

实验 2 报告

学号: 2019K8009929039

姓名: 周鹏宇

箱子号: 50

一、 实验任务 (10%)

本次实验内容为寄存器堆的仿真，同步、异步 RAM 仿真和综合以及调试给出的存在五个 bug 的 LED 文件。前两者仅需按部就班的导入文件并进行仿真和综合即可，根据生成的波形可以了解寄存器堆和RAM 的功能，同时也可以起到是否操作成功的检验作用（若无法生成波形图或波形图存在问题，则说明之前的操作存在缺陷）。而第三者除了导入文件和仿真外，还增加了上板验证的功能，通过上板后拨动开关和检查晶体管与 LED 灯的反应，可以验证是否将 bug 全部排除。

二、 实验设计 (0%)

三、 实验过程 (90%)

(一) 实验流水账

表格 1: 实验流水账

时间	工作	结果
2021.9.3 18:30-18:40	导入寄存器堆文件，仿真并观察波形	顺利完成并记录
2021.9.3 18:45-19:15	导入同步、异步 RAM 文件，仿真和综合，同时观察波形图	顺利完成并记录
2021.9.3 19:30-21:30	导入 LED 文件，对错误进行修正后试图上板	无法与板子连接，并把 vivado 损坏
2021.9.3 22:30-23:00	再次上板并记录	顺利完成

(二) 子任务一

在完成文件导入和仿真后，结果如下图：

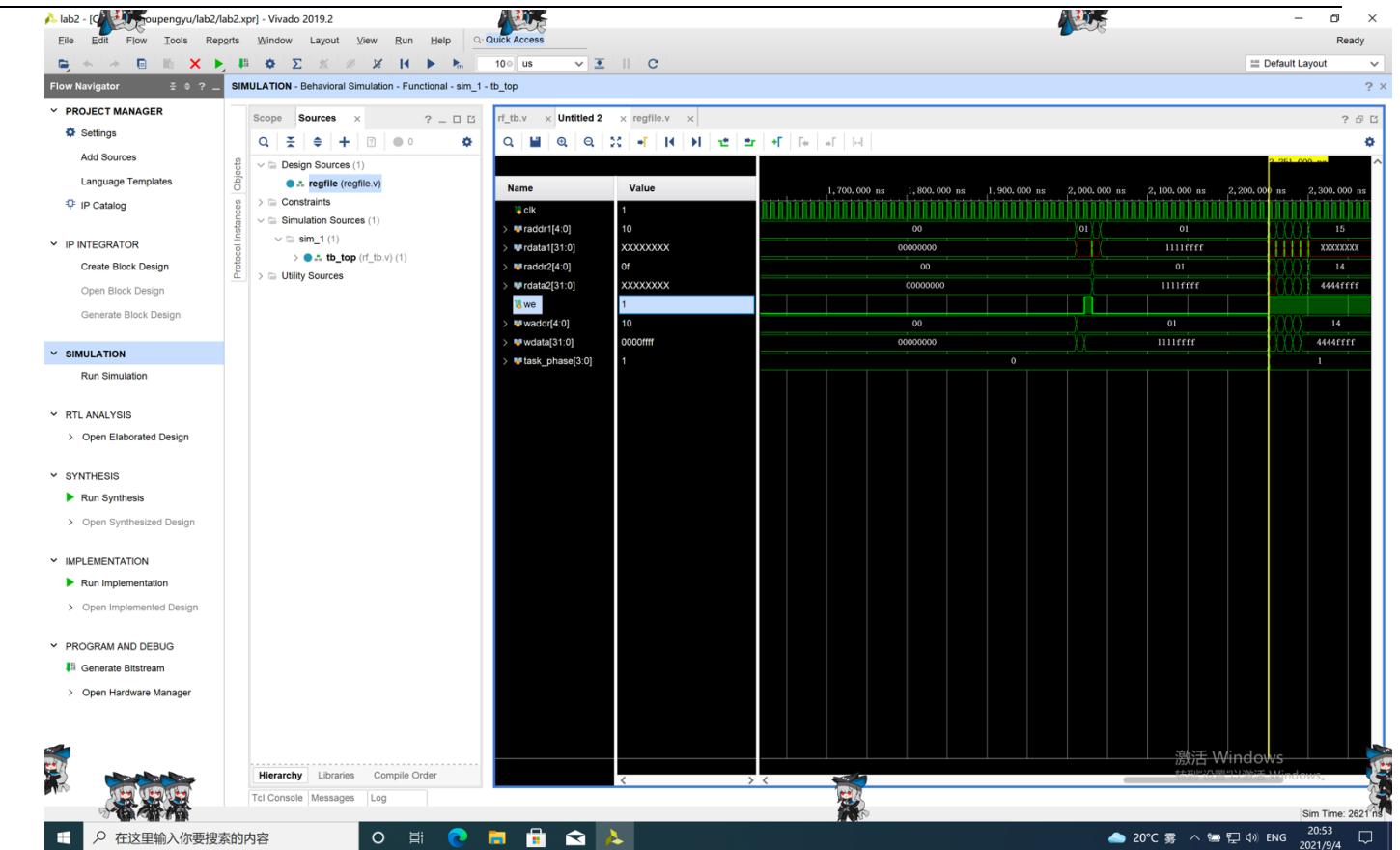


Figure 1: 子任务一

由截图可以看出，在 we 信号拉低时，数据并不写入 RF，在 we 信号拉高时，寄存器堆向地址 01 写入了 ffff，而在之后的周期，rdata1 和 rdata2 都从地址 01 中读出了数据 ffff。

具体情况如下：

1、零号寄存器

无论零号寄存器中为何值，寄存器堆都只能从其中读出 0，如下图所示，0 号寄存器中无数据，但 rdata 仍为 0。

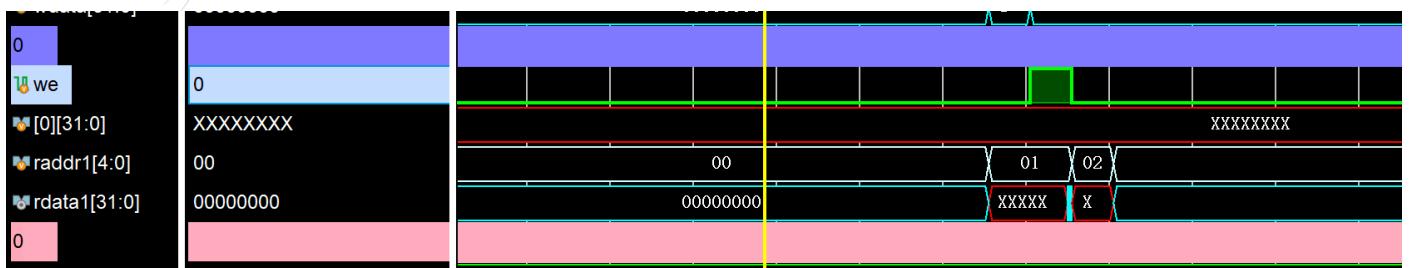


Figure 2 寄存器堆零号寄存器读

2、同步写

同步写即在时钟周期上升沿完成写入，无论是阅读源码还是看波形图都能看出

```
// WRITE
always @(posedge clk) begin
    if (we) rf[waddr] <= wdata;
end
```

Figure 3 寄存器堆写



Figure 4 寄存器写的波形图

注意波形中，在 we 信号拉高的时刻，其并未立刻向寄存器堆中写入 1111ffff，而是在下一个时钟周期的上升沿才写入。

3、异步读

其读出部分为组合逻辑，与 clk 信号无关，故为异步读。

```
// 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
```

