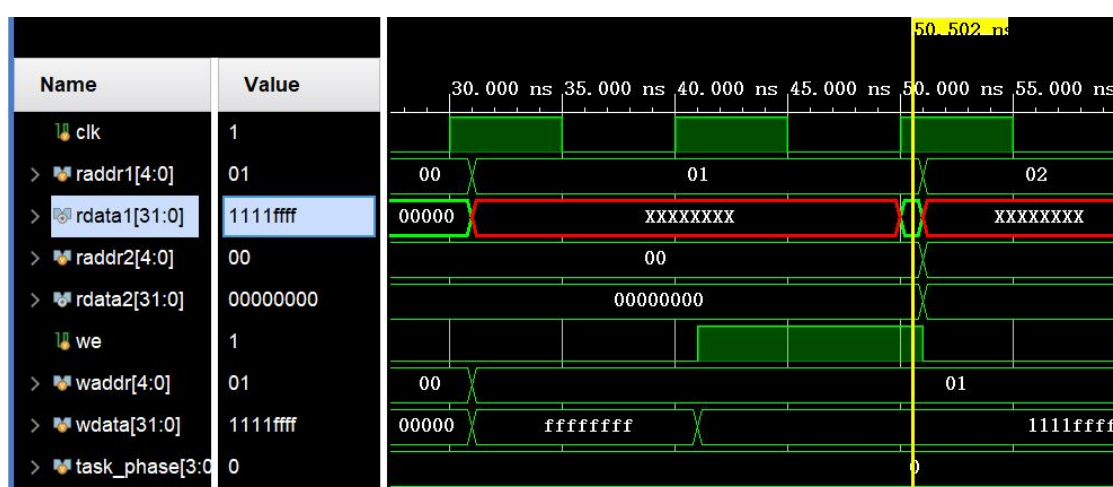
然后开始第 0 部分，首先要将 `waddr` 赋值为 1，`wdata` 赋值为 `0xffffffff`，`raddr1` 赋值为 1，意味着向 1 号寄存器写入数据 `0xffffffff`，1 号堆要读取的寄存器是 1 号，不过由于 `we` 为 0，所以暂时还没有进行写入。





如图，此时 waddr 为 1，wdata 为 0xffffffff，1 号堆读取了 1 号寄存器，但是由于还没有写入，所以是 X 不定值。

接下来 we 赋值为 1，waddr 赋值为 1，wdata 赋值为 0x1111ffff。



可以看见，we，waddr，wdata 的赋值同时进行，因为读取操作只有当时钟上升沿才会触发，所以当在一个时钟上升沿时，rdata1 变成了寄存器 1 的值 1111ffff。

然后将 we 赋值为 0，raddr1 赋值为 2，raddr2 赋值为 1，即 1 号堆要存储 2 号寄存器的值，2 号堆存储 1 号寄存器的值。



如图，rdata1 变成了不定值，rdata2 变成了 1111ffff。

再之后又把 addr1 改为 1 号寄存器。所以 1 号堆也要存 1111ffff。



接下来开始第 1 部分：

```
#10;
task_phase = 4'd1;
we        = 1'b1;
wdata     = 32'h0000ffff;
waddr     = 5'h10;
raddr1    = 5'h10;
raddr2    = 5'h0f;
#10;
wdata     = 32'h1111ffff;
waddr     = 5'h11;
raddr1    = 5'h11;
raddr2    = 5'h10;
#10;
wdata     = 32'h2222ffff;
waddr     = 5'h12;
raddr1    = 5'h12;
raddr2    = 5'h11;
#10;
wdata     = 32'h3333ffff;
waddr     = 5'h13;
raddr1    = 5'h13;
raddr2    = 5'h12;
#10;
wdata     = 32'h4444ffff;
waddr     = 5'h14;
raddr1    = 5'h14;
raddr2    = 5'h13;
#10;
raddr1    = 5'h15;
raddr2    = 5'h14;
#10;
```

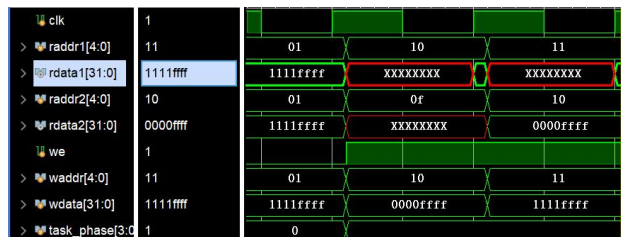
首先向 16 号寄存器写入 0x0000ffff，然后让 1 号堆读取 16 号寄存器，2 号堆读取 15 号寄存器。



