

LAB 7

小组成员 (按姓氏倒序) :张梓堃、谭茜、林孟颖

箱子号:79

LAB 7

一.实验任务概览

二.实验设计

(一) 总体设计思路

1. 从SRAM→类SRAM
2. 设计类SRAM-AXI转接桥

(二) 流水级的重要变化

1. IF 阶段
 - 1.1 下一条取指PC的确定
 - 1.2 指令buffer的引入
2. EXE级变化: ready_go的调整

(三) 重要模块1设计: 读请求通道状态机

- 1.工作原理
- 2.状态机状态定义
- 3.状态机状态转移重要信号

(四) 重要模块2设计: 读响应通道状态机

- 1.工作原理
- 2.状态机状态定义
- 3.状态机状态转移重要信号

(五) 重要模块3设计: 写数据&写地址通道状态机

- 1.工作原理
- 2.状态机状态定义
- 3.状态机状态转移重要信号

(六) 重要模块4设计: 写响应通道状态机

- 1.工作原理
- 2.状态机状态定义
- 3.状态机状态转移重要信号

三.实验过程

(一) 实验流水

(二) 实践任务14: 添加类 SRAM总线支持

- 错误1: nextpc确定信号未存于寄存器
- 错误2: 丢弃一条指令时未置空指令buffer
- 错误3: fs_allowin理解不到位
- 错误4: 信号未定义
- 错误5: br_stall判断未考虑data_ok

- 错误6：错把访存最低二位地址强制置零
- 错误7：中断返回信号未与valid信号挂钩
- 错误8：MEM阶段的例外信号未与上valid
- 错误9：取指连续cancel时处理不当
- 错误10：上板使用部分RANDOM_SEED无法通过测试

(三) 实践任务15：添加 AXI 总线支持

- 错误1：状态机状态设计不合理
- 错误2：data_ok和addr_ok实现有误
- 错误3：给出valid信号后变更地址
- 错误4：弄混写响应通道的主从端
- 错误5：读请求通道未排除写请求
- 错误6：未考虑写地址和写数据同时握手成功的情况
- 错误7：指令RAM的数据实现有误
- 错误8：读数据和地址通道的计数器实现有误
- 错误9：未用寄存器保存rid的值
- 错误10：wdata漏接
- 错误11：AXI输出端不来自寄存器Q端的后遗症
- 错误12：未处理好IF级cancel后的握手信号丢弃问题

(四) 实践任务16：完成 AXI 随机延迟验证

- 错误1：地址重复握手
- 错误2：预取指级阻塞处理不当
- 错误3：指令buffer有效位清零时机不对

四.实验总结：几个印象深刻debug的痛点

一.实验任务概览

- 增加握手信号，将CPU访问SRAM的接口修改为类SRAM总线接口；
- 设计“类SRAM-AXI”转接桥，完成固定延迟的上板验证；
- 完善先前设计，完成随机延迟验证。

二.实验设计

• (一) 总体设计思路

- 1. 从SRAM→类SRAM

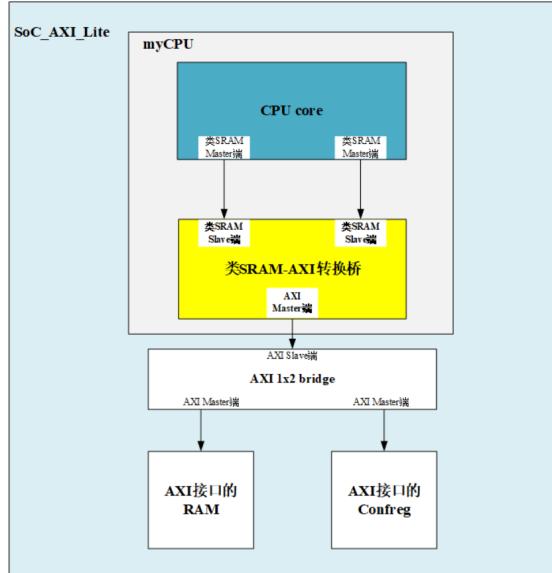
二者的本质区别在于是否有发出请求的下一拍就可以得到数据相应，故需要额外增加握手信号对流水线进行控制。

引入 data_ok 和 addr_ok 信号，用于判断当前请求是否已经被CPU外部接受，若有 req和ok信号同时拉高，意味着完成一次握手，我们希望在这个时候就允许流水级后流。

而若要其能达到控制流水线流动的效果，就需要对我们的 `ready_go` 信号或者 `allow_in` 信号做调整，在本次实验中，选择在握手还未成功时不许 `ready_go` 拉高，并未对 `allow_in` 做额外调整。

- 2. 设计类SRAM-AXI转接桥

目前我们的CPU需要有两个类SRAM接口，而常见的简单CPU通常只有一个接口，故需要设计一个转接桥将两个接口和一个AXI接口对接，如下图所示：



而为了区分信号究竟来自哪个类SRAM接口，我们需要引入 `id` 对当前的读写任务做一个标识。此后数据返回时也根据返回的 `id` 将结果返回相应的接口。

在处理读写任务的时候，我们设计了四个状态机，分别用于记录读请求通道、读响应通道、写地址写数据通道、写响应通道的状态。读写分离可提高流水线的效率，同时读请求&读响应通道的分离使得来自不同类SRAM接口的读请求处理更紧密。

• (二) 流水级的重要变化

- 1. IF 阶段

1.1 下一条取指PC的确定

此时下一条取指地址的生成不仅仅与当前拍的控制信号 `br_taken`、`wb_ex`、`ertn_flush` 有关，还与上一拍的流水级cancel相关信号有关。如，若出现了上一拍恰为分支跳转指令，而在本拍收到了来自写回级的例外信号 `wb_ex`，此时送往指令RAM的PC应该为例外处理入口而非上一拍的分支跳转的目标地址，综合而言有优先级如下：

```
1 | ex_r > wb_ex > ertn_flush_r > ertn_flush > br_taken_r > br_taken
```

其中 `_r` 后缀表前一次未处理的cancel的对应信号。

1.2 指令buffer的引入

若当前数据握手成功，但IF级还不允许数据流入的情况下，可能导致指令的丢失与错位。故引入指令buffer，在上述情况下暂存指令，并在取到的指令顺利流入IF级时将buffer有效位置零。

- 2. EXE级变化：ready_go的调整

- IF级也做了相应的调整，因其与此原理相近，故只展开介绍了EXE的调整。

引入了握手信号后，EXE阶段的ready_go信号不再仅仅与ALU是否完成当前计算的 `alu_complete` 有关，暂不考虑需要使用ALU类的指令的情况，可分为如下两种情况讨论：

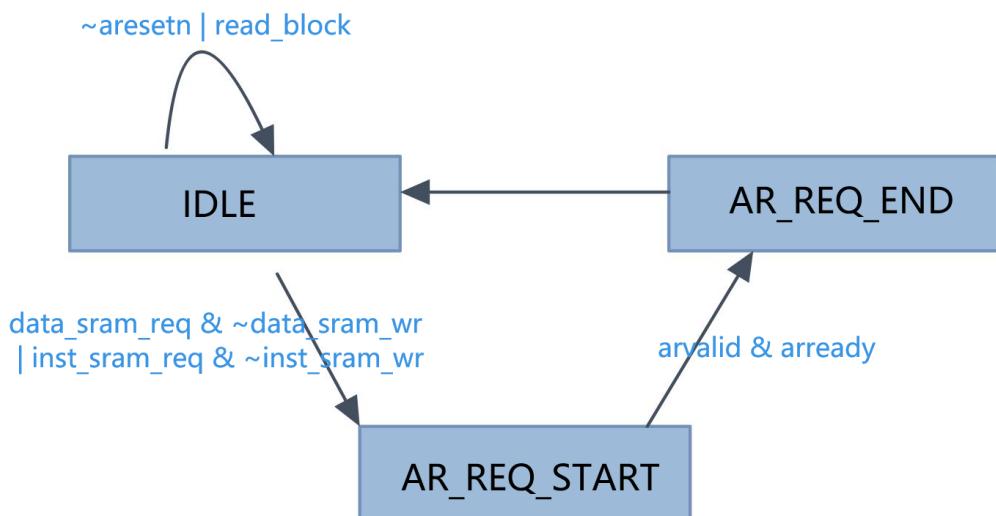
- 当前无访存需求：`ready_go` 可放心置为1；
- 当前有访存请求：需要握手成功后才允许流向下一流水级，

故将`ready_go`信号调整如下：

```
1 assign es_ready_go      = alu_complete & (~data_sram_req |  
2   a_sram_req & data_sram_addr_ok);
```

• (三) 重要模块1设计：读请求通道状态机

- 1. 工作原理



- 2. 状态机状态定义

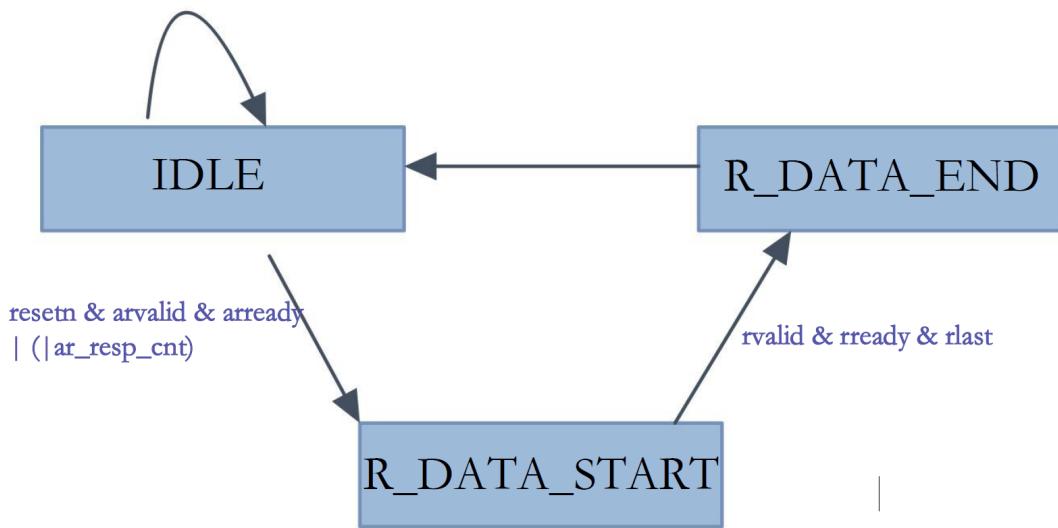
Name	Func
IDLE	表示读请求通道空闲
AR_REQ_START	表示开始处理读请求
AR_REQ_END	表示读请求通道一次握手成功

- 3. 状态机状态转移重要信号

`read_block`：判断当前是否需要阻塞读请求。当读写地址相同且写请求握手成功但未处理完毕时，需要阻塞写请求。

• (四) 重要模块2设计：读响应通道状态机

- 1. 工作原理



- 2. 状态机状态定义

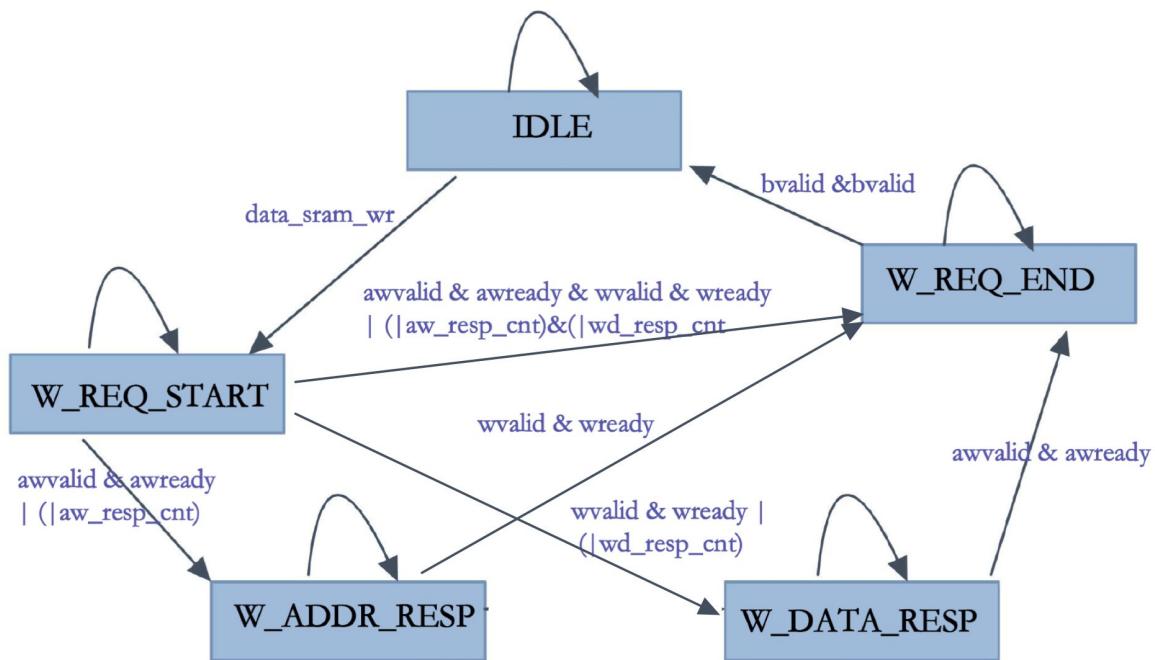
Name	Func
IDLE	表示读响应通道空闲
R_DATA_START	表示开始准备读数据
R_DATA_END	表示读数据已经成功返回

- 3. 状态机状态转移重要信号

- `ar_resp_cnt`：一个计数器，统计当前地址握手成功但还未返回数据的读请求个数；
- `rlast`：此处并不重要（甚至可以忽略），但此次设计还是将它一并放在判定条件中，原因是若后续需实现DMA突发传输or其他数据分组传输的情况，`rlast`就对控制状态机转移有至关重要的作用。

• (五) 重要模块3设计：写数据&写地址通道状态机

- 1. 工作原理



- 2. 状态机状态定义

Name	Func
IDLE	表示写请求通道空闲
W_REQ_START	表示开始处理写请求
W_ADDR_RESP	表示写地址握手成功
W_DATA_RESP	表示写数据握手成功
W_REQ_END	表示写请求发送成功

- 3. 状态机状态转移重要信号

- `aw_resp_cnt`：一个计数器，统计当前写地址握手成功但还未收到写响应通道回复的指令数；
- `wd_resp_cnt`：一个计数器，统计当前写数据握手成功但还未收到写响应通道回复的指令数；

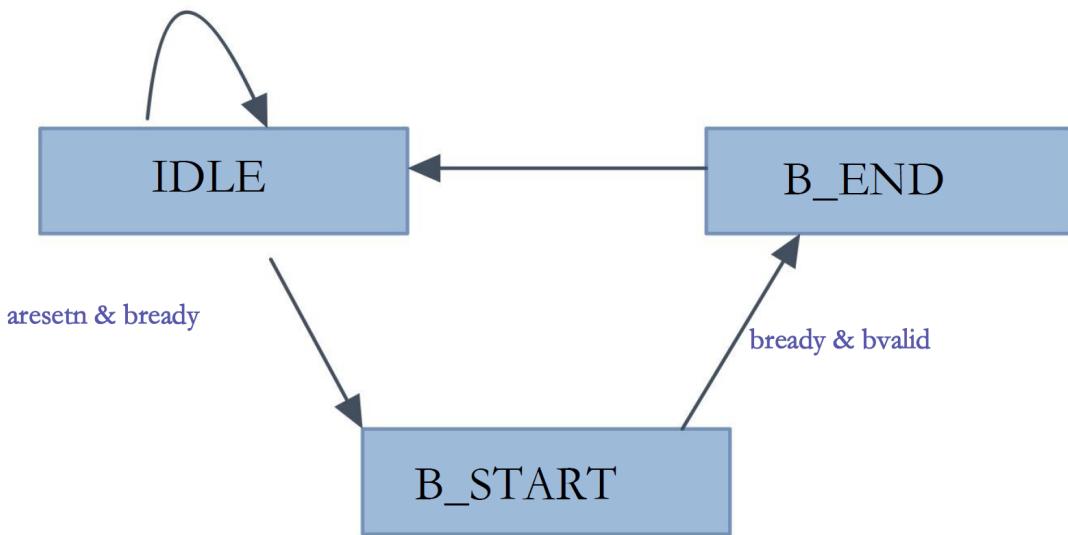
对这两个信号，我们可有如下断言：

- 二者其中任意一个不为0，写请求通道忙；
- 二者同时非0，可代表一次写请求发送成功；
- 二者仅其中一个非0，则需根据是写地址握手成功还是写数据握手成功分别跳转到 `W_ADDR RESP` 或 `W_DATA RESP`；
- `bvalid` 和 `bready`：当二者同时拉高，将写请求通道置为IDLE，这样处理的原因是我们设计中 `bready` 的判断信号有赖于写请求状态机的状态：

```
1 |ign bready = w_current_state[4];
```

• (六) 重要模块4设计：写响应通道状态机

- 1. 工作原理



- 2. 状态机状态定义

Name	Func
IDLE	表示写响应通道空闲
B_START	表示写响应通道开始工作
B_END	表示写响应已经成功返回

- 3. 状态机状态转移重要信号

`bready` : 当写数据&写地址通道处于最后一个阶段时（表示二者都握手成功），将 `bready` 拉高，进行写响应的处理。

三. 实验过程

• (一) 实验流水

- 10月10日 20:00-21:00 阅读exp14讲义；
- 10月10日 21:00-24:00 exp14代码雏形；
- 10月11日 21:00-23:00 exp14 debug；
- 10月12日 20:00-22:00 完成exp14；
- 10月24日 21:00-24:00 阅读理解exp15讲义；
- 10月25日 8:00-10:00 exp15代码雏形；
- 10月25日 20:00-次日2:00 exp15 debug；
- 10月26日 9:00-10:00 完成exp15；
- 10月26日 10:00-12:30 完成exp16；

• (二) 实践任务14：添加类 SRAM总线支持

- 错误1：nextpc确定信号未存于寄存器

(1) 错误现象

写回时PC和金标准不一致：

```
-----  
Test begin!  
-----  
[ 3157 ns] Error!!!  
    reference: PC = 0x1c02c788, wb_rf_wnum = 0x17, wb_rf_wdata = 0x00000001  
    mycpu     : PC = 0x1c00f0d4, wb_rf_wnum = 0x01, wb_rf_wdata = 0x1c00f0d8  
-----
```