Figure 5 寄存器读代码

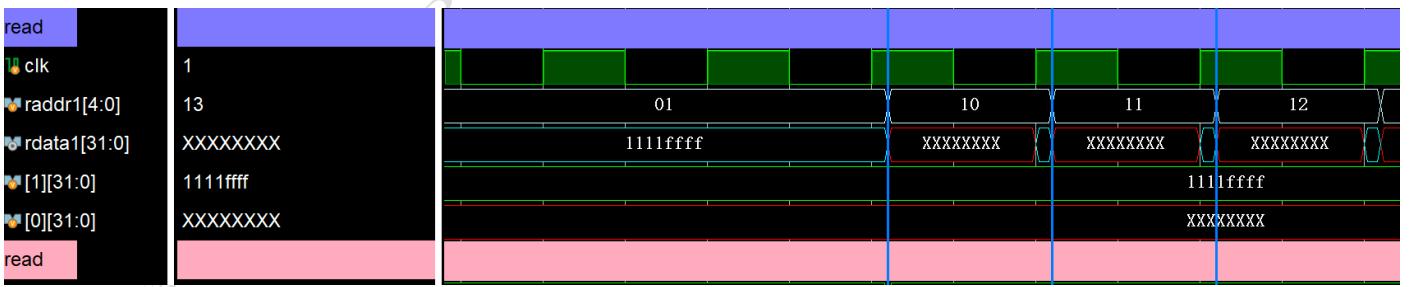


Figure 6 寄存器读波形

注意到波形图中 mark 的部分，在 raddr1 变化的瞬间，rdata1 立刻变化，与周期无关。

(三) 子任务二

4、仿真行为对比分析

(1) 同步 RAM

在 figure 8 之前，已经分别向 RAM 地址为 00f1,00f2,00f3 和 00f4 的区域写入了

0000ff11,0000ff22,0000ff33 和 0000ff44，而在 figure 8 中可以注意到，当信号 ram_addr 变为 00f1 的下一个时钟周期的上升沿，ram_rdata 才变为 0000ff11，即同步 RAM 的读出是同步的。

而在 figure 7 中，可以注意到当 ram_wen 信号拉高后，在时钟信号并未发生变化时，ram_wdata 已经写入 00f0，换言之，同步 RAM 的写是异步的。

(2) 异步 RAM

如图 figure 9 和 figure 10 所示，在 figure 9 中，当 ram_addr 发生变化时，ram_rdata 立刻发生改变（此时 ram_wen 信号为 0，故为读出），与 clk 信号无关，也即异步 RAM 的读出是异步的。而在 figure 10 中可以看到，若异步 RAM 为异步写，则在 ram_wen 升起后的 ram_rdata 应当变为 0000ff00,故异步 RAM 的写入是同步的。



