组原第三次实验报告

学号：2312966 姓名：林晖鹏 班次：张金老师

1. 任务一：寄存器堆实验

1.1 实验目的

针对任务一寄存器堆实验，完成仿真，思考并回答问题：为什么寄存器堆要设计成“两读一写”？

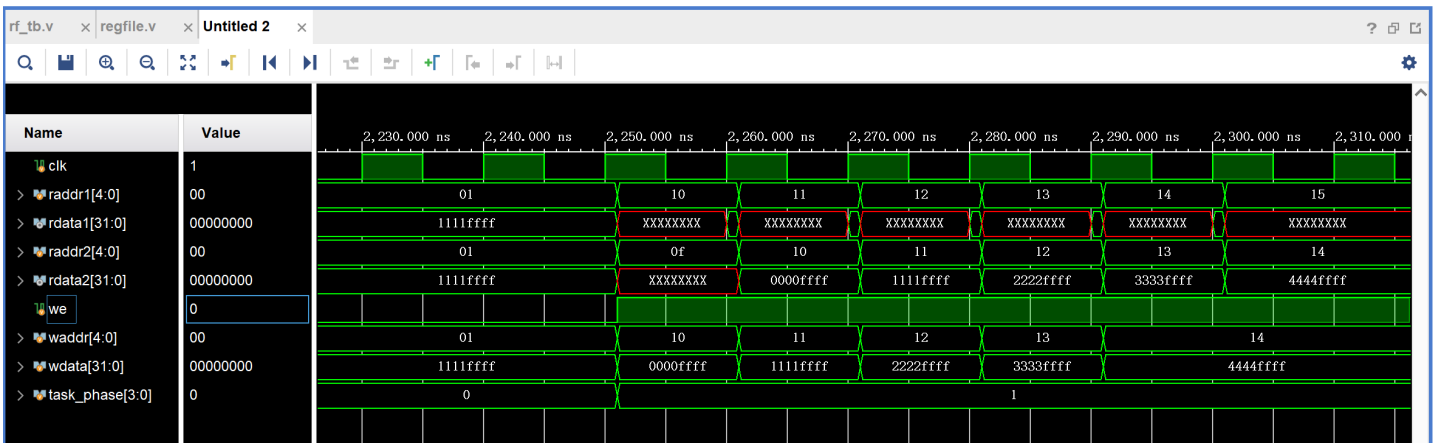
1.2 实验流程

- 实验流程很简单，只是把lab2中的相关文件导入project中，然后进行仿真。
- 这里按照书本上的操作，尝试将所有变量在仿真中显示。

1.3 代码分析

这个代码设计了一个寄存器堆，支持根据地址同时读取两个数，并写入一个数。比较特殊的是，为了简化指令集设计，这里读端的使能端始终为1。

1.4 实验结果



仿真part1截图

实验结果分析：

按照书本上的操作，设置看全部变量信息，看到了更多的变量信息，方便分析结果。

可以看到，当时钟上升沿并且we==1的时候，在特定地址写入数据。

但是看part2的仿真实验可以知道，读取的是异步的，不依赖于时钟周期。

而当读取地址拥有数据的时候，立刻读取数据。

1.5 问题思考

- 存在大量的逻辑计算是二元的，两读一写能够在写入上一条指令的结果的同时，读取下一条指令的两个输入数据，这样实现了流水工作。而不是读取完一条指令的数据之后，需要等待写入结束才能开始下一条指令的读取。
- 单周期内能够完成更多操作，同时读取两个数据
- 这样的设计满足了常见的RISC指令集的执行需求。在RISC架构中，大多数算术逻辑指令（如 `ADD`，`SUB`，`AND` 等）采用**三操作数**格式。