(2) 分析定位过程

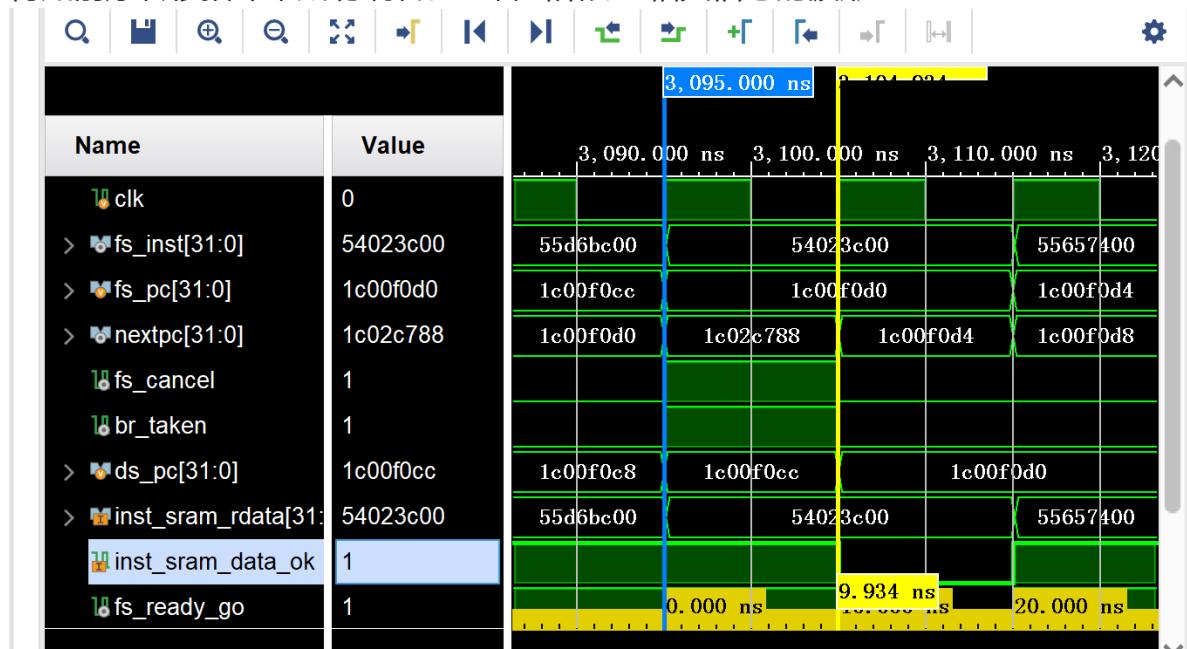
查看当前PC前的指令：

```

1 1c00f0cc <inst_test>:
2 inst_test():
3 1c00f0cc: 55d6bc00    bl 120508(0x1d6bc) # 1c02c788
<n1_lu12i_w_test>
4 1c00f0d0: 54023c00    bl 572(0x23c) # 1c00f30c <idle_1s>
5 1c00f0d4: 55657400    bl 91508(0x16574) # 1c025648
<n2_add_w_test>
6 1c00f0d8: 54023400    bl 564(0x234) # 1c00f30c <idle_1s>
7 1c00f0dc: 56ab5400    bl 174932(0x2ab54) # 1c039c30
<n3_addi_w_test>

```

得知前序转移指令未成功跳转。查看IF阶段PC相关信号的波形：



得知在光标指示处判断出转移应该跳转，但由于前一个周期读出了顺序pc对应的指令，导致该指令流向了ID，使得其br_taken信号只维持了一周期，并且沿着错误的指令执行。

(3) 错误原因

未在判断出 `fs_cancel` 时把 `fs2ds_valid` 拉低，只需把该判断放在 `fs_ready_go` 中来做：

```

1 assign fs_ready_go      = (inst_sram_data_ok | inst_buf_valid) &
~inst_discard & ~fs_cancel;
2 assign fs_allowin       = ~fs_valid | fs_ready_go & ds_allowin |
fs_cancel;
3 assign fs2ds_valid      = fs_valid & fs_ready_go;

```

但即使这样修改，还是会在同样的位置出错，原因是原本判断 `br_taken` 时与上了 `ds_valid`，而 `ds_valid` 实现如下：

```

1  always @(posedge clk) begin
2      if(~resetn)
3          ds_valid <= 1'b0;
4      else if(wb_ex)
5          ds_valid <= 1'b0;
6      else if(br_taken)
7          ds_valid <= 1'b0;
8      else if(ds_allowin)
9          ds_valid <= fs2ds_valid;
10     end
11

```

在光标指示的下一个周期 `ds_valid` 就会被赋予清零，导致 `br_taken` 被判断为0。若仅在 `fs2ds_valid & ds_allowin` 的时候才允许pc更新的话讲错过分支指令跳转的目标地址，故应该在IF阶段设立寄存器暂存pc值：

```

1  assign nextpc      = wb_ex? ex_entry:
2                                ertn_flush? ertn_entry:
3                                br_taken ? br_target : seq_pc;
4  assign nextpc_valid = {32{nextpc_from_r}} & nextpc_r |
{32{~nextpc_from_r}} & nextpc;
5  always @(posedge clk) begin
6      if(~resetn) begin
7          nextpc_r <= 32'b0;
8          nextpc_from_r <= 1'b0;
9      end
10     // 若当前nextpc来源为寄存器且其内对应地址已经获得了来自指令SRAM的
11     ok, 后续nextpc不再从寄存器中取
12     else if(nextpc_from_r & pf_ready_go)
13         nextpc_from_r <= 1'b0;
14     // 若遇到需要取消流水级的情况, 若当前nextpc还不能流入IF级, 需要暂
15     存入寄存器
16     else if(fs_cancel & ~pf_ready_go) begin
17         nextpc_r <= nextpc;
18         nextpc_from_r <= 1'b1;
19     end
20 end

```

则后续取指都使用 `nextpc_valid`。

(4) 修正效果

PC往后更新：

```

-----[ 60797 ns] Error!!!
reference: PC = 0x1c025648, wb_rf_wnum = 0x17, wb_rf_wdata = 0x00000002
mycpu    : PC = 0x1c025648, wb_rf_wnum = 0x01, wb_rf_wdata = 0x1c02564c
-----
```

(5) 归纳总结 (可选)

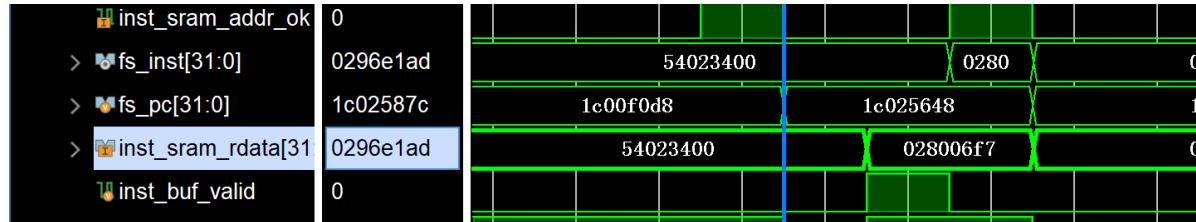
- 错误2：丢弃一条指令时未置空指令buffer

(1) 错误现象

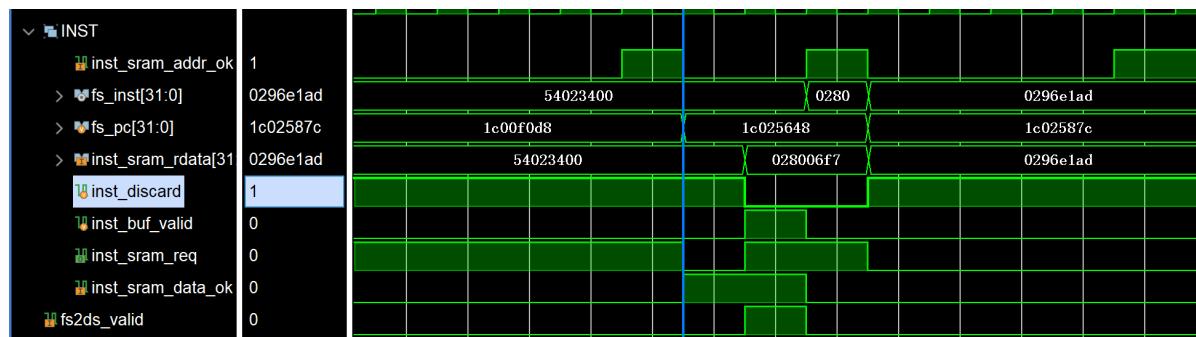
如上，猜测是指令与PC未对上号。

(2) 分析定位过程

查看fs_pc等于上述值的波形：



如图中光标所示，fs_pc对应的inst还是上一个周期的指令，同时注意到当前是分支跳转指令，再查看是否需要扔掉一条指令：



意识到在需要扔掉一条指令的情况下不应该判定指令buffer内的值有效。查看指令buffer有效位的实现：

```
1 // 设置寄存器，暂存指令，并用valid信号表示其内指令是否有效
2 always @(posedge clk) begin
3     if(~resetn) begin
4         fs_inst_buf <= 32'b0;
5         inst_buf_valid <= 1'b0;
6     end
7     else if(fs2ds_valid & ds_allowin) // 缓存已经流向下一流水级
8         inst_buf_valid <= 1'b0;
9     else if(~inst_buf_valid & inst_sram_data_ok) begin
10        fs_inst_buf <= fs_inst;
11        inst_buf_valid <= 1'b1;
12    end
13 end
```

(3) 错误原因

在需要丢弃一条指令的情况下，不应该将buffer置为有效，修正如下：

```
1 // 设置寄存器，暂存指令，并用valid信号表示其内指令是否有效
2 always @(posedge clk) begin
3     if(~resetn) begin
4         fs_inst_buf <= 32'b0;
5         inst_buf_valid <= 1'b0;
6     end
7     else if(fs2ds_valid & ds_allowin) // 缓存已经流向下一流水级
8         inst_buf_valid <= 1'b0;
9     else if(~inst_buf_valid & inst_sram_data_ok &
~inst_discard) begin
10        fs_inst_buf <= fs_inst;
11        inst_buf_valid <= 1'b1;
12    end
13 end
```

(4) 修正效果

```
[ 52000 ns] Test is running, debug_wb_pc = 0x1c02d4bc
-----
[ 60817 ns] Error!!!
reference: PC = 0x1c025648, wb_rf_wnum = 0x17, wb_rf_wdata = 0x00000002
mycpu   : PC = 0x1c02564c, wb_rf_wnum = 0x17, wb_rf_wdata = 0x00000002
-----
```

pc往后更新。

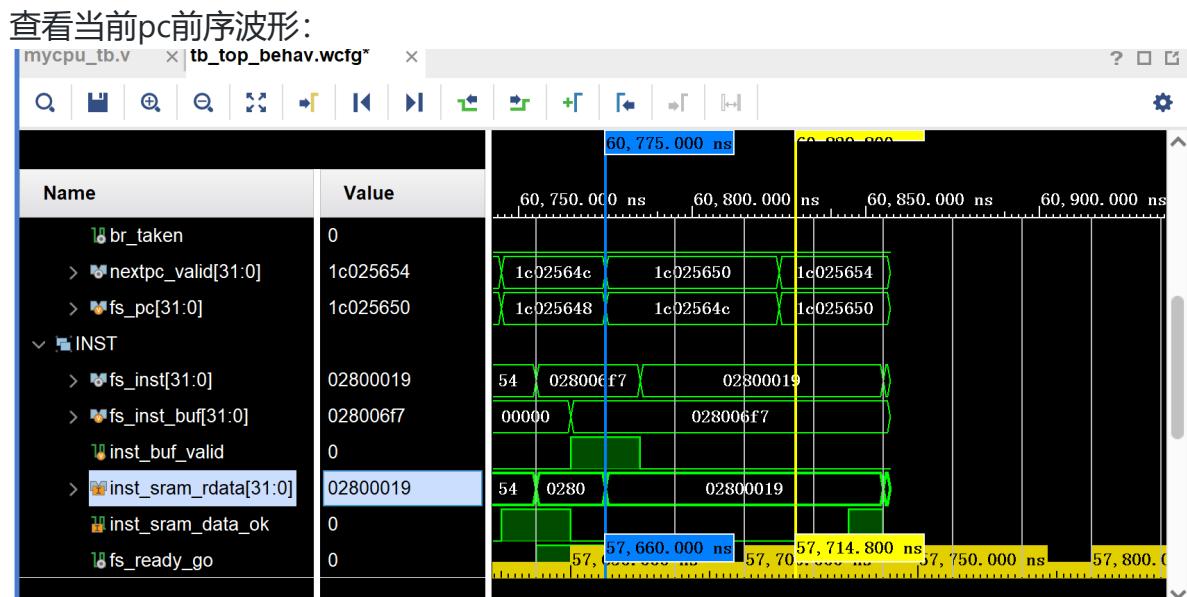
(5) 归纳总结 (可选)

- 错误3：fs_allowin理解不到位

(1) 错误现象

如图所示，PC与指令错位了。

(2) 分析定位过程



意识到此时buffer中的值被判定为有效，导致指令被赋予为上一个周期的指令。再往前看，发现在PC为 1c025648 时discard拉高，而我原本的allowin实现如下：

```
1 |     assign fs_allowin      = ~fs_valid | fs_ready_go & ds_allowin |
    |     inst_discard;
```

这将导致该条PC被冲掉。

(3) 错误原因

`fs_allowin` 不应该或上 `inst_discard`。

(4) 修正效果

修改后pc往后更新：

```
[1612000 ns] Test is running, debug_wb_pc = 0x1c052818
-----
[1615857 ns] Error!!!
reference: PC = 0x1c06170c, wb_rf_wnum = 0x0a, wb_rf_wdata = 0xc822c7e8
mycpu   : PC = 0x1c06170c, wb_rf_wnum = 0x0a, wb_rf_wdata = 0xxxxxxxxx
-----
$finish called at time : 1615897 ns : File "D:/Desktop/cdp_edc_local/mycpu_env_14/soc_verify/soc_hs_bram/testbench/mycpu_tb.v" Line 16
```

(5) 归纳总结 (可选)

- 错误4：信号未定义

(1) 错误现象

如上，写回数据未定义。

(2) 分析定位过程

查看当前pc对应指令：

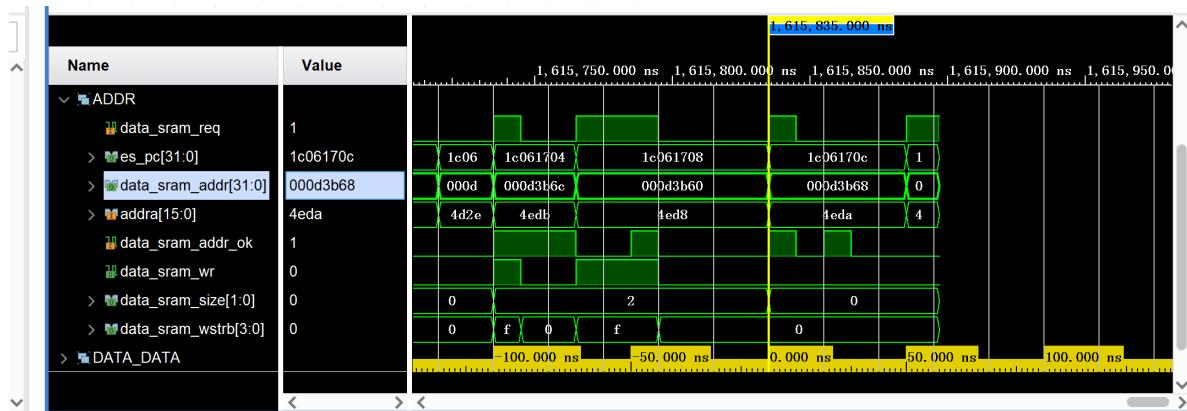
```

1 1c0616fc: 02801184 addi.w $r4,$r12,4(0x4)
2 1c061700: 02bfe185 addi.w $r5,$r12,-8(0xff8)
3 1c061704: 299aa084 st.w $r4,$r4,1704(0x6a8)
4 1c061708: 299aa0a5 st.w $r5,$r5,1704(0x6a8)
5 1c06170c: 289aa18a ld.w $r10,$r12,1704(0x6a8)
6 1c061710: 289aa086 ld.w $r6,$r4,1704(0x6a8)
7 1c061714: 289aa0a4 ld.w $r4,$r5,1704(0x6a8)
8 1c061718: 289aa0a6 ld.w $r6,$r5,1704(0x6a8)

```

为ld指令。

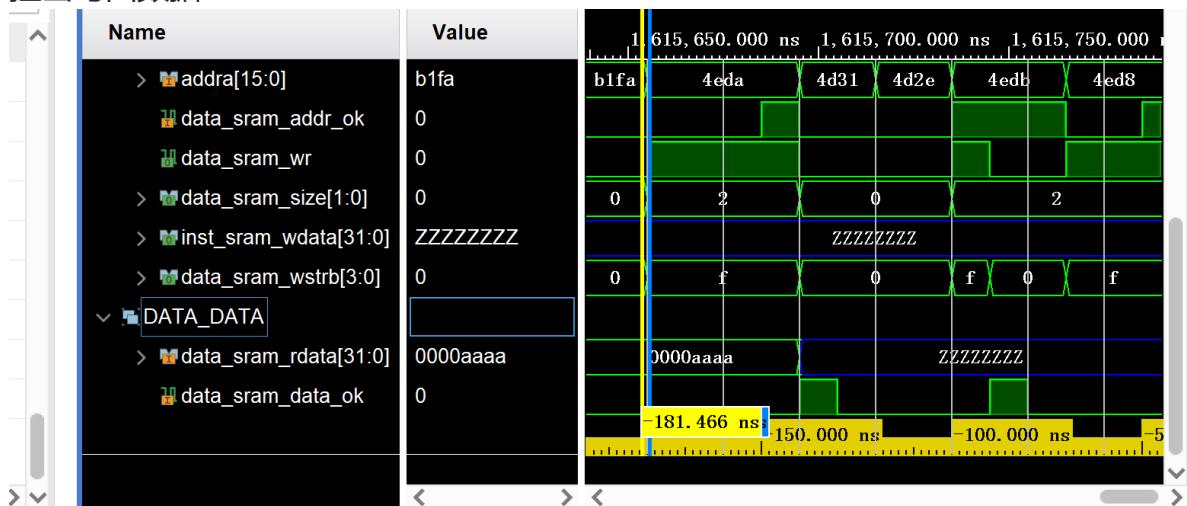
查看当前指令对应写地址：



查看最近一次写入该地址的时刻：



拉出写回数据：



发现写回数据未定义。

(3) 错误原因

顶层模块漏连接 `data_sram_wdata`。

(4) 修正效果

通过若干测试：

```
--> [5958805 ns] Number 8'd40 Functional Test Point PASS!!!
      [5962000 ns] Test is running, debug_wb_pc = 0x1c02f904
-----
[5965077 ns] Error!!!
  reference: PC = 0x1c02f9cc, wb_rf_wnum = 0x04, wb_rf_wdata = 0x000d1434
  mycpu    : PC = 0x1c02f9cc, wb_rf_wnum = 0x04, wb_rf_wdata = 0x000d1428
-----
$finish called at time : 5965117 ns : File "D:/Desktop/cdp_ede_local/mycpu_env_14/soc_verify/soc_hs_bram/testbe
```

