实验 2 报告

学号：2017K8009922026
姓名：康齐瀚
箱子号：63

一、实验任务（10%）

本次实验由三个子任务组成：

1. 建立寄存器堆的工程，并利用提供的设计代码和 testbench 代码进行仿真，按照波形描述寄存器堆的读写行为
2. 分别建立同步 RAM 和异步 RAM 的工程，利用提供的代码进行仿真，实现和综合。通过波形图描述同步 RAM 和异步 RAM 的读写行为，并分析其在时序和资源利用率上的异同
3. 找出并修改课程提供的源代码中的 bug，要求最后的设计能上板运行

子任务 1 的目的在于帮助我们理解寄存器堆的工作原理，同时再次熟悉 Vivado 软件的使用流程以及对波形图的使用和分析能力。理解好寄存器堆也是以后设计 CPU 的基础。

子任务 2 的目的在于帮助加深对时序关系的理解，因为 CPU 的设计中将大量使用到时序逻辑。

子任务 3 的目的在于培养我们的调试分析能力。

二、实验设计（0%）

三、实验过程（90%）

（一）实验流水账

2019/9/5：18：00~19：00 建立了寄存器堆工程并添加了所有源文件，进行了对波形的分析，大致完成了子任务 1

2019/9/6：8：40 ~ 9：40 建立同步 RAM 工程并进行了分析

2019/9/6：18：00 ~ 21：00 完成同步 RAM 工程报告并建立异步 RAM 分析

2019/9/7：12：00 ~ 14：30 完成 debug 任务

（二）子任务一

1. 波形图信号分组及颜色说明：

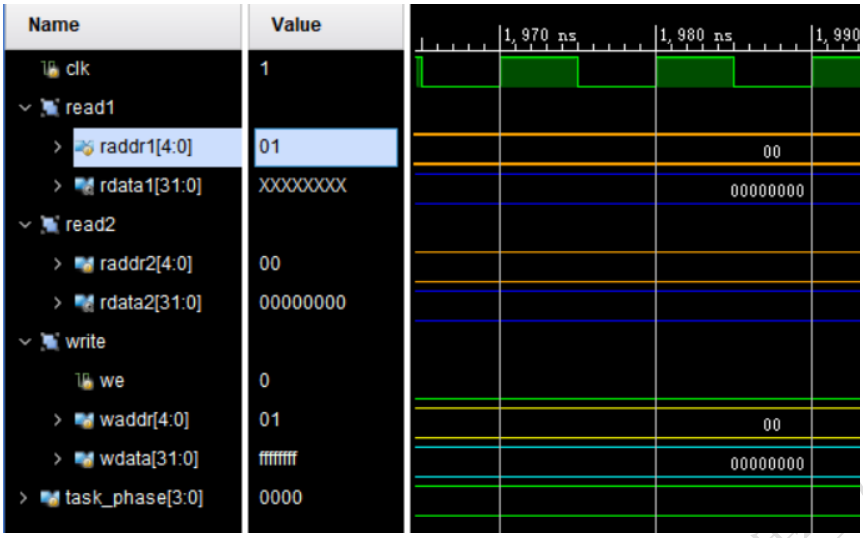


图 1：信号颜色及分组

根据寄存器堆的读写性质可以将信号分为 3 组：

(1) read1 由 raddr1 和 rdata1 组成，它们表示了读寄存器堆的第一个寄存器编号以及读出的数据

(2) read2 由 raddr2 和 rdata2 组成，它们表示了读寄存器堆的第二个寄存器编号以及读出的数据

(3) write 由 we，waddr 和 wdata 组成，它们表示写使能信号，写入的寄存器编号以及写入的数据

在颜色上，地址统一采用黄色系的颜色，数据采用蓝色系的颜色，且 write 组的信号颜色更浅，更利于分辨所有数据在波形图中均以 16 进制形式标出

2. 读写行为描述

总体上看，此次仿真波形的变化可分为三个阶段：

(1) 从开始仿真到 2000 ns 时刻，没有数据写入寄存器堆，一直在读取 0 号寄存器的数据

(2) 从 2000 ns 到 2200 ns 左右，有短暂的数据写入，读取了 1 号寄存器的数据

(3) 2200ns 之后，一直有数据写入，读取 13，14，15 号寄存器的数据。

行为描述 1：

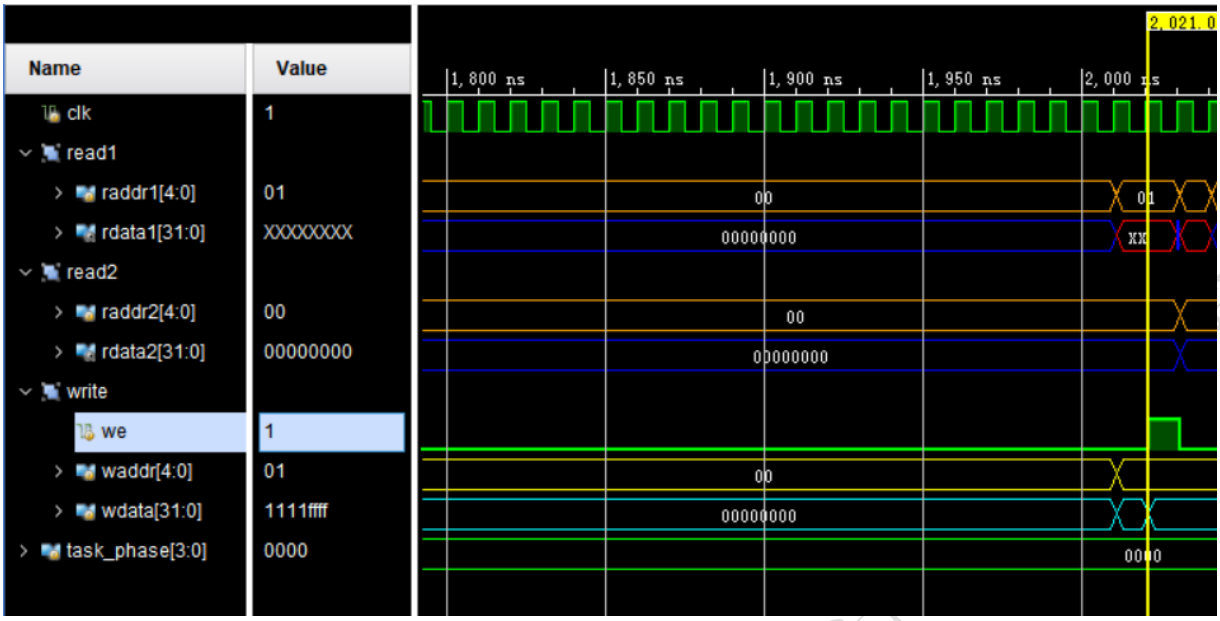


图 2：Part0 波形图(1)

从上图可以看出，在 we 信号拉高之前，没有任何数据写入，raddr1 和 raddr2 均为 0，因此只能从寄存器堆中读出数据 0

行为描述 2：

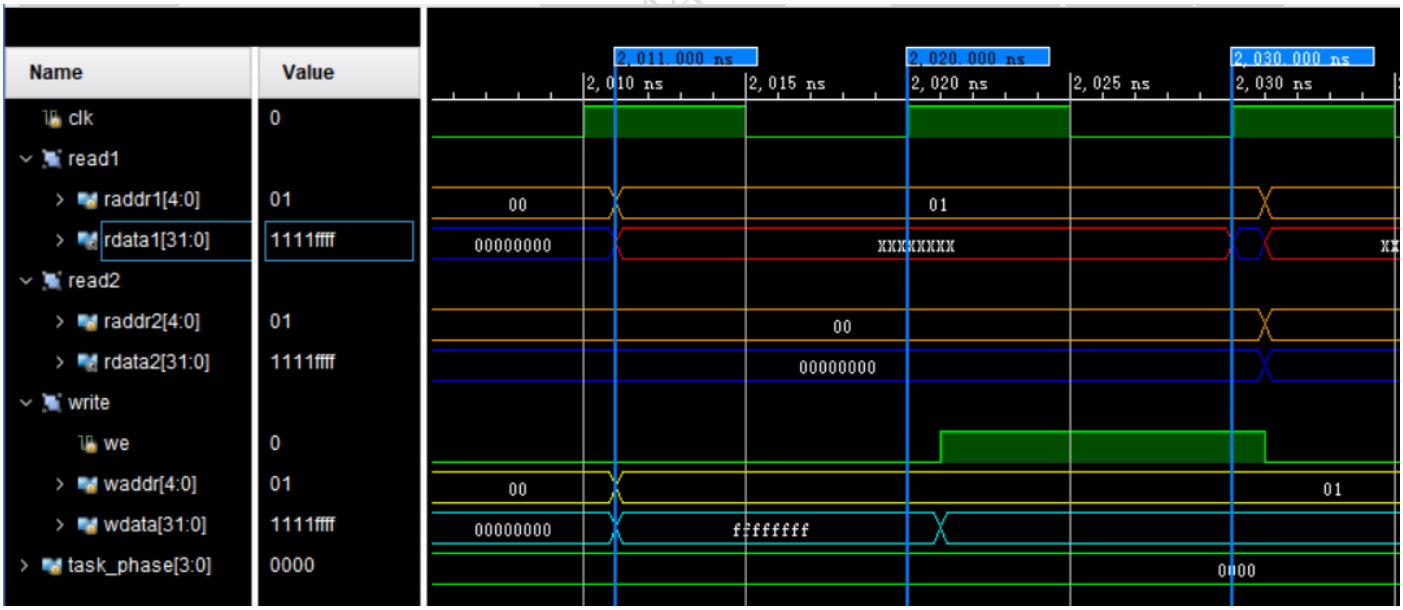


图 3： Part0 波形图(2)