---

2. 任务二：同步RAM和异步RAM的仿真、综合与实现。

2.1 实验要求

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，分析同步ram和异步ram各自的特点和区别。

2.2 实验流程

a. 根据手册上面的流程，分别建立两个工程，并设置异步ip核以及同步ip核。

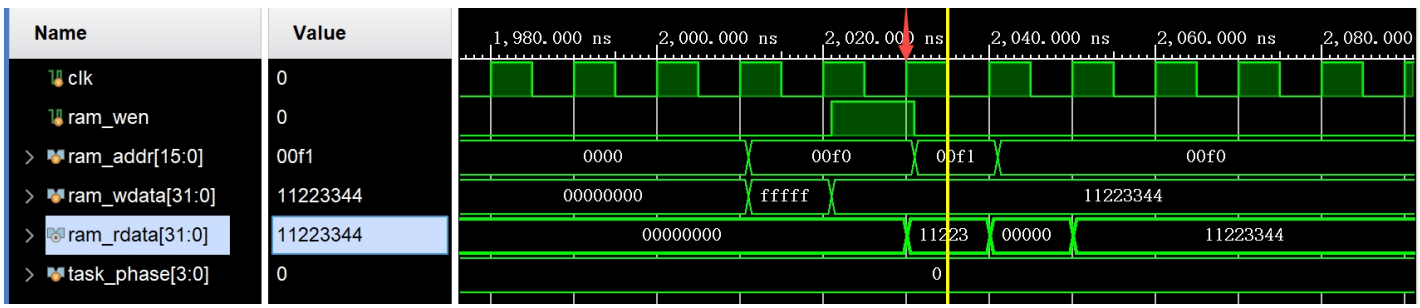
b. 进行仿真实验，并分析结果。

c. 生成时序结果和资源利用率，并进行分析。

2.3 实验结果

2.3.1 仿真结果：

同步RAM：

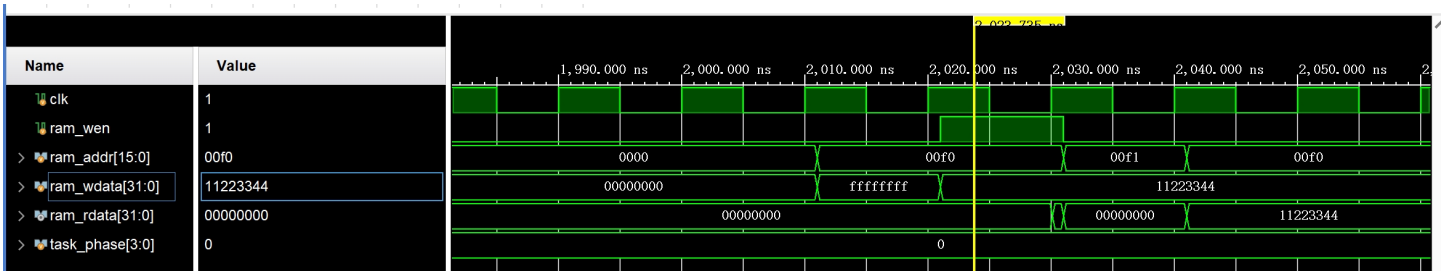


同步RAM仿真截图

分析：

在同步RAM中，我们观察图中红色指向的上升沿周期，可以发现，wdata的写入和rdata的读取都在时钟周期上升沿的时候完成，说明同步RAM由时序控制，依赖时钟信号同步操作。

异步RAM：



异步RAM仿真截图

分析：

在异步RAM中，我们可以发现，写入数据是不依赖于时钟周期的，而是当ram_wen==1的时候，只要出现变化，就写入新值，但是这里的读取仍然是依赖于时钟周期。

2.3.2 时序结果

我们将同步异步放在一起分析：

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP	Start	Elapsed	Run Str
synth_1 (active)	**constrs_1**	**synth_design Complete!**								**0**	**0**	**0.0**	**0**	**0**	**4/2/25, 2:30 PM**	**00:00:37**	**Vivado**
impl_1	constrs_1	route_design Complete!	2.439	0.000	0.571	0.000	0.000	0.155	0	122	4	58.0	0	0	4/2/25, 2:31 PM	00:01:56	Vivado

同步RAM时序结果

synth_1 (active)	**constrs_1**	**synth_design Complete!**									**0**	**0**	**0.0**	**0**	**0**	**4/2/25**
impl_1	constrs_1	route_design Complete, Failed Timing!	-3.648	-219.1	0.702	0.000	0.000	0.385	0	41357	0	0	0.0	0	0	4/2/25