Figure 7 同步 RAM 的写入

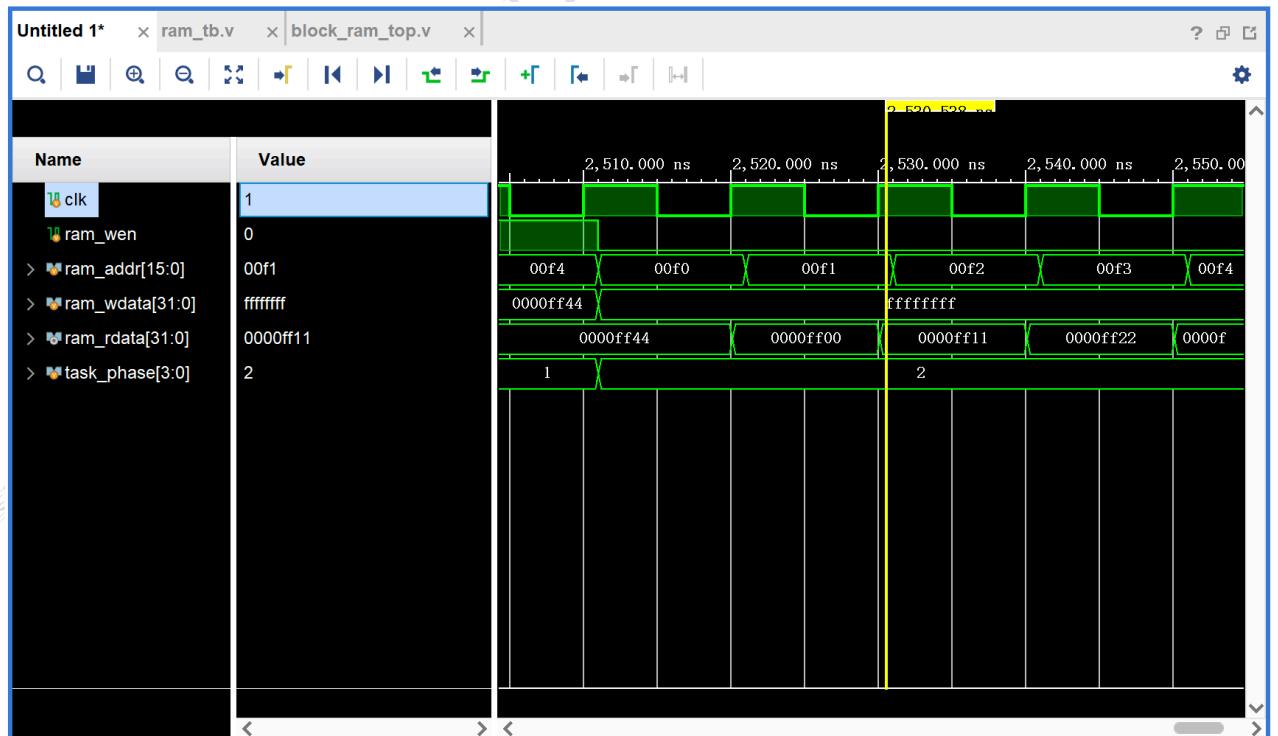


Figure 8 同步 RAM 的读出

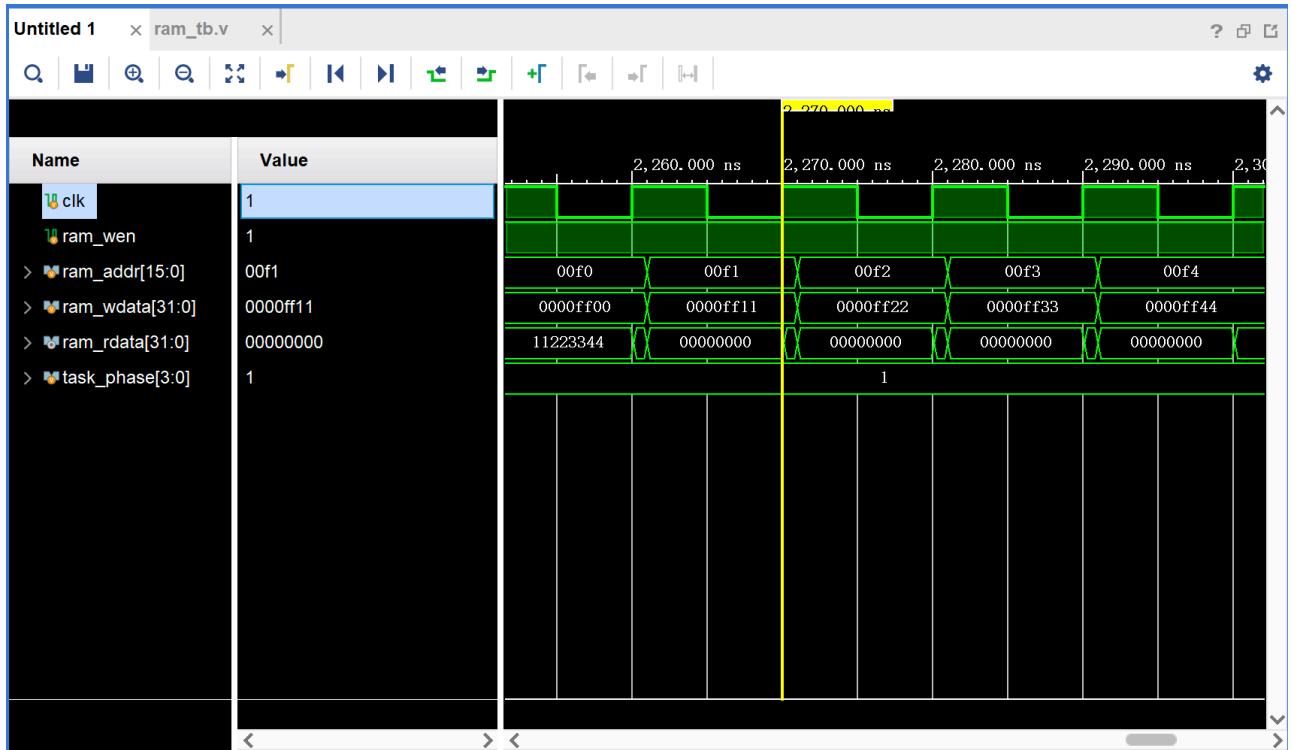


Figure 9 异步 RAM 的写入

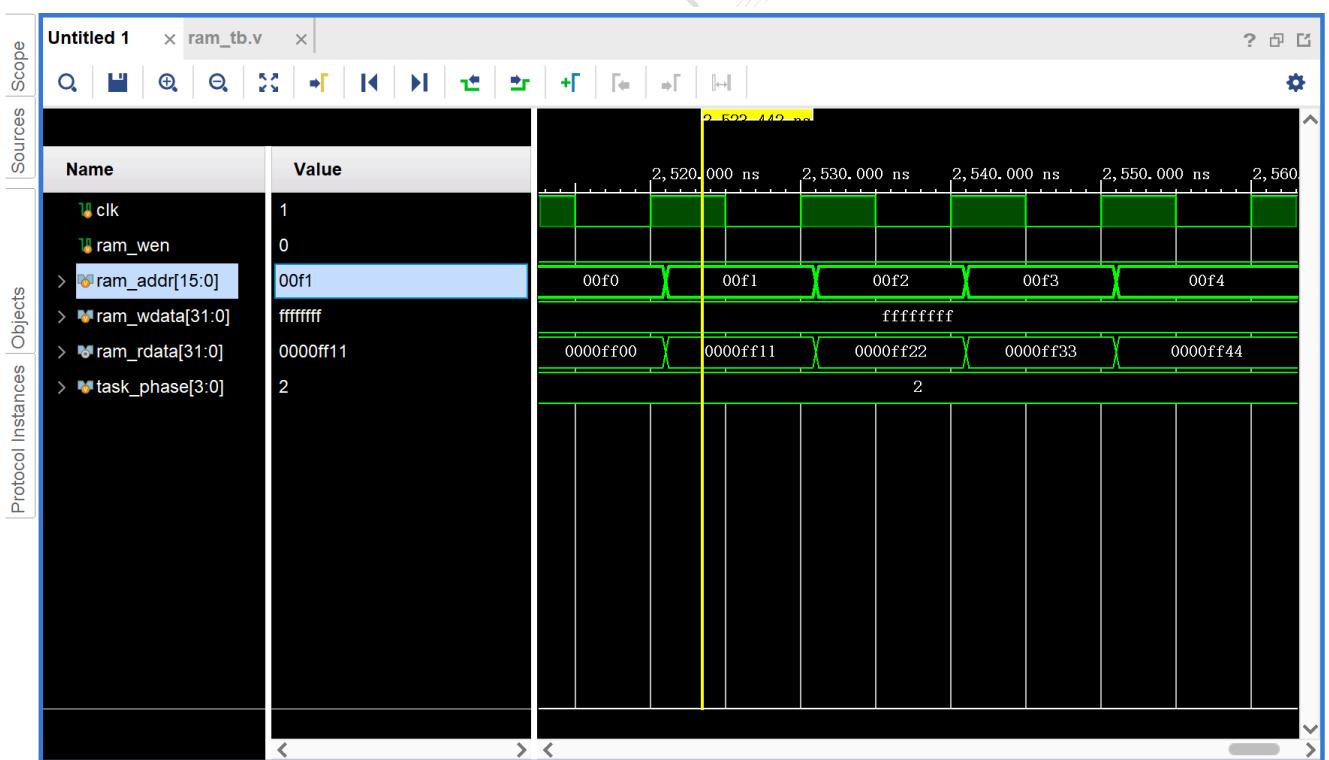


Figure 10 异步 RAM 的读出

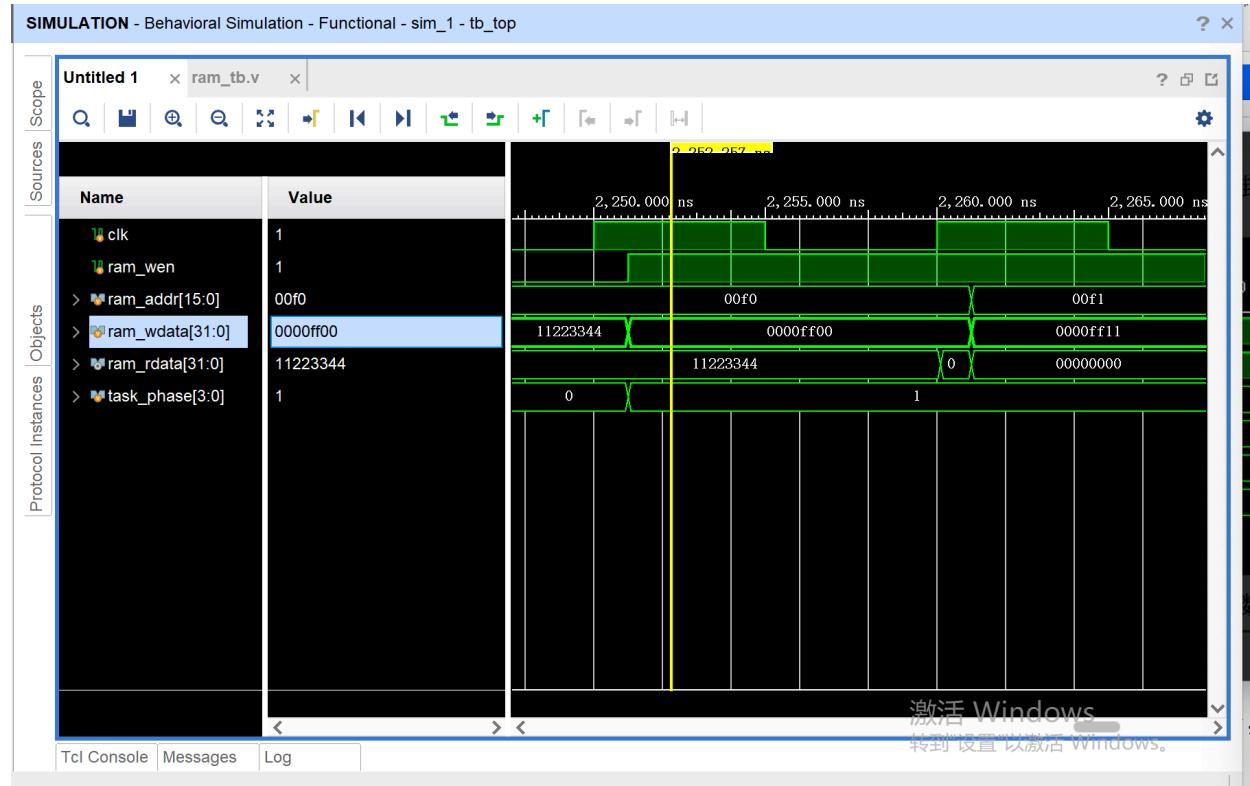


Figure 11 判断异步 RAM 写为异步还是同步

5、时序、资源占用对比分析

(1) 时序分析

如 figure 12 所示，同步 RAM 的 WNS 为 0.802，为正值，即其时序满足极好，而其 TNS 的值为 0，即所有路径的总违约值极小；相较而言，如 figure 13 所示，异步 RAM 的 WNS 为 -7.002，时序满足较好，而 TNS 的值为 -7290.607，两者的值都远大于同步 RAM。

(2) 资源占用分析

同步 RAM 的 LUT 和 FF 占用比都为 1%，而 BRAM 占用比为 16%，IO 占用比为 21%，BUFG 占用比为 3%；异步 RAM 的 LUT 占用比为 31%，LUTRAM 占用比为 71%，IO 占用比为 21%，BUFG 占用比为 3%，即相比而言，异步 RAM 对 LUT 和 RAM 占用比都显著高于同步 RAM，而 IO 占用比和 BUFG 占用比与同步 RAM 相同。