(5) 归纳总结 (可选)

- 错误5：br_stall判断未考虑data_ok

(1) 错误现象

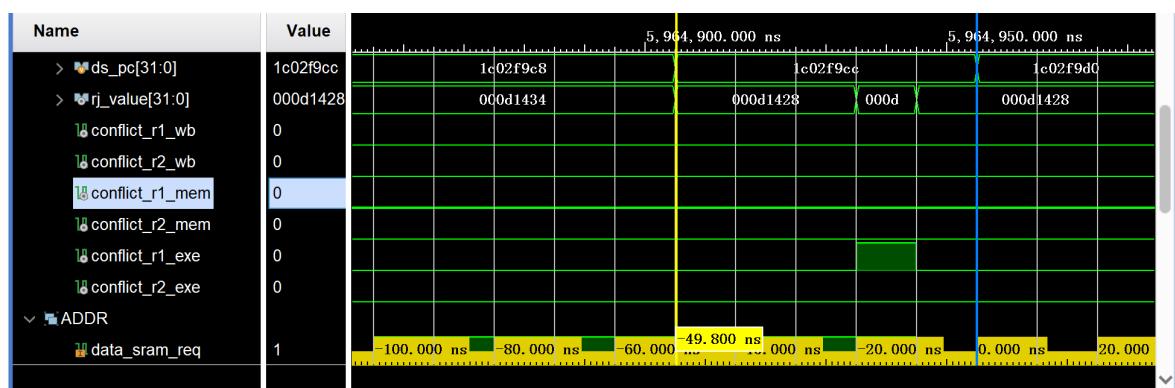
如图，写回数据有误。

(2) 分析定位过程

查看指令类型：

33953	1c02f9c0:	29b920a5	st.w	\$r5,\$r5,-440(0xe48)
33954	1c02f9c4:	28392190	ld.b	\$r16,\$r12,-440(0xe48)
33955	1c02f9c8:	28b92085	ld.w	\$r5,\$r4,-440(0xe48)
33956	1c02f9cc:	28b920a4	ld.w	\$r4,\$r5,-440(0xe48)
33957	1c02f9d0:	28b920a6	ld.w	\$r6,\$r5,-440(0xe48)
22052	1c02f9d1:	5c1cadfa	hno \$r15 \$r16 712a(0x1c0a) # 1c02f9d0 <inst errors>	

意识到此处存在RAW相关，查看conflict信号：



(3) 错误原因

发现由于冲突信号和 `xx2xx_valid` 信号有关，其中 `es2ms_valid` 只有效了一拍，且进跟着的 `ms2ws_valid` 并未立刻拉高（原因是访存指令还在等待数据返回），导致后续选取alu操作数的时候有误，导致读取数据地址有误。

故此时 `br_stall` 的判断不能仅仅为判断exe阶段是否为ld类指令，还应该等待mem阶段的ld类指令等到 `data_sram_data_ok`，才可以继续执行。

(4) 修正效果

通过当前测试点：

```
[6382000 ns] Test is running, debug_wb_pc = 0x1c021624
----[6387945 ns] Number 8'd44 Functional Test Point PASS!!!
[6392000 ns] Test is running, debug_wb_pc = 0x1c052954
-----
[6392497 ns] Error!!!
reference: PC = 0x1c052968, wb_rf_wnum = 0x10, wb_rf_wdata = 0x9ef0fca8
mycpu    : PC = 0x1c052968, wb_rf_wnum = 0x10, wb_rf_wdata = 0x9efafca8
```

(5) 归纳总结 (可选)

- 错误6：错把访存最低二位地址强制置零

(1) 错误现象

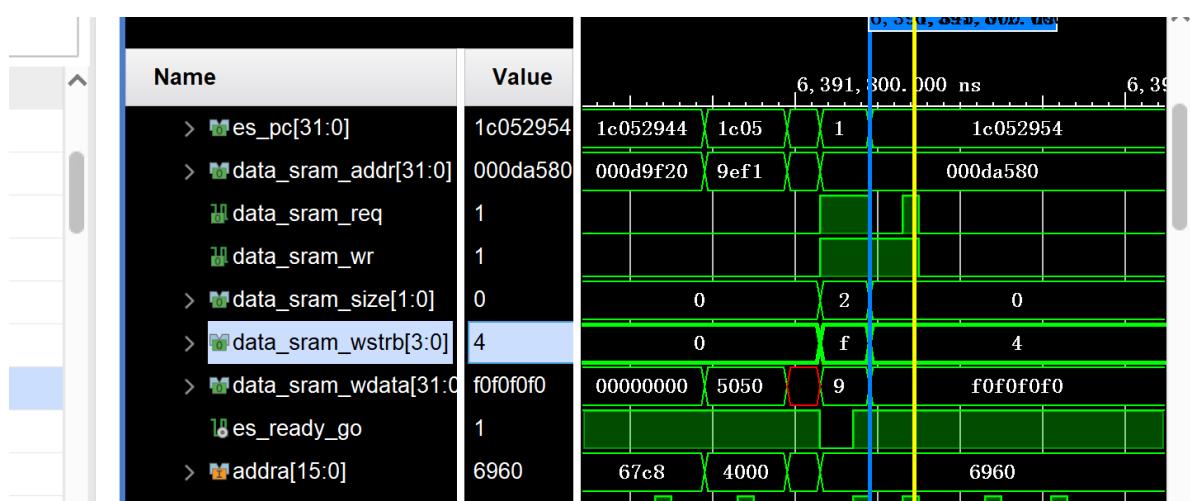
如上，写回数据有误。

(2) 分析定位过程

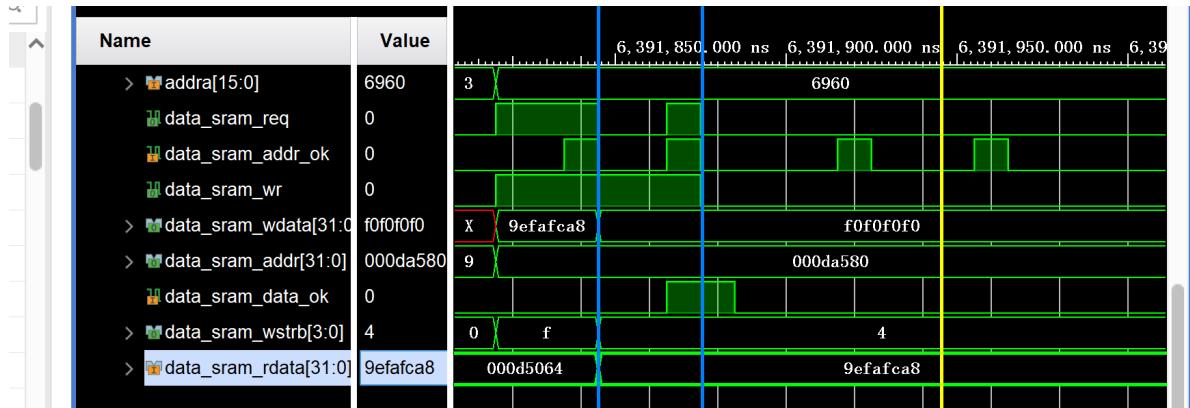
查看指令类型：

69872	1c05294c:	02b2a1ef	addi.w	\$r15,\$r15,-856(0xca8)
69873	1c052950:	2999818e	st.w	\$r14,\$r12,1632(0x660)
69874	1c052954:	2919898d	st.b	\$r13,\$r12,1634(0x662)
69875	1c052958:	02801184	addi.w	\$r4,\$r12,4(0x4)
69876	1c05295c:	02bff185	addi.w	\$r5,\$r12,-4(0xffc)
69877	1c052960:	29998084	st.w	\$r4,\$r4,1632(0x660)
69878	1c052964:	299980a5	st.w	\$r5,\$r5,1632(0x660)
69879	1c052968:	28998190	ld.w	\$r16,\$r12,1632(0x660)
69880	1c05296c:	289980a4	ld.w	\$r4,\$r5,1632(0x660)
69881	1c052970:	28998085	ld.w	\$r5,\$r4,1632(0x660)
69882	1c052974:	289980a6	ld.w	\$r6,\$r5,1632(0x660)
69883	1c052978:	5c20b5f0	bne	\$r15,\$r16,8372(0x20b4) # 1c054a2c <inst_>
69884	1c05297c:	152c91ee	lu12i.w	\$r14,-433009(0x9648f)

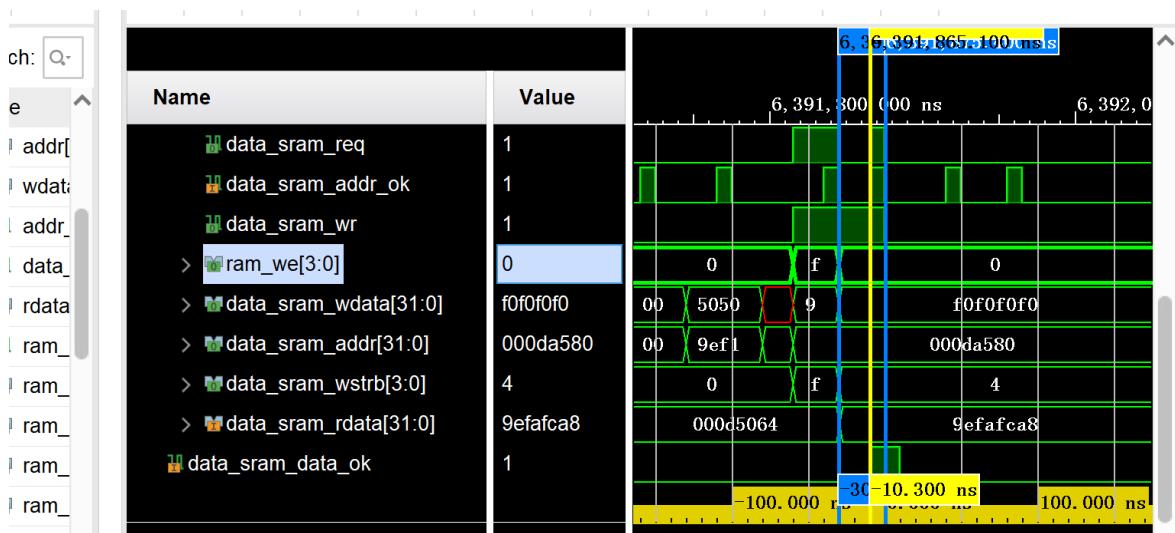
猜测时ld.b时处理有误，查看前序波形：



发现此时发生了连续读写的情况，单独拉出DATA SRAM的信号：



发现由于data_ok拉高了两拍，且恰好在两次写操作的交界处。但理论上data_ok拉高了两拍，应该有数据全部正确写入。在piazza提问后受老师提醒查看data_ram的波形：



发现第二次并没有成功写入 (ram_we=0)，查看ram_we的实现：

```
1 o ram
2 e [3:0] size_decode = size==2'd0 ?
3   dr[1:0]==2'd3,addr[1:0]==2'd2,addr[1:0]==2'd1,addr[1:0]==2'd0} :
4     size==2'd1 ?
5   dr[1],addr[1],~addr[1],~addr[1]} :
6     4'hf;
7 ign ram_en    = req && addr_ok;
8 ign ram_we    = {4{wr}} & wstrb & size_decode;
9 ign ram_addr  = addr;
@assign ram_wdata = wdata;
```

发现其还与addr有关，查看soc_lite_top模块：

```

9      .wdata          (data_sram_wdata  ),
10     .addr_ok        (data_sram_addr_ok),
11     .data_ok         (data_sram_data_ok),
12     .rdata          (data_sram_rdata  ),
13
14 //sSlave
15     .ram_en         (data_ram_en       ),
16     .ram_we         (data_ram_we       ),
17     .ram_addr        (data_ram_addr      ),
18     .ram_wdata        (data_ram_wdata      ),
19     .ram_rdata        (data_ram_rdata      ),
20 //from confreg
21     .ram_random_mask (ram_random_mask[3:2])
22 );
23 data_ram data_ram
24

```

其直接连的是未经过映射的地址（未去掉最低二位），而我最早在写前序实验时为了避免地址错误的情况直接强制把写地址的最低二位置零：

```

assign data_sram_wstrb = es_mem_we;
assign data_sram_size = {2{op_st_b}} & 2'b0 | {2{op_st_h}};
assign data_sram_addr = {es_alu_result[31:2], 2'b0};
assign data_sram_wdata[7:0] = es_rkd_value[7:0];
assign data_sram_wdata[15:8] = op_st_h ? es_rkd_value[7:0];

```

故会导致译出的 `size_decode` 为0，致使最后一次未成功写入。

(3) 错误原因

写地址的最后两位不应该“人工对齐”。

(4) 修正效果

来到后续测试点：

```

[6612000 ns] Test is running, debug_wb_pc = 0x1c02ede0
[6622000 ns] Test is running, debug_wb_pc = 0x1c02f09c
[6632000 ns] Test is running, debug_wb_pc = 0x1c02f330
----[6641665 ns] Number 8'd46 Functional Test Point PASS!!!
[6642000 ns] Test is running, debug_wb_pc = 0x1c00f318
[6652000 ns] Test is running, debug_wb_pc = 0x1c00f078
[6662000 ns] Test is running, debug_wb_pc = 0x1c00f078
[6672000 ns] Test is running, debug_wb_pc = 0x1c00f078

```

(5) 归纳总结 (可选)

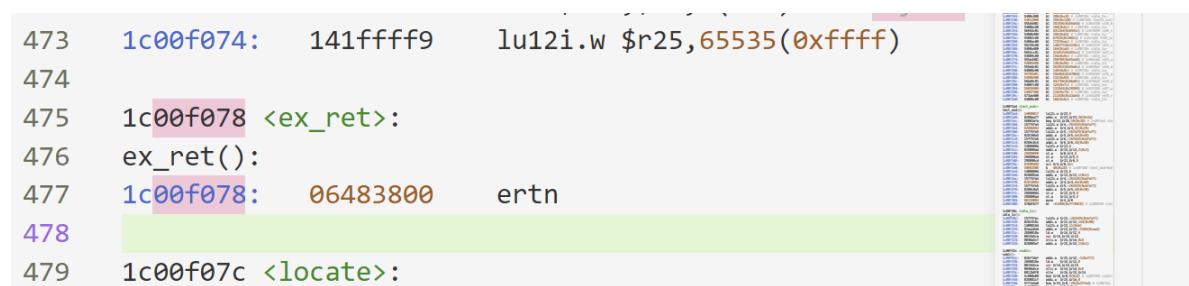
- 错误7：中断返回信号未与valid信号挂钩

(1) 错误现象

如图所示，PC卡死。

(2) 分析定位过程

查看当前指令：



```
473  1c00f074: 141fffff9    lu12i.w $r25,65535(0xffff)
474
475  1c00f078 <ex_ret>:
476  ex_ret():
477  1c00f078: 06483800    ertn
478
479  1c00f07c <locate>:
```

查看wb_pc等于该值时的波形：



(3) 错误原因

意识到 `ertn_flush` 未与 `valid` 挂钩，导致其一直拉高，致使流水线阻塞。

按如下方式修改：

```
1 | assign {ws_except_ale, ws_except_adef, ws_except_ine, ws_except_int,
2 |   except_brk,
3 |     ws_except_sys, ws_ertn} = ws_except_zip;      //  
ertn_flush=inst_ertn  
3 | assign ertn_flush = ws_ertn & ws_valid;
```

(4) 修正效果

来到新的错误点：

```

----[6641665 ns] Number 8' d46 Functional Test Point PASS!!!
[6642000 ns] Test is running, debug_wb_pc = 0x1c00f318

[6651877 ns] Error!!!
reference: PC = 0x1c050b74, wb_rf_wnum = 0x19, wb_rf_wdata = 0x00000001
mycpu : PC = 0x1c050c24, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x2f000000

```

(5) 归纳总结 (可选)

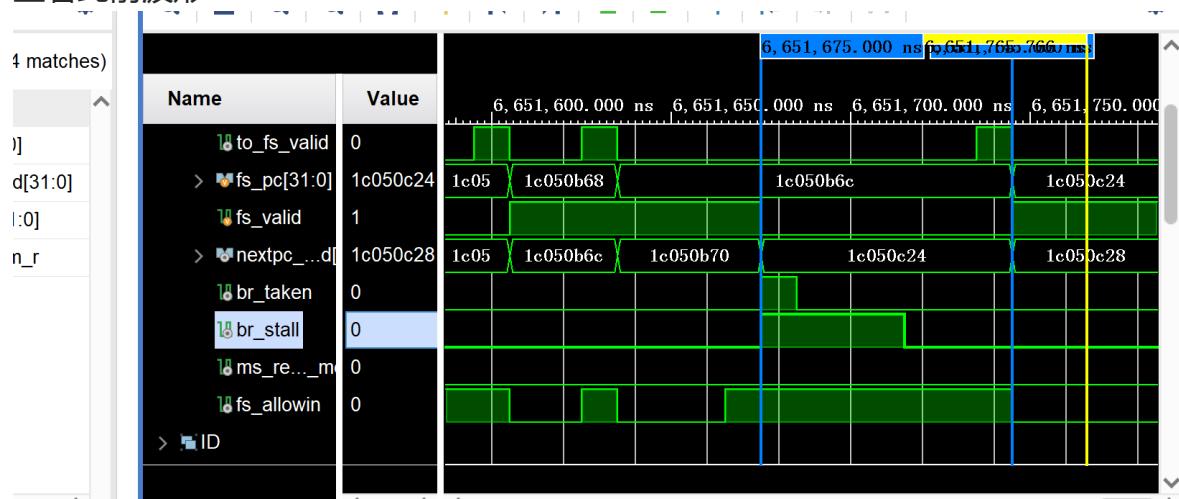
- 错误8：MEM阶段的例外信号未与上valid

(1) 错误现象

PC和金标准不对应。

(2) 分析定位过程

查看此前波形：



发现在首个蓝色光标处判断出了分支跳转有效。查看当前pc前序指令：

```

1 50b60 <syscall_pc2>:
2 50b60: 002b0000    syscall 0x0
3 50b64: 2980027b    st.w    $r27,$r19,0
4 50b68: 2880126d    ld.w    $r13,$r19,4(0x4)
5 50b6c: 5c00b9bb    bne $r13,$r27,184(0xb8) # 1c050c24 <inst_error>
6 50b70: 5c00b73e    bne $r25,$r30,180(0xb4) # 1c050c24 <inst_error>
7 50b74: 03800419    ori $r25,$r0,0x1
8 50b78: 29800279    st.w    $r25,$r19,0