异步RAM时序结果

指标	WNS	TNS	WHS	THS	TPWS
同步RAM	+2.439 ns	0.000 ns	+0.571 ns	0.000 ns	0.000 ns
异步RAM	-3.648 ns	-219.1 ns	+0.702 ns	0.000 ns	0.000 ns

- WNS - 最差负时序裕量：所有时序路径中，实际信号到达时间与要求到达时间的最差差值。
- TNS - 总负时序裕量：所有违例路径的时序裕量总和（仅统计负值）。
- WHS - 最差保持时间裕量：信号在寄存器输入端需保持稳定的最小额外时间余量。
- THS - 总保持时间违例：所有保持时间违例的裕量总和。
- TPWS - 最差脉冲宽度裕量：时钟信号脉宽是否满足最小宽度要求

分析：

同步RAM的时序报告显示 **WNS=+2.439ns** 且 **TNS=0ns**，说明所有路径均满足建立时间要求，最差路径仍有2.439ns的余量。保持时间指标 **WHS=+0.571ns** 和 **THS=0ns** 进一步验证了数据的稳定锁存。

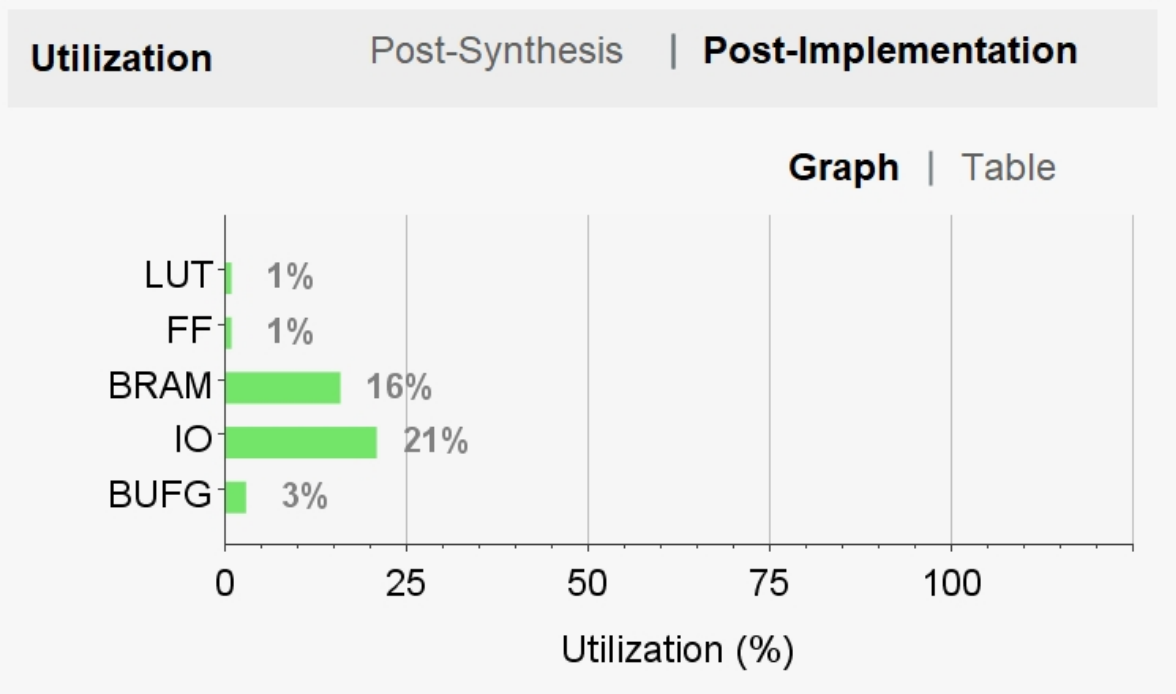
异步RAM的时序报告显示严重违例：**WNS=-3.648ns** 和 **TNS=-219.1ns**，表明关键路径延迟超出预期3.648ns，且多路径累计违例达219.1ns。尽管保持时间达标（**WHS=+0.702ns**），但建立时间违例会引发功能错误。此类问题通常源于异步控制逻辑的组合路径过长（如地址解码或数据使能链），或缺乏足够的延迟匹配。需通过插入寄存器切割长路径，或改用握手协议（如Req/Ack）替代纯组合逻辑。TPWS=0ns说明时钟/脉冲宽度无异常，问题集中在数据通路的时序收敛。

