计算机组成原理实验报告

班级： 张金老师　姓名：王众　学号：2313211

实验老师：董前琨　实验地点：实验楼306　实验时间：2025.3.30

一、实验目的

（1）对寄存器堆的读写方式有一定的了解

（2）了解并分析同步ram和异步ram各自的特点和区别

（3）对数字逻辑电路进行设计和调试

二、实验内容说明

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，再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”?

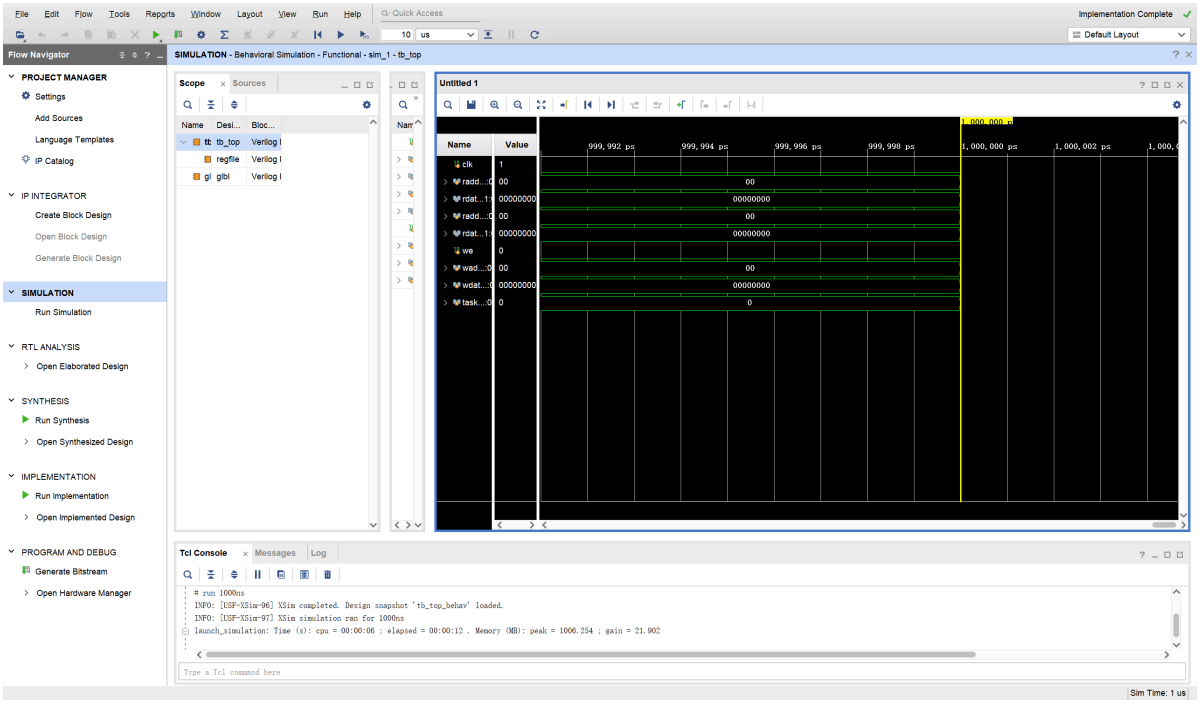
（2）针对任务二同步ram和异步ram实验，可以参考实验指导手册中的存储器实验，注意同步和异步需要分开建工程，然后仿真，在感想收获中分析同步ram和异步ram各自的特点和区别。