Marker1: raddr1 由 00 变为 01，由于目前还未向寄存器堆的 01 号寄存器中写入数据，因此读出的数据不确定，rdata1 为 X

Marker2：此处为时钟上升沿，但 we 信号处于低位无效，对应的 wdata，即 0xffffffff 无法写入寄存器堆 01 号寄存器，因此此时读出的 rdata 仍为 X

Marker3: 此处为时钟上升沿，we 信号高位有效，因此 wdata 数据将写入 waddr 对应的寄存器中，即 1 号寄存器，此时 raddr1 仍保持为 01，写入寄存器堆的数据立刻读出，因此 rdata1 立即变为写入数据，即 0x1111ffff

Part1：

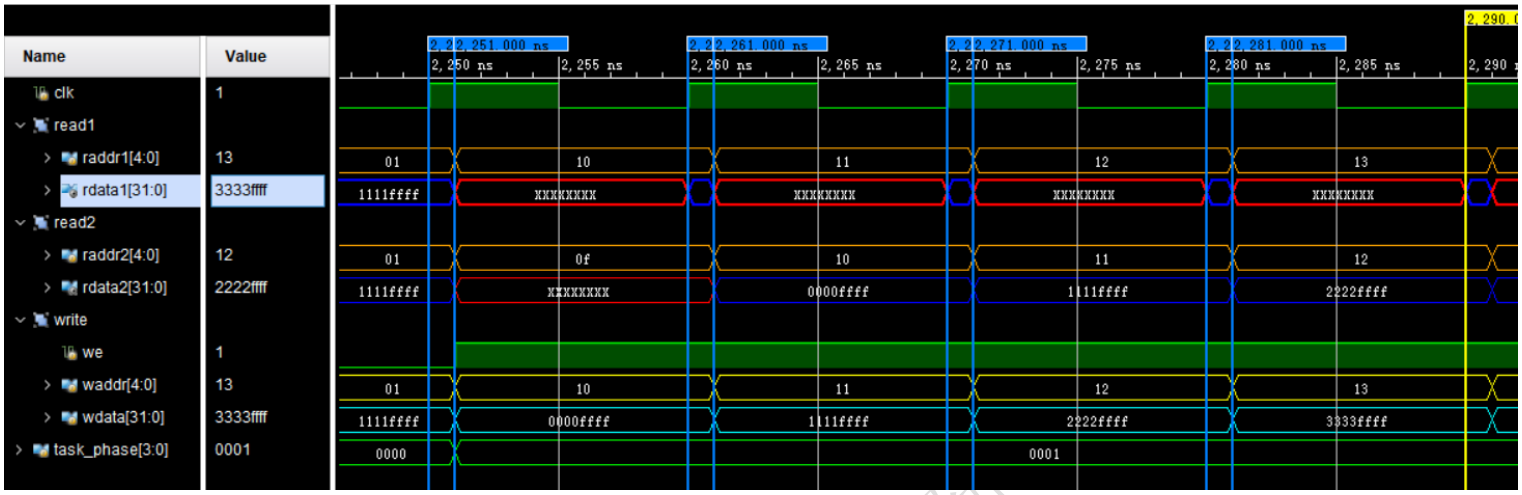


图 4：Part1 波形图

图中所有 Marker 均可两两分为一组，每一组都反映了在时钟上升沿处将数据写入某一寄存器以及马上从该寄存器读出数据的行为。

Marker1：时钟上升沿，we 信号低位无效，无数据写入。rdata1 和 rdata2 均等于 1 号寄存器的值。

Marker2：we 高位有效，但非时钟上升沿，无数据写入。raddr1 和 raddr2 分别变为 0x10 和 0x0f，rdata1 和 rdata2 是组合逻辑，立即变化，但由于 0x10 和 0x0f 寄存器中没有值，输出 X

Marker3: 时钟上升沿，we 高位有效，数据 0x0000ffff 写入 0x10 号寄存器，rdata1 等于 0x10 号寄存器的值，因此立刻变化，输出 0x0000ffff, rdata2 则不变

Marker4: 非时钟上升沿，无数据写入。raddr1 和 raddr2 分别变为 0x11 和 0x10。rdata1 输出 0x11 号寄存器的值，为 X，rdata2 输出 0x10 号寄存器的值，即刚刚写入寄存器堆的值，为 0x0000ffff

Marker5，Marker6，Marker7，Marker8 的读写情况与 Marker3,Marker4 完全相同，此处不再赘述。

Part2：

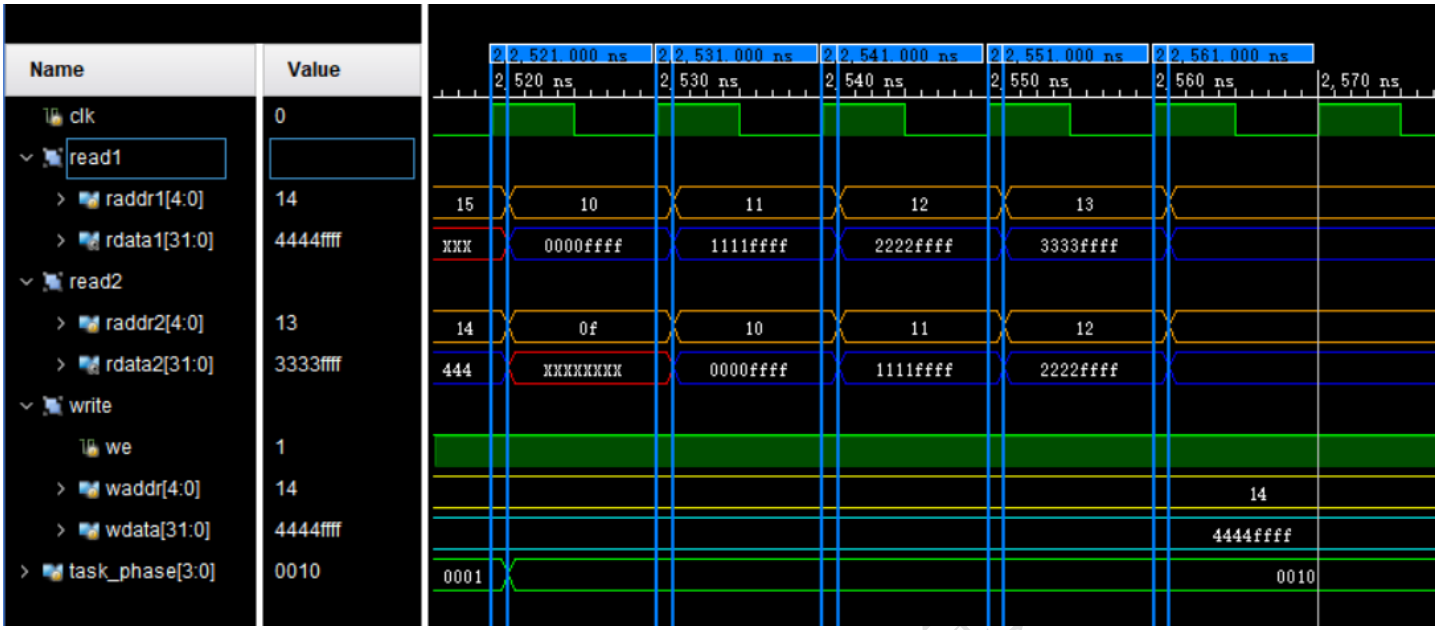


图 5：Part2 波形图

Marker1,3,5,7,9 均是在时钟上升沿处 we 信号高位有效，数据 0x4444ffff 写入寄存器堆。

Marker2,4,6,8,10 处 raddtr1 和 raddr2 变化，rdata1 和 rdata2 的输出由于是组合逻辑，因此会随 raddr1 和 raddr2 同时变化。其中 0x0f 号寄存器在 Part1 和 Part2 始终没有数据写入，因此输出 X；0x10,0x11,0x12,0x13 号寄存器在 Part1 和 Part2 阶段已经有数据写入，因此 rdata1 和 rdata2 会输出它们最后被写入的数据。

（三）子任务二

1、仿真行为对比分析

(1) 同步 RAM

同步 RAM 中所有 D 触发器状态具有相同的时序，由统一的时钟信号控制。同步 RAM 的输入信号 ram_wdata 和 ram_addr 将在时钟上升沿处对 RAM 中的相应信号进行更新。

Part0:

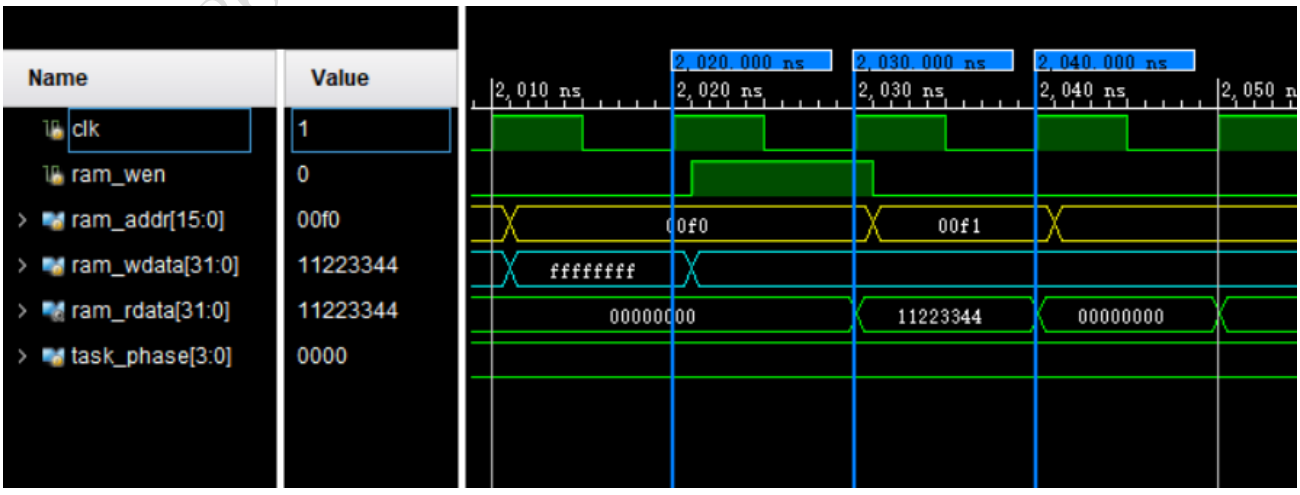


图 6：Part0 波形图

Marker1： 第一个时钟上升沿处，we 低位无效，因此无数据写入，且此时输入信号 ram_addr 为 0x00f0，因此从 Marker1 到 Marker2 这一个时钟周期内 RAM 内部的地址信号都将保持为 0x00f0，ram_rdata 的输出是组合逻辑，因此一直为 0x00000000