```

查看st.w和ld.w指令的执行：

es_mem_req	1								
> es_pc[31:0]	1c050b68	1e00f078	1e00f07e		1e050b64				1e050b68
ms_ex	0								
wb_ex	0								
DATA_SRAM									
> addr[15:0]	4001	0000	fc00		4000				4001
> data_sram_req	1								
> data_sram_addr_ok	1								
> data_sram_wr	0								
> data_sram_wdata[31:0]	f0fffffc	00000000	2f2f002f		1e050b60				f0fffffc
> data_sram_addr[31:0]	001d0004	00000000	bfaaff000		001d0000				001d0004
> data_sram_data_ok	0								
> data_sram_wstrb[3:0]	0	0			f				0

发现 `es_pc=1c050b64` 时未拉高wr，查看其实现：

```
1 |ign data_sram_wr      = ( |data_sram_wstrb) & es_valid & ~wb_ex &
~ms_ex & ~es_ex;
```

反观波形可知此时ms_ex拉高。

(3) 错误原因

`ms_ex` 未与当前流水级的 `valid` 信号挂钩，导致写使能失效。同理应该修改 `es_ex`。

(4) 修正效果

来到后续测试点：

```
[6682000 ns] Test is running, debug_wb_pc = 0x1c00f060
----[6686745 ns] Number 8'd48 Functional Test Point PASS!!!
[6692000 ns] Test is running, debug_wb_pc = 0x1c008004
[6702000 ns] Test is running, debug_wb_pc = 0x1c078738
[6712000 ns] Test is running, debug_wb_pc = 0x1c078738
[6722000 ns] Test is running, debug_wb_pc = 0x1c078738
[6732000 ns] Test is running, debug_wb_pc = 0x1c008018
[6742000 ns] Test is running, debug_wb_pc = 0x1c078744
[6752000 ns] Test is running, debug_wb_pc = 0x1c078744
[6762000 ns] Test is running, debug_wb_pc = 0x1c078744
[6772000 ns] Test is running, debug_wb_pc = 0x1c078744
[6782000 ns] Test is running, debug_wb_pc = 0x1c078744
[6792000 ns] Test is running, debug_wb_pc = 0x1c078744
```

(5) 归纳总结 (可选)

- 错误9：取指连续cancel时处理不当

(1) 错误现象

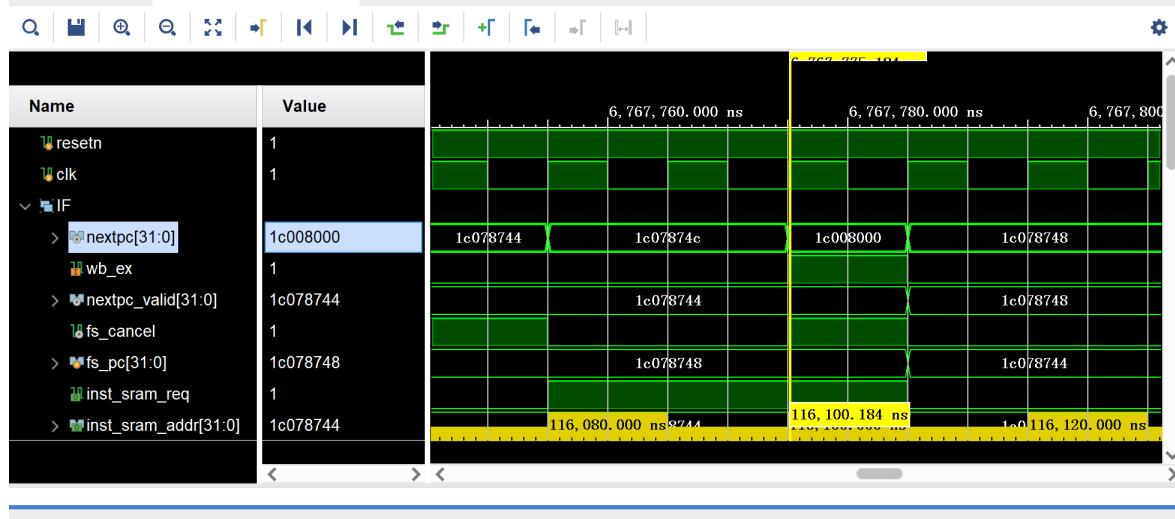
pc再度卡死。

(2) 分析定位过程

查看当前pc对应指令：

```
1770 1c078734: 0401042c csrwr $r12,0x41
1771 1c078738: 50000000 b 0 # 1c078738 <n49_ti_ex_test+0xa8>
1772 1c07873c: 1c00001b pcaddu12i $r27,0
1773 1c078740: 0280237b addi.w $r27,$r27,8(0x8)
1774 1c078744: 50000000 b 0 # 1c078744 <n49_ti_ex_test+0xb4>
1775 1c078748: 04010420 csrwr $r0,0x41
1776 1c07874c: 5c04233e bne $r25,$r30,1056(0x420) # 1c078b6c <inst_error>
```

意识到目前正在测试时钟中断。查看fs_pc的更新状况：



发现在时钟中断到来时有效的nextpc存的不是中断入口。

查看实现：

```

1 assign nextpc_valid      = {32{nextpc_from_r}} & nextpc_r |
~nextpc_from_r} & nextpc;
2 always @(posedge clk) begin
3     if(~resetn) begin
4         nextpc_r <= 32'b0;
5         nextpc_from_r <= 1'b0;
6     end
7     // 若当前nextpc来源为寄存器且其内对应地址已经获得了来自指令SRAM的ok,
8     // nextpc不再从寄存器中取
9     else if(nextpc_from_r & pf_ready_go)
10        nextpc_from_r <= 1'b0;
11     // 若遇到需要取消流水级的情况，若当前nextpc还不能流入IF级，需要暂存入寄
12
13     else if(fs_cancel & ~pf_ready_go) begin
14         nextpc_r <= nextpc;
15         nextpc_from_r <= 1'b1;
16     end
17 end

```

若当前状态为 `pf_ready_go` 且 `nextpc_from_r` 状态恰为1，`nextpc_from_r` 将被清为0，最后 `nextpc_valid` 来自于 `nextpc`（其在下一周期已经不为中断入口地址），故导致出错。

(3) 错误原因

在上述情况下应该分类讨论，如果 `nextpc_from_r` 状态恰为1，且此时判断出 `fs_cancel`，即前一个周期也有`fs_cancel`，则应该讨论优先级：如若前序诱因是 `br_taken`，且当前的诱因是 `ertn_flush` 或者 `wb_ex`，则应该以当前的为准，若当前诱因也是 `br_taken`，则应以前序寄存器中的pc为准，总体而言有：`wb_ex_r > wb_ex > ertn_flush_r > ertn_flush > br_taken_r > br_taken`。

取指所用pc实现如下：

```
1 assign nextpc      = wb_ex_r? ex_entry_r: wb_ex? ex_entry:
2                           ertn_flush_r? ertn_entry_r: ertn_flush?
3                           _entry:
4                           br_taken_r? br_target_r: br_taken ?
5  arget : seq_pc;
6 always @(posedge clk) begin
7   if(~resetn) begin
8     {wb_ex_r, ertn_flush_r, br_taken_r} <= 3'b0;
9     {ex_entry_r, ertn_entry_r, br_target_r} <= {3{32'b0}};
10    end
11   // 当前仅当遇到fs_cancel时未等到pf_ready_go，需要将cancel相关信号存
12   // 寄存器
13   else if(wb_ex & ~pf_ready_go) begin
14     ex_entry_r <= ex_entry;
15     wb_ex_r <= 1'b1;
16   end
17   else if(ertn_flush & ~pf_ready_go) begin
18     ertn_entry_r <= ertn_entry;
19     ertn_flush_r <= 1'b1;
20   end
21   else if(br_taken & ~pf_ready_go) begin
22     br_target_r <= br_target;
23     br_taken_r <= 1'b1;
24   end
25   // 若对应地址已经获得了来自指令SRAM的ok，后续nextpc不再从寄存器中取
26   else if(pf_ready_go) begin
27     {wb_ex_r, ertn_flush_r, br_taken_r} <= 3'b0;
28   end
29 end
```

(4) 修正效果

通过测试：

```

[7502000 ns] Test is running, debug_wb_pc = 0x1c07b200
[7512000 ns] Test is running, debug_wb_pc = 0x1c008028
----[7514195 ns] Number 8'd56 Functional Test Point PASS!!!
[7522000 ns] Test is running, debug_wb_pc = 0x1c008078
[7532000 ns] Test is running, debug_wb_pc = 0x1c02f730
[7542000 ns] Test is running, debug_wb_pc = 0x1c00f060
----[7546515 ns] Number 8'd57 Functional Test Point PASS!!!
[7552000 ns] Test is running, debug_wb_pc = 0x1c042e94
----[7560965 ns] Number 8'd58 Functional Test Point PASS!!!
[7562000 ns] Test is running, debug_wb_pc = 0x1c00f33c
=====
Test end!
----PASS!!!
$finish called at time : 7562985 ns : File "D:/Desktop/cdp_edc_local/mycpu_env_14/soc_verify/soc_hs_bram/testbench/mycpu_tb.
run: Time (s): cpu = 00:01:22 ; elapsed = 00:01:20 . Memory (MB): peak = 1012.230 ; gain = 0.000

```

(5) 归纳总结 (可选)

- 错误10：上板使用部分RANDOM_SEED无法通过测试

(1) 错误现象

上板修改为某些特定的随机种子会出错。

(2) 分析定位过程

将当前RANDOM_SEED宏改为上板时使用的随机种子。仿真时报错如下：

```

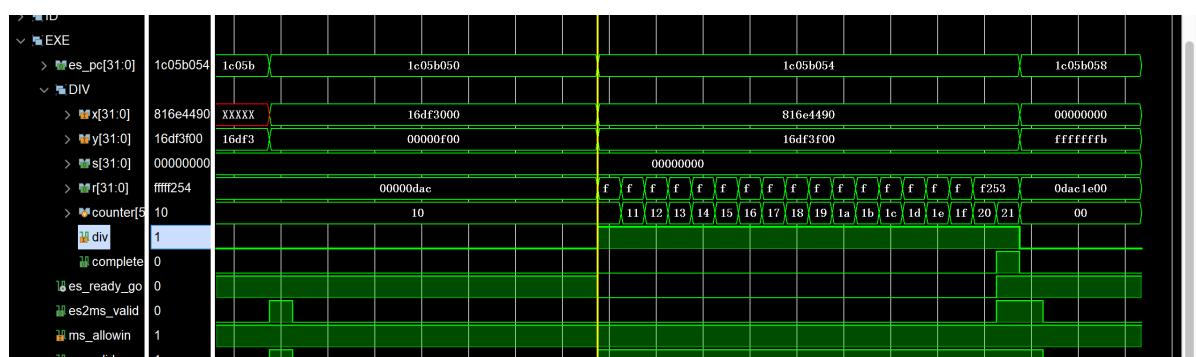
[10282000 ns] Test is running, debug_wb_pc = 0x1c05ae90
[10292000 ns] Test is running, debug_wb_pc = 0x1c05af98
-----
[10299227 ns] Error!!!
reference: PC = 0x1c05b054, wb_rf_wnum = 0x0f, wb_rf_wdata = 0xfffffffffb
mycpu    : PC = 0x1c05b054, wb_rf_wnum = 0x0f, wb_rf_wdata = 0x00000000

```

查看当前指令：

3521	1c05b044:	1502dc8c	lu12i.w \$r12,-518428(0x816e4)
3522	1c05b048:	0392418c	ori \$r12,\$r12,0x490
3523	1c05b04c:	142dbe6d	lu12i.w \$r13,93683(0x16df3)
3524	1c05b050:	03bc01ad	ori \$r13,\$r13,0xf00
3525	1c05b054:	0020358f	div.w \$r15,\$r12,\$r13
3526	1c05b058:	02bfec10	addi.w \$r16,\$r0,-5(0xffb)
3527	1c05b05c:	5c0b5e0f	bne \$r16,\$r15,2908(0xb5c) # 1c05bbb8 <inst_end>
3528	1c05b060:	1502dc4c	lu12i.w \$r12,-405560(0x9471c)

查看除法器的计算状况：



意识到由于当前alu计算完毕后未必有当前流水级可以等来下一个指令，故在此前的计数器会在等待过程中自增，导致新的指令到来时计数器未复位，故其复位信号应该与 `ds2es_valid & es_allowin` 信号挂钩，确保每次新的指令到来时计数器都被复位。

(3) 错误原因

ALU复位信号考虑不周全。

(4) 修正效果

当前RANDOM_SEED下仿真正确，上板也可正常运行。

```
----- [16082000 ns] Test is running, debug_wb_pc = 0x1c00823c
-----[16086735 ns] Number 8'd57 Functional Test Point PASS!!!
    [16092000 ns] Test is running, debug_wb_pc = 0x1c042d8c
    [16102000 ns] Test is running, debug_wb_pc = 0x1c042eb8
    [16112000 ns] Test is running, debug_wb_pc = 0x1c042ffc
-----[16117505 ns] Number 8'd58 Functional Test Point PASS!!!
    [16122000 ns] Test is running, debug_wb_pc = 0x1c00f304
=====
Test end!
----PASS!!!
$finish called at time : 16122385 ns : File "D:/Desktop/cdp_ede_local/mycpu_env_14/soc_verify/soc_h
```

(5) 归纳总结 (可选)

• (三) 实践任务15：添加 AXI 总线支持

- 给本组uu的大致思路概述：IF级和MEM级会有访存的情况，二者用ID信号区分。
- 在IF级接收到数据之前，其不会再申请下一条指令；
- 对于需要在EXE发送读数据请求、在MEM级等待读数据返回的DATA RAM，也可以保证收到数据之前不会发送下一条读/写数据请求，原因是我们将发送的req与MEM阶段的allowin挂钩。但是有可能出现
- 以上，我们只需要给取指和取数各设立一个寄存器，并用ID做index即可。

- 错误1：状态机状态设计不合理

(1) 错误现象

指令没流起来：

```

[2082000 ns] Test is running, debug_wb_pc = 0x00000000
[2092000 ns] Test is running, debug_wb_pc = 0x00000000
[2102000 ns] Test is running, debug_wb_pc = 0x00000000
[2112000 ns] Test is running, debug_wb_pc = 0x00000000
[2122000 ns] Test is running, debug_wb_pc = 0x00000000
[2132000 ns] Test is running, debug_wb_pc = 0x00000000
[2142000 ns] Test is running, debug_wb_pc = 0x00000000
[2152000 ns] Test is running, debug_wb_pc = 0x00000000
[2162000 ns] Test is running, debug_wb_pc = 0x00000000
[2172000 ns] Test is running, debug_wb_pc = 0x00000000
[2182000 ns] Test is running, debug_wb_pc = 0x00000000
[2192000 ns] Test is running, debug_wb_pc = 0x00000000

```

(2) 分析定位过程

查看握手过程，发现valid和ready信号似乎逻辑正确，再查看读数据，rdata读出了正确的数据，但是返回给CPU的数据有误：



查看实现：

```

1 assign data_sram_rdata = buf_rdata[0];
2 assign data_sram_addr_ok = arid[0] & ar_current_state[2]; // arvalid
   ready的下一拍
3 assign data_sram_data_ok = rid[0] & r_current_state[2]; // rvalid &
   ready的下一拍
4 assign inst_sram_rdata = buf_rdata[1];
5 assign inst_sram_addr_ok = ~arid[0] & ar_current_state[2]; // 
   alid & arready的下一拍
6 assign inst_sram_data_ok = ~rid[0] & r_current_state[2]; // rvalid &
   ready的下一拍

```

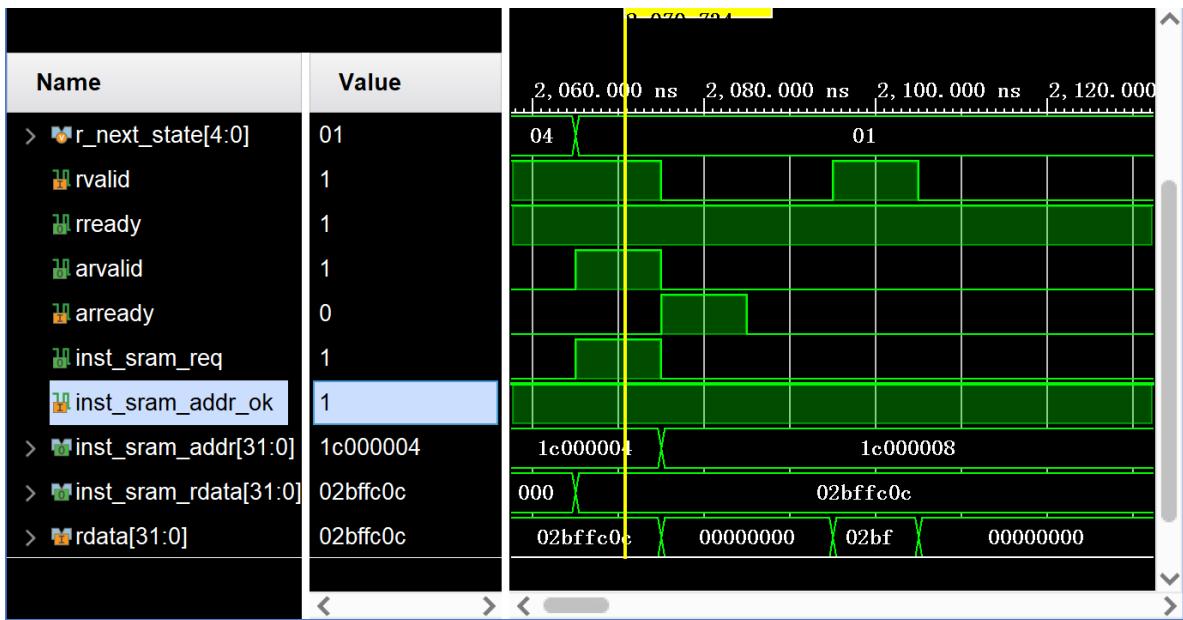
发现index搞反，指令rdata应从第一个寄存器拿。修改后依旧卡住：

```

[1472000 ns] Test is running, debug_wb_pc = 0x1c000000
[1482000 ns] Test is running, debug_wb_pc = 0x1c000000
[1492000 ns] Test is running, debug_wb_pc = 0x1c000000
[1502000 ns] Test is running, debug_wb_pc = 0x1c000000
[1512000 ns] Test is running, debug_wb_pc = 0x1c000000
[1522000 ns] Test is running, debug_wb_pc = 0x1c000000
[1532000 ns] Test is running, debug_wb_pc = 0x1c000000