2.3.3 资源利用率

我们先来解释相关指标：

LUT：查找表利用率、FF：触发器利用率、BRAM：块RAM、IO：输入输出接口、BUFG：全局时钟缓冲器、LUTRAM：分布式RAM。

同步RAM：

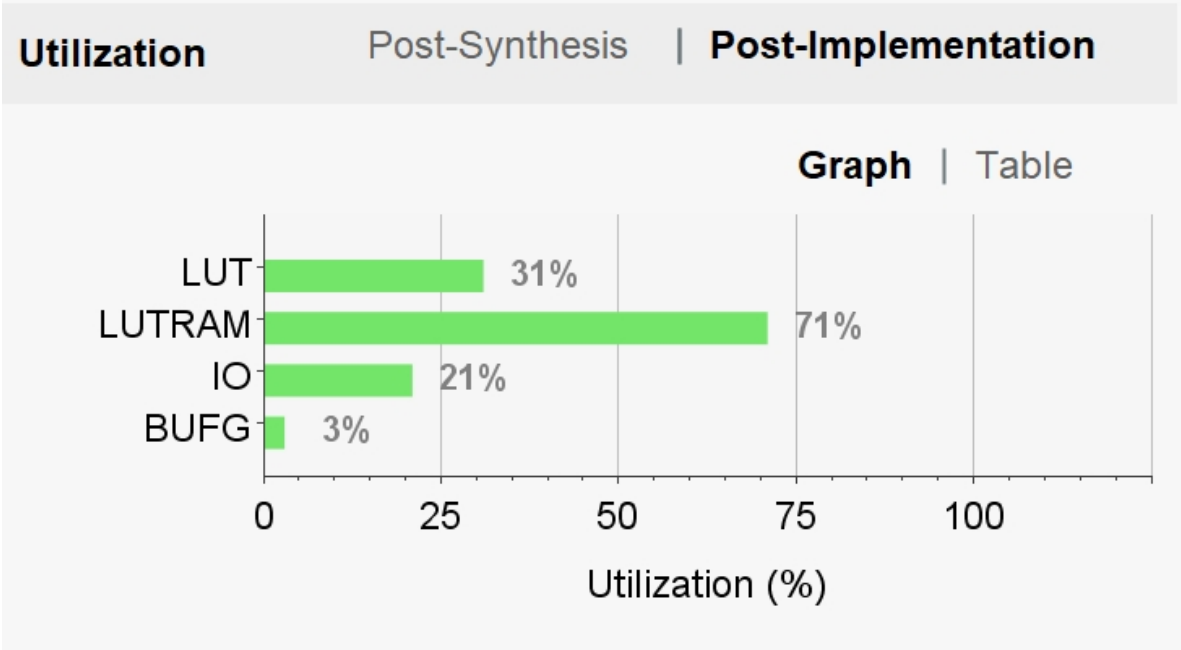


同步RAM资源利用率

分析：

同步RAM设计资源占用集中于BRAM和IO，逻辑部分（LUT/FF）优化良好，适合作为存储模块集成到更大系统中。

异步RAM：



异步RAM资源利用率

分析：

LUTRAM利用相当高，LUT也比较高，说明查找开销不小。异步RAM在接口和控制逻辑上开销较大，需优化信号分配。

2.4 同步异步RAM对比

从本次实验来看，同步异步的差别主要体现在是否依赖时钟信号来操作，所以同步的接口也会比异步的接口多一个时钟信号。通过分析时序结果和资源利用率，我们还能得到其他区别。

特点与差别：

- 同步依赖时钟信号，接口多一个时钟信号；而异步根据信号变化响应。
- 所有操作和时钟信号边沿对齐，有周期概念；而异步没有周期概念。
- 同步RAM访问速度快、吞吐量高、有固定延迟（与周期有关）；而异步RAM访问速度慢、吞吐量小、无固定延迟
- 同步RAM设计相对简单；异步RAM相对复杂。