Marker2：第二个时钟上升沿处，we 高位有效，输入信号为 0x00f0，从 Marker2 到 Marker3 这一个时钟周期内 RAM 内部的地址信号都将保持为 0x00f0，由于 we 高位有效，输入数据 ram_wdata 写入 RAM 的 0x00f0 处。ram_data 的输出是组合逻辑，在 RAM 的 0x00f0 位置数据更新之后也更新为 0x11223344，直到 Marker3 处 ram_addr 改变

Marker3： 第三个时钟上升沿处，we 处于低位无效，输入信号 ram_addr 为 0x00f1,RAM 内部的地址信号变为 0x00f1 并保持直到下一个时钟周期，ram_rdata 输出 RAM 在 0x00f1 处的内存数据，但由于之前没有数据写入，因此输出为 0x00000000

Part1:

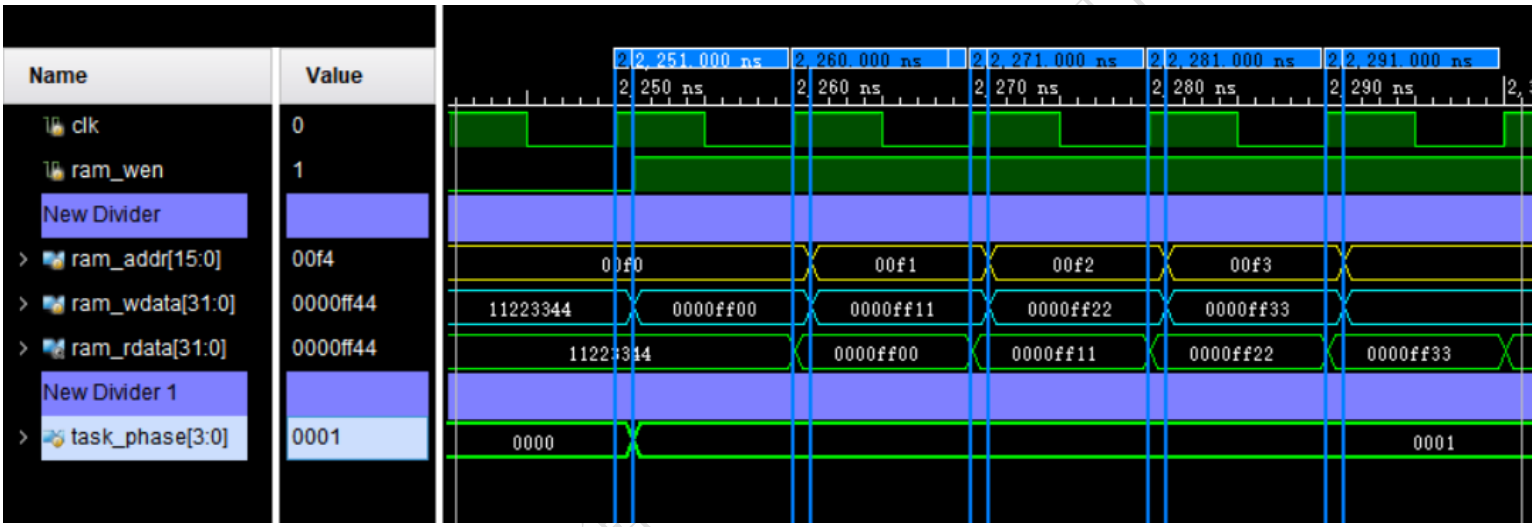


图 7：Part1 波形图

Marker1:时钟上升沿处，we 低位无效，因此没有数据写入，此时输入信号 ram_addr 为 0x00f4,RAM 内部读写地址信号为 0x00f4,读出数据为 0x11223344，这是在 Part0 阶段最后写入 0x00f4 地址处的数据。

Marker2：we 高位有效，但是非时钟上升沿，因此 ram_wdata 没有变化

Marker3：时钟上升沿，ram_we 高位有效，ram_addr 为 0x00f0, 数据 0x0000ff00 写入 RAM0x00f0 位置，读取数据也为 0x00f0 位置的数据，即 ram_rdata=0x0000ff00

Marker4：非时钟上升沿，ram_addr 和 ram_wdata 改变不影响 RAM 中的数据

Marker5：时钟上升沿，ram_we 高位有效，ram_addr 为 0x00f1, 数据 0x0000ff11 写入 RAM0x00f1 位置，ram_rdata 读取 0x00f1 位置处的数据，即 ram_rdata=0x0000ff11

Marker6：非时钟上升沿，ram_addr 和 ram_wdata 的改变不影响 RAM 中数据

Marker7 和 Marker9 处的读写行为类似于 Marker3 和 Marker5 处，Marker8 和 Marker10 的读写行为类似于 Marker4 和 Marker6 处，此处不再赘述。

Part1 阶段结束时，有 RAM[0x00f0]=0x0000ff00, RAM[0x00f1]=0x0000ff11,RAM[0x00f2]=0x0000ff22, RAM[0x00f3]=0x0000ff33

Part2:

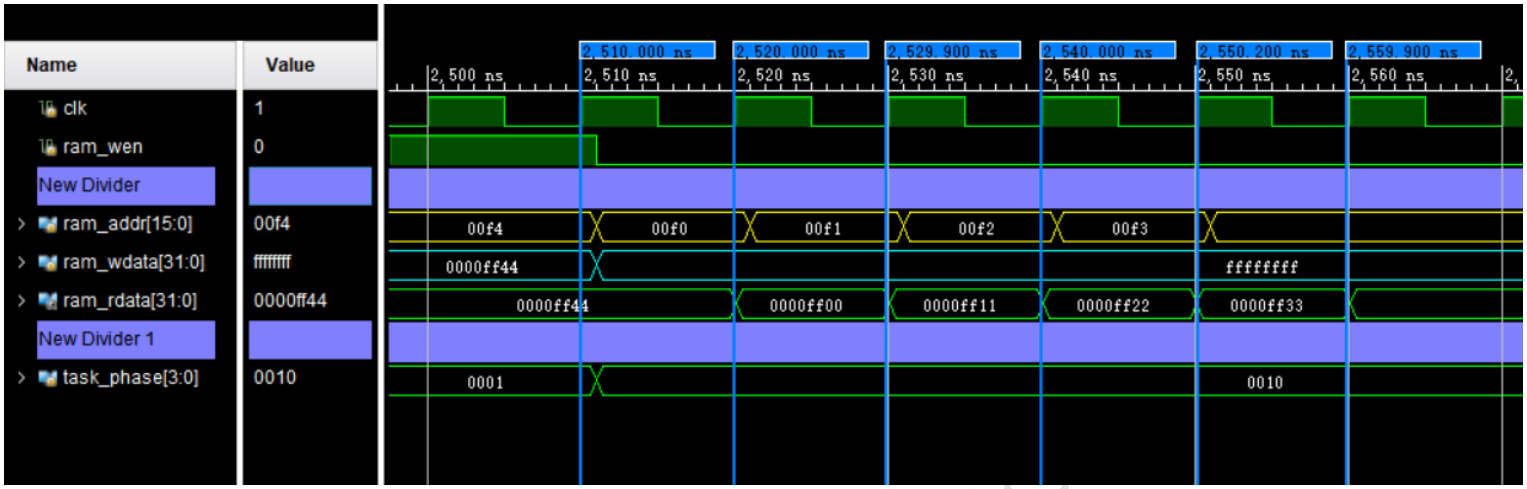


图 8：Part2 波形图

Marker1：时钟上升沿，ram_wen 高位有效，数据 0x0000ff44 写入 0x00f4 位置，同时 ram_rdata 读出该单元数据 0x0000ff44

Marker2：时钟上升沿，ram_wen 低位无效，无数据写入，读出 0x00f0 位置数据，ram_rdata=0x0000ff00

Marker3：时钟上升沿，ram_wen 低位无效，无数据写入，读出 0x00f1 位置数据，ram_rdata=0x0000ff11

Marker4：时钟上升沿，ram_wen 低位无效，无数据写入，读出 0x00f2 位置数据，ram_rdata=0x0000ff22

Marker5：时钟上升沿，ram_wen 低位无效，无数据写入，读出 0x00f3 位置数据，ram_rdata=0x0000ff33

Marker6：时钟上升沿，ram_wen 低位无效，无数据写入，读出 0x00f4 位置数据，ram_rdata=0x0000ff44

(2) 异步 RAM

异步 RAM 通常没有统一的时钟信号控制，其触发器的状态变化依赖于某个信号的变化

异步 RAM 的写操作仍然受时钟控制，但是读操作受 ram_addr 的变化而变化

Part0:

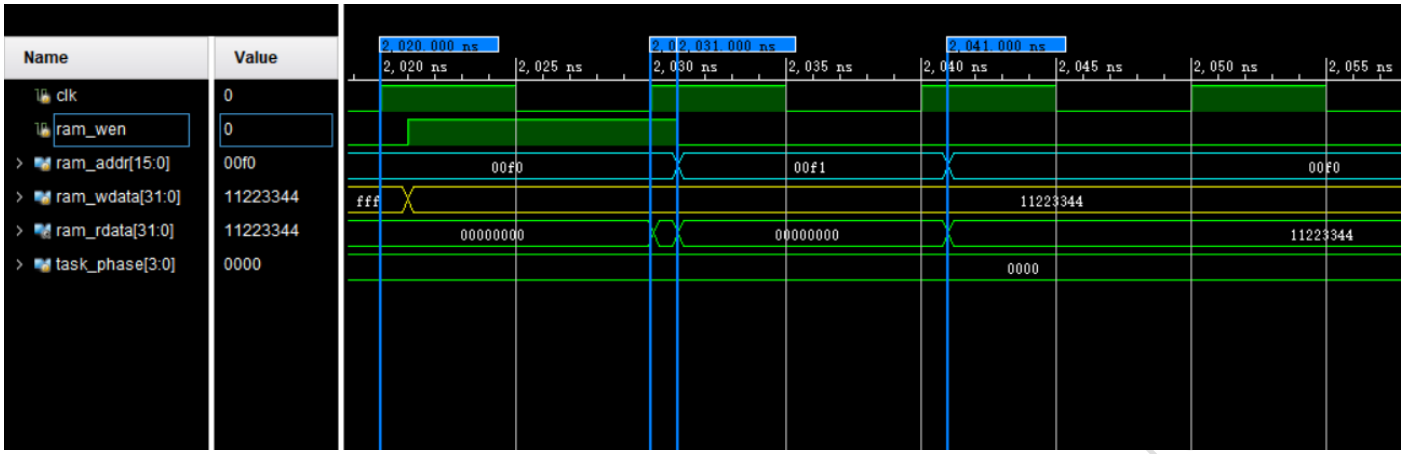


图 9：Part0 波形

Marker1: 时钟上升沿处，ram_wen 信号低位无效，无数据写入，此时 ram_rdata 对应于 ram_addr=0x00f0 读出的数据，即 0x00000000

Marker2:时钟上升沿处，ram_wen 信号高位有效，数据 0x11223344 写入 ram_addr=0x00f0 对应的内存单元，同时此时 ram_rdata 立刻输出 ram_addr 内存单元的数据，即 0x11223344

Marker3:此处 ram_addr 发生变化，ram_rdata 立刻发生变化，输出 0x00f1 位置的数据，即 0x00000000

Marker4:ram_addr 变化，ram_rdata 立即变化，输出 0x00f0 位置的数据，由于此处数据已经在 Marker2 时写入，0x11223344, 因此 ram_rdata=0x11223344

Part1:

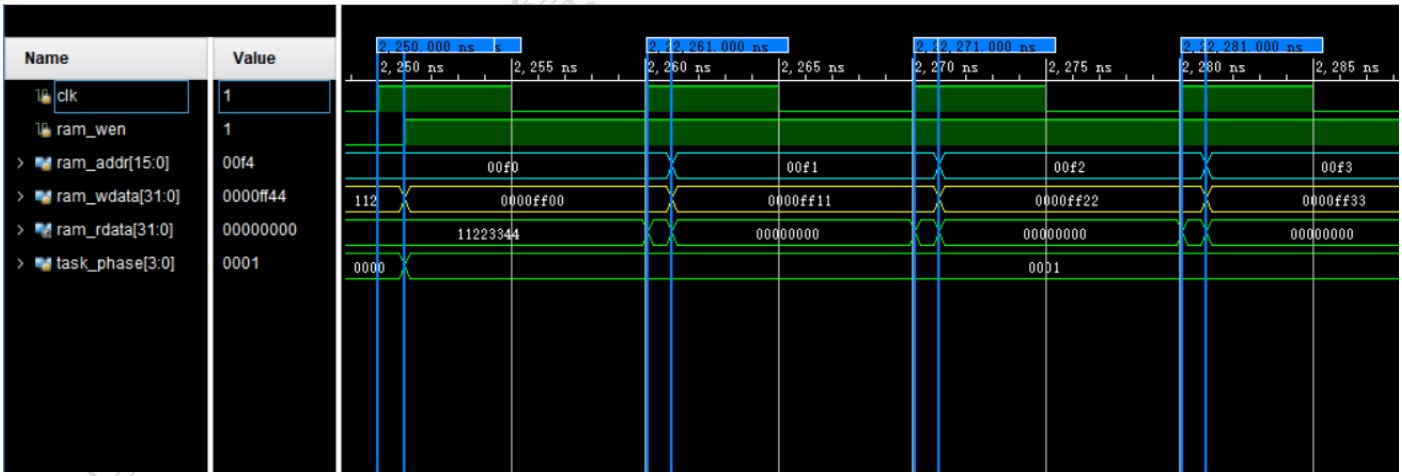


图 10：Part1 波形

Marker1: 时钟上升沿处，ram_wen 低位无效，因此没有数据写入 ram_rdata 输出 0x00f0 内存单元的数据

Marker2:ram_wen 高位有效，但是未处于时钟上升沿，且 ram_addr 无变化，因此无数据写入，ram_rdata 也无变化

Marker3:时钟上升沿，ram_wen 高位有效，数据 0x0000ff00 写入此时的 ram_addr 即 0x00f0 位置，ram_rdata 立刻输出 0x00f0 位置的数据，即输出 0x0000ff00

Marker4:非时钟上升沿，无数据写入，ram_addr 变为 0x00f1，ram_rdata 立即变为输出 0x00f1 处的内存数据，即输出 0x00000000

Marker5 和 Marker6，Marker7 和 Marker8 的行为类似于 Marker3 和 Marker4，此处不再赘述

Part1 阶段结束时，有 RAM[0x00f0]=0x0000ff00, RAM[0x00f1]=0x0000ff11,RAM[0x00f2]=0x0000ff22, RAM[0x00f3]=0x0000ff33

Part2:

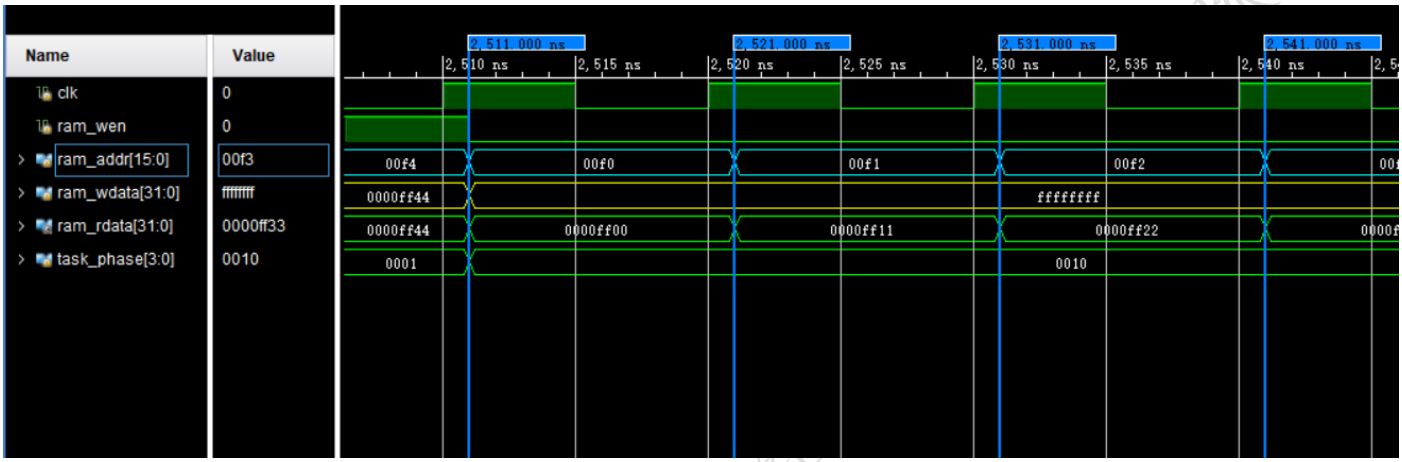


图 11：Part2 波形

Marker1~Marker2: ram_addr 为 0x00f0, ram_rdata 输出 0x00f0 位置的数据，输出 0x0000ff00

Marker2:ram_addr 变化为 0x00f1, ram_rdata 立刻输出 0x00f1 位置的数据，输出 0x0000ff11

Marker3:ram_addr 变化为 0x00f2, ram_rdata 立刻输出 0x00f2 位置的数据，输出 0x0000ff22

Marker4:类似于上述 Marker 的变化

(1)同步 RAM 和异步 RAM 的异同:

同步 RAM 的读写都是由时钟统一控制的，而异步 RAM 的读写则不同

同步 RAM 和异步 RAM 的写操作均由时钟同一控制，即在时钟上升沿处若 ram_wen 信号有效则直接写入，但同步 RAM 的读操作则取决于时钟上升沿处的 ram_addr 信号，在时钟上升沿处，该信号用于更新 RAM 内部的读写地址，因此输出信号 ram_rdata 也只有可能在时钟上升沿处才会发生变化；异步 RAM 的读行为则根据 ram_addr 的变化实时更新，行为上更像是组合逻辑，类似于寄存器