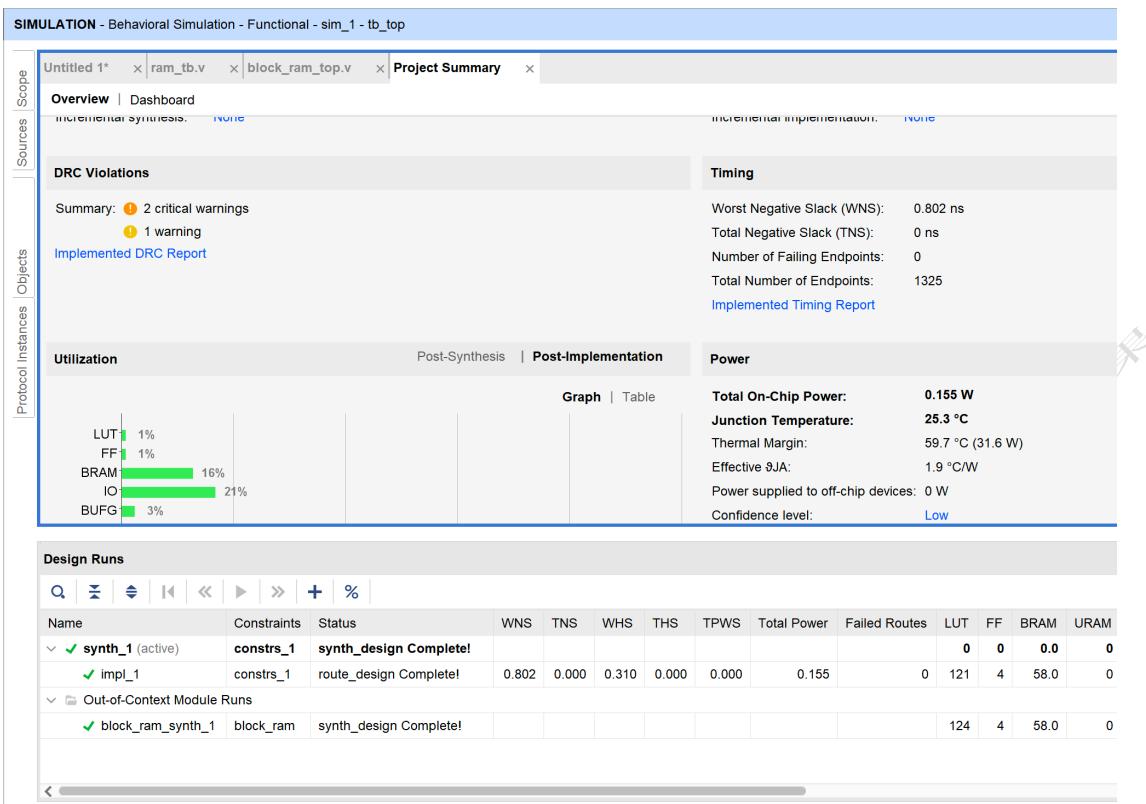


Figure 12 同步 RAM 时序和资源占用比分析

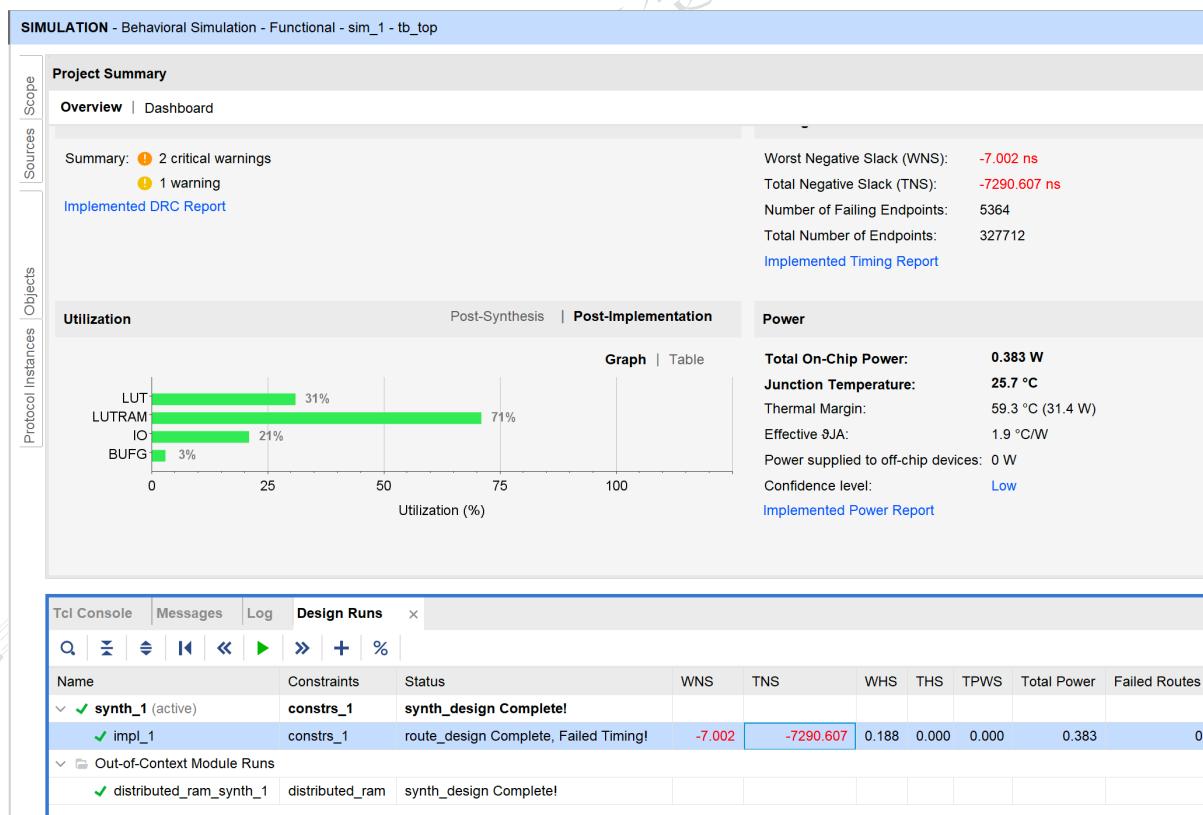


Figure 13 异步 RAM 时序和资源占用比分析

6、总结

同步 RAM 和异步 RAM 的不同首先体现在读写的时序逻辑上，同步 RAM 的读是同步的，而异步 RAM 的读是异步，写是同步的；其次，相较于异步 RAM，同步 RAM 的 WNS 和 TNS 更小，也即更容易上板成功；最后，异步 RAM 的 LUT 和 RAM 的资源占用比显著高于同步 RAM，而 IO 和 BUFG 的占用比与同步 RAM 几乎没有差别。

(四) 子任务三

1、错误 1：信号出现不定植

(1) 错误现象

信号 show_data, nxt_a_g 和 num_a_g 都是不定植

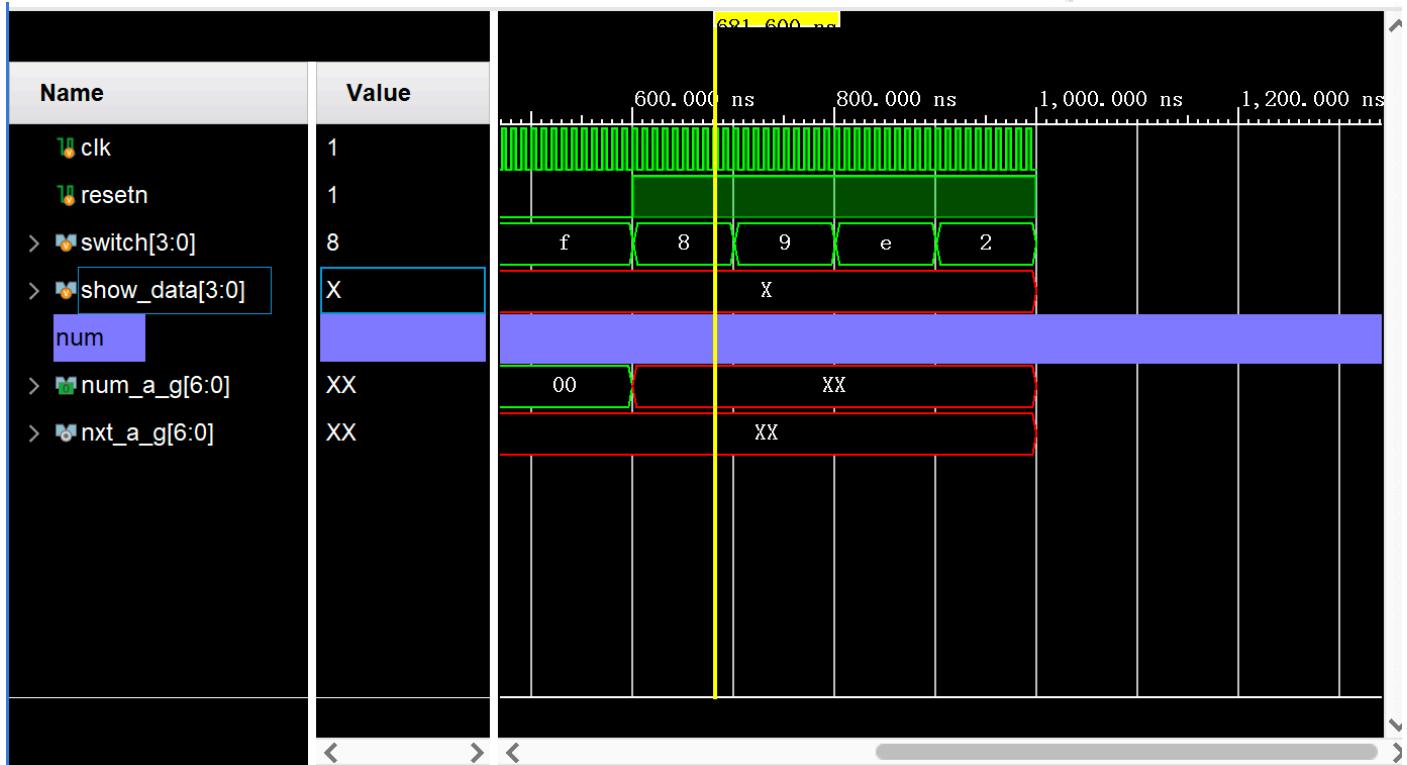


Figure 14 不定植信号

(2) 分析定位过程

首先找一下这几个信号的关联，可以注意到代码中有

```

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'd7 ? 7'b1110000 : //7
      show_data==4'd8 ? 7'b1111111 : //8
      show_data==4'd9 ? 7'b1111011 : //9
      keep_a_g ;

