南開大學
Nankai University

计算机学院

计算机组成原理实验报告

实验三

姓名：林盛森

学号：2312631

专业：计算机科学与技术

2025 年 4 月 8 日

目录

1 任务一：寄存器堆实验

1.1 实验步骤

1. 首先依次导入相关文件。
2. 接着对电路进行仿真测试，但是我发现仿真测试后波形图上全为 0；

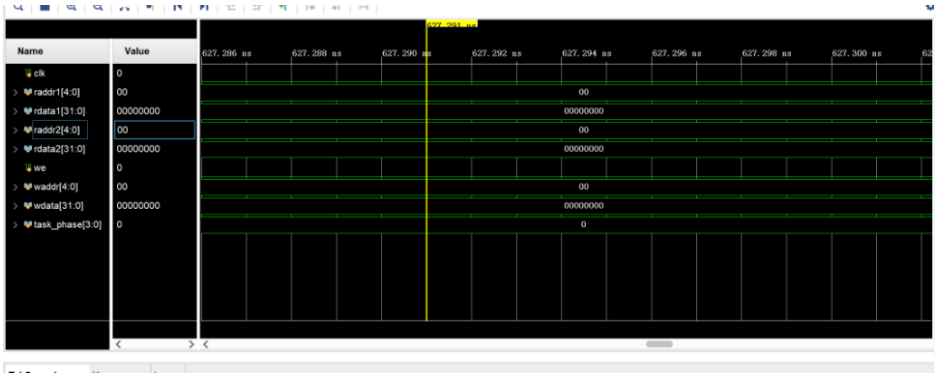

图 1.1

在看了 rf_tb 仿真测试文件之后，我发现在 2000ns 之前，是初始化阶段，使 addr 和 data 均为 0，使能端 we 也为 0，为后续对寄存器的读写做准备，在 2000ns 之后电路开始工作，准确时间应为 2000+10+1=2011ns，在此时开始输入了相应的数据进行仿真，也就是说我们仿真测试的默认最大运行时间小于开始测试的时间，所以造成在仿真时看到的全为初始化的 0；

```
39      waddr  =  5'd0;
40      wdata  = 32'd0;
41      we     =  1'd0;
42      task_phase =  4'd0;
43      #2000;
44
45      $display("==============================");
46      $display("Test Begin");
47      #1;
48      // Part 0 Begin
49      #10;
50      task_phase = 4'd0;
51      we         = 1'b0;
52      waddr      = 5'd1;
53      wdata      = 32'hffffffff;
54      raddr1     = 5'd1;
55      #10;
56      we         = 1'b1;
57      waddr      = 5'd1;
58      wdata      = 32'h1111ffff;
```

图 1.2: rf_tb 代码

我们可以在 simulation_settings 中修改单次运行默认最大时间，如图所示修改为 10000ns，这样我们就可以看到相应的波形图。

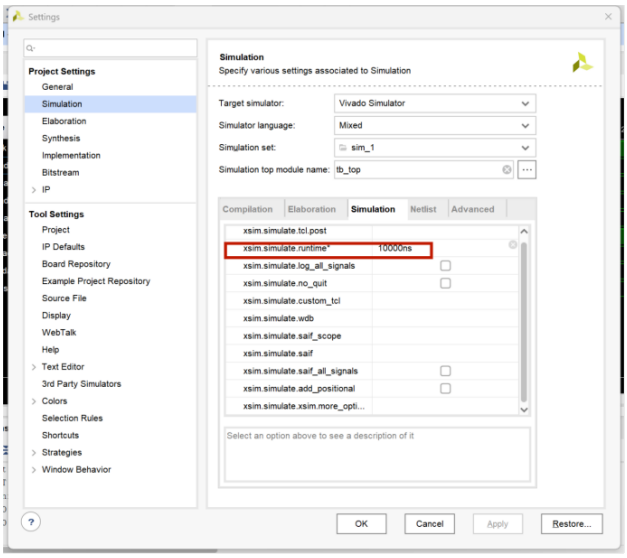

图 1.3: simulation_settings

3. 我们对 3 个阶段的波形图依次分析；首先是 part 0，经过计算，得到 part 0 的开始时间为 2011ns，观察波形图，首先 raddr1 被设置为 01，也就是要去读出寄存器地址为 01 处所存的内容，但由于我们还没有对此寄存器进行写的操作，即没有对其进行初始化，所以读出的值 rdata1 显示 XXXXXXXX，we 此时为 0，故我们输入的 ffffffff 并不会写入寄存器 01，之后 we 为 1，我们向 waddr 所存的寄存器地址即 01 寄存器中写入 wdata1111ffff，可以看到读出 01 寄存器的值为 1111ffff，表明我们写入寄存器的操作成功，而对于读取 02 寄存器，返回的值为 XXXXXXXX，也是由于我们未对 02 寄存器初始化，再之后两个读取端同时读取 01 寄存器的值，为 1111ffff，至此 part0 阶段结束，我们完成了一个时钟信号一次写入两次读出的仿真验证。

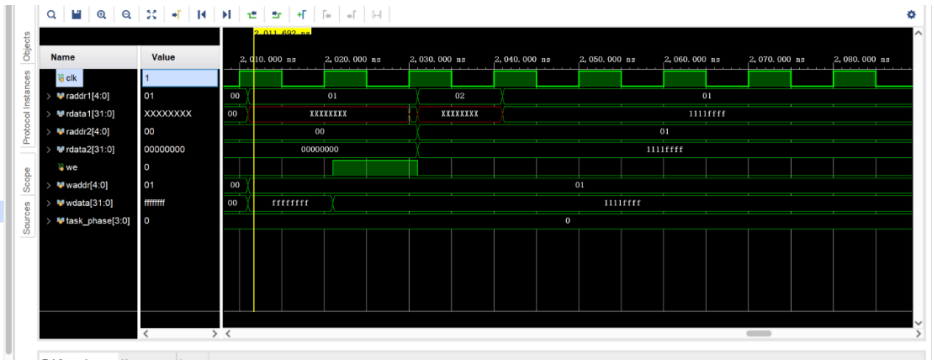

图 1.4: part0 阶段波形图

接着是 part 1 阶段，相关逻辑也是类似，观察下图，在第一个时钟，读取 10，0f 寄存器，但是由于我们没对其初始化，所以读出都为 XXXXXXXX，此时 we 为 1，所以我们成功将 0000ffff 写入 10 号寄存器；第二个时钟周期，访问 11，10 号寄存器，可以看到从 11 号寄存器读出的值为 XXXXXXXX，这也是由于我们未对其进行写操作，而对于 10 寄存器，由于在第一个时钟将 0000ffff 写入了其中，所以成功读出，并且此时 we 为 1，成功向 11 号寄存器中写入 1111ffff；后续阶段也类似。总之，在 part1 阶段，rdata1 依次读取 10、11、12、13、14、15 号寄存器，读出的都为未初始化的 XXXXXXXX，rdata2 依次读取 10，11，12，13，14 号寄存器，为相应的上一个周期刚写入的数据，能够成功读出，we 始终为 1，依次向 10、11、12、13、14 号寄存器写入数据。