2、时序、资源占用对比分析

(1)资源占比

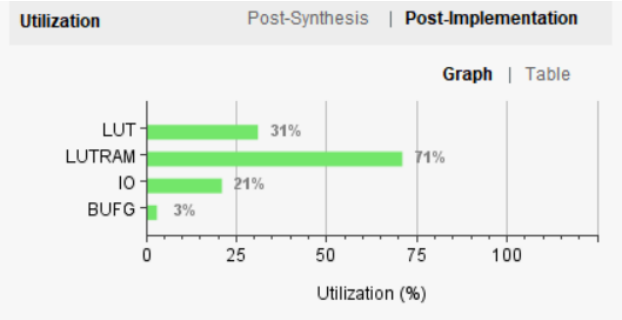


图 12：异步 RAM 资源占比

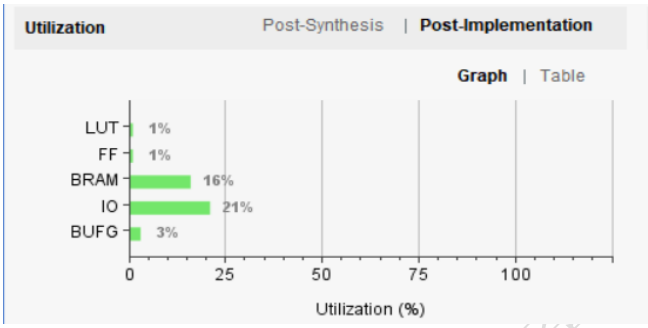


图 13：同步 RAM 资源占比

同步 RAM 和异步 RAM 在 IO 使用率和 BUFG 使用率上相同。
异步 RAM 占用更多的 LUT 资源和 LUTRAM 资源

（2）时序结果对比

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS
synth_1 (active)	constrs_1	synth_design Complete!					
impl_1	constrs_1	route_design Complete, Failed Timing!	-6.536	-6406.483	0.153	0.000	0.000

图 14：异步 RAM 时序结果

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS
synth_1 (active)	constrs_1	synth_design Complete!					
impl_1	constrs_1	route_design Complete!	0.805	0.000	0.538	0.000	0.000

图 15：同步 RAM 时序结果

图中的 WNS 表示表示最差负时序余量，也就是说该值负得越大，表示时序违约情况越严重
可以看出此次综合中异步 RAM 的 WNS 为-6.536，表示在本次实验中设置的 50MHz 频率下此异步 RAM 达不到时序要求，而同步 RAM 的 WNS 为正，没有违约，能达到时序要求

3、总结

同步 RAM 和异步 RAM 的写行为相同，都是在时钟的上升沿写入，同步的 RAM 同时也在时钟上升沿完成读操作，异步 RAM 则根据 ram_addr 的不同立即变换输出不同的数据。在生成大块 RAM 方面，同步 RAM 更具优势，首先同步 RAM 的占用资源更少，其次在访问频率高的情况下，异步 RAM 的时序违约情况严重

（四）子任务三

1、错误 1：Z 输出

（1）错误现象

从开始时刻到仿真结束，num_csn 信号的输出始终为 Z

（2）分析定位过程

首先分析 num_csn 信号的来源，num_csn 信号作为顶层模块 show_sw 的输出信号，在 show_sw 模块的内部实现中，实例化了 show_num 模块，以 show_num 模块的输出 num_csn 作为顶层模块的输出。

分析波形图：

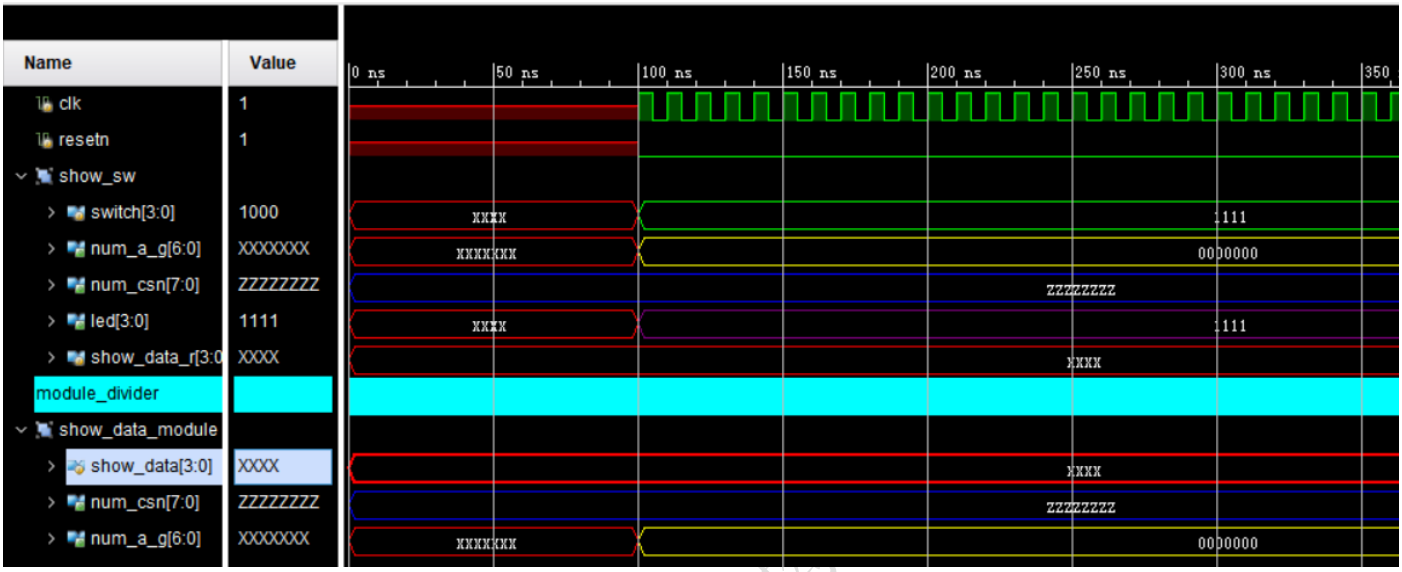


图 16：错误 1

可以看出在两个模块中，num_csn 均为 Z

产生 Z 的原因可能有两个：(1) wire 信号从未赋值；(2)模块未连接相应的信号

对第一种情况，考察代码中 show_data 模块的 num_csn 语句：

```
//digital number display
assign num_csn = 8'b0111_1111;
```