```

Figure 15 信号关联 I

```

begin
  if ( !resetn )
    begin
      num_a_g <= 7' b000000;
    end
  else
    begin
      num_a_g <= nxt_a_g;
    end
end

```

Figure 16 信号关联 II

两个片段，故三者都依赖于 show_data 的值，而后检查 show_data，可以注意到有：

```

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

```

Figure 17 show_data 作为 reg 型变量却从未被赋值

之后阅读代码发现给 show_data 赋值的语句仅有此一句，故原因找到。

(3) 错误原因

Show_data 作为 reg 型变量却从未被赋值。

(4) 修正效果

直接将注释符号删除即可，效果如下：

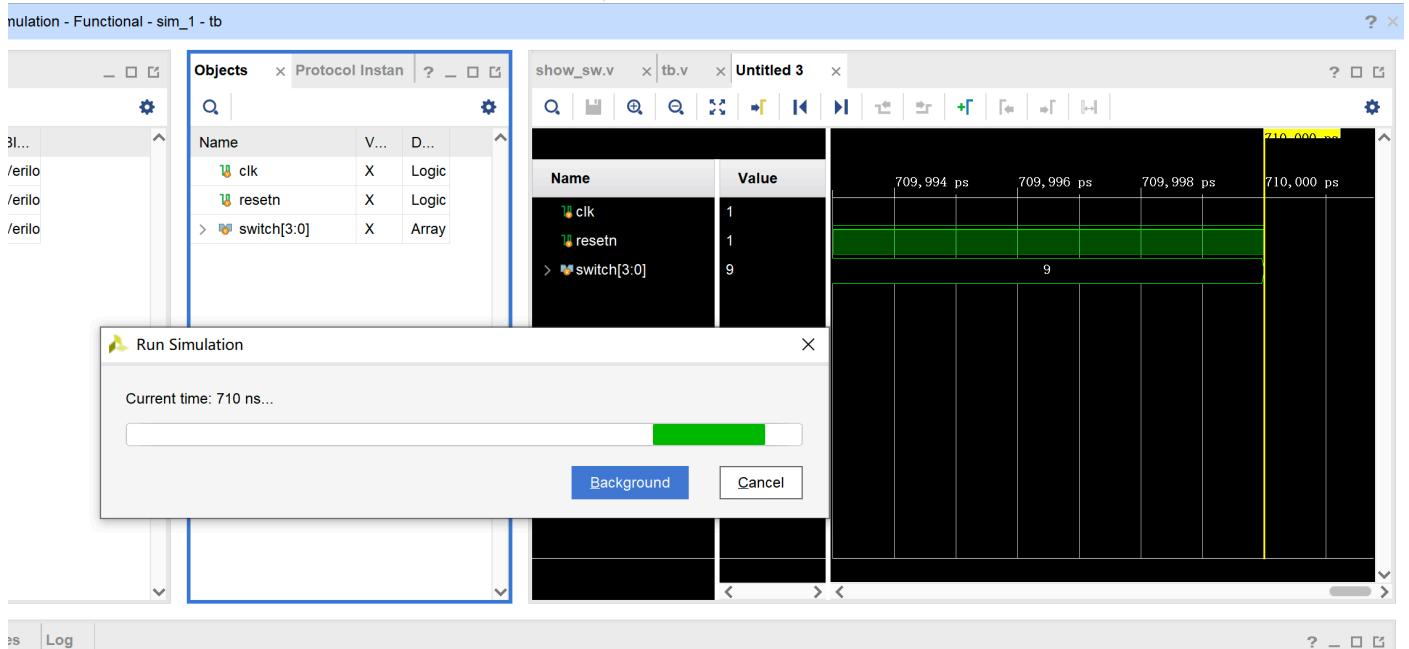


Figure 18 修改 show_data 后的效果可以注意到出现了新的问题

在修正上述错误后的效果如下所示，可以注意到 show_data, num_a_g 和 nxt_a_g 的值恢复正常。

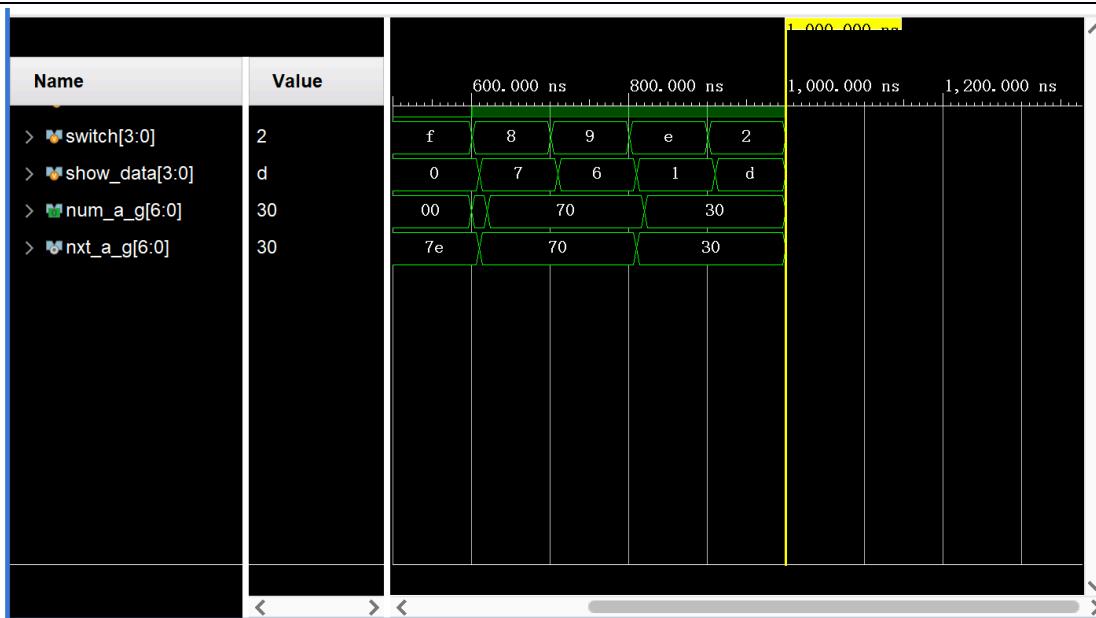


Figure 19 恢复正常的信号

(5) 归纳总结（可选）

个人感觉出现不定值更有可能的原因是 reg 信号被多驱动了，笔者在上学期计算机组成原理实验中曾遇到过类似情况，本次试验中出现的因为未赋值而导致出现 X 的现象，只要在写的时候稍加注意，对自己使用的信号有一定了解，大抵就不会出现，当然，如果是复用别人的代码而后乱改信号名，改了却又没完全改，那就是另一种情况了……

2、错误 2：波形停止

(1) 错误现象

书接上文（figure 13）

(2) 分析定位过程

课件中提及，出现波形停止的原因是“造成这种情况的原因是模拟组合环路的计算达到次数上限后自动停止仿真了”，换言之，出现组合回路了，此时重新检查代码，发现：

```
//keep unchange if show_dtaa>=10
wire [6:0] keep_a_g;
assign keep_a_g = num_a_g + nxt_a_g;

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'd7 ? 7'b1111000 : //7
                show_data==4'd8 ? 7'b1111111 : //8
                show_data==4'd9 ? 7'b1111011 : //9
                                keep_a_g ;
endmodule
```