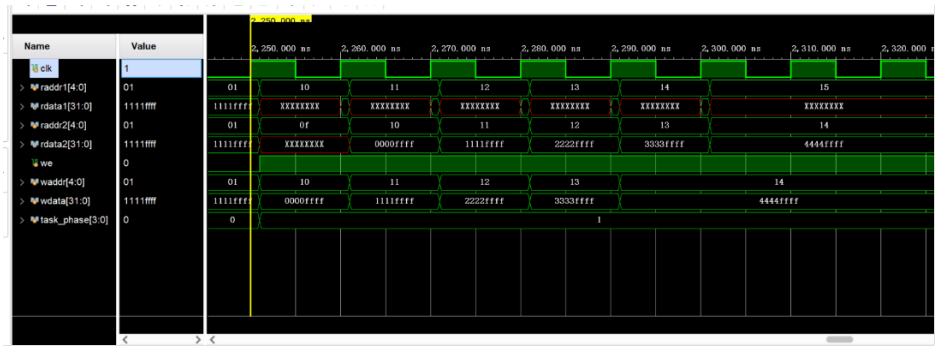

图 1.5: part1 阶段波形图

最后是 part 2 阶段，rdata1 依次读取 10 14 号寄存器，rdata2 依次读取 0f、10 到 13 寄存器，we 为 1，向 14 寄存器写入 4444ffff，我们可以看到读出的内容都为我们 part1 阶段存储的内容，而对于 0f 寄存器，由于为对其初始化，所以仍显示 XXXXXXXX。至此仿真结束，我们完成了对寄存器堆两读一写功能的验证。

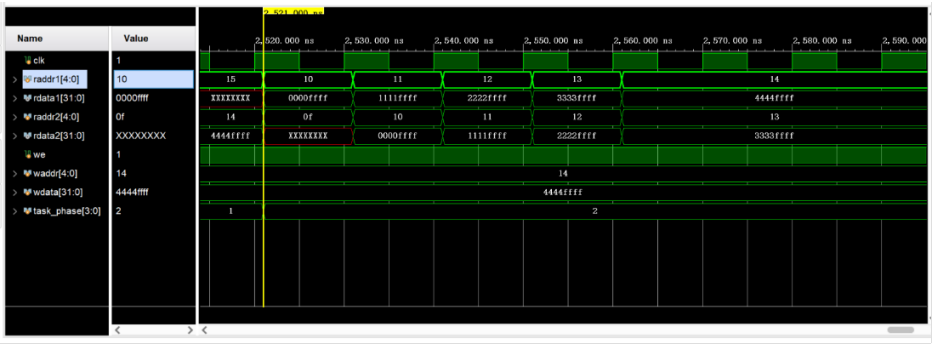

图 1.6: part2 阶段波形图

1.2 思考题：为什么寄存器堆设置为两读一写

1. 两读：对于大多数的指令来说，一般都是对两个操作数进行操作，比如 add 指令，我们需要目的操作数和源操作数，也就是要同时从寄存器堆中读取访问两块内容，所以我们需要设置两个独立的读取端口，达到指令的设计要求。

2. 一写：要是用了多个写端口，可能会发生冲突，造成对数据的危害，并且在大多数情况下一条指令只有一个目的操作数，也就是说只需要一个写入端口。

3. 两读一写的设计原则使得对寄存器堆进行操作时，实现了并行，从而支持流水，能够加快数据读取速率，减少了延迟，并且避免了冲突，实现了可靠性。

2 任务二：同步 RAM 和异步 RAM 实验

2.1 同步 RAM 实现与仿真

2.1.1 定制同步 RAM IP 核

1. 首先在 Project Manager 点击 “IP Catalog”。

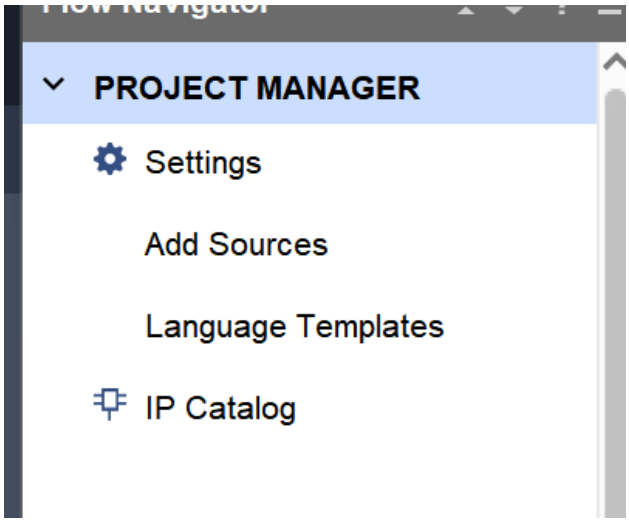

图 2.7: 在 project manager 中选择 IP Catalog

2. 接着在 IP Catalog 中选择 Memories and Storage Elements->RAM&ROMS&BRAM->Block Memory Generator。

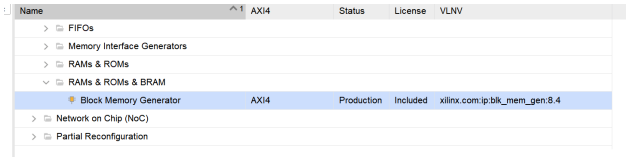

图 2.8: 在 IP Catalog 中选择 Block Memory Generator

3. 在打开的 IP 定制界面中设置 RAM 参数

(1) 在 RAM 界面的 “Basic” 选项卡里将 IP 重命名为 “block_ram”，内存类型设为单端口 RAM，不要勾选 “Byte Write Enable”。

(2) 在 “PortAOptions” 选项卡中将 RAM 深度设为 65536，宽度设为 32, 使能端口设为 “Always Enabled”’ 不要勾选 ‘‘Primitives Output Register”，其他保持默认设置即可，然后点击 “OK”。