可以看出 num_csn 语句是被赋值了的，因此很大可能是第二种情况。

观察 show_sw 中对 show_data 模块的实例化：

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

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

可以看出 num_csn 的信号连接有问题，Vivado 已给出了可能的错误提示。

（3）错误原因

show_sw 模块调用 show_num 模块时 num_csn 信号未连接

（4）修正效果

将括号内的 num_scn 修改为 num_csn 即可

修改后仿真波形图如下所示：

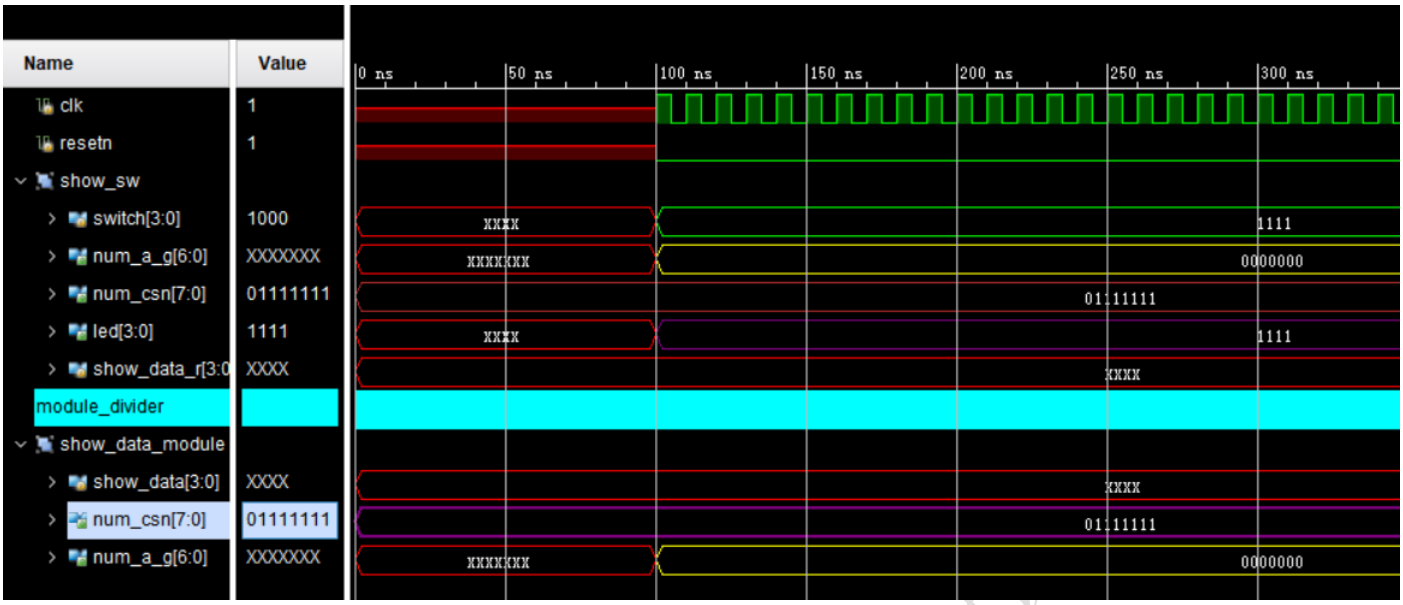


图 17：修改 1

修改了之后 num_csn 信号就变为了正确的输出,表明修改有效。

（5）归纳总结（可选）

以后在调用模块时注意模块的接口与自己定义的信号相对应。

2、错误 2：X 输出

（1）错误现象

show_data 信号的输出始终为 X

（2）分析定位过程

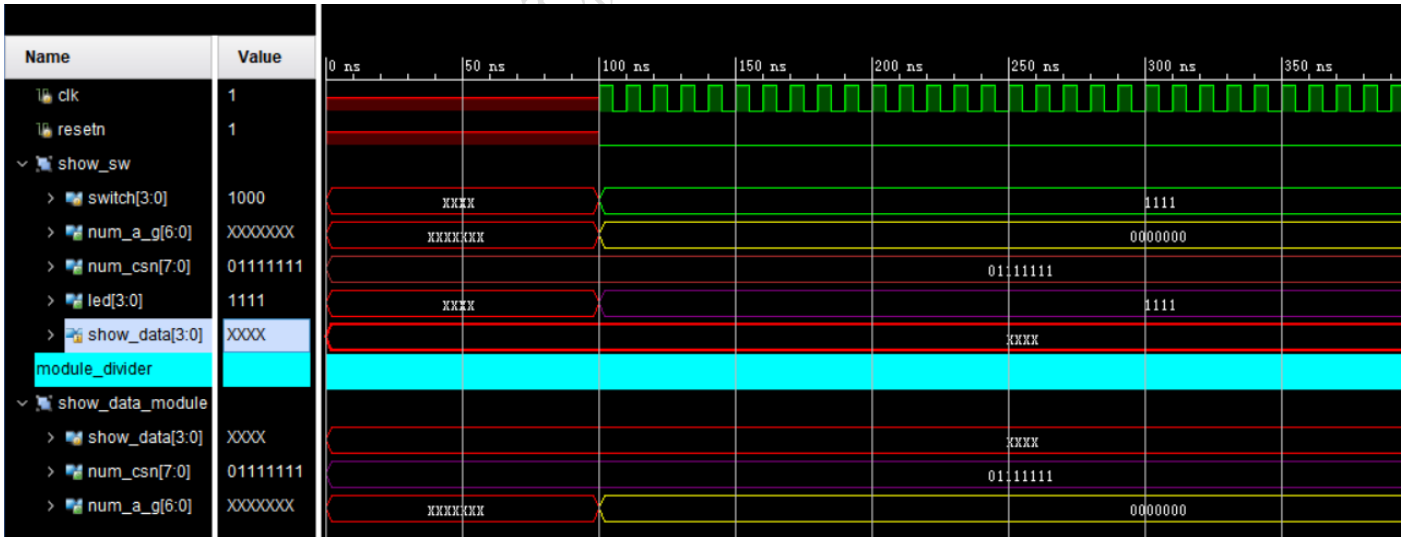


图 18：错误 2

show_data 信号在 show_sw 模块中产生，并作为 show_num 模块的一个输入信号，因此在 show_num 模块中 show_data 信号不大可能被改变，show_num 模块也不大可能是引起该问题的原因。

在 show_sw 模块中定义了寄存器变量 show_data, 其为 X 的原因可能是：

(1) 从未被赋值

(2) show_data 信号本身由多个其他的信号驱动

观察所有 show_sw 模块中与 show_data 信号相关的语句，我们发现 show_data 没有被赋值：

```
25    always @(posedge clk)
26    begin
27    //    show_data    <= ~switch;
28    end
```

这就是导致 show_data 一直表现为 X 的原因

（3）错误原因

show_data 信号从未被赋值

（4）修正效果

把注释的//去掉，仿真波形图如下所示：

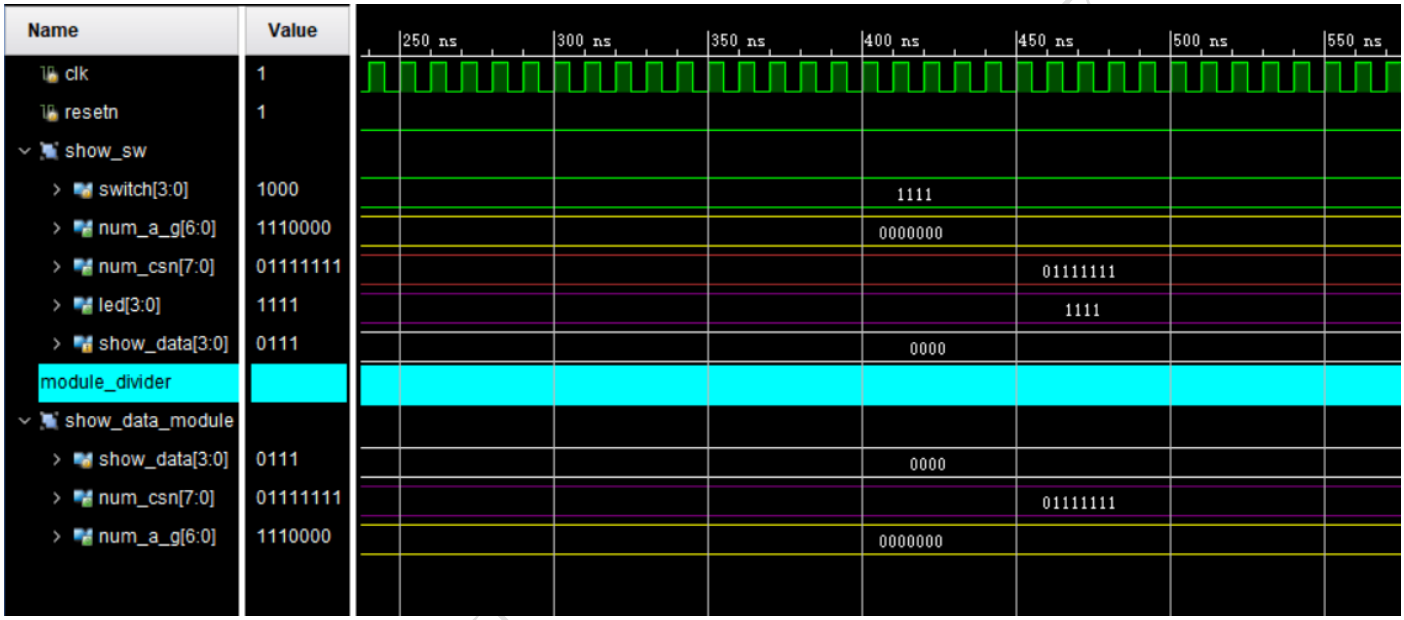


图 19：修改 2

可以看到 show_data 信号不再是 X, 而是输入信号 switch 按位取反之后的结果

（5）归纳总结（可选）

reg 类型的信号要在 always 块里赋值，而且注意一个 always 块通常只给一个信号赋值，方便 debug

3．错误 3：波形停止

（1）错误现象

在完成错误 2 的修改之后进行仿真时，仿真停在了 710ns 的位置，如下图所示：

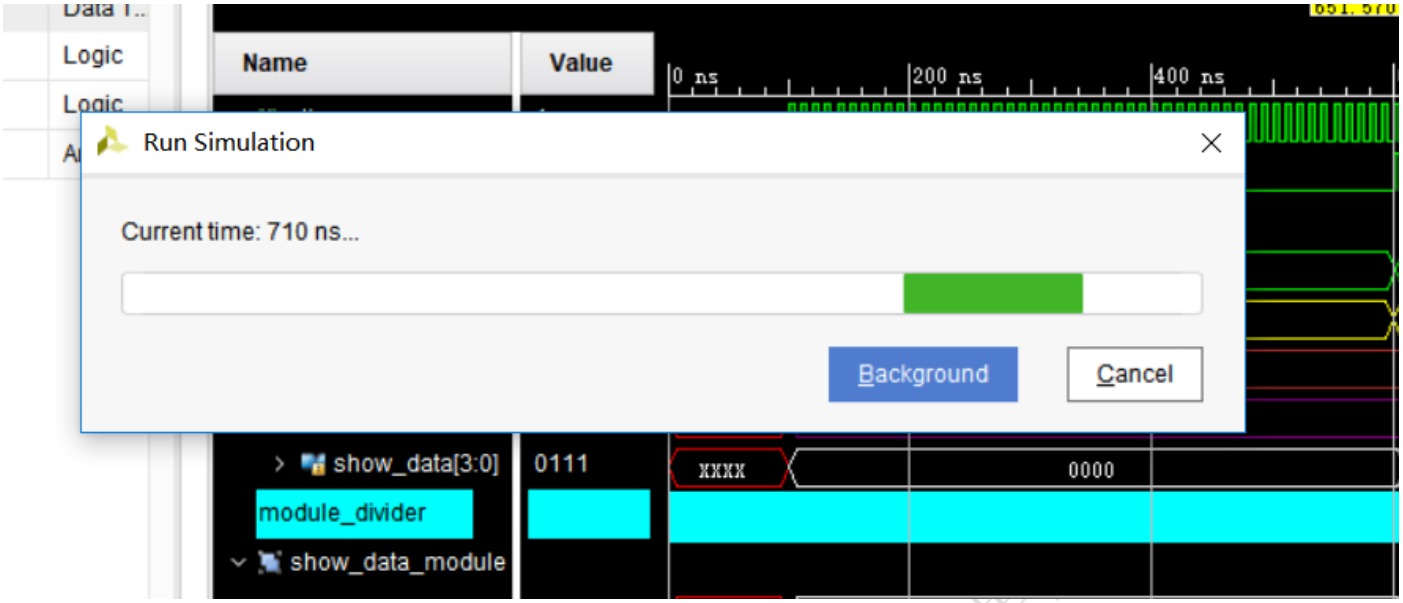
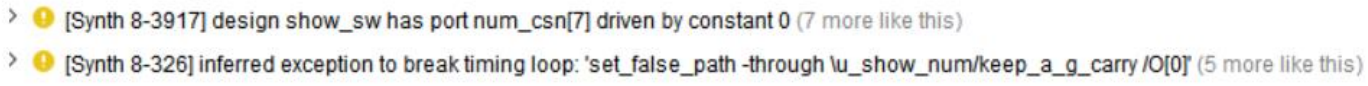