```

发现addr_ok一直拉高导致实际上握手未成功而req已经拉低：



查看addr_ok的实现：

```

1 assign inst_sram_addr_ok = ~arid[0] & ar_current_state[2]; // lid & arready的下一拍
2 always @(*) begin
3   case(ar_current_state)
4     IDLE:begin
5       if(aresetn & arvalid & ~read_block)
6         ar_next_state = AR_REQ_START;
7       else
8         ar_next_state = IDLE;
9     end
10    AR_REQ_START:begin
11      if(arvalid & arready)
12        ar_next_state = AR_REQ_END;
13      else
14        ar_next_state = AR_REQ_START;
15    end
16    AR_REQ_END:begin
17      ar_next_state = AR_REQ_END;
18    end
19  endcase
20 end

```

(3) 错误原因

状态机 AR_REQ_END 写错，下一个状态应该为IDLE。

(4) 修正效果

PC向后更新：

```

=====
Test begin!
-----
[ 2177 ns] Error!!!
reference: PC = 0x1c00f07c, wb_rf_wnum = 0x04, wb_rf_wdata = 0xbfffff000
mycpu   : PC = 0x1c000008, wb_rf_wnum = 0x0c, wb_rf_wdata = 0xffffffff
-----

```

- 错误2：data_ok和addr_ok实现有误

(1) 错误现象

如上，PC更新有误。

(2) 分析定位过程

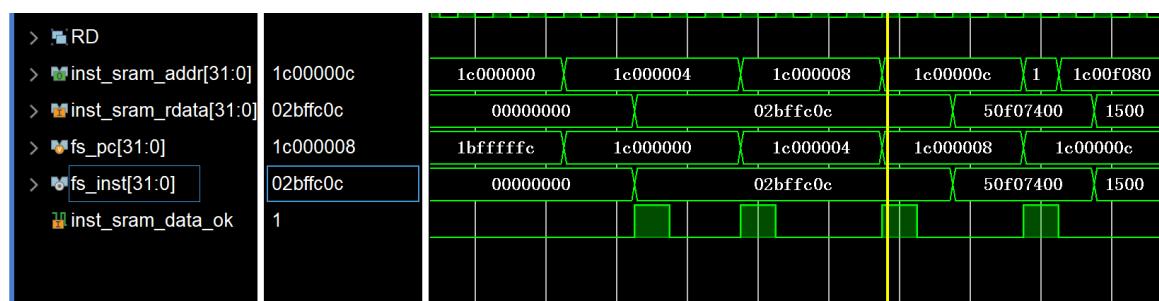
查看当前指令：

```

1
2 00000 <_start>:
3 nel_entry():
4 00000: 02bffc0c    addi.w $r12,$r0,-1(0xffff)
5 00004: 02bffc0c    addi.w $r12,$r0,-1(0xffff)
6 00008: 50f07400    b 61556(0xf074) # 1c00f07c <locate>
7 0000c: 1500000c    lu12i.w $r12,-524288(0x80000)

```

查看取指拿到的数据：



发现PC为1c000008时返回data_ok时取到的是上一条指令。查看波形：



发现第三个data_ok到来时（图中黄色光标）rddata正确，但是传递给cpu的数据错误。仔细阅读讲义，查看data_ok和addr_ok应该拉高的时机：

对于 `addr_ok` 信号来说，当对应的是读事务的时候，其对应 AXI 总线的 `already`。当对应的是写事务的时候，没有简单的对应信号，它的含义其实是 AXI 总线上的 `awready` 和 `wready` 都

已经或正在为 1。此处的对应关系有些复杂，读者在设计“类 SRAM-AXI”转接桥时需要关注。

对于 `data_ok` 信号来说，当对应的是 `读事务` 的时候，其对应 AXI 总线的 `rvalid`；当对应的 `写事务` 的时候，对应 AXI 总线的 `bvalid`。

起初我的实现如下：

```
1 assign inst_sram_addr_ok = ~arid[0] & ar_current_state[2]; //  
alid & already的下一拍  
2 assign inst_sram_data_ok = ~rid[0] & r_current_state[2]; // rvalid &  
rready的下一拍
```

如果按照讲义的思路应该实现为如下：

```
1 assign inst_sram_addr_ok = ~arid[0] & arvalid & already;  
2 assign inst_sram_data_ok = ~rid[0] & rvalid & rready;
```

然而这两种都不十分准确，原因是对于地址的握手，`arvalid & already` 同时拉高的时刻给出的地址就是有效的，此时应该采用讲义中的实现方式；而对于数据的握手，由于我们需要从 buffer 中读出数据，则 `data_ok` 应该在 `rvalid & rready` 握手完成的下一个时钟上升沿拉高，应该采取我原本的实现方式，故总的逻辑如下：

```
1 assign inst_sram_data_ok = ~rid[0] & r_current_state[2]; // rvalid &  
ady的下一拍  
2 assign inst_sram_addr_ok = ~arid[0] & arvalid & already;
```

(3) 错误原因

`data_ok` 和 `addr_ok` 的逻辑混乱。

(4) 修正效果

PC 往后更新。

```
[ 2227 ns] Error!!!
reference: PC = 0x1c00f07c, wb_rf_wnum = 0x04, wb_rf_wdata = 0xbfaaff000
mycpu     : PC = 0x1c00f07c, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x800000000
```

- 错误3：给出valid信号后变更地址

(1) 错误现象

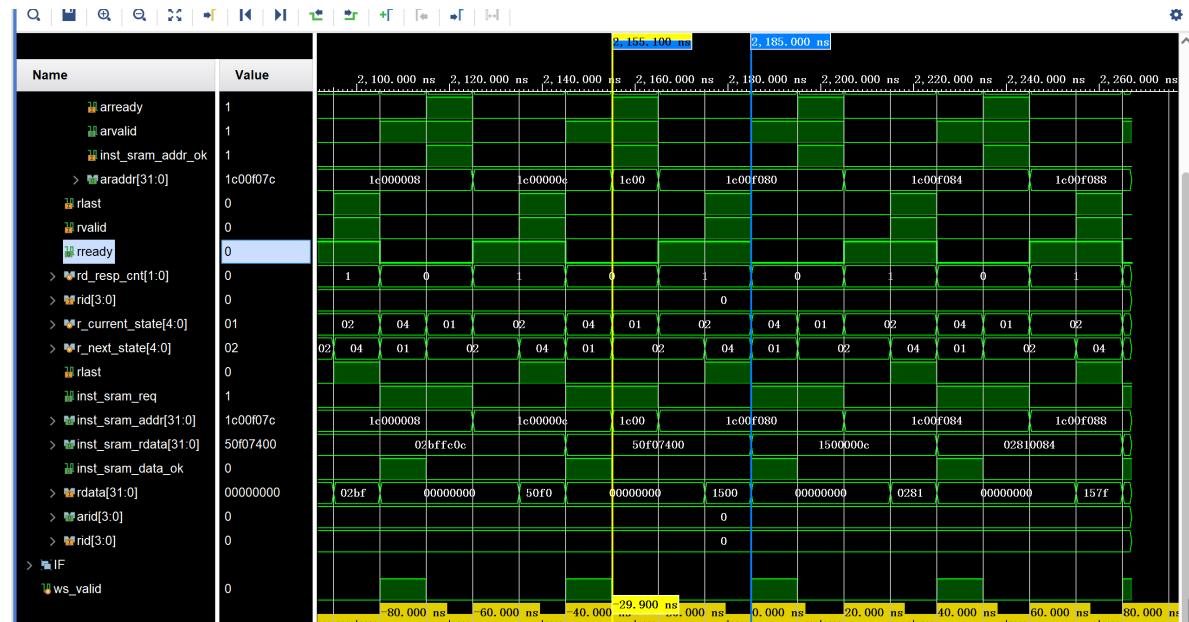
如上，写回数据和地址皆有误。

(2) 分析定位过程

查看当前pc对应的指令：

```
1 0f07c <locate>:
2 ate():
3 0f07c: 157f5fe4    lu12i.w $r4,-263425(0xbfaaff)
4 0f080: 02810084    addi.w $r4,$r4,64(0x40)
5 0f084: 157f5fe5    lu12i.w $r5,-263425(0xbfaaff)
```

查看当前pc实际取出的指令：



第一个光标指示处是地址握手成功的时间点，而第二个光标指示处是取出数据时间点，可见握手地址无误，但是取出的指令确实错误的。查看先前代码执行流：

```

1c000000 <_start>:
kernel_entry():
1c000000: 02bfffc0c addi.w $r12,$r0,-1(0xffff)
1c000004: 02bfffc0c addi.w $r12,$r0,-1(0xffff)
1c000008: 50f07400 b 61556(0xf074) # 1c00f07c <Locate>
1c00000c: 1500000c lu12i.w $r12,-524288(0x80000)
1c000010: 028005ad addi.w $r13,$r13,1(0x1)

```

猜测是取到了 1c00000c 中的指令。仔细阅读讲义：

5. 在 AXI Master 端的读请求、写请求、写数据通道上，如果 Master 输出的 valid 置为 1 的时候对应的 ready 是 0，那么在 ready 信号变为 1 之前，不允许 Master 变更该通道的所有输出信号。这一点与类 SRAM 总线不同，类 SRAM 总线允许中途更改请求。

再观察波形，得知在当前pc值为 1c00000c 时 arvalid 已经拉高一次，故导致读出的数据有误。则应该在判断出 fs_cancel 时将pf阻塞，直到等到data_ok，再发出新的读数请求，引入 pf_block 信号：

```

1 assign to_fs_valid      = pf_ready_go & ~pf_cancel & ~pf_block;
2
3 always @(posedge clk) begin
4     if(~resetn)
5         pf_block <= 1'b0;
6     else if(fs_cancel)
7         pf_block <= 1'b1;
8     else if(inst_sram_data_ok)
9         pf_block <= 1'b0;
10 end
11
12 assign inst_sram_req    = fs_allowin & resetn & ~br_stall &
~pt_block;

```

(3) 错误原因

忽略对于AXI总线，给出valid信号后不可随意变更地址，故一旦发出了valid信号且需要变更地址，无论是否收到add_ok，都需要阻塞，直到等到data_ok。

(4) 修正效果

PC往后更新：

```

[ 402000 ns] Test is running, debug_wb_pc = 0x1c00f0b4
[ 412000 ns] Test is running, debug_wb_pc = 0x1c00f0b4
[ 422000 ns] Test is running, debug_wb_pc = 0x1c00f0b4
[ 432000 ns] Test is running, debug_wb_pc = 0x1c00f0b4
[ 442000 ns] Test is running, debug_wb_pc = 0x1c00f0b4
[ 452000 ns] Test is running, debug_wb_pc = 0x1c00f0b4
[ 462000 ns] Test is running, debug_wb_pc = 0x1c00f0b4
> WARNING: [Simulator 45-99] Cannot open source file /wrk/2019.2/continuous/2019.11.06.2'

```

- 错误4：弄混写响应通道的主从端

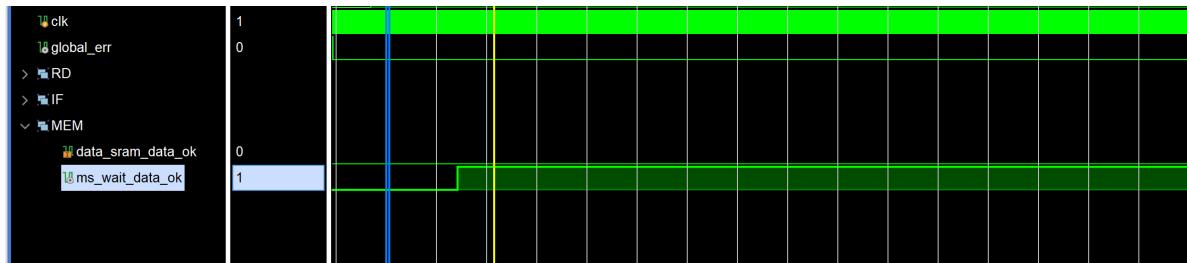
(1) 错误现象

PC卡死，查看当前指令：

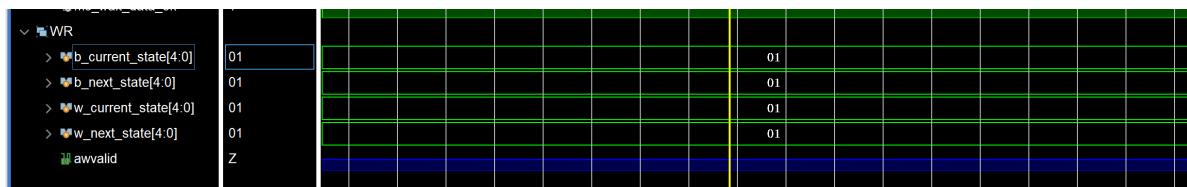
```
1 0f0b0: 02bffdef addi.w $r15,$r15,-1(0xffff)
2 0f0b4: 1400001a lu12i.w $r26,0
3 0f0b8: 2980008d st.w   $r13,$r4,0
4 0f0bc: 298000ae st.w   $r14,$r5,0
5 0f0c0: 298000cf st.w   $r15,$r6,0
```

(2) 分析定位过程

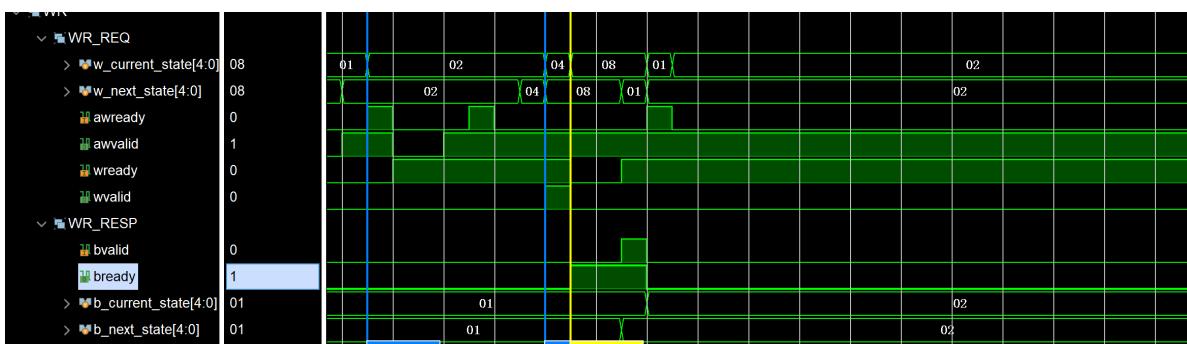
得知此时处于MEM阶段的是st类指令，猜测其握手有误导致流水线阻塞，查看信号：



始终未等到data_ok。查看相关信号：



发现awvalid未定义，意识到漏实现该信号。修改后仍卡在原处，再查看握手过程：



在两个蓝色光标处已经完成地址和数据的acknowledge，但是写响应通道在bready拉高后未及时进入下一个状态，查看实现逻辑：

```

1 //写响应通道状态机次态组合逻辑
2 always @(*) begin
3   case(b_current_state)
4     IDLE:begin
5       if(aresetn & bvalid)
6         b_next_state = B_START;
7       else
8         b_next_state = IDLE;
9     end
10    B_START:begin
11      if(bready & bvalid)
12        b_next_state = B_END;
13      else
14        b_next_state = B_START;
15    end
16    B_END:begin
17      b_next_state = IDLE;
18    end
19  endcase
20 end

```

(3) 错误原因

意识到写响应通道中CPU是接收端，应该在bready拉高时从IDLE进入下一个状态。

(4) 修正效果

PC往后更新：

Time [ns]	Action	debug_wb_pc	Data
1972000	Test is running	0x1c024690	...
1982000	Test is running	0x1c02484c	...
1992000	Test is running	0x1c024a08	...
2002000	Test is running	0x1c024bc4	...
2012000	Test is running	0x1c024d80	...
2022000	Test is running	0x1c024f3c	...
2032000	Test is running	0x1c0250fc	...
2042000	Test is running	0x1c0252b8	...
2052000	Test is running	0x1c025474	...
2062000	Test is running	0x1c025630	...
2072000	Test is running	0x1c0257ec	...
2082000	Test is running	0x1c0259a8	...
2092000	Test is running	0x1c025b64	...
2102000	Test is running	0x1c025d20	...
2112000	Test is running	0x1c025edc	...