（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、具体实验过程

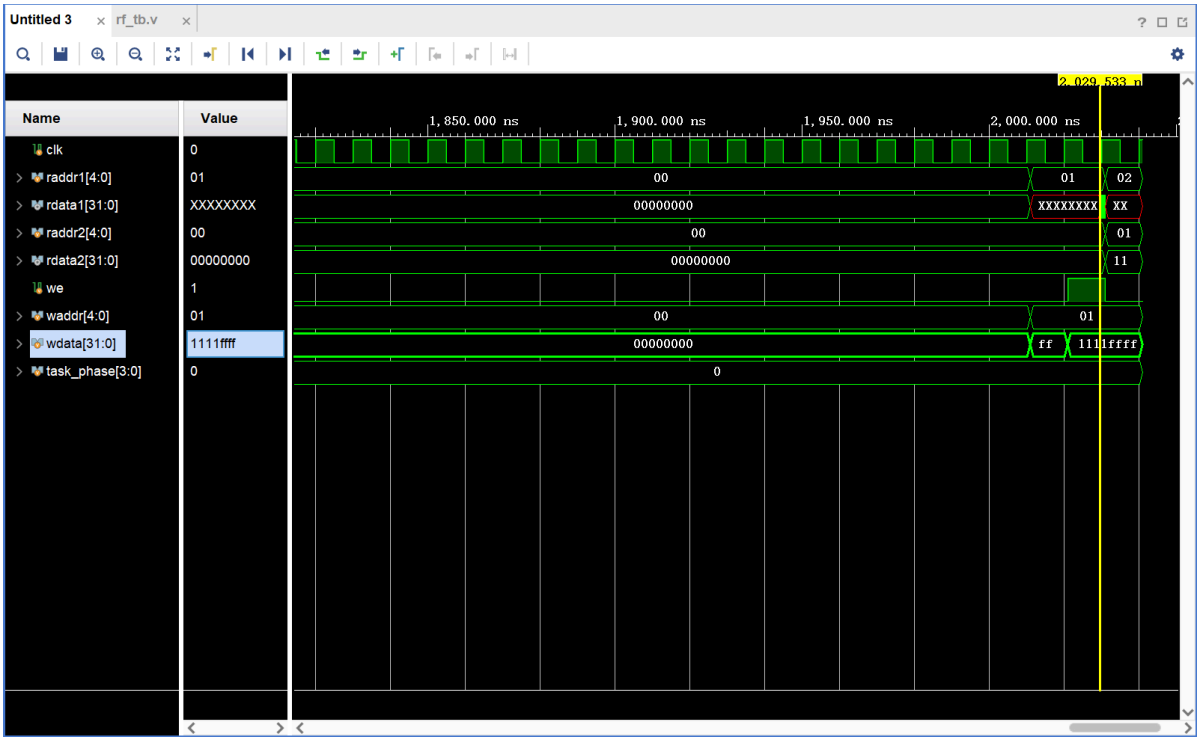
（1）寄存器堆的仿真实验

在根据实验指导手册的操作后我们得到了以下仿真实验图：

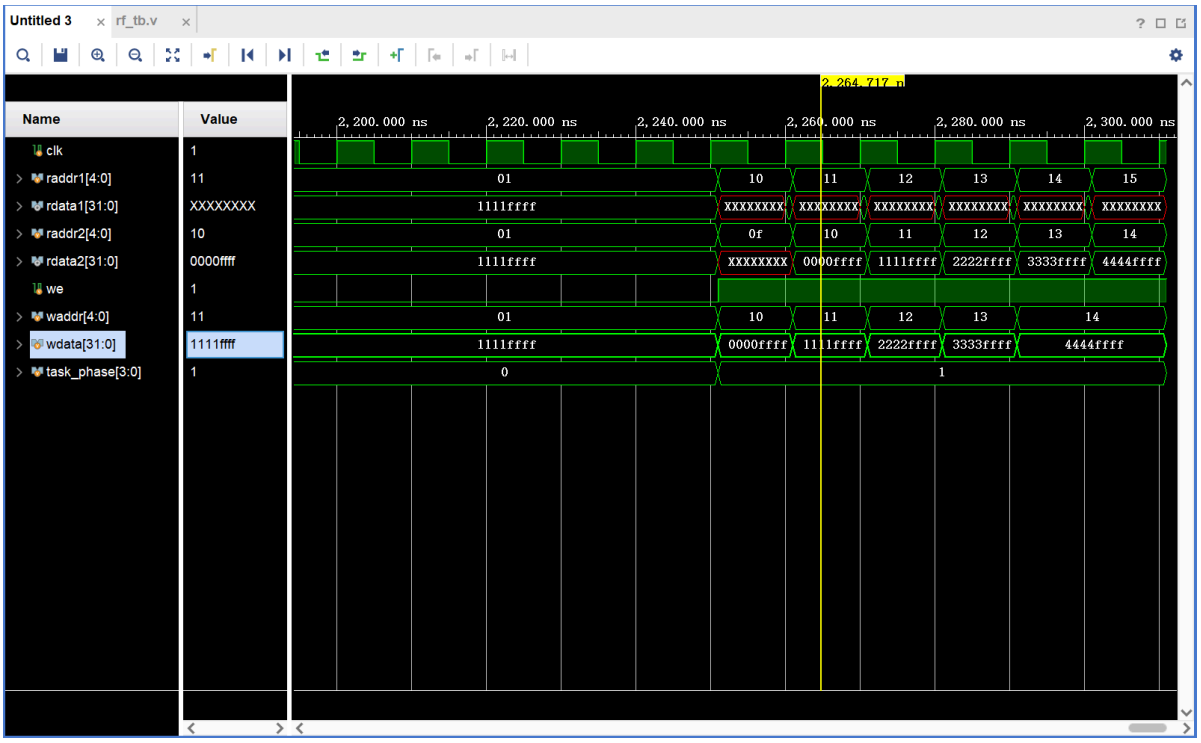


1.分析代码（结合图示分析）：

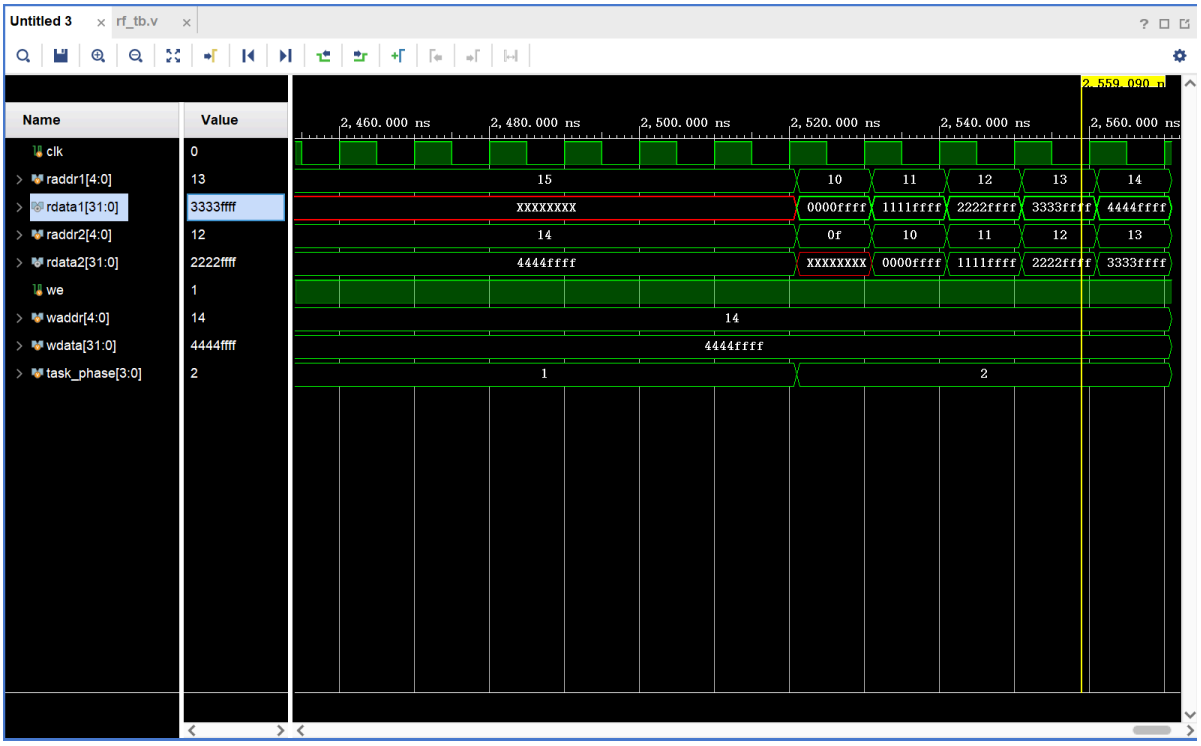
`part1`的目的是测试寄存器的基本读写功能，只在1个地址`5'd1`写入`0xffffffff`，再写入数据`0x1111ffff`,通过`raddr1`和`raddr2`进行数据的读取。我们可以看到wdata的数值变为`ffffffff`和`1111ffff`。



在`part1`中从地址`5'h10`开始，依次写入`0x0000ffff`、`0x1111ffff`、`0x2222ffff`等数据。每次写入后，通过`raddr1`和`raddr2`读取当前地址和前一个地址的数据，验证写入和读取的正确性。我们可以看到寄存器每输入一个数值，便会进行一次读的操作保证输入的正确性。



在`part2`中从地址`5'h10`开始，依次读取数据，同时`raddr2`读取前一个地址的数据验证之前写入的数据是否被正确保存。此时用于写入数据的`wdata`一直保持最后一次输入的状态`4444ffff`。同时用于读数据的`radder1`和`` `radder2 ``都在用于检验数据是否被正确保存。



2.为什么寄存器堆要设计成“两读一写”?

1)支持RISC指令的双操作数读取。在典型的RISC（精简指令集计算机）架构中，大多数算术逻辑运算（如`ADD`，`SUB`，`AND`，`OR`等）采用“三操作数”格式：`OP Rd, Rs1, Rs2`。Rs1 和 Rs2 是源操作数（需要从寄存器堆读取）Rd 是目标操作数（需要写回寄存器堆）。因此，每个时钟周期需要同时读取两个寄存器的值（Rs1 和 Rs2），并可能写入一个寄存器（Rd）。“两读一写”的设计正好满足这一需求。