图 20：错误 3

（2）分析定位过程

波形停止往往是由于 RTL 中存在组合环路

先停止仿真，对设计进行综合：



该 Critical Warning 说明设计中可能存在组合环

在关闭了仿真的页面中，发现了下图：

```
90    assign     keep_a_g = num_a_g + nxt_a_g;
91
92    assign nxt_a_g = show_data==4'd0 ? 7'b1111110 :   //0
93                     show_data==4'd1 ? 7'b0110000 :   //1
94                     show_data==4'd2 ? 7'b1101101 :   //2
95                     show_data==4'd3 ? 7'b1111001 :   //3
96                     show_data==4'd4 ? 7'b0110011 :   //4
97                     show_data==4'd5 ? 7'b1011011 :   //5
98                     show_data==4'd7 ? 7'b1110000 :   //7
99                     show_data==4'd8 ? 7'b1111111 :   //8
100                    show_data==4'd9 ? 7'b1111011 :   //9
101                                      keep_a_g   ;
```

说明仿真到了 90 这一行就一直没有进行下去，可能是 keep_a_g 信号出了问题，keep_a_g 信号由 num_a_g 信号和 nxt_a_g 信号相加残生，分别最总这两个信号可以发现 num_a_g 信号是一个 reg 信号，由时钟驱动更新，而 nxt_a_g 信号是 wire 信号，由组合逻辑产生。观察 nxt_a_g 信号的来源，发现它是由 show_data 的值决定的一个多路选择器，且在 show_data 超过 10 的情况时会被赋值为 keep_a_g, 这就是一个组合环。

（3）错误原因

nxt_a_g 信号处赋值产生了组合环

（4）修正效果

考虑在 keep_a_g 信号的赋值中避免用到 keep_a_g 信号本身。根据 keep_a_g 信号的意义，其作用是用来记录上一次的 num_a_g 信号，并在 show_data 大于等于 10 的时候保持上一次的信号不变，因此只需将 keep_a_g 信号赋值为 num_a_g 信号即可。修改后的仿真波形如下所示：

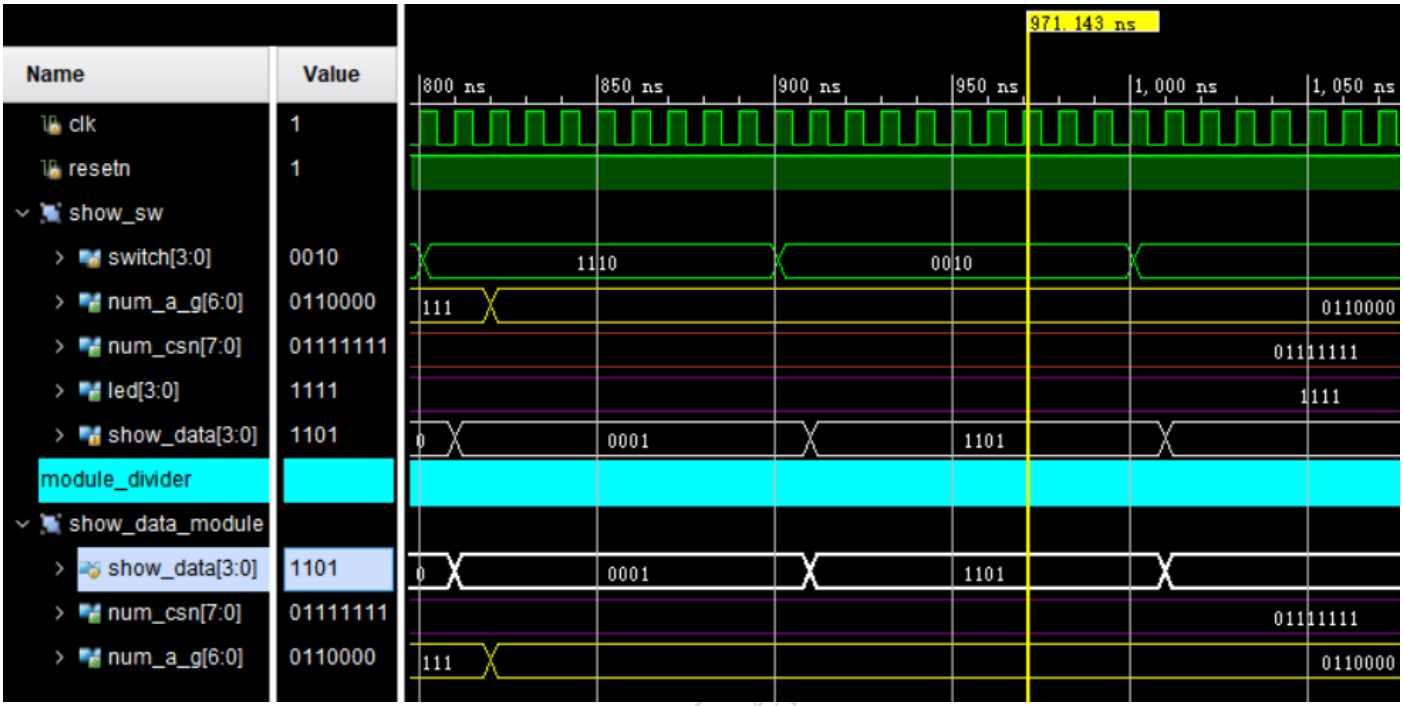


图 21：修改 3

可以看出在修改了之后，仿真能顺利进行，而且在 show_data 为 1101 即 13 时，num_a_g 仍保持为 show_data 为 1 时的 0110000

（5）归纳总结（可选）

在设计信号间的关系时需要优先考虑好信号之间的依赖关系，避免出现组合环

4、错误 4：LED 信号输出异常

（1）错误现象

从开始时刻到仿真结束，led 信号的输出时钟为 1111

（2）分析定位过程

分析代码可知，led 的输出是由信号 prev_data 决定的，而 prev_data 的值又是在 always 块中根据 show_data 和 show_data_r 的值来确定的。led 涉及的信号放在同一组中，如下图所示：

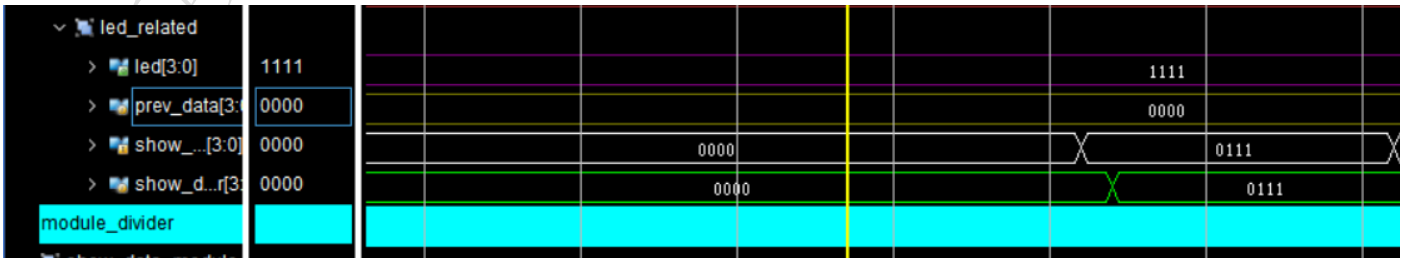


图 22：错误 4

由于 prev_data 一直为 0000，因此推测是 always 块里的赋值出了问题。

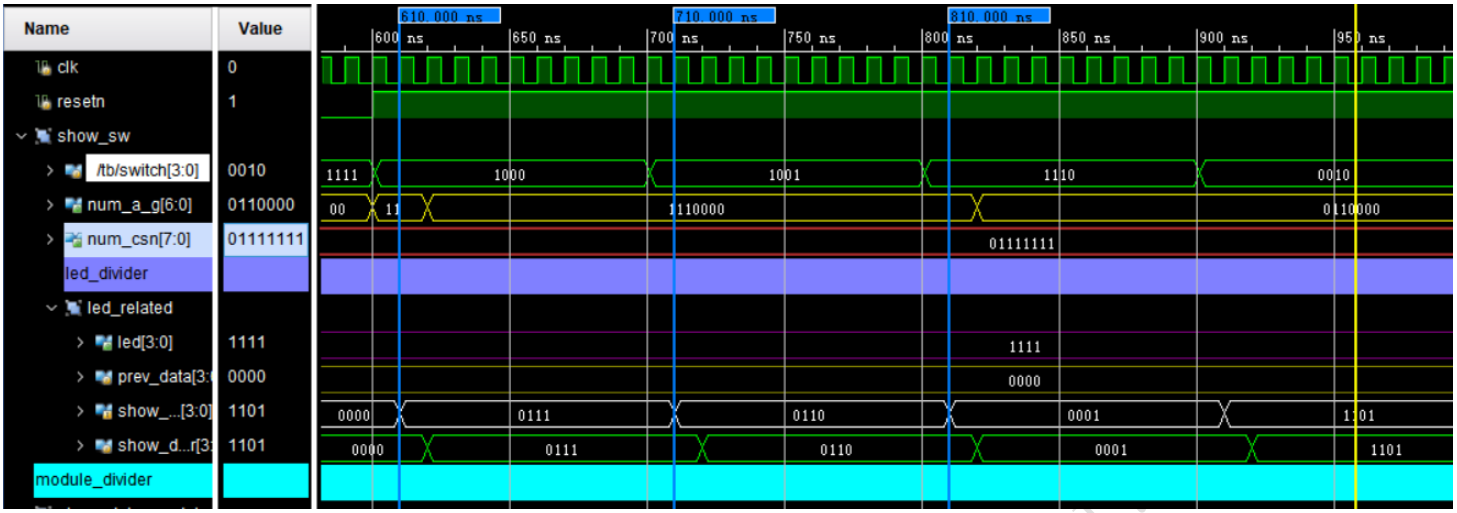


图 23：错误 4

可以看到在图中两个 divider 之间关于 led 的几个信号中，在时钟上升沿处，show_data 和 show_data_r 并不相同，prev_data 的写法，prev_data 应该被赋值为 show_data_r 才对，但并没有这么做，因此去看 show_data_r 的相关模块发现，show_data_r 的赋值使用的是阻塞赋值：

```
30  always @(posedge clk)
31  begin
32      show_data_r = show_data;
33  end
```