---

3. 任务三：数字逻辑电路的设计与调试

3.1 实验要求

本实验提供了一个有5个bug的数字逻辑电路设计源码，要求根据正确功能修改bug。希望熟悉掌握vivado调试的方法。

3.2 代码bug修改

3.2.1 确认成员变量

一开始并不知道一些变量的含义，通过查看xdc约束文件对应接口，能够推出信息：

代码块

```
1       output      [7 :0] num_csn,   //表示数码管显示的位置
2       output      [6 :0] num_a_g,  //表示数码管的译码
```

3.2.2 显示更多信息

一开始仿真，我们发现只显示了三个变量，不利于我们去观察波形图。所以我们检查了仿真文件，发现调用模块的时候，有几个参数空缺了，导致不显示变量变化情况。我们做以下修改之后再仿真，就能看见大部分需要的变量情况了：

代码块

```
1   //666666666666666这里用wire，reg无法接口
2   wire [7:0] num_csn;
3   wire [6:0] num_a_g;
4   wire [3:0] led;
```

代码块

```
1   show_sw  u_show_sw(
2       .clk    (clk    ),
3       .resetn (resetn ),
4
5       .switch (switch ),    //input
6
7       .num_csn(num_csn),   //new value
8       .num_a_g(num_a_g),
9
10      .led    (led)    //previous value
11  );
```

3.2.3 修改一

书本中也提示了，拨码开关是拨上为低电平，拨下为高电平，但是功能要求实现的是拨上为1，拨下为0。所以我们这里要做一个取反处理。

代码块

```
1   //new value
2   always @(posedge clk)
3   begin
```

```
4   //1111111111111111
5       show_data    <= ~switch;
6   end
```

3.2.4 修改二

show_data_r 是前一步的数据，show_data是当前的数据。正确功能希望实现当前的和前一步的数据不一样时候，prev_data 保存先前的数据。

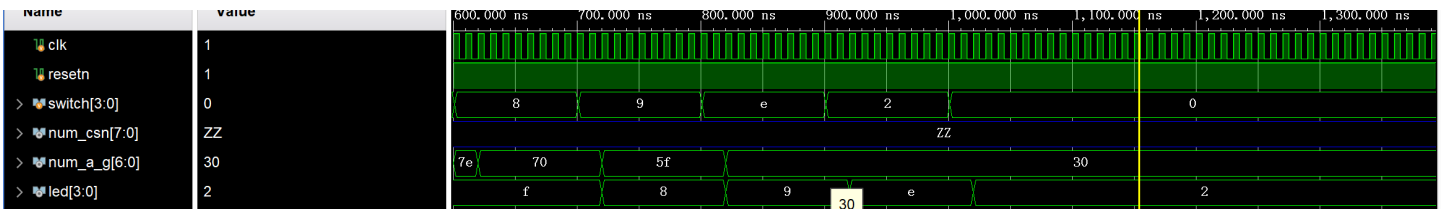
在修改之前，由于 `show_data_r = show_data` 是阻塞赋值，在非阻塞赋值之后进行赋值，所以总是在 `prev_data <= show_data_r` 完成之后才赋值，但是此时前一步的和当前的数值肯定相等，导致除了初始化时候，后面这个prev_data一直不变。我们也可以通过**分析波形图**来发现这个bug。

这里我们将阻塞赋值修改为非阻塞赋值，这样，由于阻塞赋值是并行的，所以此时记录的是之前的值。

代码块

```
1   always @(posedge clk)
2   begin
3   //2222222222222222
4       show_data_r <= show_data;
5   end
6   //previous value
7   always @(posedge clk)
8   begin
9       if(!resetn)
10      begin
11          prev_data <= 4'd0;
12      end
13      else if(show_data_r != show_data)
14      begin
15          prev_data <= show_data_r;
16      end
17  end
```

3.2.5 修改三



通过分析波形，我们发现在num_csn这一行里面，一直是ZZ，结合书上所说，这是未赋值导致的，所以我们回过头来发现，这里变量名一开始写成了num_scn 。

修改成num_csn 之后，波形恢复正常了。

代码块

```
show_num u_show_num(
        .clk        (clk      ),
        .resetn     (resetn   ),

        .show_data  (show_data),
        //3333333333333333
        .num_csn    (num_scn  ),
        .num_a_g    (num_a_g  )
);
```