2）如果寄存器堆只有一个读端口，那么当一条指令需要同时读取两个操作数时，就必须分两个周期完成，这会降低CPU的吞吐量。

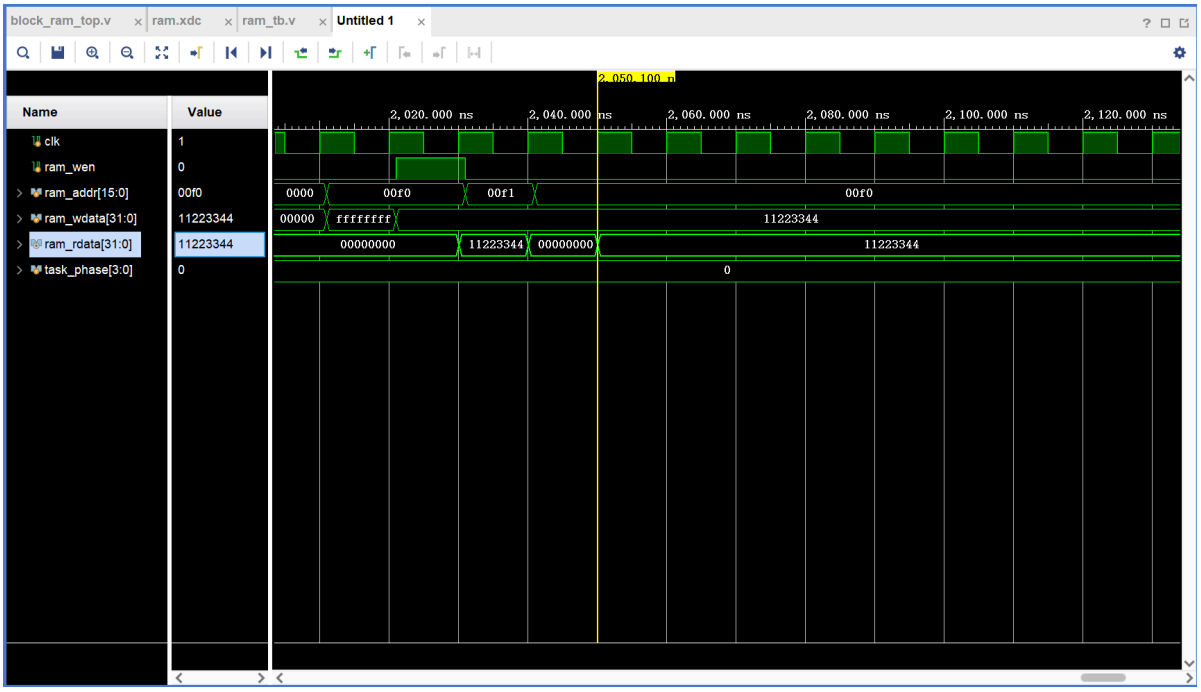
3）与流水线（Pipeline）的配合：现代CPU采用流水线技术，指令的执行分为多个阶段（如取指、译码、执行、访存、写回）。“两读一写”的设计可以很好地匹配流水线的需求，避免数据冲突。

（2）同步RAM和异步RAM仿真、综合与实现

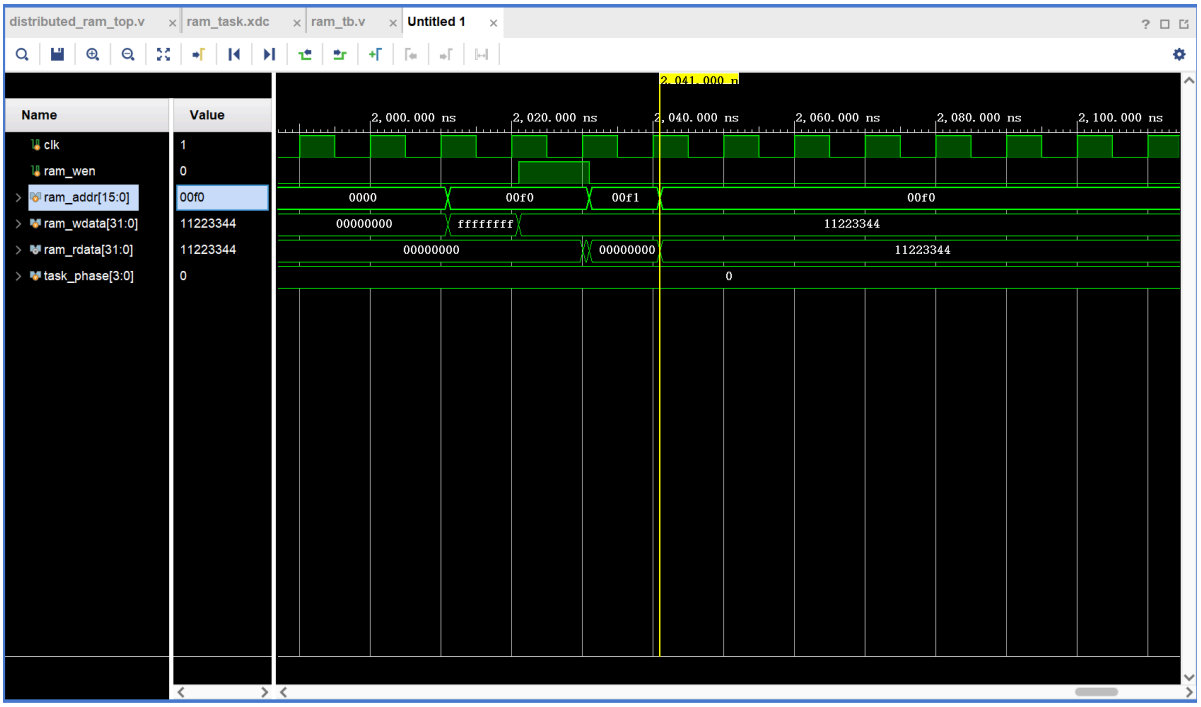
part0:

`part0`的代码的任务的是先将写使能置为0，地址设为`ffffffff`,然后将`ram_wen`置为1，向地址写入数据`11223344`，最后关闭写使能，先读取`16'hf1`和`16'hf0`。仿真截图如下：