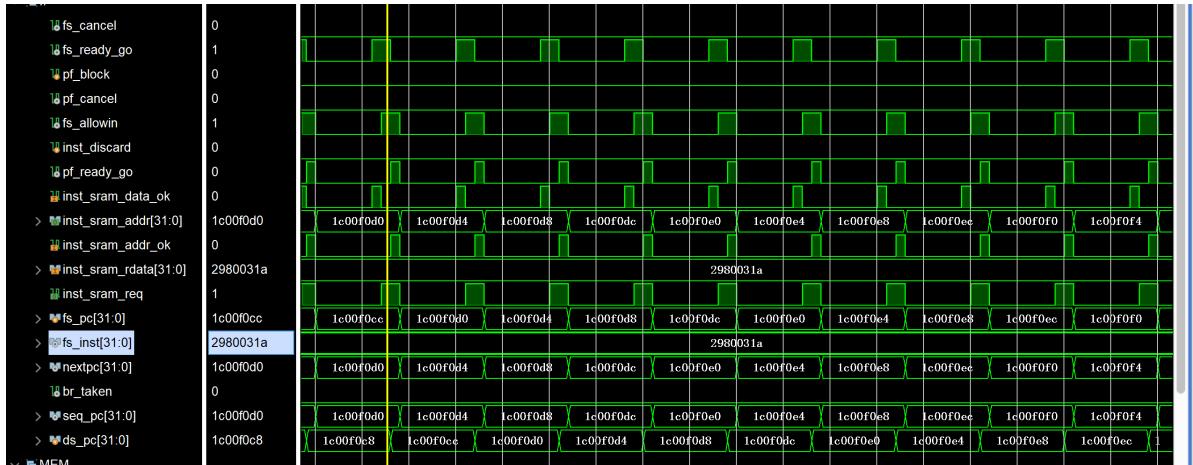
- 错误5：读请求通道未排除写请求

(1) 错误现象

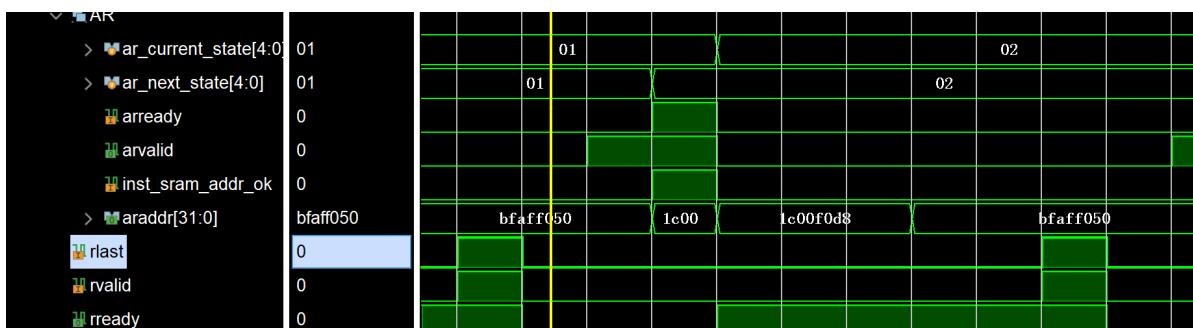
PC更新，但未打印通过测试点信息。

(2) 分析定位过程

查看指令，发现fs_inst一直存储一个值：



查看握手过程：



取指时误认为取数的握手信号是当前的握手信号，原本实现如下：

```
1 assign arid = {3'b0, data_sram_req}; // 数据RAM请求优先于指令RAM
2 assign araddr = data_sram_req ? data_sram_addr : inst_sram_addr;
3
```

应该在读channel中排除写请求：

```
1 assign arid = {3'b0, data_sram_req & ~data_sram_wr}; // 数据RAM请求优先于指令RAM
2 assign araddr = data_sram_req & ~data_sram_wr? data_sram_addr : inst_sram_addr;
```

(3) 错误原因

读请求通道未排除写请求。

(4) 修正效果

```
[1812000 ns] Test is running, debug_wb_pc = 0x1c00f0c0
[1822000 ns] Test is running, debug_wb_pc = 0x1c00f0c0
[1832000 ns] Test is running, debug_wb_pc = 0x1c00f0c0
[1842000 ns] Test is running, debug_wb_pc = 0x1c00f0c0
[1852000 ns] Test is running, debug_wb_pc = 0x1c00f0c0
[1862000 ns] Test is running, debug_wb_pc = 0x1c00f0c0
[1872000 ns] Test is running, debug_wb_pc = 0x1c00f0c0
[1882000 ns] Test is running, debug_wb_pc = 0x1c00f0c0
[1892000 ns] Test is running, debug_wb_pc = 0x1c00f0c0
[1902000 ns] Test is running, debug_wb_pc = 0x1c00f0c0
[1912000 ns] Test is running, debug_wb_pc = 0x1c00f0c0
run: Time (s): cpu = 00:00:06 ; elapsed = 00:00:07 . Memory (MB): peak = 1158.234 ; gain = 0.000
```

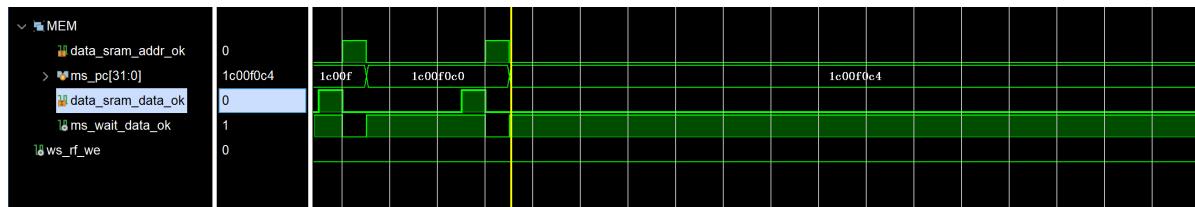
- 错误6：未考虑写地址和写数据同时握手成功的情况

(1) 错误现象

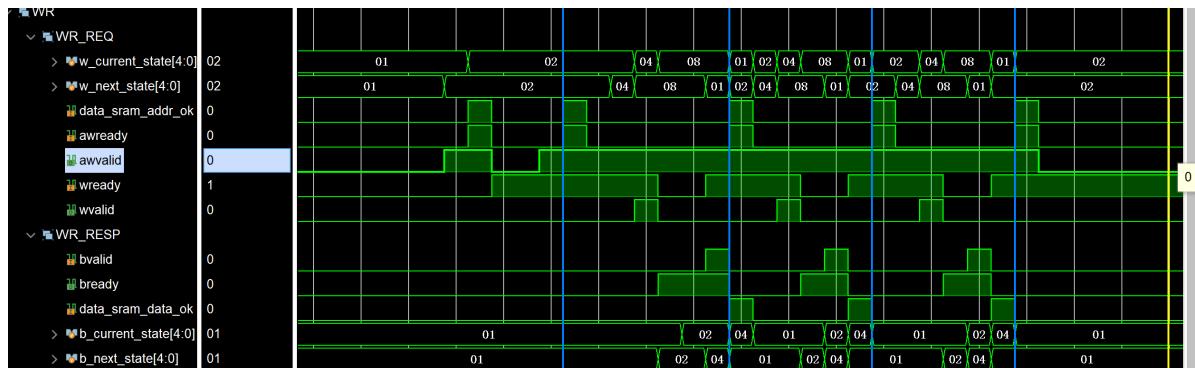
如上，修改后pc再度卡住。

(2) 分析定位过程

查看当前MEM阶段的状态：



查看当前握手情况：



发现四次addr_ok只等到了三次的data_ok。查看写请求状态机实现：

```

1 IDLE:begin
2     if(aresetn & awvalid)
3         w_next_state = AW_REQ_START;
4     else
5         w_next_state = IDLE;
6 end
7 AW_REQ_START:begin
8     if(awvalid & arready)
9         w_next_state = AW_REQ_END;
10    else
11        w_next_state = AW_REQ_START;
12    end

```

(3) 错误原因

意识到第四次请求在IDLE阶段就握手成功，而根据上述状态机其会先进入 AW_REQ_START，而该状态会继续等待 awvalid & arready，导致卡死。故应该修改如下：

```

1 //写请求&写数据通道状态机次态组合逻辑
2 always @(*) begin
3     case(w_current_state)
4         IDLE:begin
5             if(aresetn & awvalid & awready)
6                 w_next_state = AW_REQ_ACK;
7             else if(aresetn & awvalid)
8                 w_next_state = AW_REQ_START;
9             else
10                w_next_state = IDLE;
11        end
12        AW_REQ_START:begin
13            if(awvalid & awready)
14                w_next_state = AW_REQ_ACK;
15            else
16                w_next_state = AW_REQ_START;
17        end
18        AW_REQ_ACK:begin
19            if(wvalid & wready)
20                w_next_state = W_DATA_ACK;
21            else
22                w_next_state = AW_REQ_ACK;
23        end
24        W_DATA_ACK:
25            if(bvalid & bvalid)
26                w_next_state = IDLE;
27            else

```

```

28     w_next_state = W_DATA_ACK;
29   endcase
30 end

```

(4) 修正效果

PC向后更新：

```

=====
Test begin!
-----
[ 3177 ns] Error!!!
reference: PC = 0x1c00f0c8, wb_rf_wnum = 0x17, wb_rf_wdata = 0x00000000
mycpu    : PC = 0x1c02c788, wb_rf_wnum = 0x17, wb_rf_wdata = 0xxxxxxxxx
-----
$finish called at time : 3217 ns : File "D:/Desktop/cdp_ede_local/mycpu_env_15/soc_verify/soc_axi/testbench/mycp

```

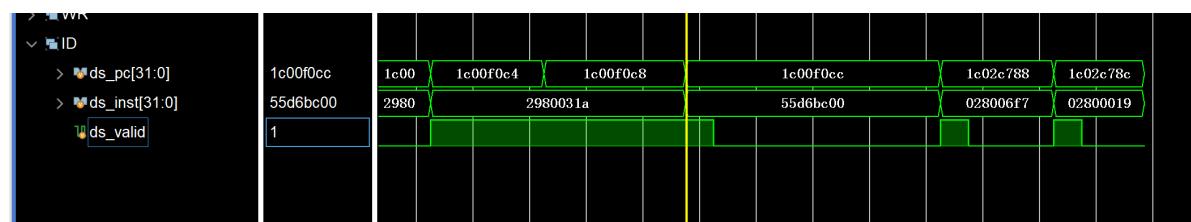
- 错误7：指令RAM的data_ok实现有误

(1) 错误现象

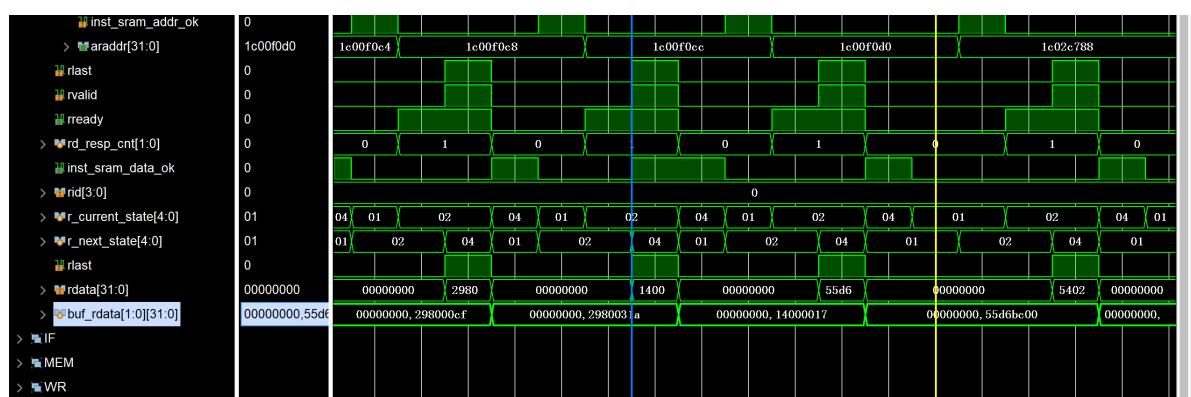
如上，写回数据未定义，且PC提前。

(2) 分析定位过程

查看ID阶段的指令和PC的对应情况：



发现出错PC处拿到了前一条指令。查看IF阶段的握手情况：



发现在rdata存入buffer前data_ok已经拉高。查看实现：

```
1 | ign inst_sram_data_ok = ~rid[0] & r_current_state[2] | ~bid & &  
| urrent_state[2]; // rvalid & rready的下一拍
```

前序修改data_sram_data_ok时忘记修改此处：

```
1 | assign inst_sram_data_ok = ~rid[0] & r_current_state[2] | ~bid &  
| lid & bready; // rvalid & rready的下一拍
```

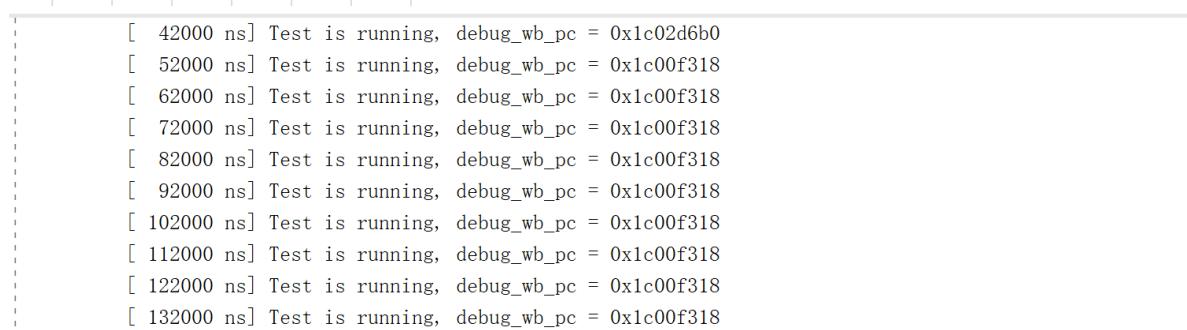
如果有写事务的话（对于指令ram无），要在bvalid & bready拉高时将data_ok拉高。

(3) 错误原因

inst_sram_data_ok实现有误。

(4) 修正效果

PC往后更新：



- 错误8：读数据和地址通道的计数器实现有误

(1) 错误现象

PC卡住。

(2) 分析定位过程

查看MEM阶段更新情况：



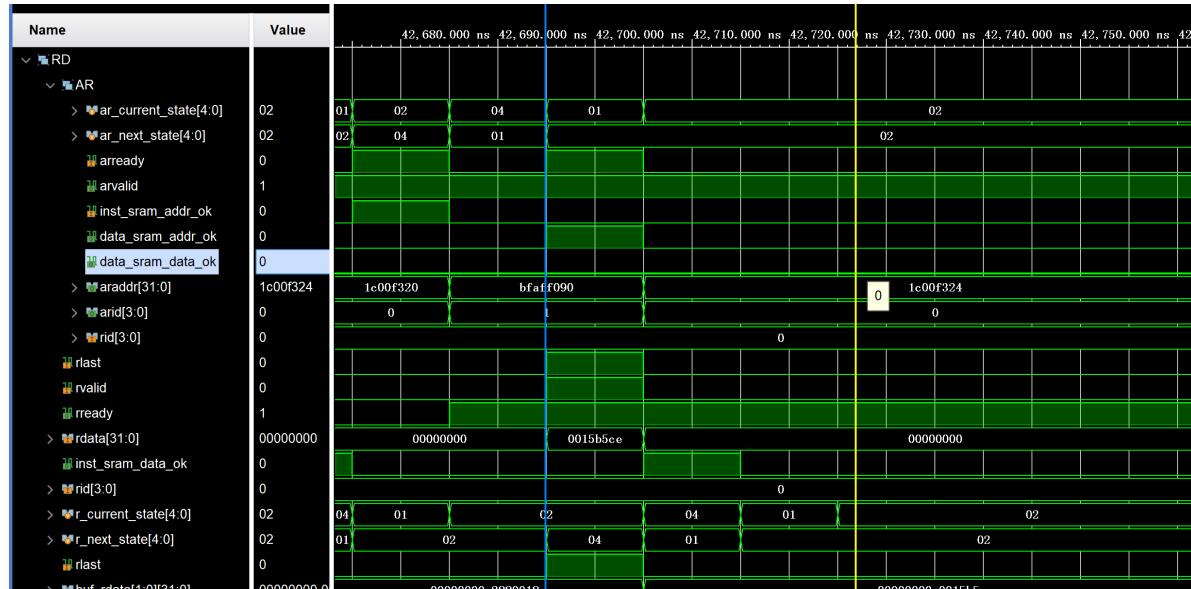
查看当前卡住的指令类型：

```

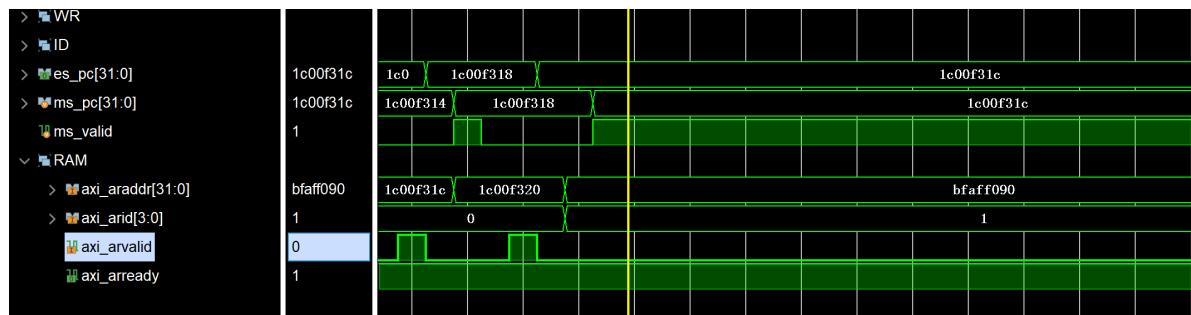
652 1c00f30c <idle_1s>:
653 idle_1s():
654 1c00f30c: 157f5fec lu12i.w $r12,-263425(0xbfaaff)
655 1c00f310: 0282418c addi.w $r12,$r12,144(0x90)
656 1c00f314: 1400016d lu12i.w $r13,11(0xb)
657 1c00f318: 02aaa9ad addi.w $r13,$r13,-1366(0xaa)
658 1c00f31c: 2880018e ld.w $r14,$r12,0
659 1c00f320: 0015b5ce xor $r14,$r14,$r13
660 1c00f324: 0040a5cf slli.w $r15,$r14,0x9
661 1c00f328: 028005ef addi.w $r15,$r15,1(0x1)
662
663 1c00f32c <idle_1s>

```

为ld指令，查看读相关信号的握手过程：



图中光标指示处地址已经握手成功，但始终未等到后续信号。查看RAM模块的实际交互情况：



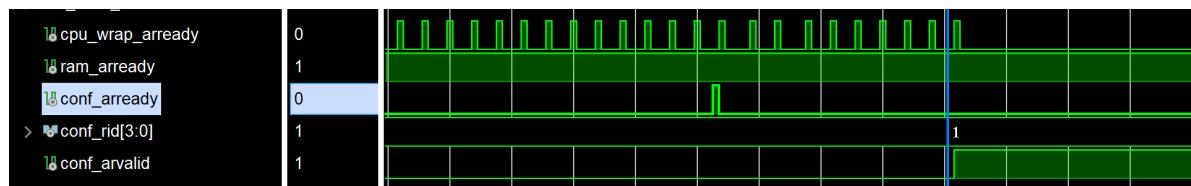
却有arvalid并未拉高。模块间连接情况如下：

```

1      -----
2      |          cpu          |
3      -----
4      inst|              | data
5      |          |          |
6      |
7      |          | 1 x 2 bridge  |
8      |          |          |
9      |          |          |
10     |          |          |
11    -----
12    | inst ram | | data ram| | confreg |
13    -----

```

询问助教得知在访问外设的时候bridge会将信号分流给confreg而非RAM，而当前指令使用的地址以 `bfaf` 打头，是访问外设，故应该检查其握手情况：



发现其`arready`未拉高。进入模块内部继续查看信号：

```

1 assign arready = ~busy & (!R_or_W | !awvalid);
2
3 always @(posedge aclk)
4 begin
5     if(~aresetn)
6         begin
7             R_or_W      <= 1'b0;
8             buf_id     <= 4'b0;
9             buf_addr   <= 32'b0;
10            buf_len    <= 8'b0;
11            buf_size   <= 3'b0;
12        end
13        else
14        if(ar_enter | aw_enter)
15        begin
16            R_or_W      <= ar_enter;
17            buf_id     <= ar_enter ? arid : awid ;
18            buf_addr   <= ar_enter ? araddr : awaddr ;
19            buf_len    <= ar_enter ? arlen : awlen ;
20            buf_size   <= ar_enter ? arsize : awsize ;
21        end
22    end
23
24 always @(posedge aclk)