这样我们就完成了同步 RAM IP 核的定制。

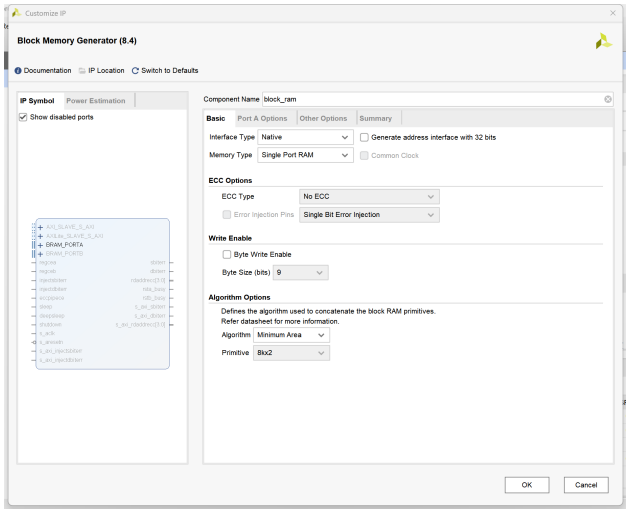

图 2.9: 设置同步 RAM 的 Basic 参数

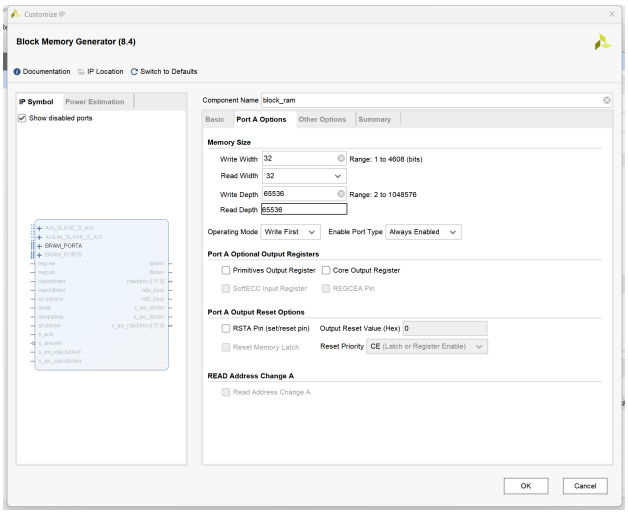

图 2.10: 设置同步 RAM 的 Port A Options 参数

2.1.2 同步 RAM 仿真分析

1. 首先是初始化阶段，在 2000ns 之前，我们把使能端 wen，操作地址 addr，写入数据 wdata 都初始化为 0，为后续实验做准备。

2. 接着在 2011ns 我们进入 part0 阶段。首先我们可以看一下 addr 为 00f1,rdata 为 11223344 这一处，我们发现，我们之前并未对 00f1 这个地址写入数据，却返回 11223344，仔细观察不难发现，这里的 rdata 读出的是 addr 上一次存储地址的内容，也就是 00f0 中的内容，通过观察可以发现读写操作总是在时钟上升沿才发生，而读操作开始时比 addr 更新早了 1ns，这也就是为什么我们进行读操作时总是对 addr 存储的上个地址进行操作。总之，在 part0 阶段，我们将 11223344 写入了 00f0。

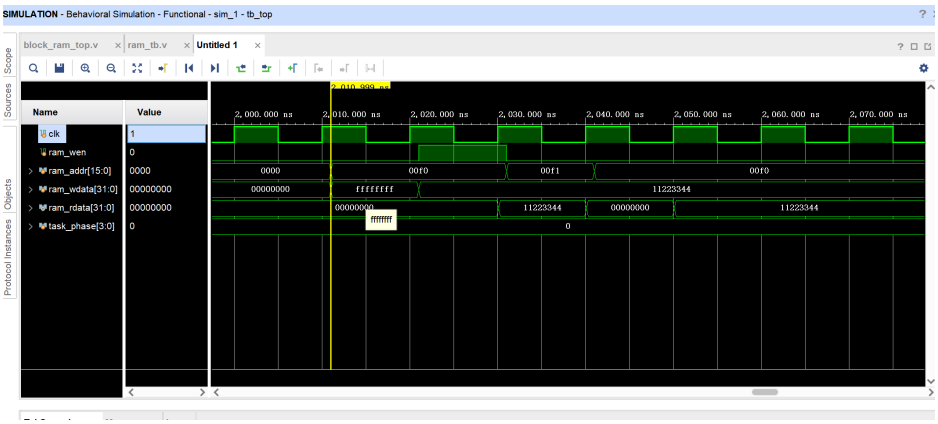

图 2.11: part 0 阶段波形图

3. 接着是 part1 阶段，使能端始终为 1，通过观察时钟与其他存储，我们可以知道这一阶段依次将 0000ff00、0000ff11、0000ff22、0000ff33、0000ff44 写入了 00f0、00f1、00f2、00f3、00f4，并且总是在下一个周期读出上一个周期存储的值。

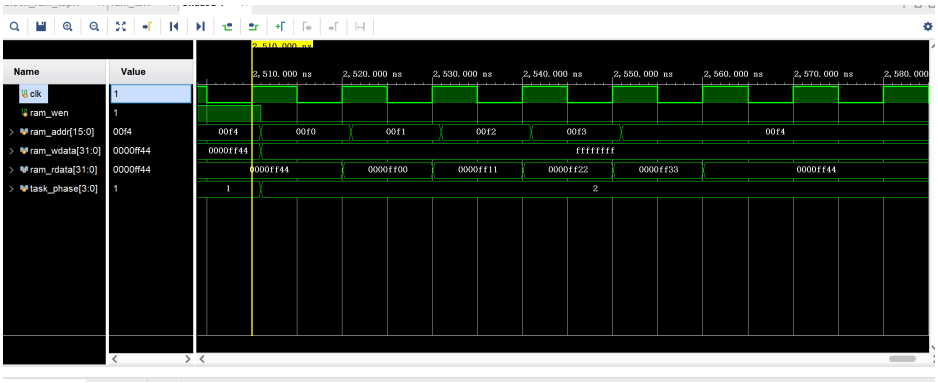

图 2.12: part 1 阶段波形图

4. 在 part2 阶段，使能端始终为 0，不进行写操作，故不会写入 ffffffff，可以看到我们总是在时钟上升沿读出前一个地址中的内容，依次从 00f0、00f1、00f2、00f3 中读出 0000ff00、0000ff11、0000ff22、0000ff33，至此，同步 ram 仿真结束。