3.2.6 修改四

在数码管译码器这里，发现缺少了六。再结合约束文件中对应的接口，查阅接口知道对应的数字笔画位置。然后检查其他没问题之后，增加6的译码：

代码块

```
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
                 //4444444444444444
                 show_data==4'd6 ? 7'b1011111 :   //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   ;
```

3.2.7 修改五

检查代码的时候，没看懂一开始这个keep_a_g为啥设置为目前值未来值相加。这里要求实现当show_data大于等于10的时候，nxt_a_g赋值给keep_a_g，但是要求nxt_a_g不变。

所以这里我们用一个三目运算符修改成当大于等于10的时候，赋值给自己。由于阻塞赋值是串行的，先完成这个语句才去赋值nxt_a_g。所以符合正确功能。

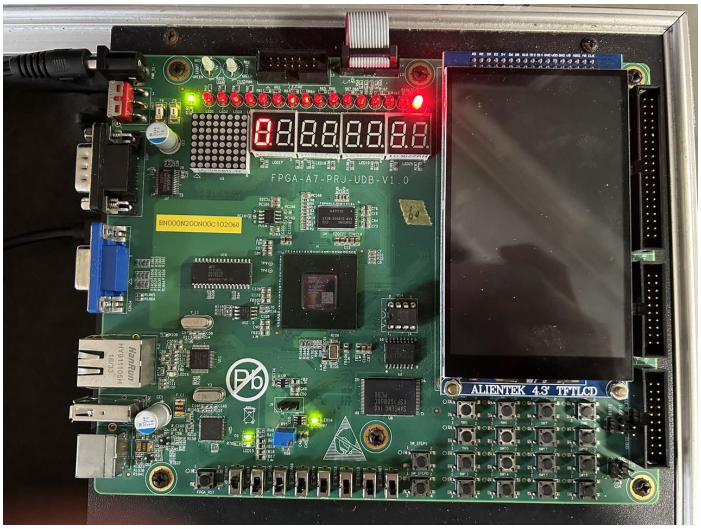
代码块

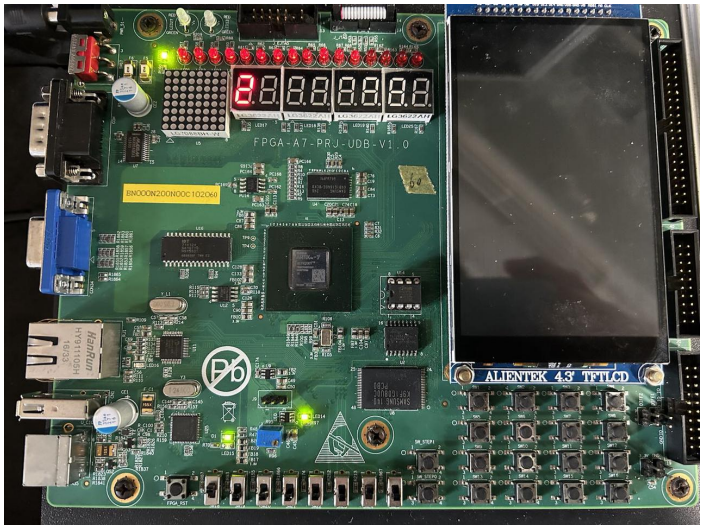
```
//keep unchange if show_dtaa>=10
wire [6:0] keep_a_g;
//55555555555555555555555555
```

```
4    assign      keep_a_g  = show_data>=10 ? keep_a_g :num_a_g;
```