Figure 20 组合环路

就此找到。

(3) 错误原因

错误原因自然是组合环路，将此句改为‘assign keep_a_g = num_a_g;’即可。

值得注意的是，如果改为‘assign keep_a_g = nxt_a_g;’，在之后的 simulation 中不会出现问题，但无法完成生成比特流文件，其本质仍然是重复赋值。

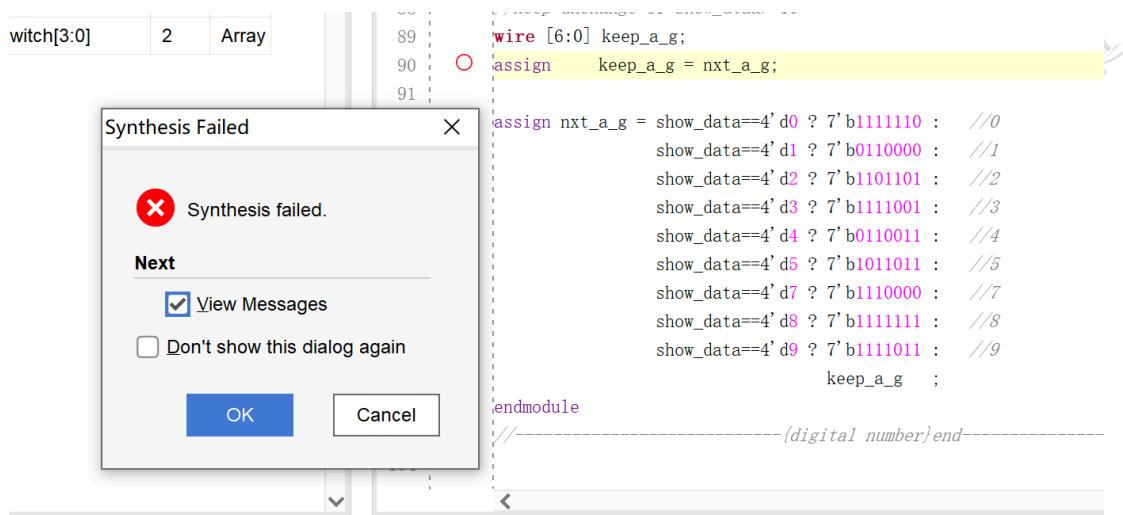


Figure 21 生成比特流文件时出错

(4) 修正效果

成功跑完仿真了。

(5) 归纳总结（可选）

组合逻辑环的确是一个比较常见的问题，而且不是很容易看出来，上学期有很多同学费劲千辛万苦跑完几十个测试点，却因为出现了环路而导致后续的生成比特流文件失败，通常情况下可以看 vivado 给出的 warning 来确定位置就是了。

3、错误 3：功能错误

(1) 错误现象

注意 figure 14，可以看到，当要显示的数字是 6 和 7 时，二者的 num_a_g 竟然是相同的，这显然出现了严重的偏差。

(2) 分析定位过程

直接找回 num_a_g 的赋值过程，即 figure 10 和 figure 11，可以注意到 figure 11 中少了一种情况——当要显示的值为 6 的情况。

(3) 错误原因

漏了一个情况，补上就是了，如图所示。

```

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'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 ;

```

Figure 22 修改后的 nxt_a_g

(4) 修正效果

本错误的验证留待上板后确认。

(5) 归纳总结（可选）

功能错误大抵是每次实验必然出现的错误了，如果能一遍写成而无需担心功能错误，想必其 verilog 编程的能力已经高到不知道哪里去了，这类问题很难找到一个通解，以笔者上学期的计算机组成原理实验课为例，每次都会出现意想不到的惊喜，例如指令实现错误，跳转错误，使能信号异常等等，只能说寄希望于之后的功能错误都和本次一样易于看出罢……

4、错误 4：信号为“Z”

(1) 错误现象

信号 num_csn 为'Z'

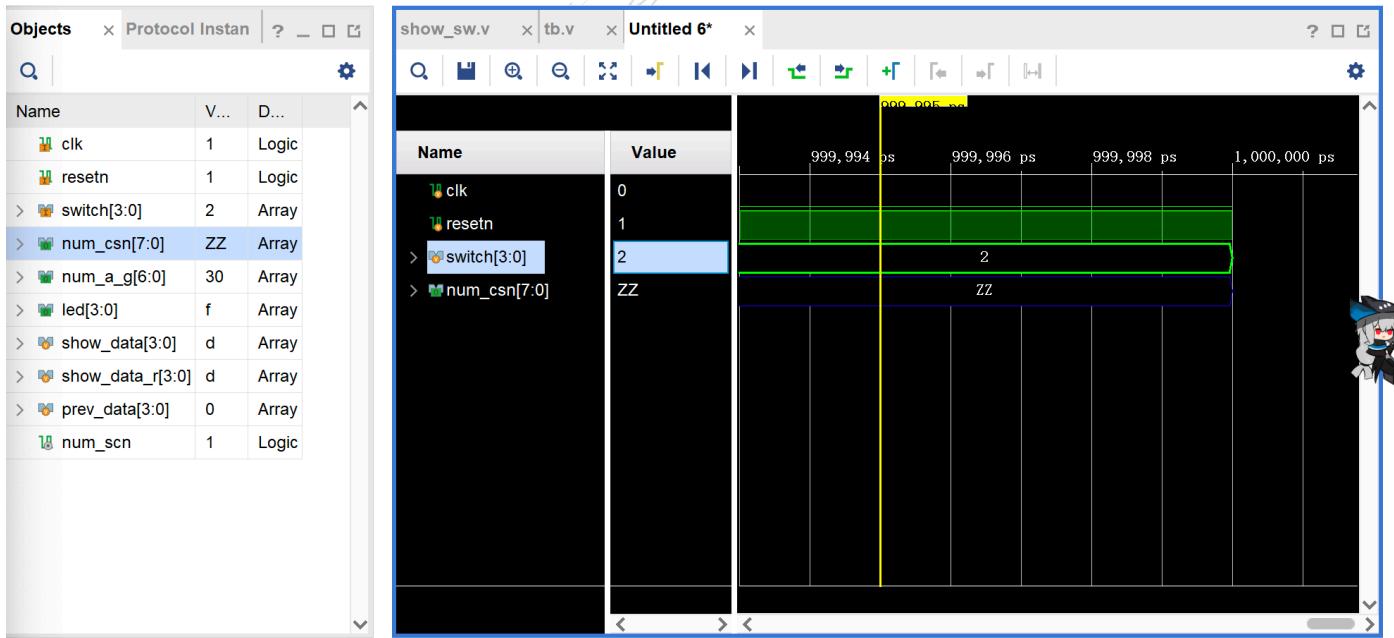


Figure 23 csn 信号为 Z

(2) 分析定位过程

直接找 num_csn 信号相关的代码部分，发现在模块调用时有一处很显然的错误。

```

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