此时一个上升沿以后，1 号堆存入了 16 号寄存器的值 0000ffff，而由于 15 号寄存器没

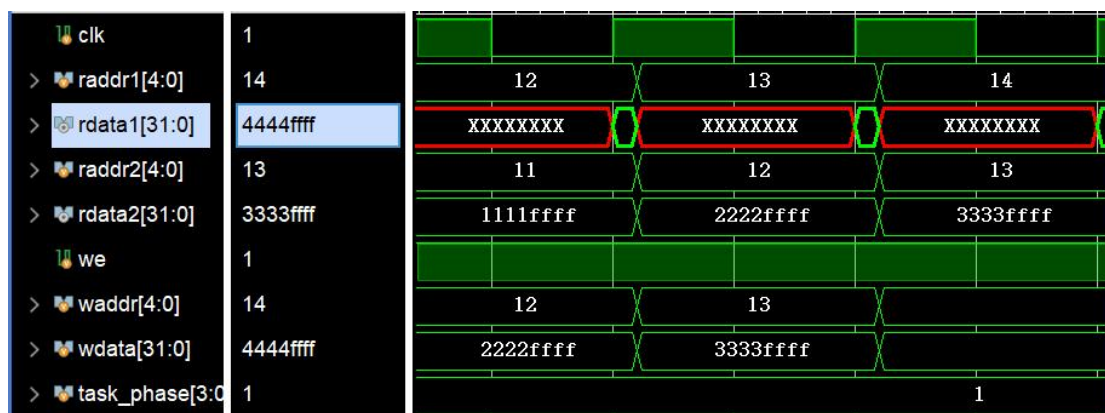
有写入，所以 2 号堆存入的是不定值 X。

然后向 17 号寄存器写入 0x1111ffff，然后让 1 号堆读取 17 号寄存器，2 号堆读取 16 号寄存器。



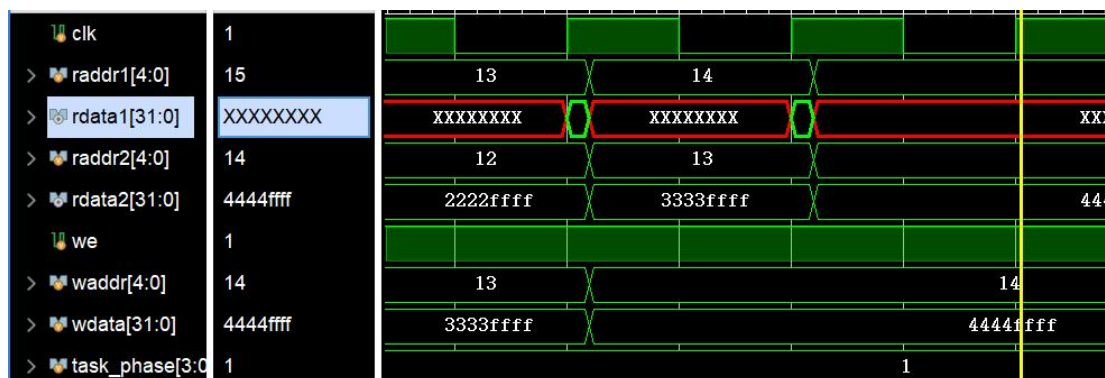
可见，此时 1 号堆存入了 17 号寄存器的值 1111ffff，2 号堆存入了 16 号寄存器的值 0000ffff。

之后由按照这种格式往后类推，到向 20 号寄存器写入 0x4444ffff，然后让 1 号堆读取 20 号寄存器，2 号堆读取 19 号寄存器。



结果与之前的类似。

然后，raddr1 改为 21 号寄存器，raddr2 改为 20 号寄存器，这是 21 号寄存器没有写入，所以 1 号堆存入不定值 X。



可见，1 号堆为不定值 X，2 号堆是 20 号寄存器的值 4444ffff。

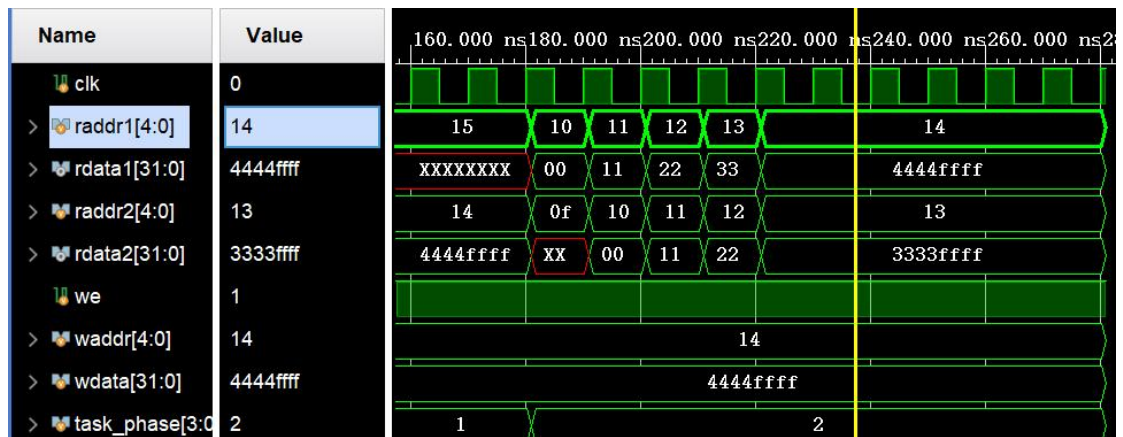
最后开始第 2 部分：

```

#10;
task_phase = 4'd2;
we         = 1'b1;
raddr1     = 5'h10;
raddr2     = 5'h0f;
#10;
raddr1     = 5'h11;
raddr2     = 5'h10;
#10;
raddr1     = 5'h12;
raddr2     = 5'h11;
#10;
raddr1     = 5'h13;
raddr2     = 5'h12;
#10;
raddr1     = 5'h14;
raddr2     = 5'h13;