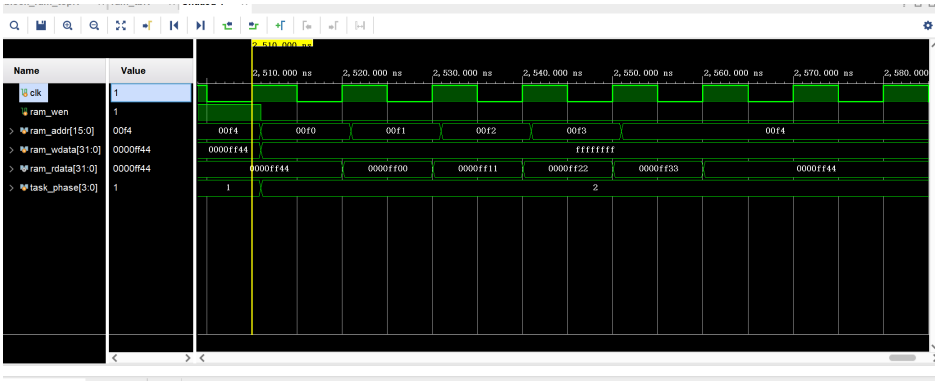

图 2.13: part 2 阶段波形图

2.2 异步 RAM 实现与仿真

2.2.1 定制异步 RAM IP 核

1. 首先在 Project Manager 点击 “IP Catalog”。

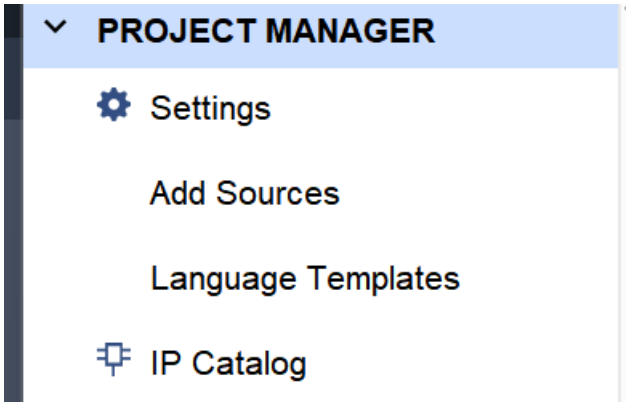

图 2.14: 在 project manager 中选择 IP Catalog

2. 接着在 IP Catalog 中选择 Memories and Storage Elements->RAMs&ROMs->Distributed Memory Generator。

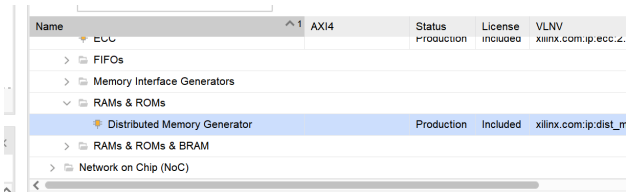

图 2.15: 在 IP Catalog 中选择 Distributed Memory Generator

3. 在打开的 IP 定制界面中设置 RAM 参数，在“memory config”界面将 IP 重命名为“distributed_ram”，内存类型设为单端口 RAM, 深度设为 65536, 宽度设为 32。其他保持默认设置即可。至此完成了异步 RAM IP 核的定制。

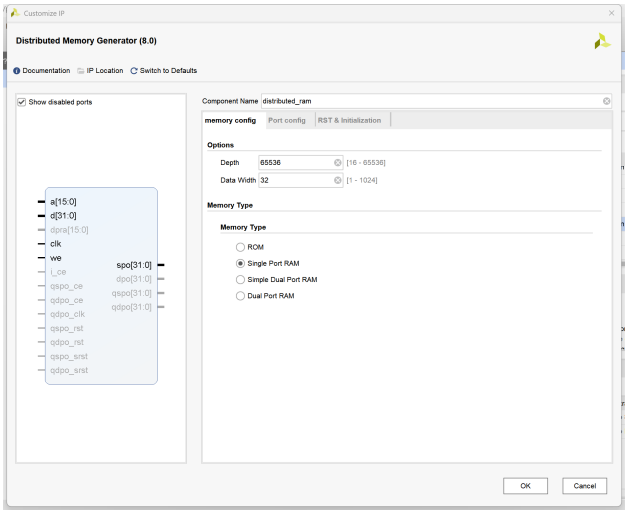

图 2.16: 设置异步 RAM 的参数

2.2.2 异步 RAM 仿真分析

通过观察波形图我们可以看到读写操作不再与时钟同步。

1. 首先是初始化阶段，在 2000ns 之前，我们把使能端 wen，操作地址 addr，写入数据 wdata 都初始化为 0，为后续实验做准备。

接着进入 part0 阶段，首先开始使能端为 0，不会向 00f0 中写入 fffffff，当使能端为 1 时，向 00f0 中写入了 11223344，再之后使能端又变为 0，由于未对 00f1 进行写操作所以显示 00000000，而 00f0 处显示了 11223344，这也验证了之前的写操作成功执行，在 part0 阶段，完成了把 11223344 写入 00f0 的操作。

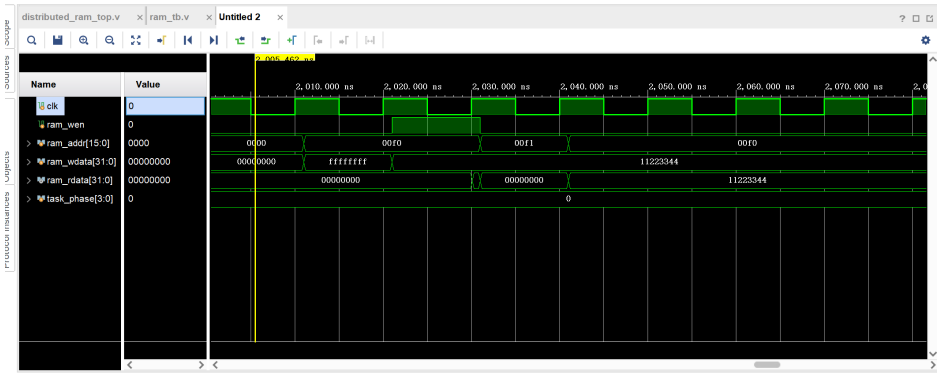

图 2.17: part 0 阶段波形图

2. 接着进入 part1 阶段，使能端始终为 1，我们将 0000ff00 写入了 00f0 中，接着依次向 00f1，00f2，00f3，00f4 中写入了 0000ff11、0000ff22、0000ff33、0000ff44，可以看到由于我们之前未对这些地址进行写操作所以访问读取时全返回为 0，在向 00f4 写入 0000ff44 之后的下一个周期，我们可以看到成功从其中读出了内容。