同步：



异步：

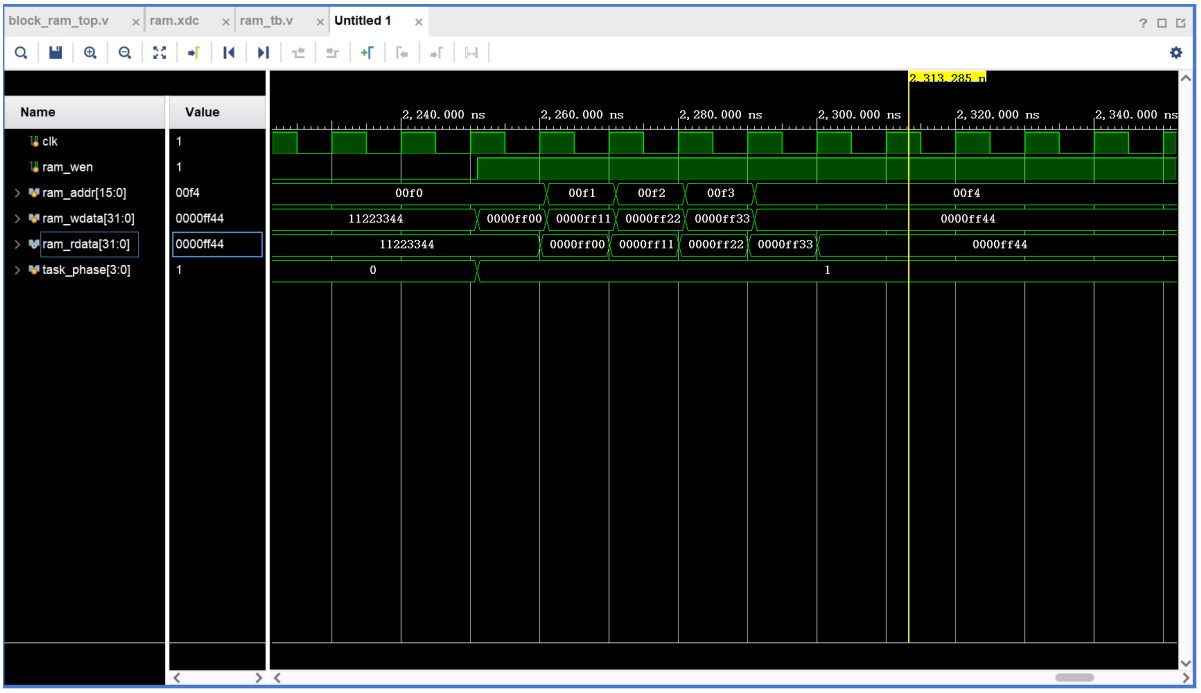


我们可以看见，在同步ram中，读取数据值变量只有在时钟跳转升降时会发生改变，但是在异步中只要读取的地址改变，那么读取的数据值会同步改变。

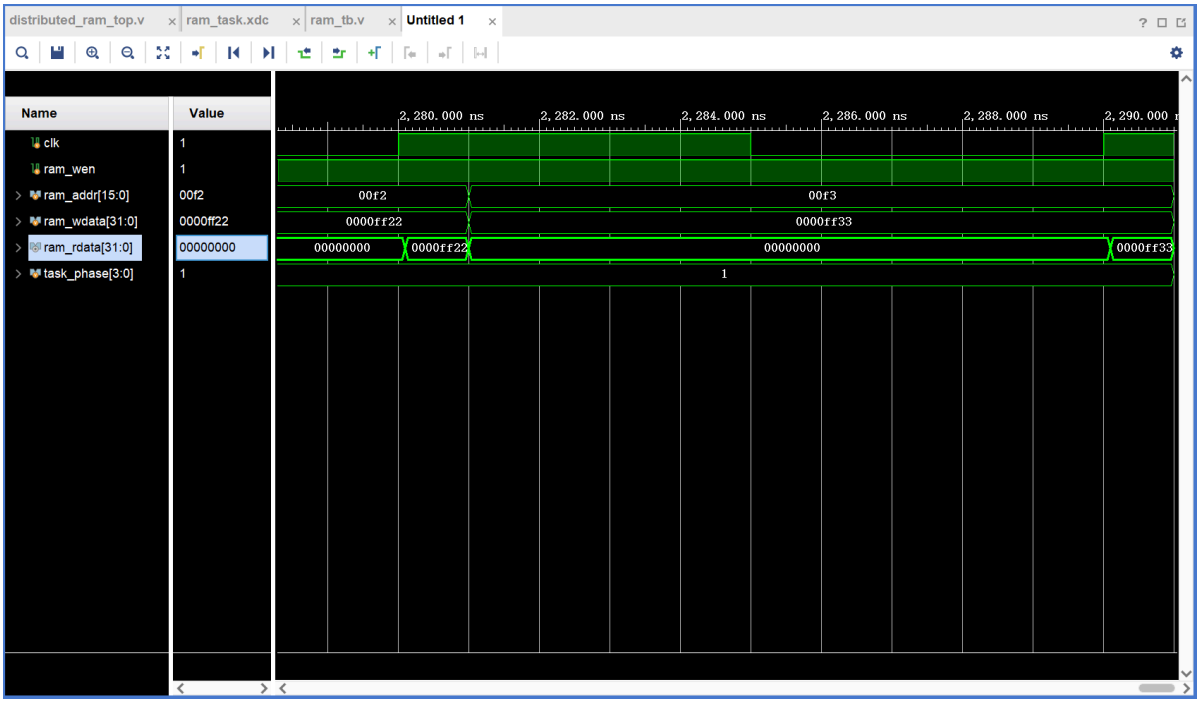
part1:

在`part1`部分的代码中，首先使写使能保持为1，然后向连续地址写入不同的数据。得到的仿真如下：

同步：



异步：

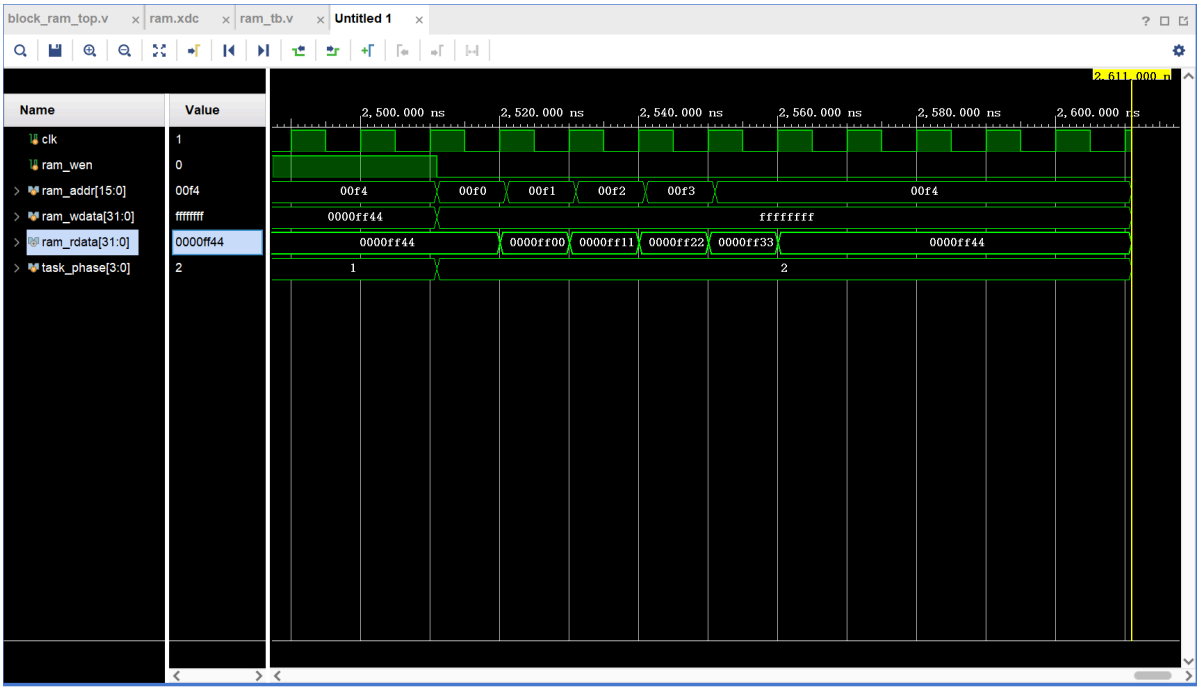


我们可以看到，对于同步电路，还是在时钟信号的跳转出进行数据的更改，并且数据的读写是同步的。但是对于异步电路，并不会受到时钟信号的影响来更改数据，但是异步电路的读写是异步的，所以会读到`00000000`这样的数据。