```

首先，让 1 号堆读取 16 号寄存器，2 号堆读 15 号，之后让 1 号堆读取 17 号寄存器，2 号堆读 16 号，让 1 号堆读取 18 号寄存器，2 号堆读 17 号，让 1 号堆读取 19 号寄存器，2 号堆读 18 号，让 1 号堆读取 20 号寄存器，2 号堆读 19 号。除了 15 号寄存器没有写入，其余寄存器都是之前已经写入的。



如上图，这里读取的数据和第 1 部分写入的一致，与我们分析的一致，实验正确。

## 任务二：同步 RAM 与异步 RAM 仿真

### 同步 RAM 仿真实验

```

ram_addr   = 16'd0;
ram_wdata  = 32'd0;
ram_wen    = 1'd0;
task_phase = 4'd0;

```

首先所有端口初始化为 0。

```

task_phase = 4'd0;
ram_wen    = 1'b0;
ram_addr   = 16'hf0;
ram_wdata   = 32'hffffffff;
#10;
ram_wen    = 1'b1;
ram_addr   = 16'hf0;
ram_wdata   = 32'h11223344;
#10;
ram_wen    = 1'b0;
ram_addr   = 16'hf1;
#10;
ram_wen    = 1'b0;
ram_addr   = 16'hf0;

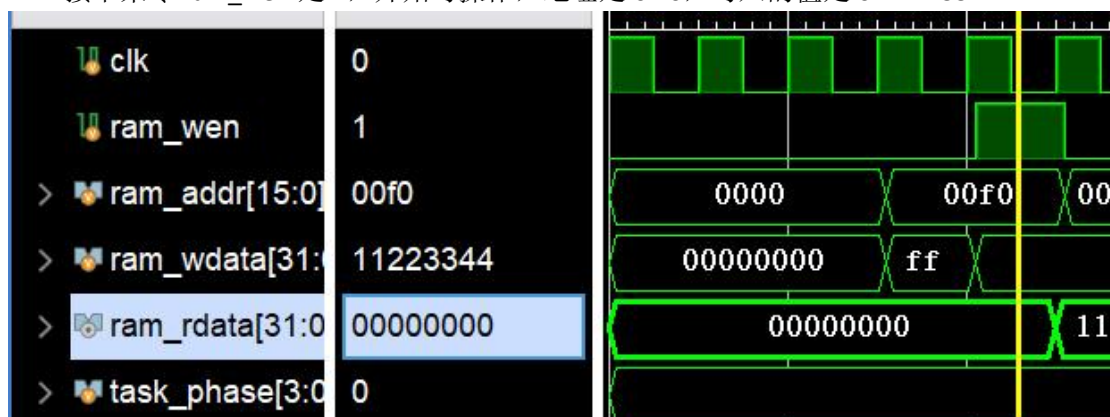
```

开始第0部分，先指定 ram 中的地址是 f0，然后写入的数据是 0xffffffff。ram\_wen 是 0，代表此时进行读操作，因此无法写入。



如上图，各个端口数据成功修改。此时 ram\_rdata 是全零，因为还没有写入。

接下来令 ram\_wen 是 1，开始写操作，地址是 0xf0，写入的值是 0x11223344.