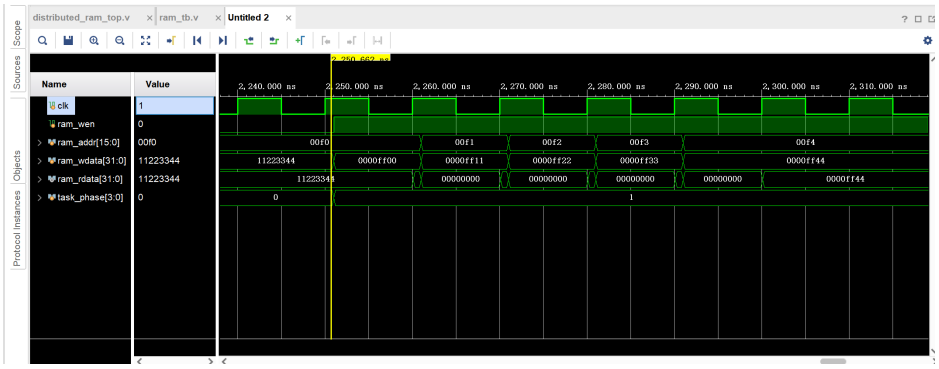

图 2.18: part 1 阶段波形图

3. 接着进入 part2 阶段，使能端始终为 0，由于在 part1 阶段我们成功将相应数据写入了相应寄存器，所以我们在此阶段可以从 00f0、00f1、00f2、00f3、00f4 中依次读出 0000ff00、0000ff11、0000ff22、0000ff33、0000ff44，并且可以知道此时不会把 ffffffff 写入相应寄存器，这点可以通过我们读取了 00f4 中的内容后的下一个时间读出的仍为 0000ff44 来验证。至此，异步 ram 仿真结束。

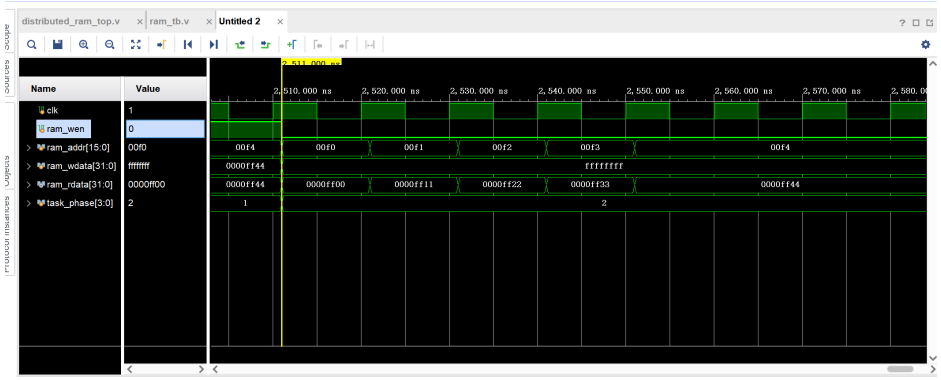

图 2.19: part 2 阶段波形图

2.3 查看时序结果和资源利用率

2.3.1 同步 RAM 结果

1. 通过观察同步 RAM 的时序结果，我们可以看到其 WNS 值为 2.439，由于 WNS 表示为最长路径的违约值，则说明所有路径都满足时序的要求，说明时序满足极好，而 TNF 值为 0，TNF 表示的是总负时序裕量，也就是所有 WNS 的和，其值为 0 说明没有冲突，所有时序约束都已满足。

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP	Start	Elapsed
synth_1 (active)	constrs_1	synth_design Complete!								0	0	0.0	0	0	4/3/25, 3:56 PM	00:00:18
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	4/3/25, 3:57 PM	00:01:54
Out-of-Context Module Runs																
block_ram_synth_1	block_ram	synth_design Complete!								124	4	58.0	0	0	4/3/25, 3:54 PM	00:01:03

图 2.20: 同步 RAM 时序结果

2. 观察同步 RAM 的资源利用率，LUT 为 1%,FF 为 1%，BRAM 为 16%,IO 为 21%,BUFG 为 3%。

通过查阅相关资料，我了解到 LUT 表示的是一个基本逻辑单元，用于实现组合逻辑功能，FF 表示的是一个基本存储单元，用于实现时序逻辑，BRAM 是 FPGA 中的嵌入式存储器块，通常用于存储较大的数据结构，可以实现深度较大的 RAM，IO 是 FPGA 中的接口资源，用于与外部设备进行通信，BUFG 用于将时钟信号分发到 FPGA 的各个部分，确保时钟信号的稳定性和同步性。

较小的 LUT、FF 利用率也就说明了该电路需要用到的逻辑单元很少，逻辑结构比较简单，不需要太多的逻辑单元，BRAM 利用率为 16%，说明利用了一些存储器块用于存储数据，IO 利用率为 21%说明进行了一定程度的读写操作，利用了一些 IO 端口，而 BUFG 利用率低，说明设计复杂度较低。

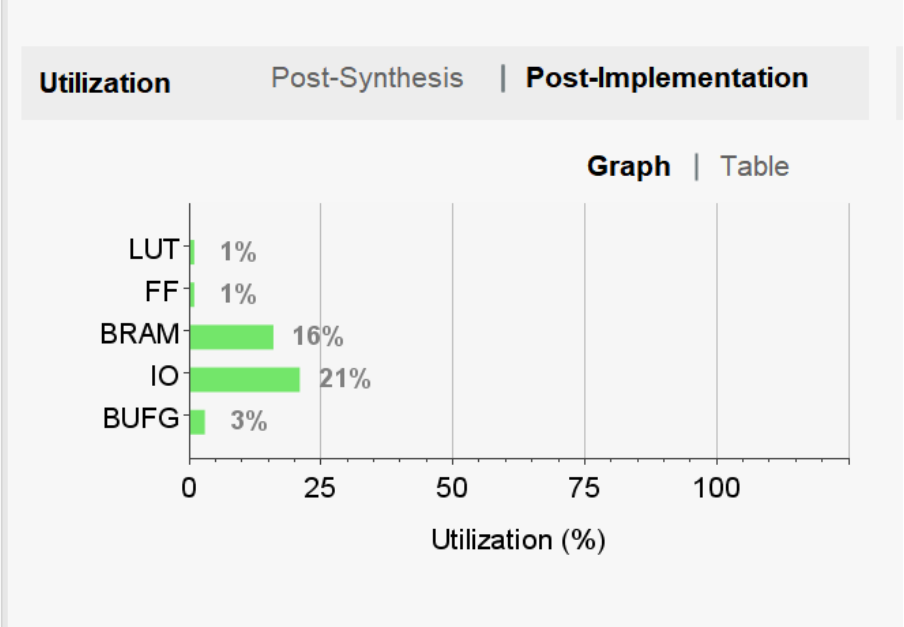

图 2.21: 同步 RAM 资源利用率

2.3.2 异步 RAM 结果

1. 观察异步 RAM 时序结果，其 WNS 值为-3.648，也就是最长路径的违约值为-3.648，不如同步 RAM 的时序性，TNS 值为-219.196，说明与同步 RAM 相比，其时序满足性较差。