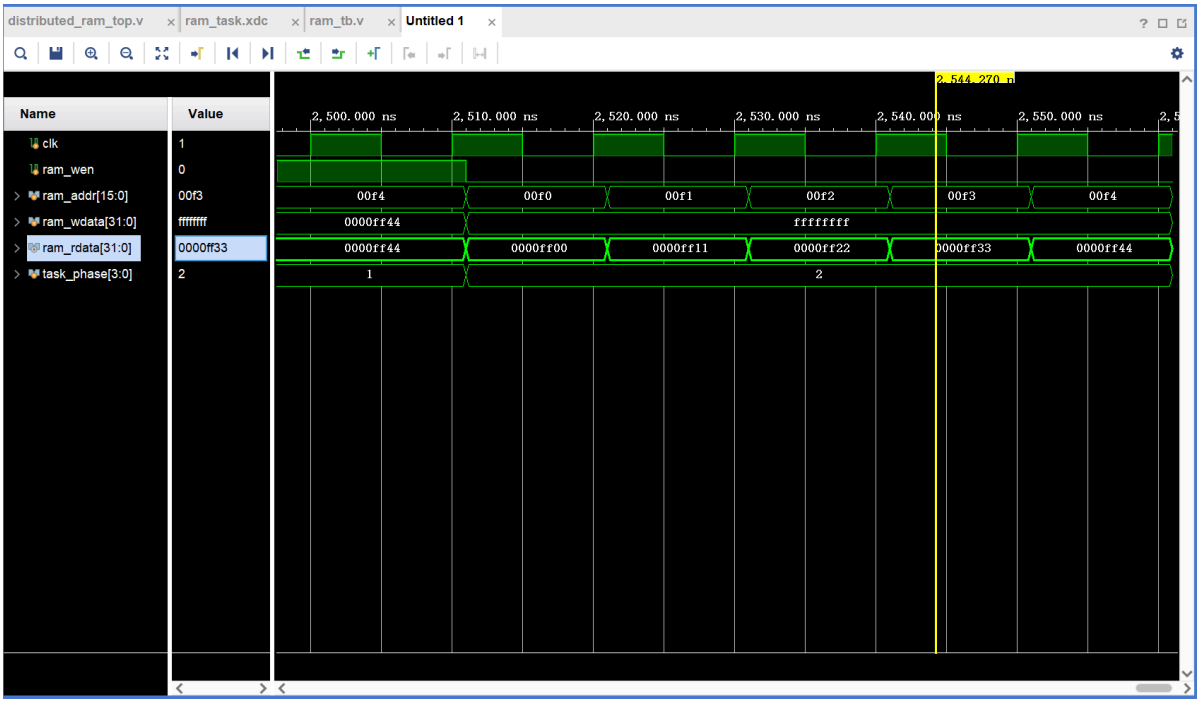
part2:

`part2`代码关闭了写使能，并一次读取之前写入的数据。得到的仿真如下：

同步：



异步：

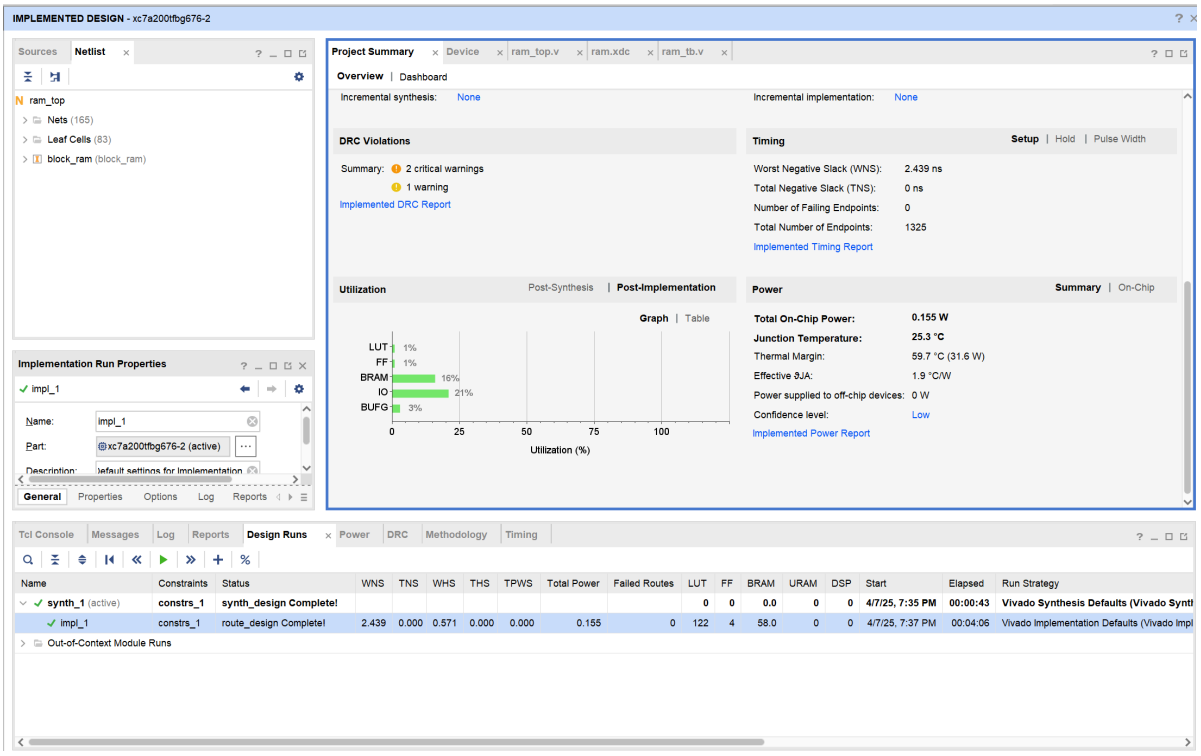


与`part0`部分的情况相似，同步的数据变换需要在时钟信号变化时进行跳转，但是异步的数据是根据地址的变化同步变化的。

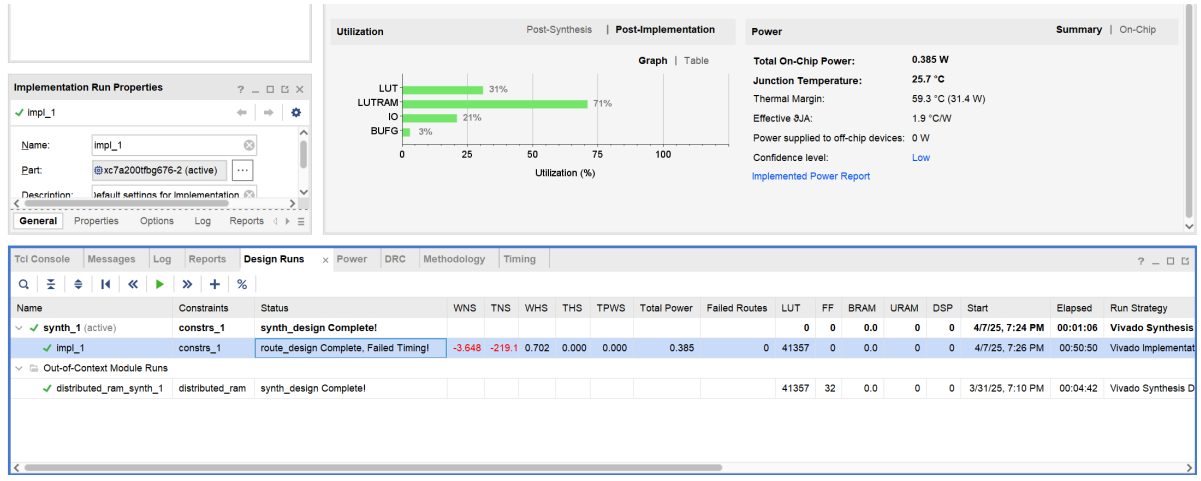
总结来说：当读写行为同时存在时，同步RAM的读写行为是同步的，但是可能会存在一个周期的延迟。但是异步RAM的读写行为是分开的。并且同步RAM的数据变换需要时钟信号进行跳转的瞬间进行，但是异步RAM并不需要遵守时钟信号。所以，同步RAM因为时钟的原因竞争风险比较低，异步RAM的竞争风险比较高。所以同步RAM适用于高速系统如CPU缓存中，异步RAM适用于低速缓存之类的系统中。