如图，ram\_wen 为 1，写入的数据是 11223344。因为还没有读取，所有 ram\_rdata 是全 0。

接下来令 ram\_wen 是 0，开始读操作，然后修改读取地址为 0xf1。这个地址还没有写入，因此应该是全 0。

clk	1
ram_wen	0
> ram_addr[15:0]	00f1
> ram_wdata[31:0]	11223344
> ram_rdata[31:0]	00000000
> task_phase[3:0]	0

如图，与分析的一致。

接下来继续读取，地址改为 0xf0，这是各个写入 0x11223344 的地址。

clk	1
ram_wen	0
> ram_addr[15:0]	00f0
> ram_wdata[31:0]	11223344
> ram_rdata[31:0]	11223344
> task_phase[3:0]	0



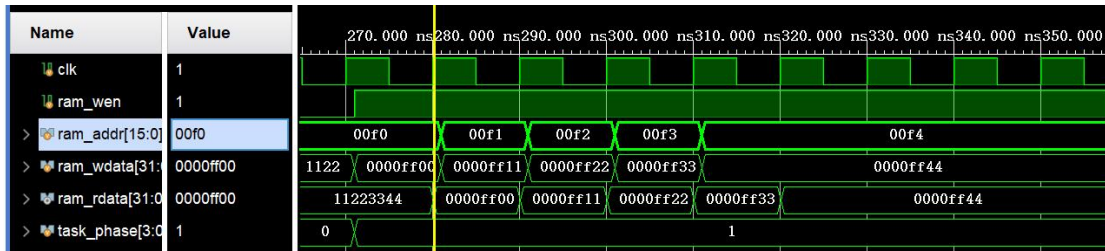
如图，读取的是我们预估的值，正确。

然后开始第 1 部分。

```
task_phase = 4'd1;
ram_wen    = 1'b1;
ram_wdata   = 32'hff00;
ram_addr    = 16'hf0;
#10;
ram_wdata   = 32'hff11;
ram_addr    = 16'hf1;
#10;
ram_wdata   = 32'hff22;
ram_addr    = 16'hf2;
#10;
ram_wdata   = 32'hff33;
ram_addr    = 16'hf3;
#10;
ram_wdata   = 32'hff44;
ram_addr    = 16'hf4;
```

这里进行了连续的写操作，在 0xf0 写入 0xff00，在 0xf1 写入 0xff11，在 0xf2 写入 0xff22，

在 0xf3 写入 0xff33，在 0xf4 写入 0xff44。

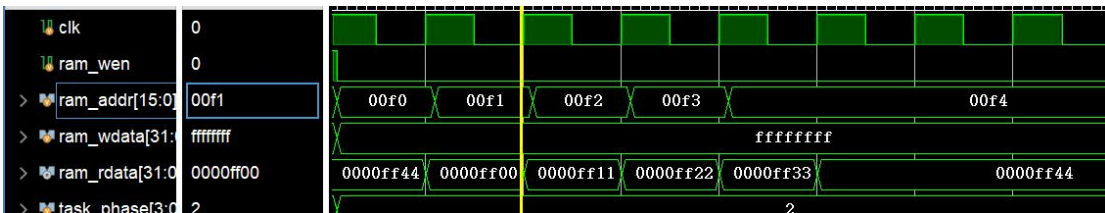


如上图，写入的顺序与预估的基本一致，但是需要注意的是，我们的 RAM 默认设定的是写后读模式，即在时钟上升沿触发时，ram\_rdata 会直接保存此时 ram\_addr 写入的值。这与波形图一致。

接下来开始第 2 阶段。

```
task_phase = 4'd2;  
ram_wen    = 1'b0;  
ram_addr   = 16'hf0;  
ram_wdata  = 32'hffffffff;  
#10;  
ram_addr   = 16'hf1;  
#10;  
ram_addr   = 16'hf2;  
#10;  
ram_addr   = 16'hf3;  
#10;  
ram_addr   = 16'hf4;  
#10;
```

此时开始了连续读操作，ram\_wen 赋值为 0，写入的数据改为 0xffffffff，当然并不会真的进行写入。开始连续读取 0xf0,0xf1,xf02,xf03,xf4 五个地址的值。



如图，每当时钟上升沿的时候，ram\_rdata 就会存入读取到的值，波形图的值与我们刚刚写入的值一致。

## 异步 RAM 仿真实验



```

ram_addr  = 16'd0;
ram_wdata = 32'd0;
ram_wen   = 1'd0;
task_phase = 4'd0;

```

首先全部初始化为 0。

```

task_phase = 4'd0;
ram_wen    = 1'b0;
ram_addr   = 16'hf0;
ram_wdata  = 32'hffffffff;
#10;
ram_wen    = 1'b1;
ram_addr   = 16'hf0;
ram_wdata  = 32'h11223344;
#10;
ram_wen    = 1'b0;
ram_addr   = 16'hf1;
#10;
ram_wen    = 1'b0;
ram_addr   = 16'hf0;

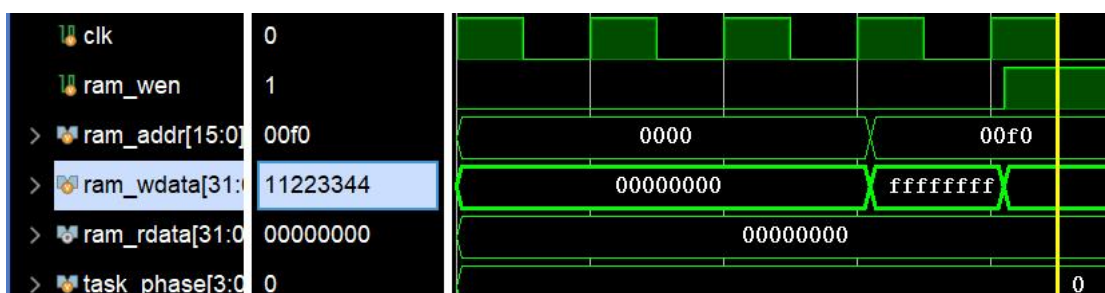
```