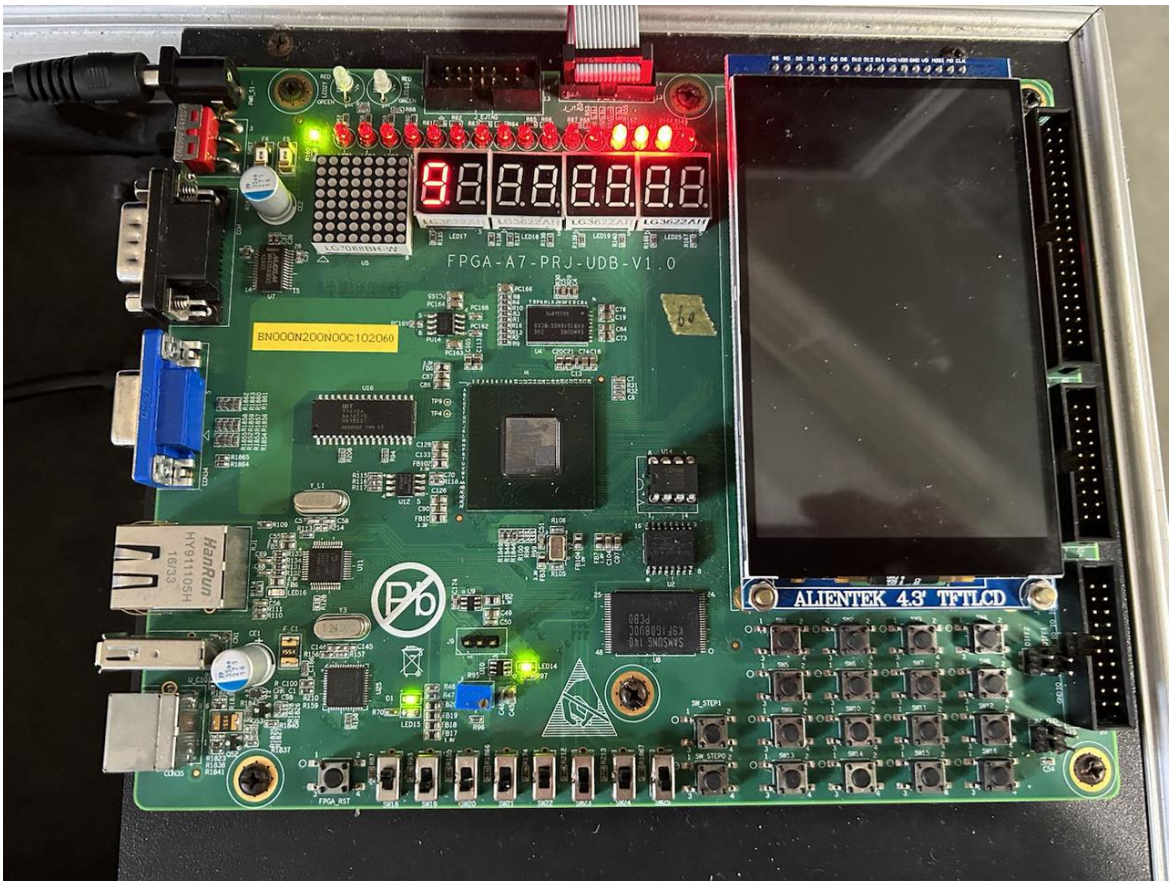
3.3 上箱验证

我们下面给出两组案例，先左后右。





这个是正常案例，可以看到基本功能正常。



这个案例我们可以看到，此时输入的大于10了，而数码管没有改变数字。另外，上面的晶体管也能显示大于等于10的数据。

4. 总结感想

调试代码心得步骤：

1. 在写完之后，尝试综合与实现，根据报错信息修改代码。这些一般是语法或者结构的问题。
2. 尝试设计测试案例去进行模拟仿真。
3. 观察仿真的波形图，思考功能是否正确。沿着时间轴往前寻找，要找到一开始错误的地方，因为一处错误可能会导致后面时间里一系列的错误，所以一开始找到的错误可能是前面出现的错误导致的，并非根本原因。
4. 在CPU的书本上介绍了各种常见错误，进行一一对应并寻找原因。
5. 进行上箱验证功能，上箱能更加直观的找出错误，个人感觉这种错误一般是阻塞和非阻塞的居多。上箱验证也能验证自己的约束文件是否写少写错。
6. 重点，这里还有一个小众bug，可能在初始化项目的时候配置选错导致上箱失败！！！都是上学期的泪。