在结束分析之后，我们对同步和异步电路进行综合与实现的操作，得到以下两个图：

同步：



异步：



我们可以看到同步的WNS显示为正数，表示及时序极好，但是异步的WNS为负数，但是没有超过300ps表示满足较好，可以上板执行。再看资源利用率：异步的资源利用率各项都基本比同步的要更好。

分析资源利用率差异的原因：异步设计在资源利用率方面通常比同步设计更好，主要原因在于其不依赖全局时钟信号，减少了时钟相关的资源消耗，并且在读写操作方面具有更高的并发性和灵活性。

异步设计的资源利用率

1. **时钟域减少**：异步设计不依赖全局时钟信号，因此不需要复杂的时钟树和时钟缓冲器，这减少了时钟相关的资源消耗。
2. **控制信号简化**：异步设计通常使用简单的握手协议进行通信，不需要复杂的时钟使能和锁存电路。
3. **低功耗**：由于异步电路仅在需要时才进行切换，而不是每个时钟周期都切换，因而可以减少不必要的切换活动，从而降低功耗。
4. **无全局同步**：异步电路不需要全局同步，因此不需要全局时钟树，这减少了全局时钟树的资源占用。

同步设计的资源利用率

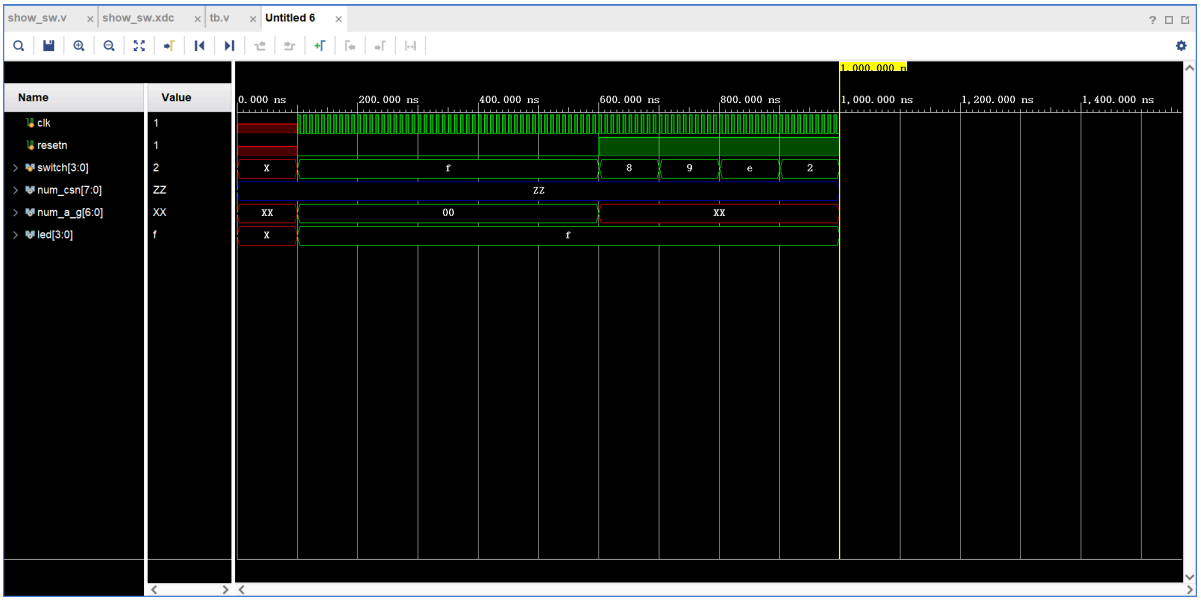
1. **时钟树资源**：同步设计依赖全局时钟信号，需要复杂的时钟树和时钟缓冲器，这增加了资源消耗。
2. **时钟使能电路**：同步设计往往需要时钟使能电路来控制时钟信号的传递，这增加了额外的资源使用。
3. **锁存器和寄存器**：同步设计需要在时钟边沿进行数据锁存，因此需要更多的锁存器和寄存器来保持数据，这增加了资源使用。
4. **一致性控制**：为了确保数据传输的一致性，同步设计需要额外的控制电路来管理时钟域和数据域的一致性。

（3）数字逻辑电路的设计与调试

我们首先需要补齐输出：在`tb.v`文件中将有关于输出的定义和有关于输出的绑定连接。

```
wire     [7 :0] num_csn;
wire     [6 :0] num_a_g;
wire     [3 :0] led;
//..................existed code
.num_csn(num_csn),
.num_a_g(num_a_g),
.led    (led)
```

可以得到以下图示：



我们按顺序对问题进行解决：

1.对于输出变量出现 z 波形问题。

1. 信号为“Z”

“Z”表示高阻，比如电路断路就会显示为高阻，这种错误往往是以下两个原因导致的：

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。

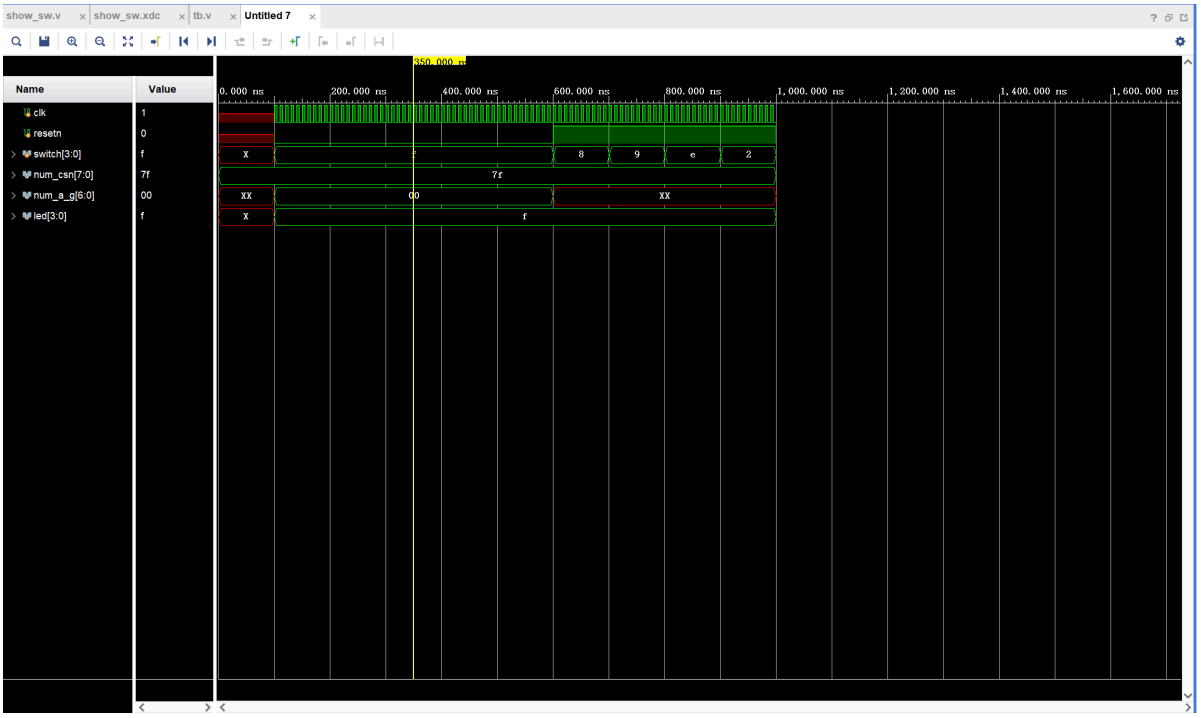
我们查阅书本可知主要原因是调用的模块信号为链接导致悬空所致。所以我们检查代码可知：