开始第 0 部分，先指定 ram 中的地址是 f0，然后写入的数据是 0xffffffff。ram\_wen 是 0，代表此时进行读操作，所以无法写入。



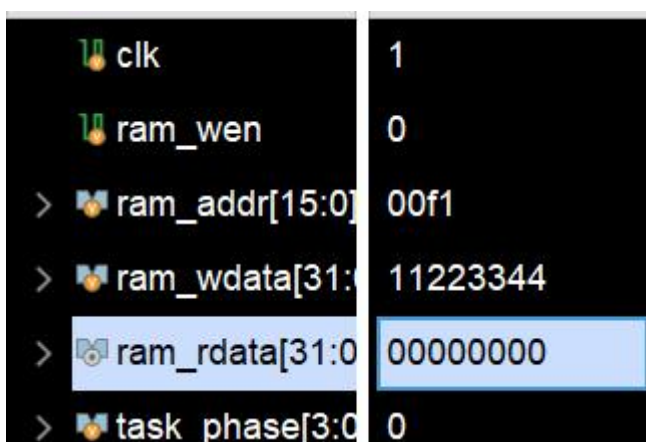
如上图，各个端口数据成功修改。此时 ram\_rdata 是全零，因为还没有写入。

接下来令 ram\_wen 是 1，开始写操作，地址是 0xf0，写入的值是 0x11223344.

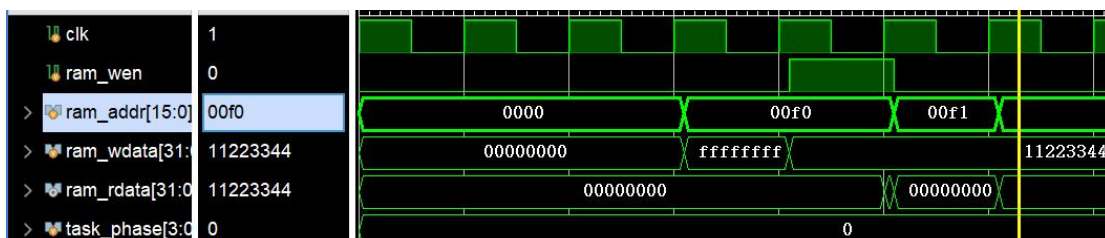


如图，ram\_wen 为 1，写入的数据是 11223344。因为还没有读取，所有 ram\_rdata 是全 0。

接下来令 ram\_wen 是 0，开始读操作，然后修改读取地址为 0xf1。这个地址还没有写入，因此应该是全 0。



接下来继续读取，地址改为 0xf0，这是各个写入 0x11223344 的地址。



可以看见，此时的 ram\_rdata 是我们写入的数据，但是与之前不同的一点是，它并没有等到时钟上升沿才进行读取，而是在使能端口改变的瞬间就开始读取。

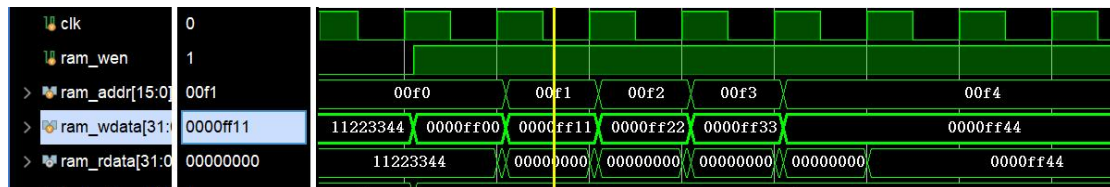
然后开始第 1 部分。

```

task_phase = 4'd1;
ram_wen    = 1'b1;
ram_wdata   = 32'hff00;
ram_addr    = 16'hf0;
#10;
ram_wdata   = 32'hff11;
ram_addr    = 16'hf1;
#10;
ram_wdata   = 32'hff22;
ram_addr    = 16'hf2;
#10;
ram_wdata   = 32'hff33;
ram_addr    = 16'hf3;
#10;
ram_wdata   = 32'hff44;
ram_addr    = 16'hf4;