```

```

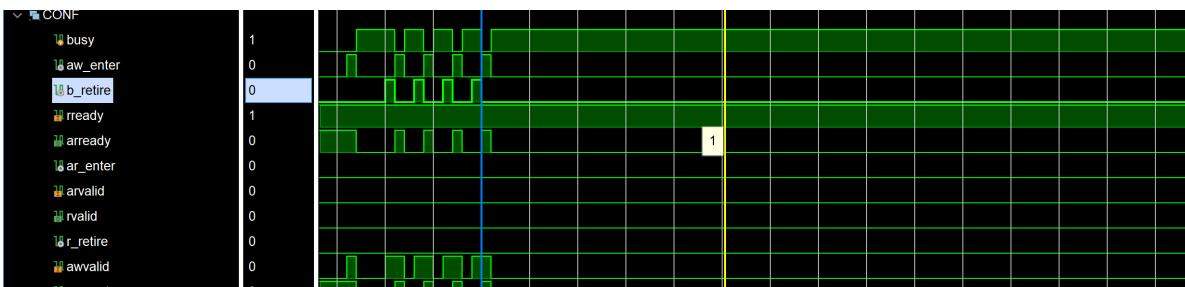
25 begin
26     if(~aresetn) busy <= 1'b0;
27     else if(ar_enter|aw_enter) busy <= 1'b1;
28     else if(r_retire|b_retire) busy <= 1'b0;
29 end

```

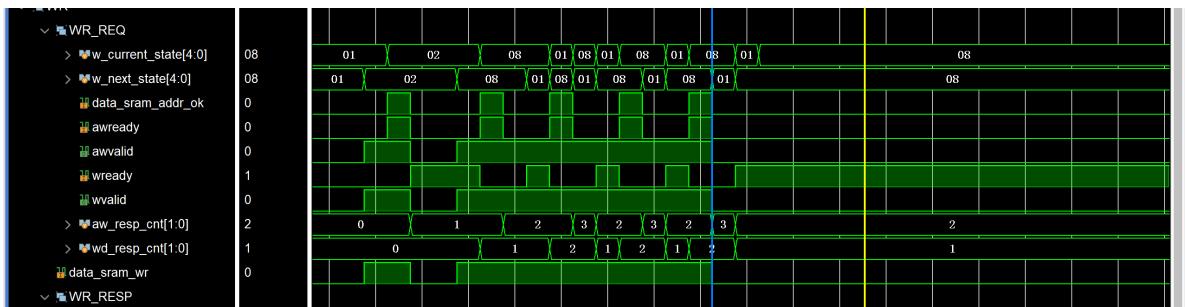
拉出相关信号：



发现confreg一直处于busy的状态。往前查看：



发现图中光标指示处发出了enter但未retire。查看此时写通道握手情况：



光标指示处显示未处理的响应不为0，导致写数据请求一直拉高，查看计数器实现：

```

1 always @(posedge aclk) begin
2     if(~aresetn) begin
3         aw_resp_cnt <= 2'b0;
4         wd_resp_cnt <= 2'b0;
5     end
6     else if(awvalid & awready)
7         aw_resp_cnt <= aw_resp_cnt + 1'b1;
8     else if(wvalid & wready)
9         wd_resp_cnt <= wd_resp_cnt + 1'b1;
10    else if(bvalid & bready) begin
11        aw_resp_cnt <= aw_resp_cnt - 1'b1;
12        wd_resp_cnt <= wd_resp_cnt - 1'b1;

```

```

13   end
14 end

```

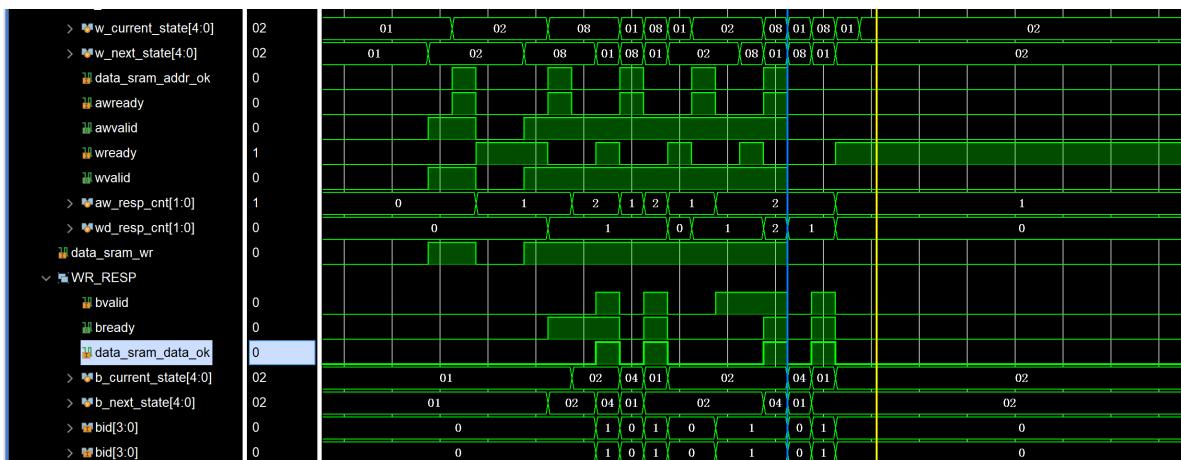
意识到未考虑写响应通道和写数据、地址通道握手同时成功的情形。故应该修改如下：

```

1 always @(posedge aclk) begin
2   if(~aresetn) begin
3     aw_resp_cnt <= 2'b0;
4   end
5   else if(awvalid & awready)
6     aw_resp_cnt <= aw_resp_cnt + {1'b0, ~(bvalid & bready)};
7   else if(bvalid & bready)
8     aw_resp_cnt <= aw_resp_cnt - 1'b1;
9 end
10
11 always @(posedge aclk) begin
12   if(~aresetn) begin
13     wd_resp_cnt <= 2'b0;
14   end
15   else if(wvalid & wready)
16     wd_resp_cnt <= wd_resp_cnt + {1'b0, ~(bvalid & bready)};
17   else if(bvalid & bready) begin
18     wd_resp_cnt <= wd_resp_cnt - 1'b1;
19   end
20 end

```

修改后计数器可正常计数，但仍缺少一次握手：



意识到在当前地址握手成功计数器>数据握手成功计数器时wvalid信号应该保持拉高。同理修改awvalid逻辑：

```

1 assign wvalid = inst_sram_wr | data_sram_wr | (wd_resp_cnt <
| resp_cnt);
2 assign awvalid = inst_sram_wr | data_sram_wr | (wd_resp_cnt>
| aw_resp_cnt);

```

不过更为简洁的是使用状态机状态：

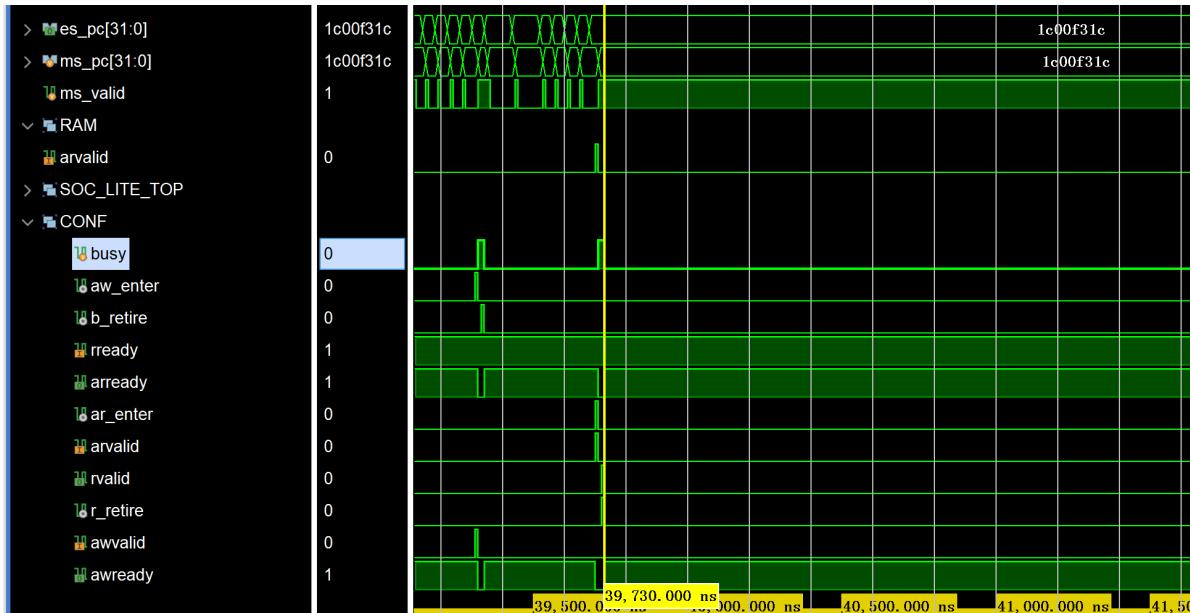
```
1 | assign wvalid = inst_sram_wr | data_sram_wr | w_current_state[1];
2 | assign awvalid = inst_sram_wr | data_sram_wr | w_current_state[2];
```

(3) 错误原因

写数据、地址通道的计数器实现有误。

(4) 修正效果

可以看到busy信号正常：



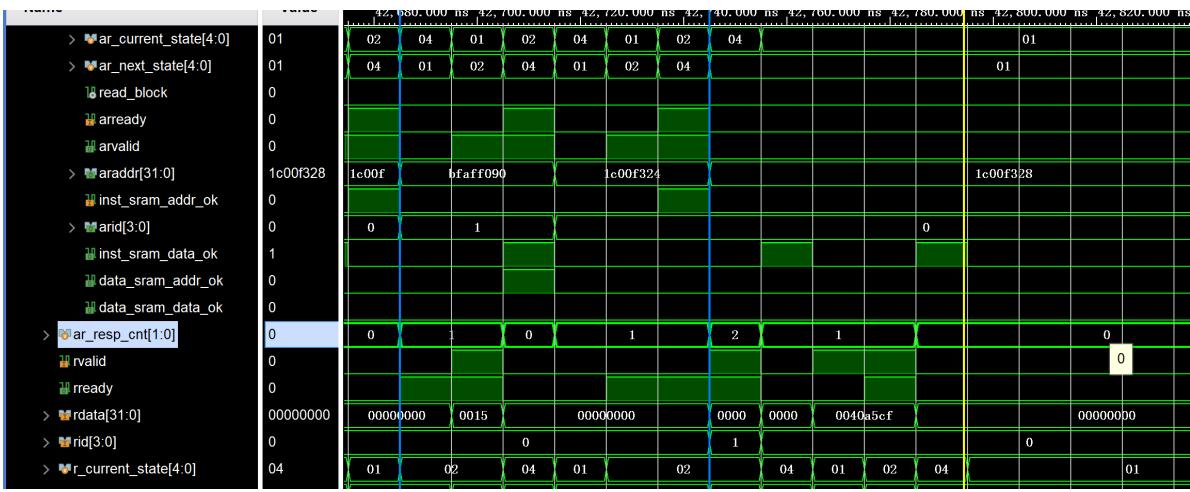
- 错误9：未用寄存器保存rid的值

(1) 错误现象

如上，即使busy信号拉低，pc仍卡死。

(2) 分析定位过程

第一个光标处完成地址握手，第二个光标处完成data握手，但data_ok未拉高：



查看data_ok实现：

```
1 | ign data_sram_data_ok = rid[0] & r_current_state[2] | bid[0] & bvalid
& bready;
```

(3) 错误原因

意识到 `r_current_state[2]` 会在握手成功的下一个周期拉高，故应该使用寄存器保存rid值：

```
1 | always @(posedge aclk) begin
2 |   if(~aresetn)
3 |     rid_r <= 4'b0;
4 |   else if(rvalid & rready)
5 |     rid_r <= rid;
6 | end
```

(4) 修正效果

PC往后更新：

```

[1102000 ns] Test is running, debug_wb_pc = 0x1c052234
[1112000 ns] Test is running, debug_wb_pc = 0x1c05261c
-----
[1119857 ns] Error!!!
      reference: PC = 0x1c06170c, wb_rf_wnum = 0x0a, wb_rf_wdata = 0xc822c7e8
      mycpu    : PC = 0x1c06170c, wb_rf_wnum = 0x0a, wb_rf_wdata = 0x00000000
-----
$finish called at time : 1119897 ns : File "D:/Desktop/cdp_edc_local/mycpu_env_15/soc_verify/soc_axi/test
run: Time (s): cpu = 00:00:13 ; elapsed = 00:00:14 . Memory (MB): peak = 1220.238 ; gain = 0.000

```

- 错误10：wdata漏接

(1) 错误现象

如图，写回数据有误。

(2) 分析定位过程

查看指令类型：

地址	操作码	参数	注释
85109	1c0616f4:	029fa16b	addi.w \$r11,\$r11,2024(0x7e8)
85110	1c0616f8:	299aa18d	st.w \$r13,\$r12,1704(0x6a8)
85111	1c0616fc:	02801184	addi.w \$r4,\$r12,4(0x4)
85112	1c061700:	02bfe185	addi.w \$r5,\$r12,-8(0xff8)
85113	1c061704:	299aa084	st.w \$r4,\$r4,1704(0x6a8)
85114	1c061708:	299aa0a5	st.w \$r5,\$r5,1704(0x6a8)
85115	1c06170c:	289aa18a	ld.w \$r10,\$r12,1704(0x6a8)
85116	1c061710:	289aa086	ld.w \$r6,\$r4,1704(0x6a8)
85117	1c061714:	289aa0a4	ld.w \$r4,\$r5,1704(0x6a8)
85118	1c061718:	289aa0a6	ld.w \$r6,\$r5,1704(0x6a8)
85119	1c06171c:	5c0fc4b	bne \$r10,\$r11,4044(0xfc) # 1c0626e8 <inst_error>
85120	1c061720:	14e2e70d	lu12i.w \$r13,464696(0x71738)

查看最近一次写入该地址时的情况（第一个光标）：



发现地址和数据握手都正常，但读出有误。再拉出更直接的信号：

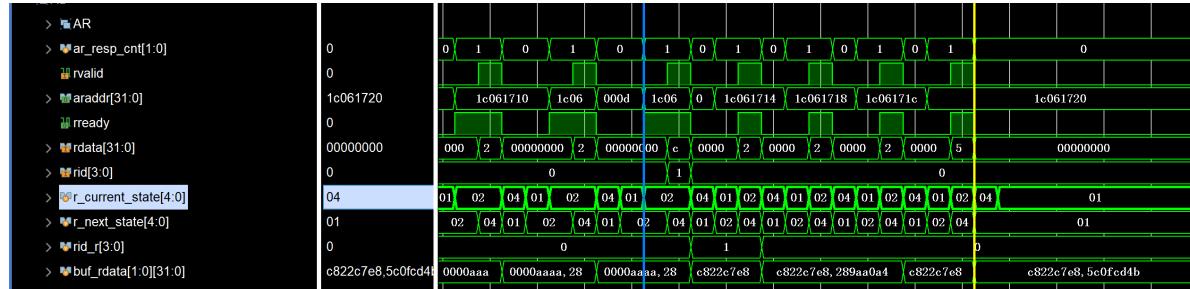


(3) 错误原因

发现wdata漏实现.....

(4) 修正效果

向后更新：



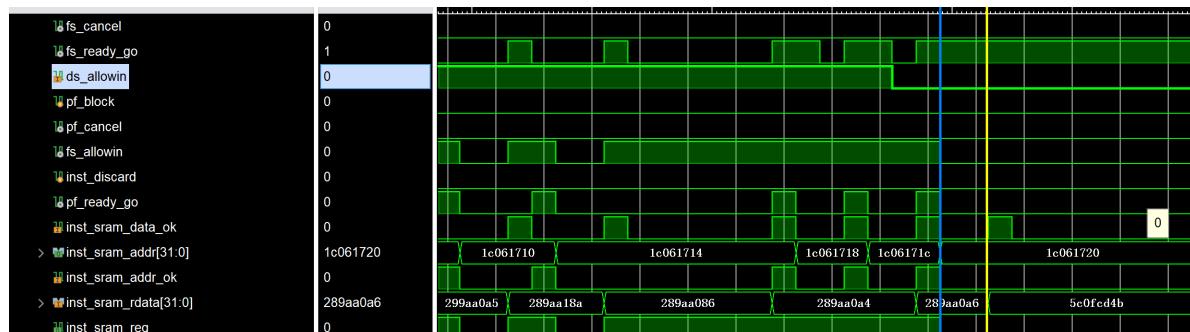
- 错误11：AXI输出端不来自寄存器Q端的后遗症

(1) 错误现象

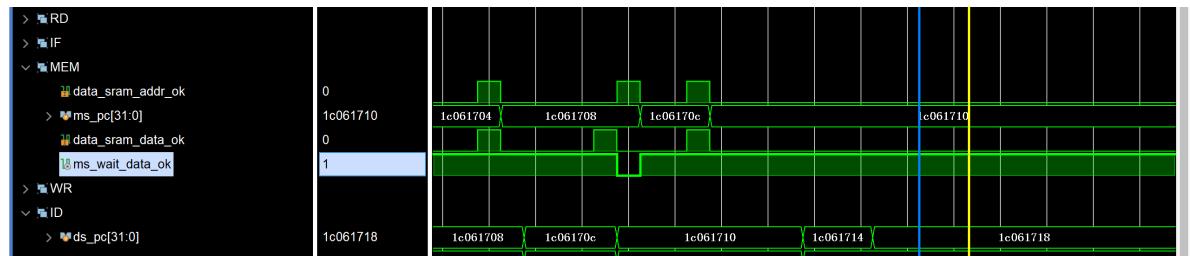
如图，在黄色光标处卡死。

(2) 分析定位过程

查看当前IF阶段：



已经取出了指令，但由于ds_allowin为0卡住。猜测是在MEM阶段出问题，查看相关信号：



未等到data_ok，当前为ld指令，查看读数据握手情况：

RD		AR										
>	ar_current_state[4:0]	01	02	04	01	02	04	01	02	04	01	02
>	ar_next_state[4:0]	02	04	01	02	04	01	02	04	01	02	04
	read_block	0										
	arready	0										
	arvalid	1										
>	araddr[31:0]	1c061714	000d3	1e061714	000d3b6c		1c061714			1e061718		
>	arid[3:0]	0	1	0	1			0				
	inst_sram_addr_ok	0										
	data_sram_addr_ok	0										
>	data_sram_addr[31:0]	000d3b6c	000d3		000d3b6c			000d3b6c			000d3b6c	
	data_sram_addr_ok	0										

如图，在蓝色光标指示处完成了数据握手，但由于前序 arvalid 拉高时对应的是另一个地址，导致实际上RAM中的握手情况如下：

RAM		RAM									
>	ram_wdata[31:0]	XXXXXXXX									XXXXXXXX
>	ram_araddr[31:0]	1c061714	000d3b68								1c061714
	ram_arvalid	0									
	ram_arready	1									
>	ram_rid[3:0]	1	0	1					0		
>	ram_rdata[31:0]	c822c7e8	289aa	c822c7e8					289aa0a4		
>	ram_arid[3:0]	0		1				0			