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

```
    .resetn     (resetn   ),
    .show_data  (show_data),
    .num_csn    (num_scn  ),//命名错误
    .num_a_g    (num_a_g  )
);
-------------------------更改
//show number: new value
show_num u_show_num(
    .clk        (clk      ),
    .resetn     (resetn   ),
    .show_data  (show_data),
    .num_csn    (num_csn  ),//命名错误
    .num_a_g    (num_a_g  )
);
```

在`.num_csn`的调用中的`num_csn`被命名为了`num_scn`导致在show信号时找不到待测信号。

解决完问题之后我们可以得到如下波形图：可以看出Z波形错误已经被解决，但是X波形错误仍然存在且是从600ns到1000ns。所以我们进一步去解决



我们查看指导书发现是reg型的变量未被赋值：

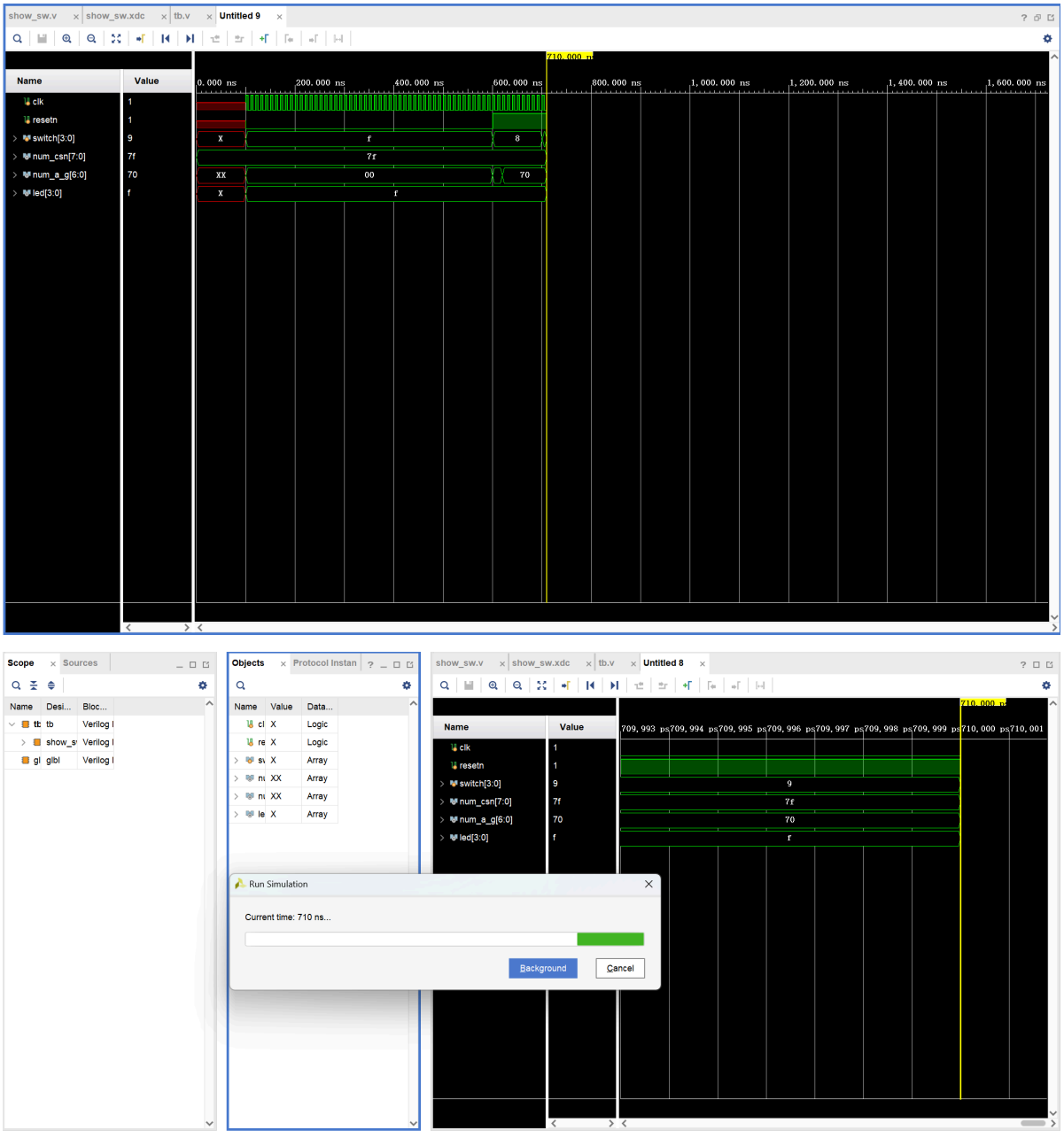
2. 信号为“X”

“X”表示不定值，这种错误往往是以下两个原因之一导致的：

1）RTL 里声明为 reg 型的变量从未被赋值。

观察代码我们可以发现，代码中的`show_data_r`只被定义而没有被赋值，所以我们将//去掉对其进行赋值。

```
begin
    //show_data_r <= show_data;
end
--------------------------------------更改
begin
    show_data_r <= show_data;//采用非阻塞式赋值
end
```