```

Figure 24num_csn 的错误

(3) 错误原因

由于拼写错误导致模块调用信号未连接。

(4) 修正效果

把拼写纠正即可。

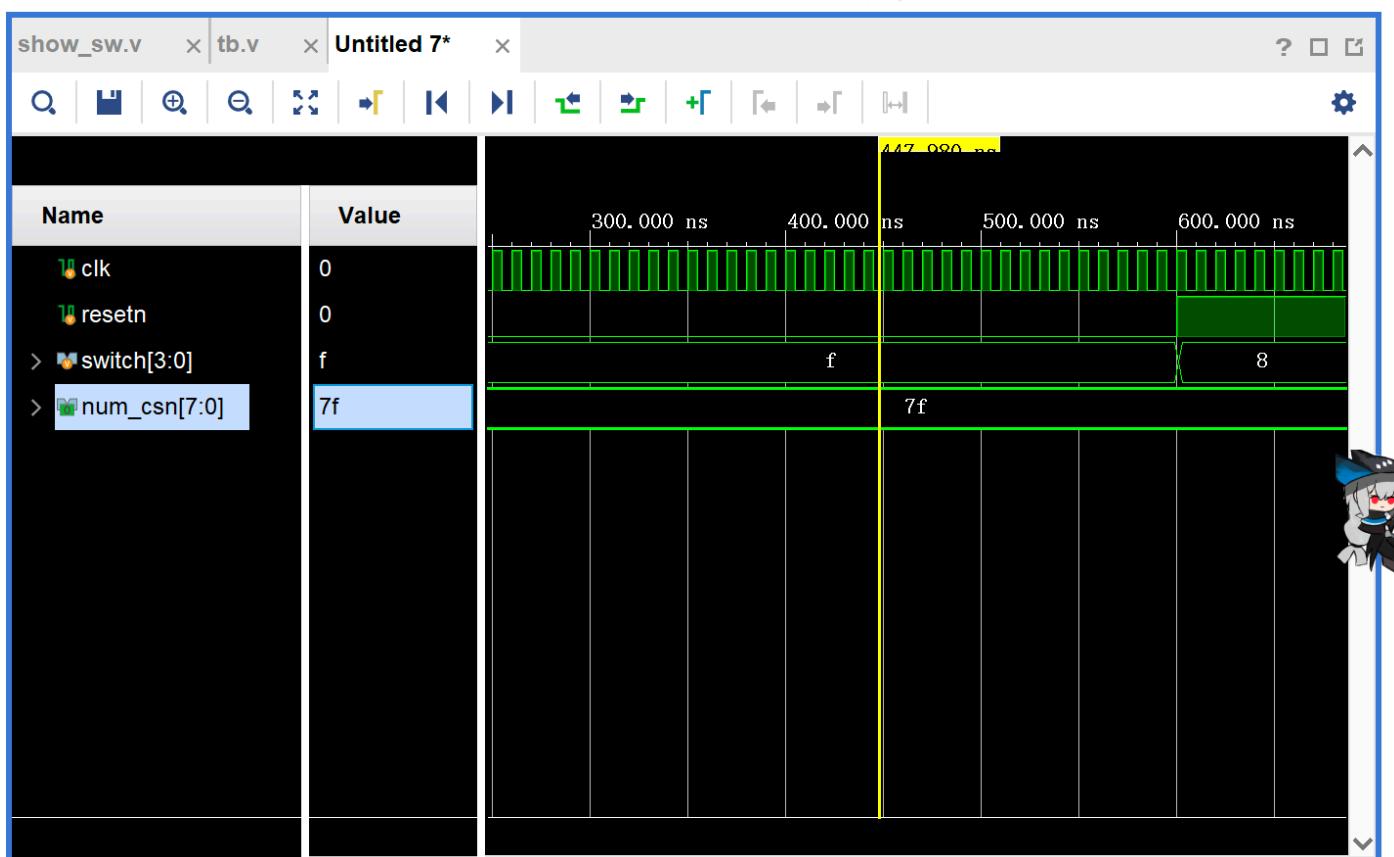


Figure 25num_csn 信号恢复正常

(5) 归纳总结（可选）

此类问题和问题一十分接近，调试和注意方式可以参考问题一。

5、错误 5：越沿采样

(1) 错误现象

prev_data 信号显示异常，一直为 0，实际上，可以注意到在每个 show_data 变化的周期时刻，prev_data 应当也

会发生相应的改变，但由于越沿采样，导致 `prev_data` 的判断条件(`show_data_r != show_data`)出现了问题，使之一直无法变化。

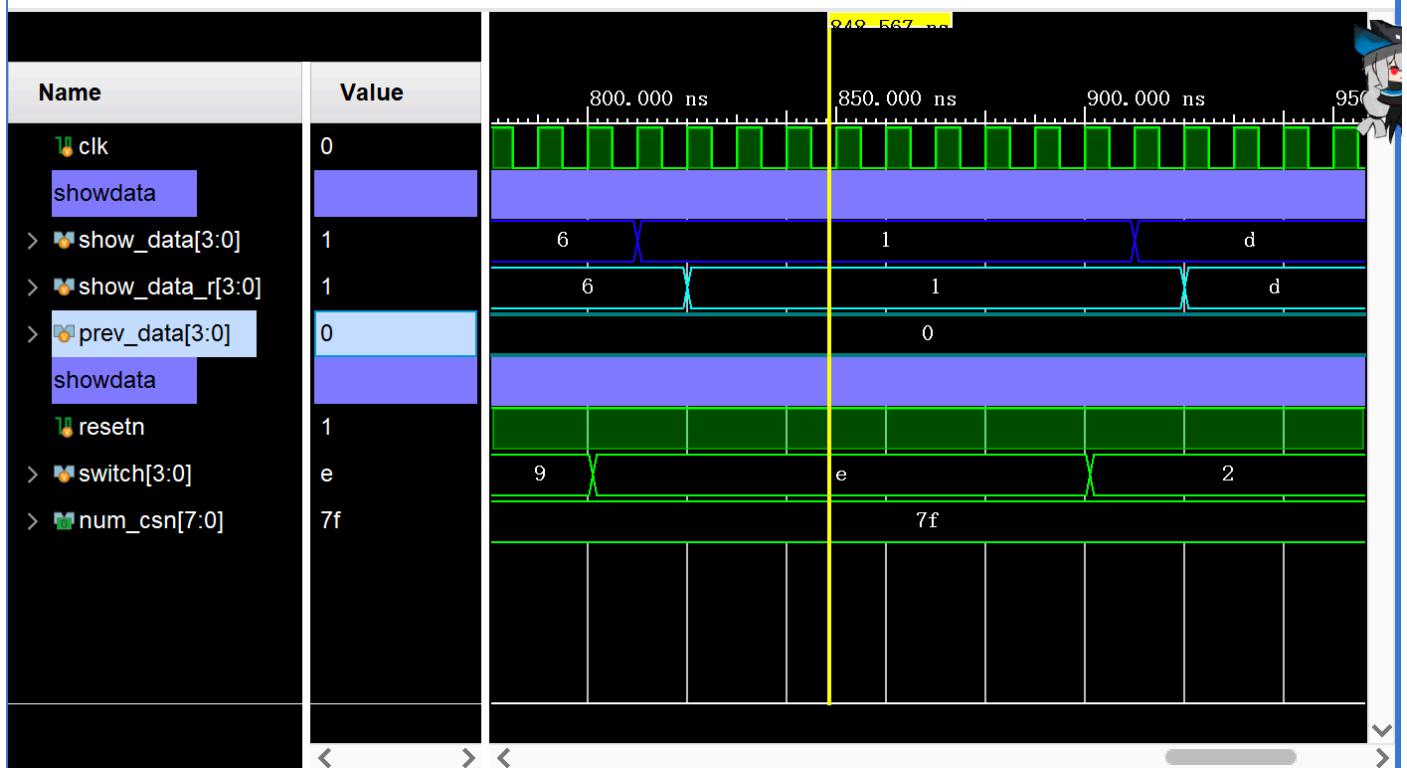


Figure 26 异常的 `prev_data` 信号

(2) 分析定位过程

直接寻找以上三个信号的相关代码，可以直接发现有一处阻塞赋值。

```

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