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP	S
Out-of-Context Module Runs															
distributed_ram_synth_1	distributed_ram	synth_design Complete!								41357	32	0.0	0	0	4
synth_1 (active)	constrs_1	synth_design Complete!								0	0	0.0	0	0	4
impl_1	constrs_1	route_design Complete, Failed Timing!	-3.648	-219.196	0.702	0.000	0.000	0.385	0	41357	0	0.0	0	0	4

图 2.22: 异步 RAM 时序结果

2. 观察异步 RAM 资源利用率，LUT 利用率为 31%,LUTRAM 利用率为 71%，IO 利用率为 21%,BUFG 利用率为 3%，其中 IO、BUFG 利用率与同步 RAM 相同。

LUT 利用率高于同步 RAM，说明与同步 RAM 相比，异步 RAM 逻辑设计需要更多的逻辑单元。LUTRAM 代表的是利用 LUT 资源实现的小型 RAM 存储器，通常用于存储小量数据，而其利用率高达 71% 说明其需要更多的存储数据的单元。与同步 RAM 相比，异步 RAM 需要更多的资源消耗。

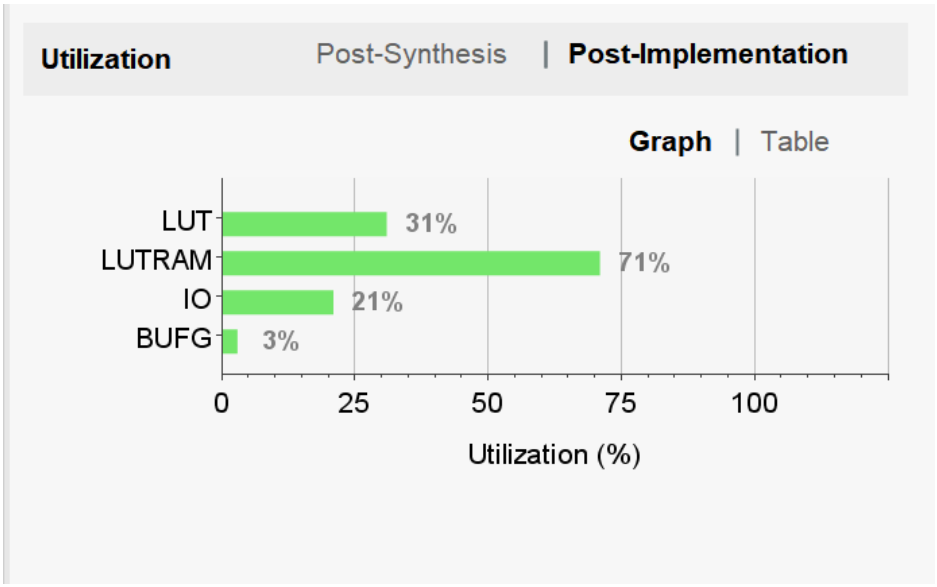

图 2.23: 异步 RAM 资源利用率

2.4 同步 RAM 和异步 RAM 各自的特点和区别

1. 首先一个明显的区别就是，同步 RAM 的读写操作依赖时钟信号，在时钟上升沿才会进行相应的读写操作，时序严格受控，而对于异步 RAM 来说，其读写操作不依赖时钟信号，由控制信号直接触发，无需时钟同步。

2. 另外，通过资源利用率及时序结果来看，异步 RAM 的稳定性和时序性与同步 RAM 相比较差。

3. 在实验时，同步 RAM 的综合实现需要的时间很短，而异步 RAM 综合实现需要很长时间，这是由于异步 RAM 时序分析复杂度高，会产生时序冲突，并且利用了更多逻辑单元和存储单元，需要更多的组合逻辑，占用了更多的资源。

3 任务三：数字逻辑电路的设计与调试实验

1. 在导入相应代码进行综合实现之后，进行仿真模拟，我们得到仿真波形图，首先可以看到，在仿真波形图上，缺少了相关数据的显示，这就需要我们去仿真文件中找出问题。

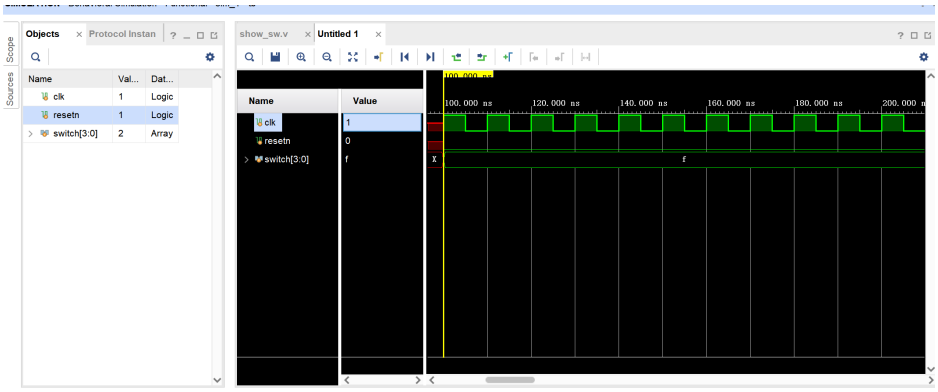

图 3.24: 在波形图上缺失数据显示

在仿真文件中查找问题，我们发现并没有对 num_csm、num_a_g、led 进行声明，我们需要在 tb 文件中加上相应的声明语句。

```
reg         clk    ;
reg         resetn;
reg [3 :0] switch;     //input
wire[7 :0] num_csn;    //new value
wire[6 :0] num_a_g;
wire[3 :0] led;         //previous value
```

2. 在进行了上述修改之后，我们再次进行仿真模拟，可以看到出现了波形为 “Z” 的情况，可能出现了以下两种情况的问题，

(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。

通过分析，由于我们之前修改了声明部分，可能在 tb 文件中对应的接口部分也没作添加修改，我们找到这段代码，可以发现确实如此，属于第二种问题，模块调用的信号未连接导致信号悬空，并且这里应该是显示的未连接的情况，我们需要在端口中加入我们添加的变量。