```

这里进行了连续的写操作，在 0xf0 写入 0xff00，在 0xf1 写入 0xff11，在 0xf2 写入 0xff22，在 0xf3 写入 0xff33，在 0xf4 写入 0xff44。



如上图，写入的顺序与预估的基本一致，但是需要注意的是，我们的异步 RAM 没有默认设定的写后读模式，因此 ram\_rdata 一致保持全 0 不变。这与波形图一致。

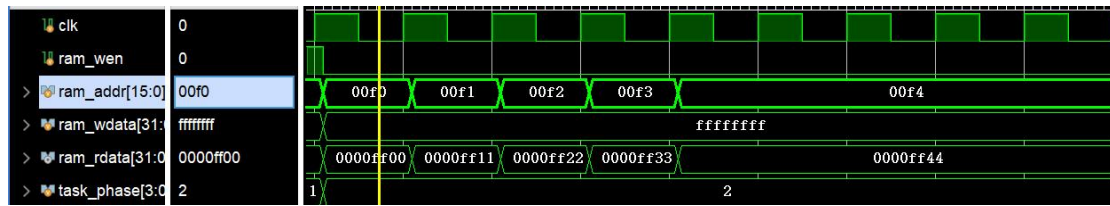
最后是第 2 阶段。

```

task_phase = 4'd2;
ram_wen    = 1'b0;
ram_addr    = 16'hf0;
ram_wdata   = 32'hffffffff;
#10;
ram_addr    = 16'hf1;
#10;
ram_addr    = 16'hf2;
#10;
ram_addr    = 16'hf3;
#10;
ram_addr    = 16'hf4;
#10;

```

此时开始了连续读操作，ram\_wen 赋值为 0，写入的数据改为 0xffffffff，当然并不会真的进行写入。开始连续读取 0xf0,0xf1,0xf2,0xf3,0xf4 五个地址的值。



如图，与同步 ram 不同的是，一旦地址修改，不需要等到时钟上升沿，ram\_rdata 就会存入读取到的值，波形图的值与我们刚刚写入的值一致。

### 任务三：数字逻辑电路的设计与调试

```

    clk    = 1'b0;
    resetn = 1'b0;
    switch = 4'h0;

    #50;
    resetn = 1'b1;

```

首先初始化，然后令 resetn 为 1，停止清零。

```
#100;
```

然后等了 100ns，由于原先 switch 为 0，取反后就是 f。

clk	1
resetn	0
switch[3:0]	0000
num_csn[7:0]	01111111
num_a_g[6:0]	00000000
led[3:0]	1111

此时 num\_a\_g 为 00000000, 不亮，led1111，代表不亮(这里的 led 灯为 0 时候亮，为 1 的时候不亮，这是因为 switch 和我们想输入的值是反的，所以 led 灯也是反的)。

```
switch = 4'hf;
```

clk	0
resetn	1
switch[3:0]	1111
num_csn[7:0]	01111111
num_a_g[6:0]	11111110
led[3:0]	0000

这时候 num\_a\_g 为 11111110, 代表 0，led0000，为之前的值 f。

```
switch = 4'h8; // ~switch: 1
```

接下来 switch 为 8，那么输入的值应该是 8 取反，也就是 7。

clk	0
resetn	1
> switch[3:0]	1000
> num_csn[7:0]	01111111
> num_a_g[6:0]	1110000
> led[3:0]	1111

这时候 num\_a\_g 为 1110000,代表 7，led1111，为之前的值 0。

```
switch = 4'h9; //switch: 6
```

接下来 switch 为 9，那么输入的值应该是 9 取反，也就是 6。

clk	0
resetn	1
> switch[3:0]	1001
> num_csn[7:0]	01111111
> num_a_g[6:0]	1011111
> led[3:0]	1000

这时候 num\_a\_g 为 1011111,代表 6，led1000，为之前的值 7。

```
switch = 4'he; //switch: 1
```

接下来 switch 为 e，那么输入的值应该是 e 取反，也就是 1。

clk	1
resetn	1
> switch[3:0]	1110
> num_csn[7:0]	01111111
> num_a_g[6:0]	0110000
> led[3:0]	1001

这时候 num\_a\_g 为 0110000,代表 1，led1001，为之前的值 6。

```
switch = 4'h2; //switch: d
```



接下来 switch 为 2，那么输入的值应该是 2 取反，也就是 d。

clk	0
resetn	1
> switch[3:0]	0010
> num_csn[7:0]	01111111
> num_a_g[6:0]	0110000
> led[3:0]	1110

由于 d 大于 9，因此这时候 num\_a\_g 为 0110000,代表 1，不变，led1110，为之前的值 1。

```
switch = 4'h0; //switch: f
```