阻塞赋值意味着在时钟上升沿需要等该赋值完成后才能在其他模块进行赋值操作，因此每次在 show_data_r 赋值完后才会在 prev_data 模块中对 prev_data 进行赋值，而此时 show_data_r 和 show_data 已经相等了，prev_data 不会再改变，一直保持为 0000，led 也一直保持为 1111

（3）错误原因

阻塞赋值使用错误

（4）修正效果

将 show_data_r 的阻塞赋值修改为非阻塞赋值，仿真波形如下图所示：

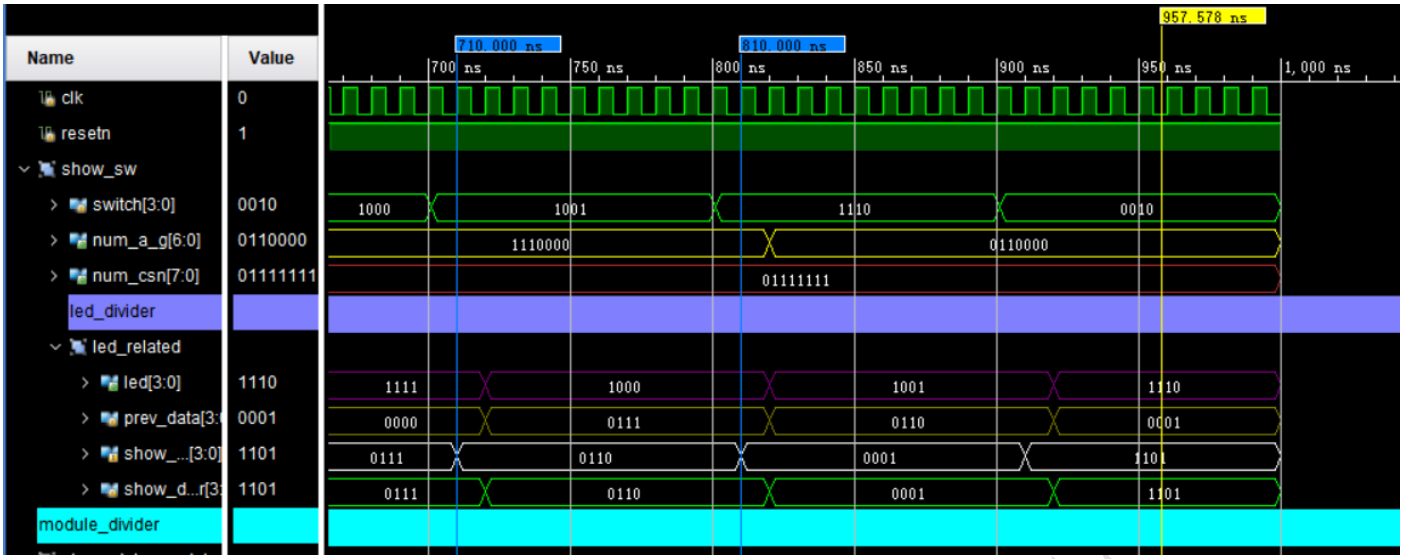


图 24：修改 4

修改完成之后 led 信号能顺利记录上一次的 switch 信号

（5）归纳总结（可选）

在 always 块里面赋值时都统一采用非阻塞赋值可以避免出现这种错误

5. 错误 5：缺少对应的输出

（1）错误现象

在观察波形时，发现 show_data 为 0110 时，num_a_g 缺少对应的输出

（2）分析定位过程

num_a_g 在时钟上升沿处由 nxt_a_g 更新，而 nxt_a_g 与 show_data 有关。

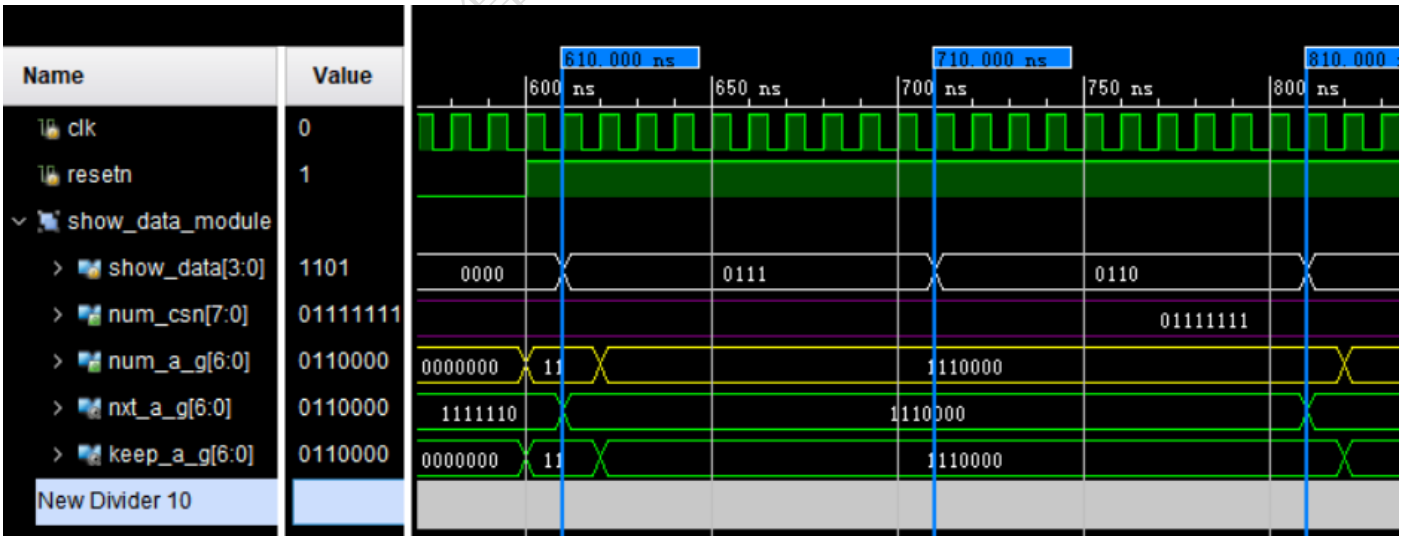


图 25：错误 5

观察波形发现 nxt_a_g 也没有变为对应值，可能是给 nxt_a_g 赋值时出错。

```
92  assign nxt_a_g = show_data==4'd0 ? 7'b1111110 :   //0
93                   show_data==4'd1 ? 7'b0110000 :   //1
94                   show_data==4'd2 ? 7'b1101101 :   //2
95                   show_data==4'd3 ? 7'b1111001 :   //3
96                   show_data==4'd4 ? 7'b0110011 :   //4
97                   show_data==4'd5 ? 7'b1011011 :   //5
98                   show_data==4'd7 ? 7'b1110000 :   //7
99                   show_data==4'd8 ? 7'b1111111 :   //8
100                  show_data==4'd9 ? 7'b1111011 :   //9
101                                    keep_a_g   ;
```

观察发现，nxt_a_g 信号在 show_data 为 6 时没有明显的赋值，因此将被继续赋值为 keep_a_g

（3）错误原因

情况考虑不完善，导致 nxt_a_g 信号缺少一个正确的赋值

（4）修正效果

增加 show_data=4’d6 的情况，仿真波形如下：

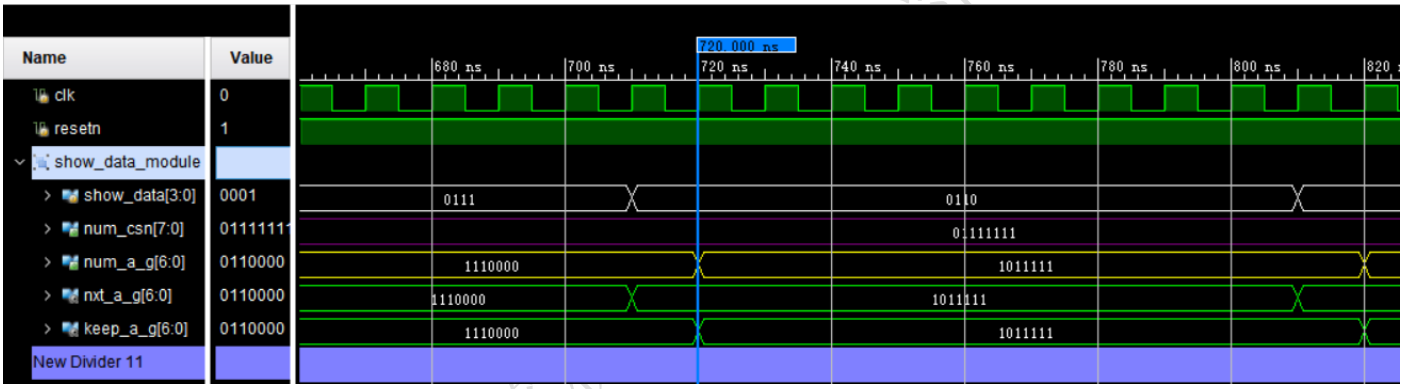


图 26：修改 5

在第一个 Marker 处，num_a_g 被赋值为了 1011111，功能完善

（5）归纳总结（可选）

情况考虑不完善导致，属于逻辑层面的问题，以后在写代码之前需要考虑周密所有的情况

四、实验总结（可选）

通过这次实验，我对时序逻辑有了更深的理解，对 Vivado 中生成 RAM 的手段有了初步的了解。同时也对 verilog 代码规范和 debug 手段有了初步的掌握。不过子任务 2 中要求对异步 RAM 和同步 RAM 的资源和时序图的解读还是不太会。