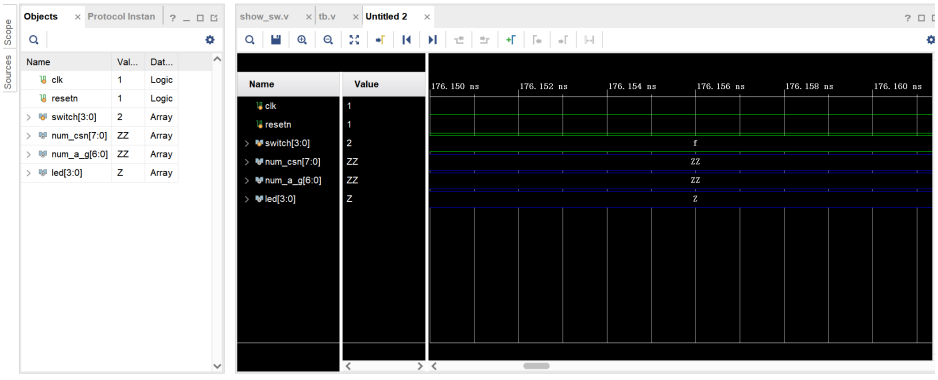

图 3.25: 波形为 “Z”

在 tb 文件中找到对应代码处，添加相应代码如下所示：

```
show_sw   u_show_sw(
    .clk     (clk     ),
    .resetn  (resetn  ),
```

```
    .switch (switch ),     //input
    .num_csn(num_csn),    //new value
    .num_a_g(num_a_g),
    .led     (led)     //previous value
);
```

在完成了如上修改之后，再次仿真，只有 num_csm 仍为 “Z” 型波形，说明对于 num_csm 仍存在之前的两种潜在问题。

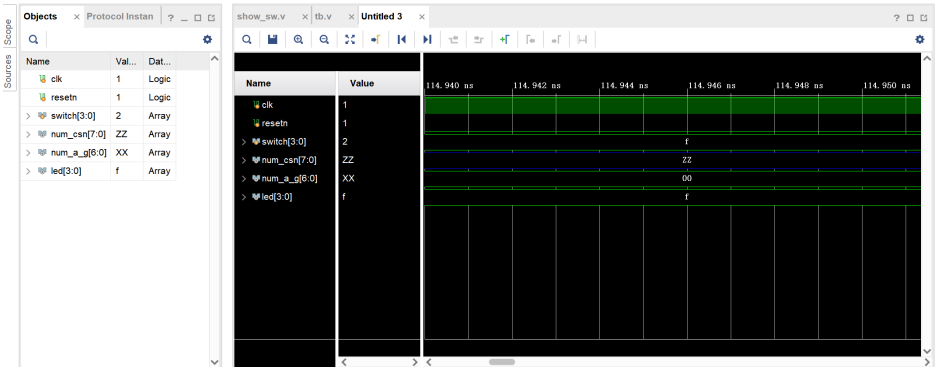

图 3.26: num_csn 波形为 “Z”

在 tb 文件中没有发现问题，我们前往查看资源文件，发现在向 show_num 函数中传参的时候变量名字写错了，我们对其进行改正。

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

        .show_data  (show_data),
        .num_csn    (num_csn  ),
        .num_a_g    (num_a_g  )
);
```

3. 再次运行仿真文件，发现在 600ns 的时候，出现了 X 型波形，而 X 波形出现的原因可能有以下两种情况：

(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。

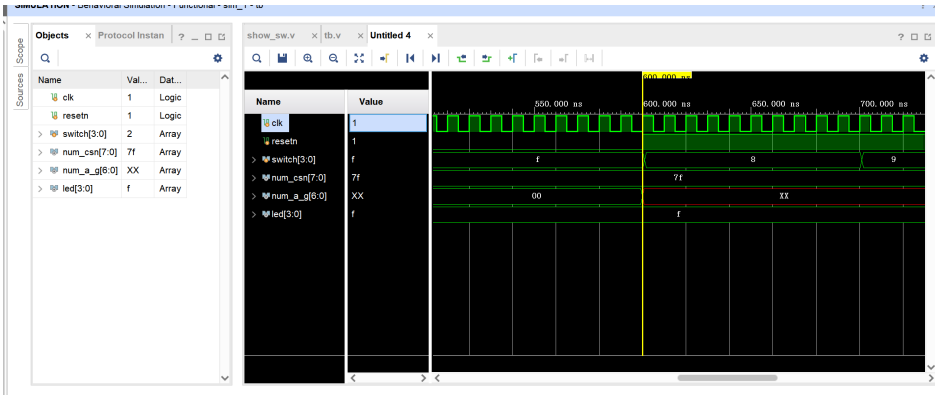

图 3.27: 波形为“X”

在这里由于没有多驱动的代码，所以应该是第一种情况，也就是 num_a_g 的赋值过程出现了问题，而其赋值过程应与 nxt_a_g 有关，而 nxt_a_g 的赋值应与 show_data 有关，通过检查 design 文件，我们发现没有对 show_data 进行相应的赋值操作，并且由于其是 reg 型变量，符合典型出错的情况。我们进行如下修改：

```
always @(posedge clk)
begin
    if(!resetn)
        show_data<=4'd0;
    else
        show_data<=~switch;
//    show_data    <= ~switch;
end
```

另外在前 100ns 时应该也要对 reg 变量赋初值，在 tb 文件中应该作如下修改：

```
initial
begin

    clk    = 1'b0;
    resetn = 1'b0;
    #100;
    #500;
    resetn = 1'b1;
end
always #5 clk = ~clk;

//set switch
initial
begin
    switch = 4'hf;
    #100;
    #500;
```

4. 在进行了上述修改之后，再次运行仿真文件，可以看到运行卡住了，图片中的这种情况正是波形停止的表现，说明出现了组合环路。

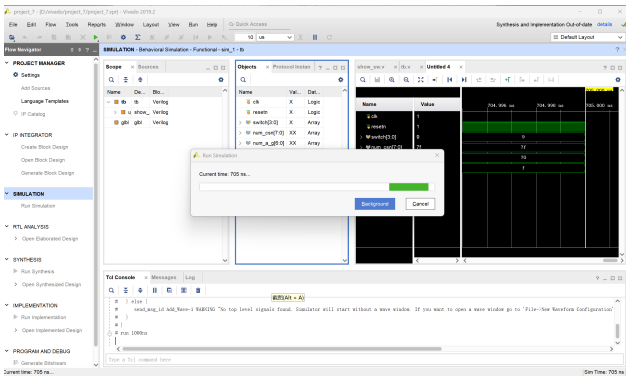

图 3.28: 发生“波形停止”

在停止仿真运行后，我们发现代码运行在如下位置，也就是说在这里发生了组合环路，对这段代码进行分析，我们可以看出，keep_a_g 的赋值需要利用到 nxt_a_g，而当 show_data 超过了 0 到 9 的范围的话，nxt_a_g 的赋值需要利用到 keep_a_g，也就造成了组合环路，所以我们可以删除 keep_a_g 的赋值需要利用到 nxt_a_g 这一行为，只把 num_a_g 赋值给 keep_a_g。

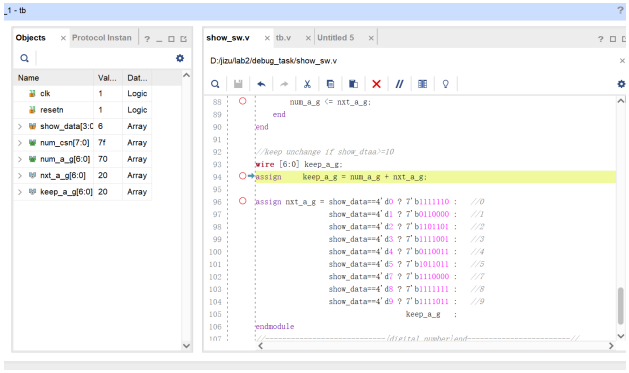

图 3.29: 停止仿真时代码中断处

修改代码如下：