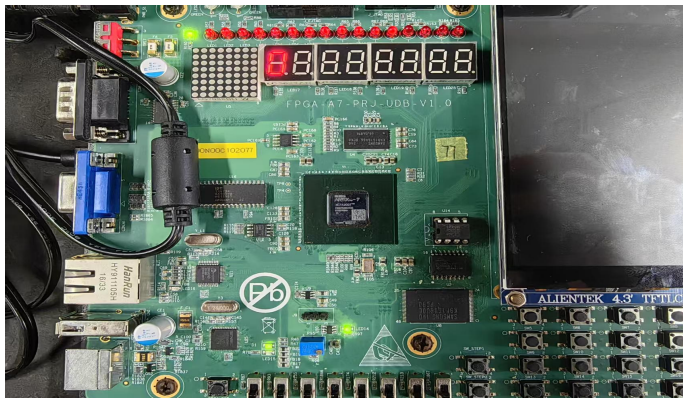
接下来 switch 为 0，那么输入的值应该是 0 取反，也就是 f。

clk	1
resetn	1
> switch[3:0]	0000
> num_csn[7:0]	01111111
> num_a_g[6:0]	0110000
> led[3:0]	0010

由于 f 大于 9，因此这时候 num\_a\_g 为 0110000,代表 1，不变，led1101，为之前的值 d。  
综上，验证成功，代码功能正确。

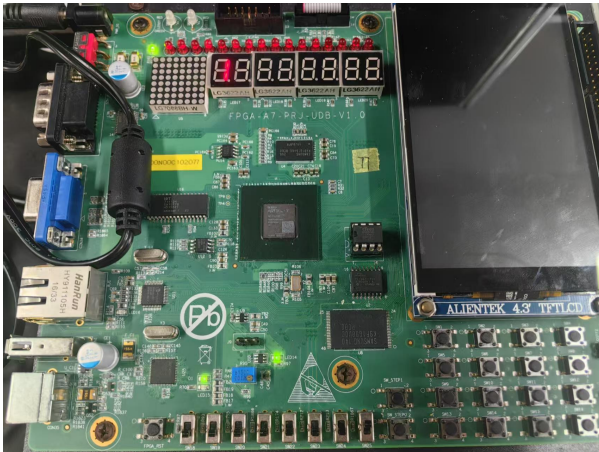
接下来使用实验箱来验证。

首先按下复位键：



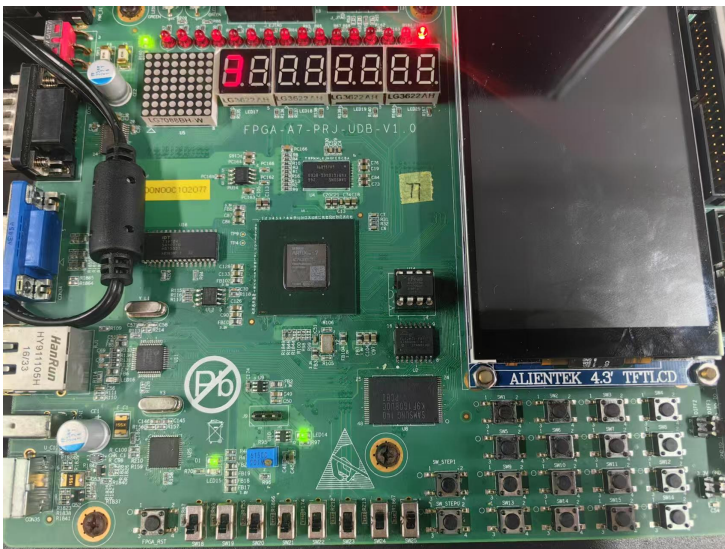
这时候 led 灯不亮，数码屏显示为 0.

接下来拨码调为 1:



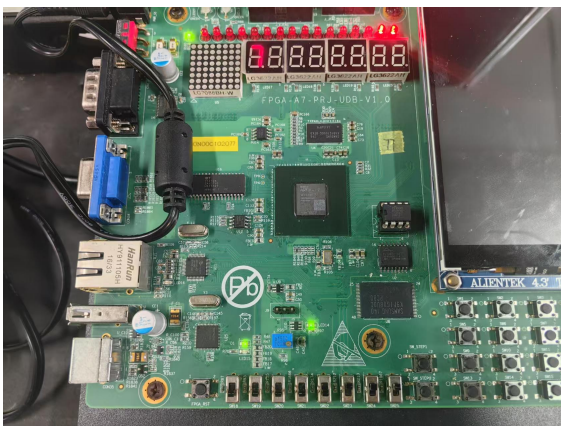
可以看见，数码屏变成了 1，而 led 灯显示上一个数字 0。

接下来拨码调为 3:



可以看见，数码屏变成了 3，而 led 灯显示上一个数字 1。

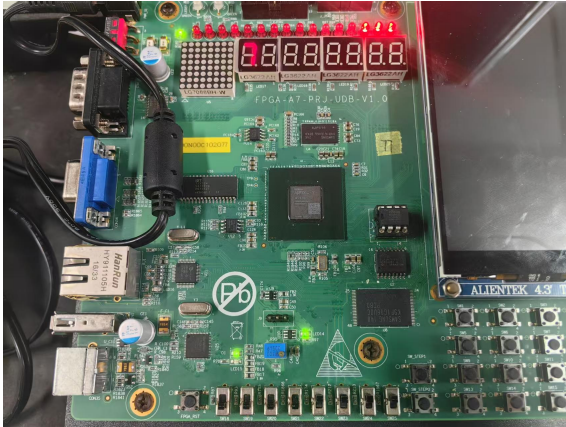
接下来拨码调为 7:



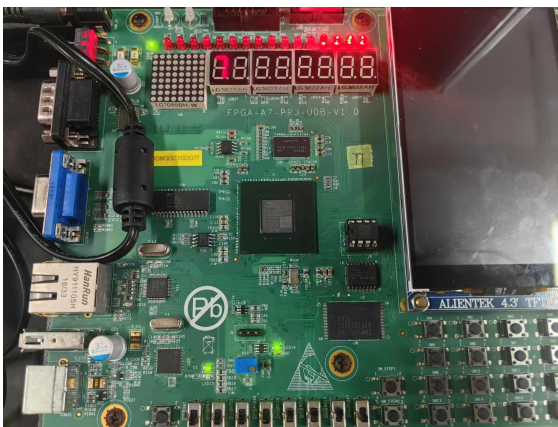
可以看见，数码屏变成了 7，而 led 灯显示上一个数字 3。

接下来我把四个拨码全部拨到上边，代表 15，大于 9:





可以看见，此时的数码屏显示的还是之前的 7，而 led 灯也变成了之前的数字 7。  
接下来我把第一个拨码拨下去，代表数字 14，还是大于 9：



因为 14 仍然大于 9，所以数码屏幕依旧不变，还是 7，而 led 灯显示上一个数据 15。

综上所述，在实验箱上验证的功能也正常。

## 五、 总结感想

### 任务一：寄存器堆仿真实验

为什么寄存器堆要设计为两读一写？

主要是为了提高 cpu 的运行效率，因为在 MIPS 体系下，对于 add, sub 等操作指令往往涉及到两个源寄存器和一个目标寄存器，我们需要从两个源寄存器里边读取到我们需要的数据，经过相应运算之后把结果写入到一个目标寄存器里边，这样就需要寄存器堆可以同时读取两个寄存器的值，并且向一个寄存器写入结果，这样可以提高 cpu 的处理效率。如果只能一读的话，寄存器堆需要读取两次，降低了速度，但如果设计更多的读取接口的话，会增大成本，让电路太复杂，得不偿失。

## 任务二：同步 RAM 与异步 RAM 仿真

同步 RAM 与异步 RAM 的特点与不同。

同步 RAM 的特点：

首先，对于同步 RAM 来说，每一次的读写操作都必须在时钟的上升沿触发，以此来保持同步性，第二点，同步 RAM 具有默认的读后写的功能，即当使能端为 1，进行写操作时，在时钟上升沿，同步 RAM 也会自动读取刚刚写入的值。这比较适合性能好，运行速度快的设备。

异步 RAM 的特点：

首先，对于异步 RAM 来说，它的每一次读写操作不需要等待时钟信号，而是随着使能端，地址等端口的变化而随时进行。并且，异步 RAM 并没有读后写的功能，读写操作严格按照使能端的信号进行。异步 RAM 比较适合慢速的设备，因为在读写时不需要等待时钟信号。

同步 RAM 与异步 RAM 的不同：

第一点，同步 RAM 的读写操作在时钟上升沿进行，而异步 RAM 是立即进行。第二点，同步 RAM 具有读后写的功能，而异步 RAM 没有。

## 任务三：数字逻辑电路的设计与调试

使用 vivado 的调试经验：

1. 调试时候最主要的方法就是看波形图，如果遇见 Z 波形，那么首先去模块里找 wire 类型的变量，看看它们有没有被正确赋值，并且要检查一下 tb 里边调用模块的时候有没有正确连接。
2. 遇见 X 波形的时候，就用看看是不是 reg 类型的量出了问题，有必要的时候可以在 always 模块里边进行一项一项的追溯。
3. 遇见波形停止的情况，这里就需要自己写代码的时候小心一点，利用 vivado 的实现功能去找出 bug，搞清楚自己需要的逻辑，然后进行修改。
4. 遇见越沿采样的情况，这应该是发现波形图里部分数据一直错误，然后去一步一步追溯检查，然后才能发现。
5. 最后，如果是功能 bug，那就需要中找出波形图里是哪一个数据出了问题，然后去模块里边进行追溯，找出问题。