```

Figure 27 `show_data_r` 阻塞赋值

(3) 错误原因

`Show_data_r` 阻塞赋值。

(4) 修正效果

改为非阻塞赋值后，效果如下：

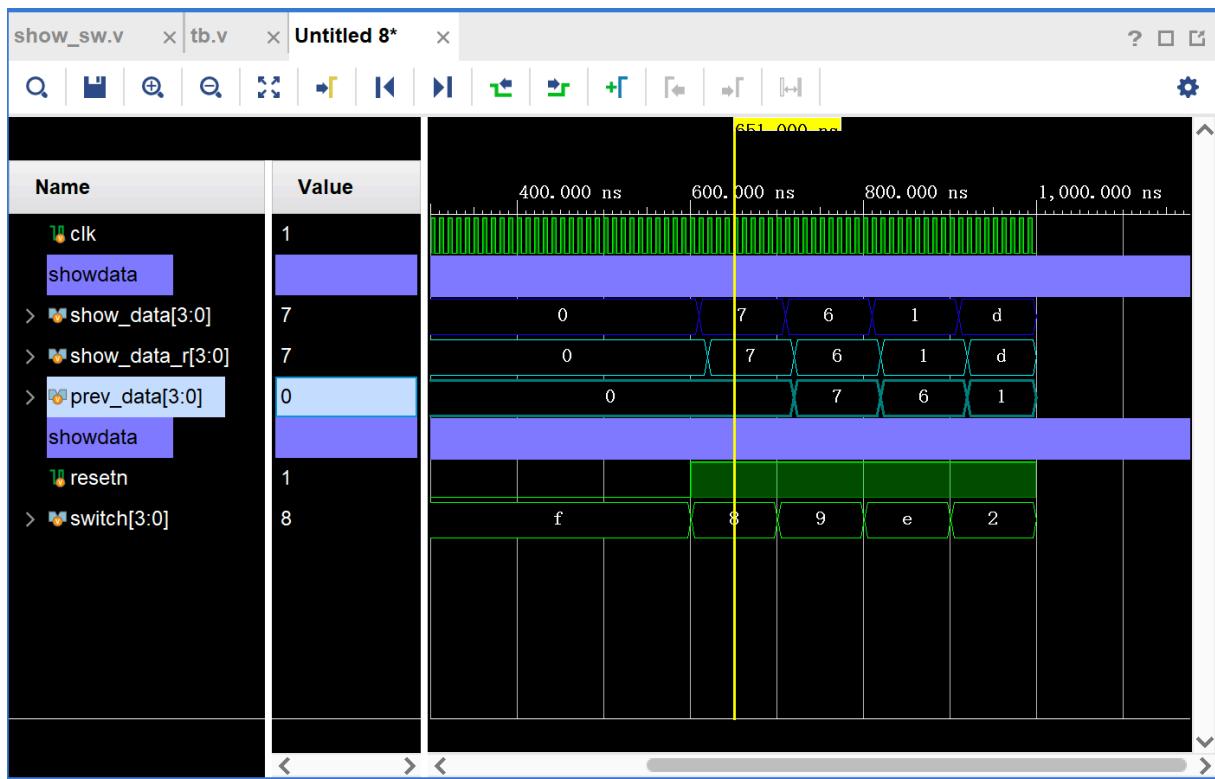


Figure 28 修改后的仿真波形图

(5) 归纳总结（可选）

这个问题只要注意编写 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即可。

最终上板效果如下：

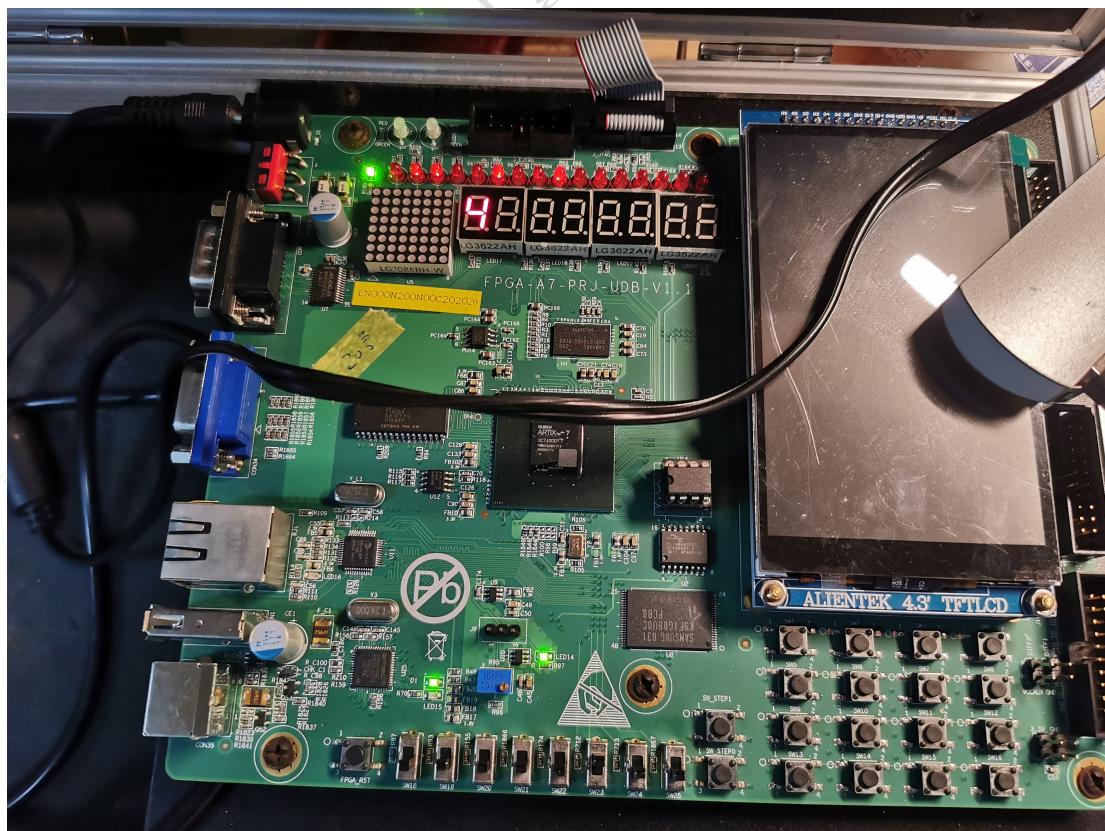


Figure 29 输入为 4，注意到 LED 显示 4



Figure 30 其后输入 5，注意到 LED 变为 5，而信号灯亮起 0100，即表示上次 LED 显示的是 4

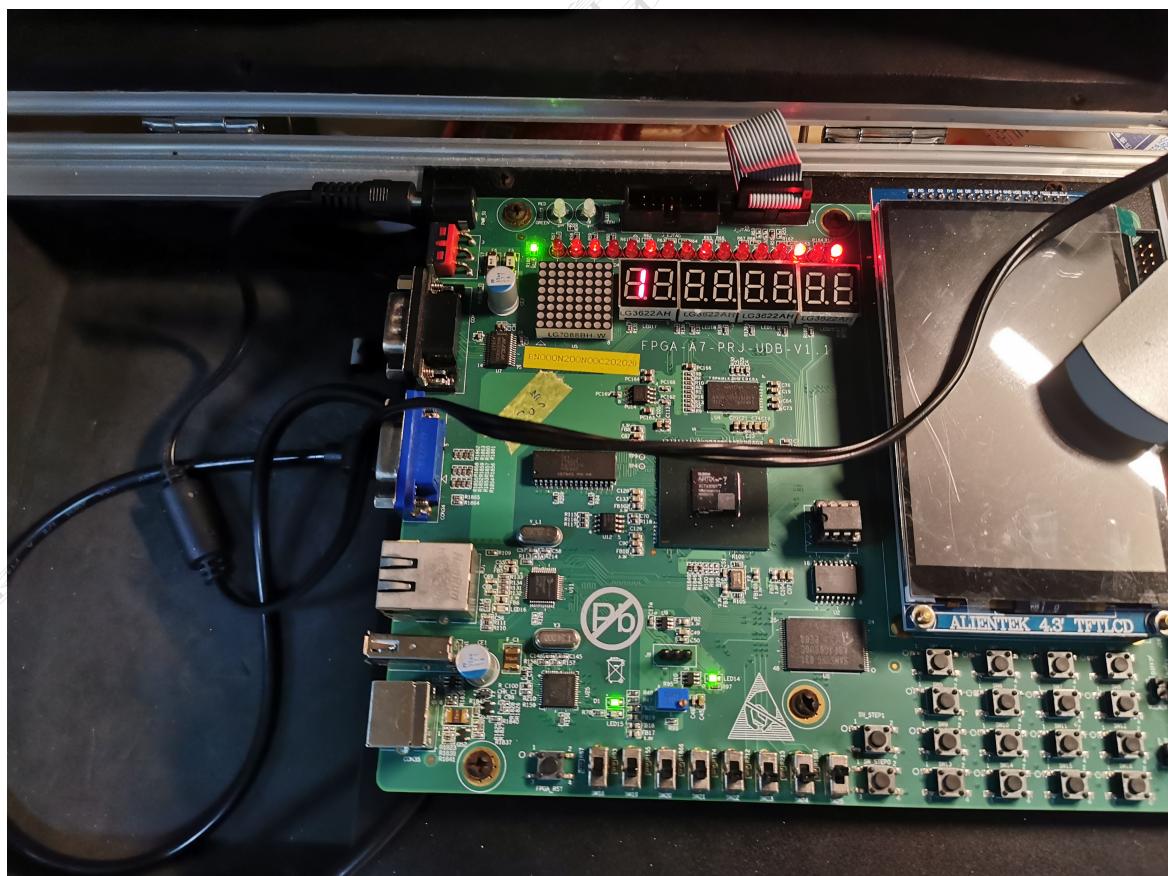


Figure 31 其后输入 1，注意到 LED 显示 0101，信号灯亮起 101，即表示上次 LED 显示的是 5

四、实验总结（可选）

本次实验主要是复习 vivado 数字逻辑设计的流程、常见 bug 与调试方式。且 bug 较为显见，甚至不要拉便能看出，希望之后的实验难度不要陡增。