我们修复之后可以得到以下的图示，我们可以看出X的波形已经被解决了，但是仿真卡在了710ns的那一步。所以我们继续进行排查

发现属于是波形停止的错误：

3. 波形停止

波形停止是指仿真在某一时刻停止，再也无法前进分毫，而仿真却显示仍然在运行，这种错误往往是 RTL 里存在组合环路导致的。波形停止示例如图 3-11 所示。

所以我们对变量之间进行检查查看有没有重复相互调用的情况，在以下代码处和`keep_a_g`和`nxt_a_g`出现了相互调用的情况。

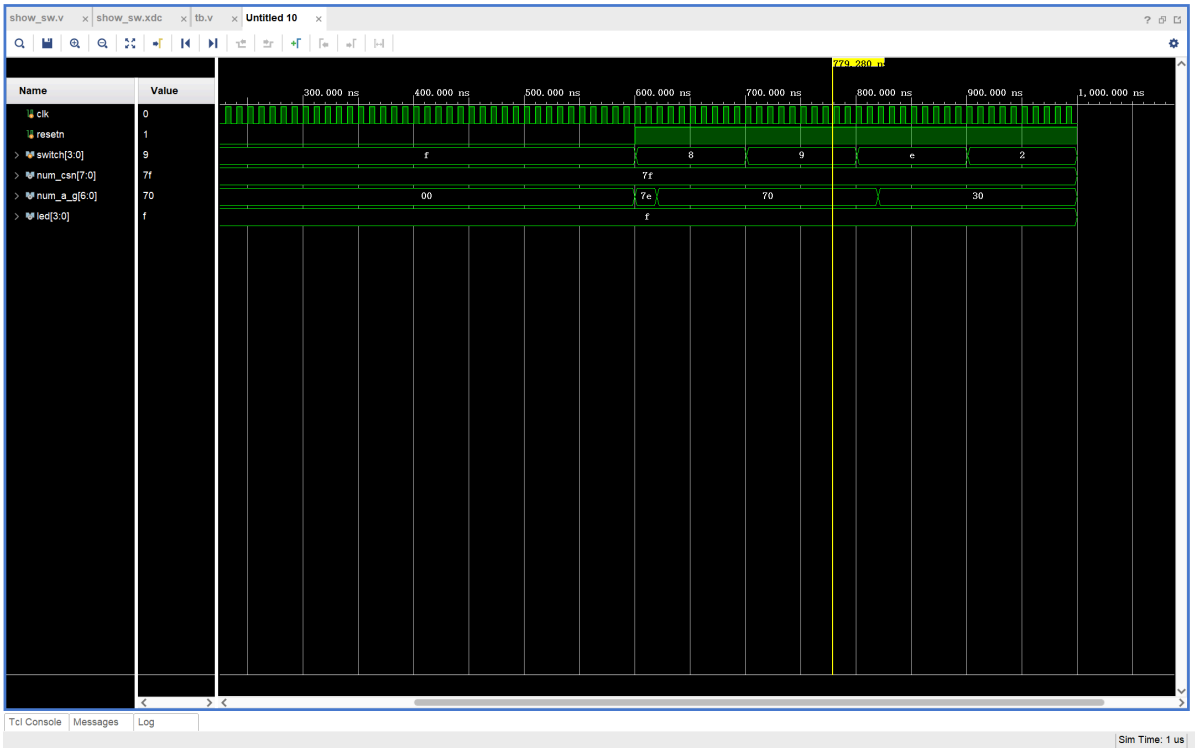
所以我们对`assign     keep_a_g = num_a_g + nxt_a_g;`代码进行删减。

```
always @(posedge clk)
    begin
        if ( !resetn )
            begin
                num_a_g <= 7'b0000000;
            end
        else
            begin
                num_a_g <= nxt_a_g;//与下面的语句相互循环
            end
    end

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'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   ;
```

得到改进后的代码：

```
wire [6:0] keep_a_g;
assign     keep_a_g = num_a_g ;//存在组合环路
```

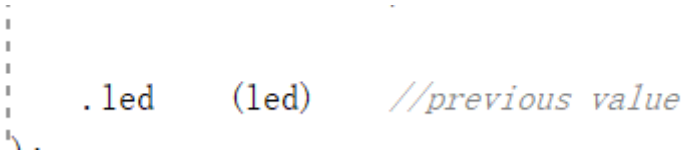
得到的波形图：图中的波形图的产生不再阻塞。



但是测试`led`灯不进行显示`previos value`，一直都是`f`所以我们继续进行`debug`。查阅指导手册可以知道：

> 除非特意设计，一般认为越沿采样是一个设计错误。针对越沿采样，我们有以下几点建议：
> 1）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。
> 2）一旦发现越沿采样的情况，追溯被采样信号，直到追溯到某一个阻塞赋值的信号，随后进行修正。

同时在这个代码中我们发现了在`always`模块中的阻塞式赋值。

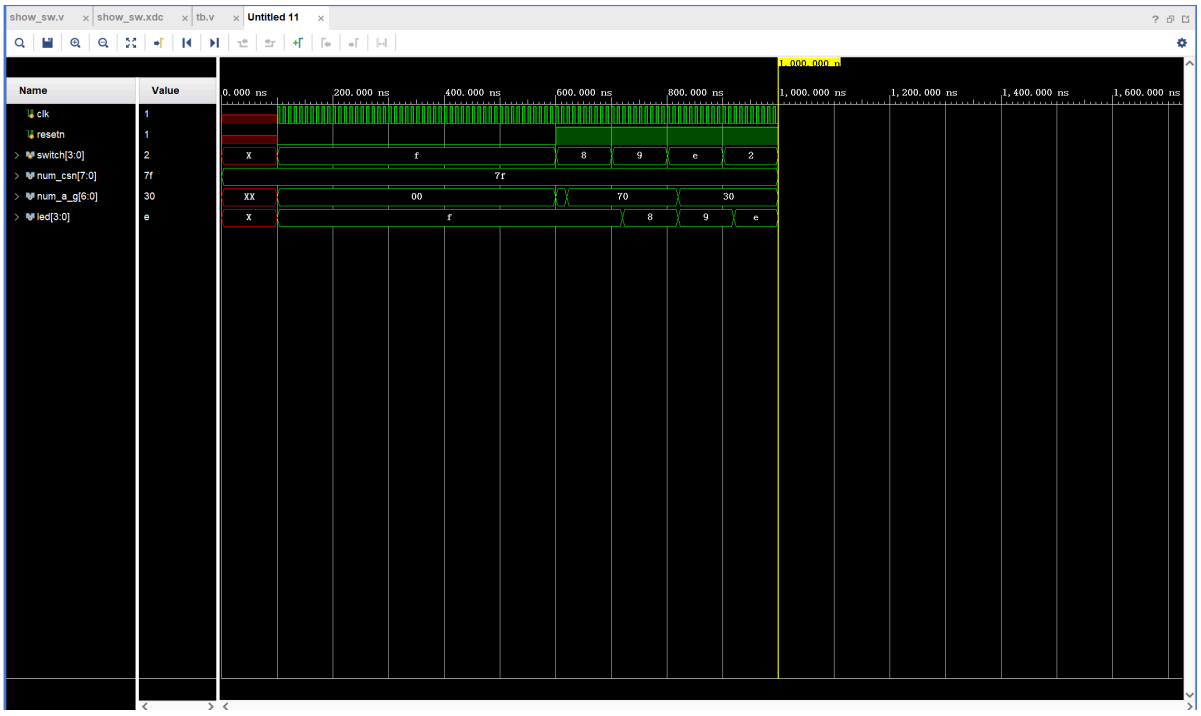


```
    .led     (led)     //previous value
);
```

```
always @(posedge clk)
begin
    show_data_r = show_data;
end
```

所以我们将其改为非阻塞式赋值：

```
always @(posedge clk)
begin
    show_data_r <= show_data;//采用非阻塞式赋值
end
```



可以看出`led`开始显示之前的赋值得到了解决。

最后是一个存在的功能bug，我们观察代码看出，代码中少了数字6的表示，同时由于数字显示屏的排列如下

```
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   ;
```

```
  a
f   b
  g
e   c
  d
```

所以我们可以得出6的数字标识为`1011111`

```
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   ;
```

上箱验证：

这个视频演示了从1到16的指示灯和4位二进制数的演示情况。

计算机组成原理lab3bug修复实验修复结果上箱演示视频*哔哩哔哩*bilibili