```
wire [6:0] keep_a_g;
assign     keep_a_g = num_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'b1110000 :   //7
                 show_data==4'd8 ? 7'b1111111 :   //8
                 show_data==4'd9 ? 7'b1111011 :   //9
                                   keep_a_g   ;
```

5. 再次运行仿真，发现波形图没问题。但还有越沿采样的 bug 没进行审查，越沿采样的 bug 比较明显，在组合逻辑电路中我们需要立即更新，用阻塞赋值，在时序逻辑中，我们要确保在时钟上升或下

降沿获得稳定的输入值，所以要用非阻塞赋值，我们只需检查源码中阻塞赋值和非阻塞赋值混用的代码，修改如下：

```
always @(posedge clk)
begin
    show_data_r <= show_data; //此处改为非阻塞赋值
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
```

至此，我们完成了所有的典型 bug 检查。6. 最后一步观察仿真波形图检验在逻辑上是否发生错误。如图所示，在 switch 为 1001 的时候，其取反为 0110，在 0 到 9 之间，应该对数码管的显示进行修改，而此处却没有修改说明存在问题。

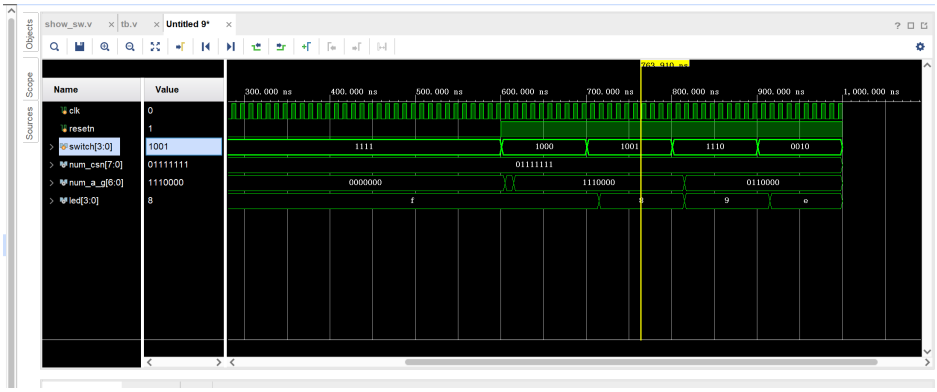

图 3.30: 功能 bug

在审查代码后我们发现，缺少了在 show_data=0110 的情况下为 nxt_a_g 的赋值，我们添加这一处：

```
wire [6:0] keep_a_g;
assign     keep_a_g = num_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
```

```
10                  show_data==4'd6 ? 7'b1011111 :    //6 新添加
11                  show_data==4'd7 ? 7'b1110000 :    //7
12                  show_data==4'd8 ? 7'b1111111 :    //8
13                  show_data==4'd9 ? 7'b1111011 :    //9
14                                    keep_a_g    ;
```

现在全部修改完毕，运行仿真后观察波形图发现完全正确。

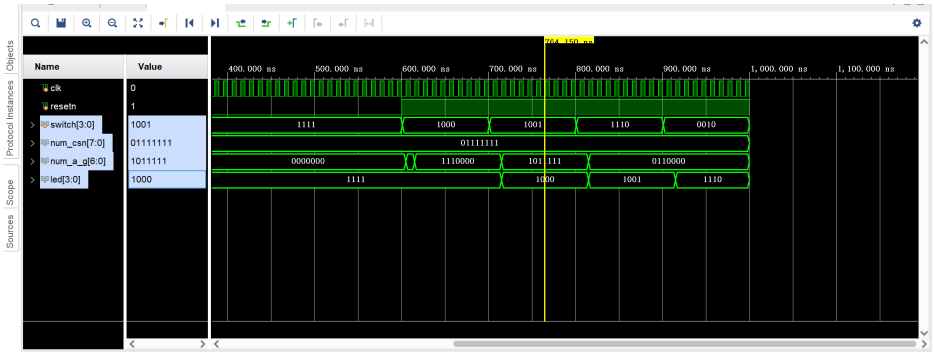

图 3.31: 修改完后的波形图

4 反思总结

1. 了解了什么是寄存器堆、同步 ram 和异步 ram，对计算机中存储模块有了更好的了解；

2. 知道了寄存器堆中设置成两读一写的原因，并且学会了分析读写过程；

3. 在同步 ram 与异步 ram 实验中，学会了如何定制一个 IP 核，并且学会进一步使用 vivado 去查看时序结果与资源利用率，并可以根据相关参数分析出时序性、资源利用的情况，另外还了解到了同步 ram 与异步 ram 各自的特点和区别；

4. 在最后的 debug 实验中，我学会了如何在 vivado 中调试一个程序，其本质与 print 大法类似，都是通过显式的内容去判断代码中可能存在的问题。另外，在这个实验中，我认为我还有以下收获：

(1) 对于 Z 型波形来说，其最有可能的就是未对 wire 型变量赋值，要注意在模块调用时把相应信号传入端口，也有可能不小心把信号名字写错导致发生 Z 型波形；

(2) 对于 X 型波形来说，其最有可能的就是未对 reg 型变量赋值，我们要学会从发现的问题出发，追溯到最根本出现的问题；

(3) 对于波形停止，其现象比较明显，我们可以中断仿真前往代码中断处，通过判断是否发生组合环路，我们可以得出相应的推断；

(4) 对于越沿采样，其现象更为明显，在组合逻辑电路中我们需要立即更新，用阻塞赋值，在时序逻辑中，我们要确保在时钟上升或下降沿获得稳定的输入值，所以要用非阻塞赋值，我们只需检查源码中阻塞赋值和非阻塞赋值混用的代码；

(5) 另外，还要注意实际输出是否符合问题目标。

5. 另外，我更好地掌握了 vivado 的使用。