记得先前在讲义看到了这样的提醒（但我和我最后的倔强让我不想改……）：

- 除了可以置为常值的信号外，所有 AXI Master 端的输出直接来自触发器 Q 端。
- 为读数据预留缓存，从 rdata 端口上收到的数据先保存到 rdata 缓存。
- AXI 上最多支持几个“已完成读请求握手但数据尚未返回”的 rdata 缓存。
- 取指对应的 arid 恒为 0，load 对应的 arid 恒为 1。
- 控制 AXI 读写的状态机分为独立的 4 个状态机：读请求、读响应和写数据共用一个，写响应一个。
- 类 SRAM Slave 端接的输入信号不用锁存后再使用。
- 数据端发来的类 SRAM 总线的读请求优先级固定高于取指端发来的类 SRAM 总线的读请求。
- 类 SRAM Slave 端输出的 addr_ok 和 data_ok 信号若是来自组合逻辑，那么这个组合逻辑中不要引入 AXI 接口上的 valid 和 ready 信号。

lin 2022/10/16 23:37:06 X

为啥？（不管了先试试

第八条我也没采纳，已经坐等踩bug了。

(3) 错误原因

Master端的输出信号来自组合逻辑，导致握手时地址改变。

(4) 修正效果

！！！速通好几个测试点：

```

[6832000 ns] Test is running, debug_wb_pc = 0x1c031864
[6842000 ns] Test is running, debug_wb_pc = 0x1c031860
[6852000 ns] Test is running, debug_wb_pc = 0x1c031864
----[6852765 ns] Number 8'd51 Functional Test Point PASS!!!
[6862000 ns] Test is running, debug_wb_pc = 0x1c00802c
[6872000 ns] Test is running, debug_wb_pc = 0x1c00801c
----[6880355 ns] Number 8'd52 Functional Test Point PASS!!!
[6882000 ns] Test is running, debug_wb_pc = 0x1c064cd0
[6892000 ns] Test is running, debug_wb_pc = 0x1c064d1c
[6902000 ns] Test is running, debug_wb_pc = 0x1c064d1c
[6912000 ns] Test is running, debug_wb_pc = 0x1c064d1c
[6922000 ns] Test is running, debug_wb_pc = 0x1c064d1c
[6932000 ns] Test is running, debug_wb_pc = 0x1c064d1c
[6942000 ns] Test is running, debug_wb_pc = 0x1c064d1c

```

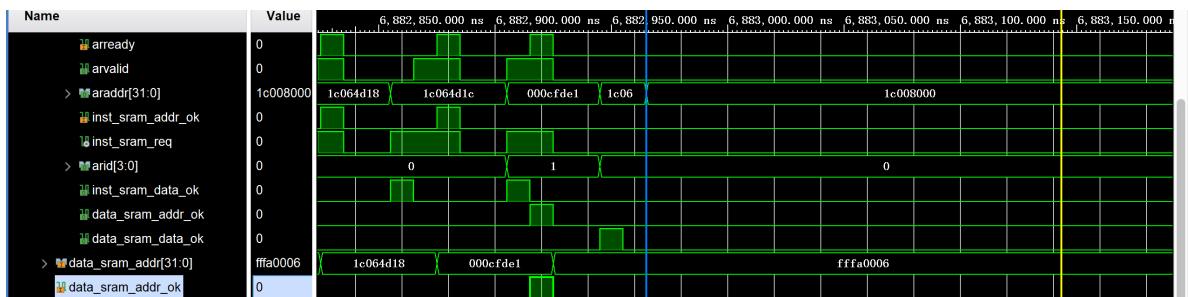
- 错误12：未处理好IF级cancel后的握手信号丢弃问题

(1) 错误现象

PC卡住（最后关头了怎么这么不争气……）。

(2) 分析定位过程

查看当前PC卡住的位置：



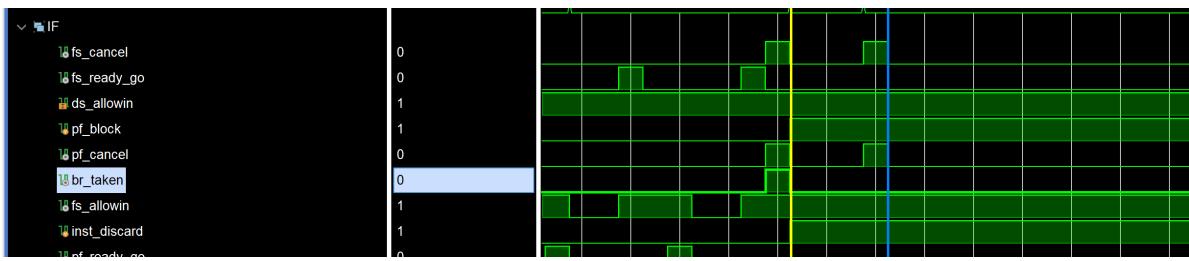
发现未发送请求。查看inst_sram_req：

```

1 assign inst_sram_req      = fs_allowin & resetn & ~br_stall &
2   block;\n
3
4 always @(posedge clk) begin
5   if(~resetn)
6     pf_block <= 1'b0;
7   else if(fs_cancel)
8     pf_block <= 1'b1;
9   else if(inst_sram_data_ok)
10    pf_block <= 1'b0;

```

再查看pf_block信号：



由于上一条恰好是br_taken指令，已经将pf_block拉高，如此要等来两个inst_sram_data_ok才可以解除block，故应该修改如下：

```

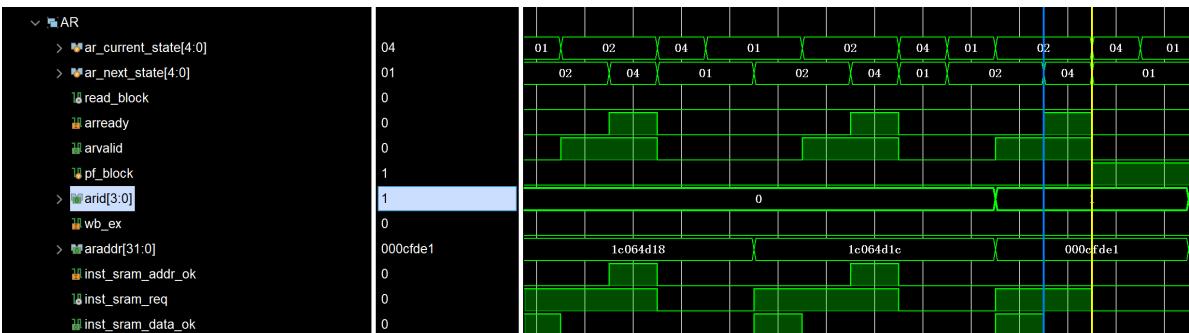
1 assign inst_sram_req      = fs_allowin & resetn & ~br_stall &
   block;\

2

3 always @(posedge clk) begin
4     if(~resetn)
5         pf_block <= 1'b0;
6     else if(fs_cancel & ~pf_block)
7         pf_block <= 1'b1;
8     else if(inst_sram_data_ok)
9         pf_block <= 1'b0;
10 end

```

但这样修改后仍会卡在原地，原因是上一次读请求时实际上进行的是读数据RAM的请求，此处无需再阻塞：



如图中光标所示，进行的实际上是数据RAM读请求的握手。

一种实现思路是确保一旦发出了req一定会响应（即在AXI转接桥中设立寄存器存储每次对指令RAM和数据RAM的读请求），但这种设计将使得有时对无用的数据还需进行一次取数（如上述我们已经判断出br_taken，需要取消上一次取指，此时恰好还未被AXI总线转接桥接收）；

另一种思路是**将转接桥的AIRD信号传回给CPU，让其接之判断上次无效的取指是否已经被接收**。这种方法不会再浪费若干周期进行无效的取指，故采用此。

```

1  always @(posedge clk) begin
2      if(~resetn)
3          pf_block <= 1'b0;
4      else if(pf_cancel & ~pf_block & ~axi_arid[0])
5          pf_block <= 1'b1;
6      else if(inst_sram_data_ok)
7          pf_block <= 1'b0;
8  end

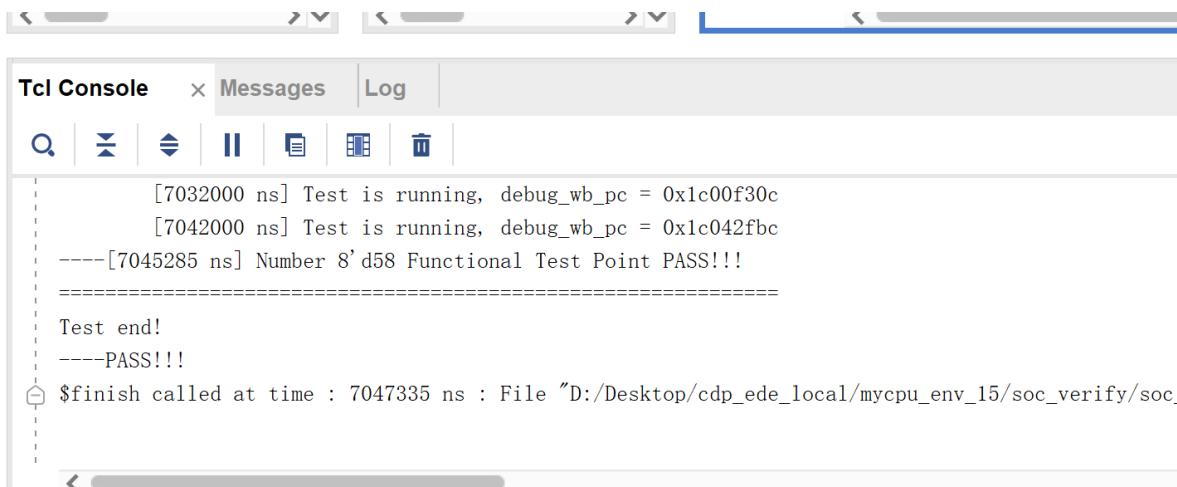
```

(3) 错误原因

未考虑IF级连续cancel的情况，且未考虑cancel时进行的是data RAM的取数请求的情况。

(4) 修正效果

妥了！



更意外之喜的是，时序居然变好了！！！！！



查看目前的最长路径：

Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clk
Path 1	0.534	0	368	cpu_resetn_reg_rep/C	u_a...AMB	19.056	0.419	18.637	20.000	cpu_clk_cl
Path 2	0.593	0	368	cpu_resetn_reg_rep/C	u_a...AMB	18.995	0.419	18.576	20.000	cpu_clk_cl
Path 3	1.030	0	368	cpu_resetn_reg_rep/C	u_a...AMB	18.554	0.419	18.135	20.000	cpu_clk_cl
Path 4	1.100	0	368	cpu_resetn_reg_rep/C	u_a...AMB	18.479	0.419	18.060	20.000	cpu_clk_cl
Path 5	1.474	0	368	cpu_resetn_reg_rep/C	u_a...AMB	18.115	0.419	17.696	20.000	cpu_clk_cl

主要的delay基本都是布局布线的延迟而非逻辑延迟 (满意(✿'◡`✿)！)

• (四) 实践任务16：完成 AXI 随机延迟验证

- 错误1：地址重复握手

(1) 错误现象

将delay低16位设置为 ffff (短延迟)，console报错如下：

```

Tcl Console x Messages Log
[11210905 ns] Number 8'd49 Functional Test Point PASS!!!
[11212000 ns] Test is running, debug_wb_pc = 0x1c00f328
[11222000 ns] Test is running, debug_wb_pc = 0x1c008574
-----
[11227727 ns] Error!!!
reference: PC = 0x1c01fb28, wb_rf_wnum = 0x19, wb_rf_wdata = 0x00000003
mycpu : PC = 0x1c01fb2c, wb_rf_wnum = 0x19, wb_rf_wdata = 0x00000003
-----
$finish called at time : 11227767 ns : File "D:/Desktop/cdp_edc_local/mycpu_env_16/soc_verify/soc_axi/testbench/mycpu_tb
run: Time (s): cpu = 00:02:35 ; elapsed = 00:02:28 . Memory (MB): peak = 2917.184 ; gain = 0.000

```

(2) 分析定位过程

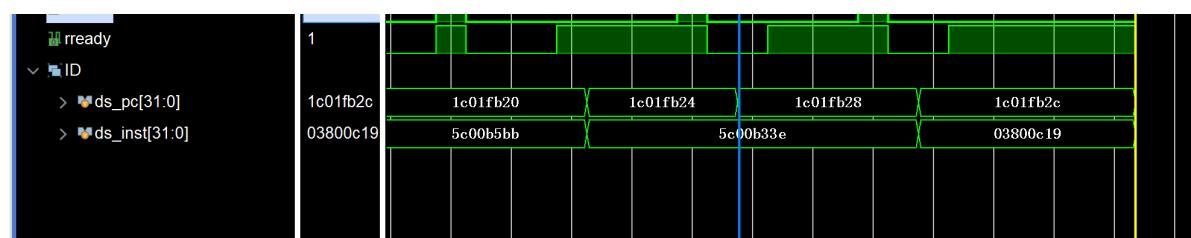
如上得知PC有误。查看当前指令：

```

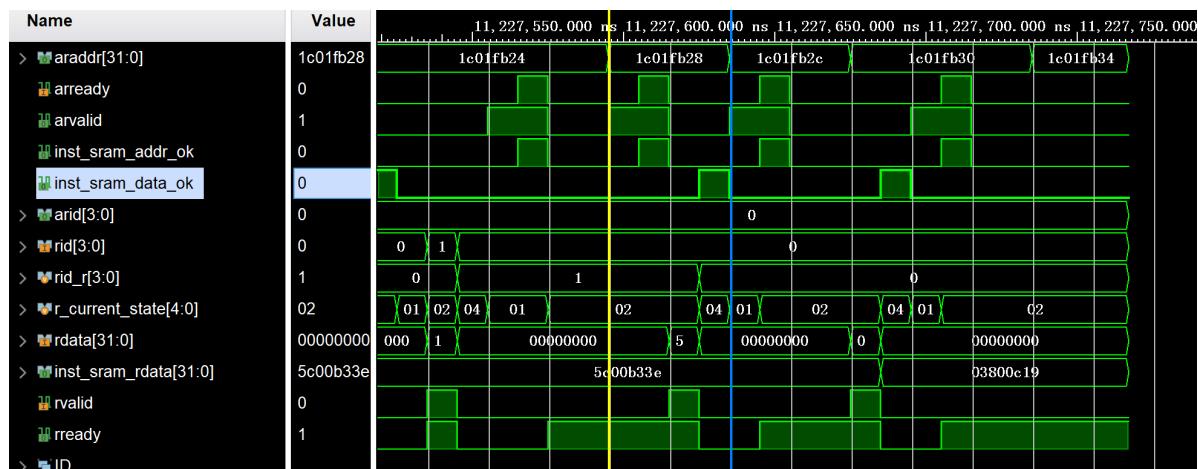
1 1fb20: 5c00b5bb bne $r13,$r27,180(0xb4) # 1c01fb24 <inst_error>
2 1fb24: 5c00b33e bne $r25,$r30,176(0xb0) # 1c01fb24 <inst_error>
3 1fb28: 03800c19 ori $r25,$r0,0x3
4 1fb2c: 29800279 st.w   $r25,$r19,0
5 1fb30: 03800c19 ori $r25,$r0,0x3
6 1fb34: 1c00001b pcaddu12i $r27,0
7 1fb38: 0280537b addi.w $r27,$r27,20(0x14)

```

查看当前ID指令和PC对应关系：



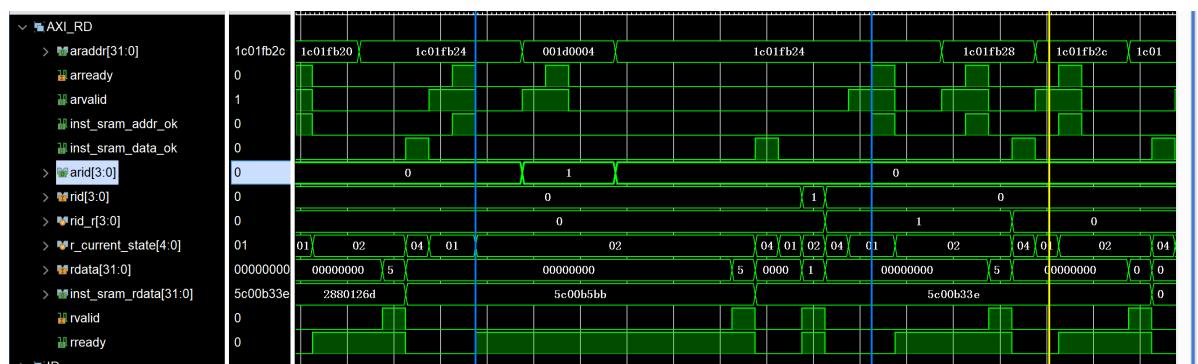
在光标指示处开始错位。查看对应地址的握手情况：



发现还未获得data_ok时拿到的还是前一条指令，再往前追溯：



发现同一条指令完成了两次地址握手：



如图，由于第一次请求地址握手成功后进行数据RAM的地址握手，但由于上次数据握手未等到data_ok所以当前请求的地址还是原来的地址，并且再度进行了握手。

一种思路是**每次都等到数据握手也结束后才发出下一个读请求**，但此种方式将使得延迟更长（对指令RAM和数据RAM的读请求不能并行）；

另一种思路是**分别给指令读地址通道和数据读地址通道维护一个ack的标志位，标识当前的握手是否已经完成**，等到数据握手完成后再拉低，避免重复请求，注意到还可以使用此标志位代替原本的读请求计数器；

还有一种思路是从IF级和MEM级入手，**当收到了addr_ok信号后把req拉低，等到data_ok信号到来后再允许req拉高**，第二种设计方法和第三种设计方法效果一致，此次设计采取第三种设计方法：

```

1 // 判断当前地址是否已经握手成功，若成功则拉低req，避免重复申请
2 always @(posedge clk) begin
3     if(~resetn)
4         inst_sram_addr_ack <= 1'b0;
5     else if(pf_ready_go)
6         inst_sram_addr_ack <= 1'b1;
7     else if(inst_sram_data_ok)
8         inst_sram_addr_ack <= 1'b0;
9 end
10 assign inst_sram_req      = fs_allowin & resetn & ~br_stall &
    block & ~inst_sram_addr_ack;

```

(3) 错误原因

由于延迟，出现了地址重复握手的情况。

(4) 修正效果

修改后如下：

```

[14382000 ns] Test is running, debug_wb_pc = 0x1c050b60
[14392000 ns] Test is running, debug_wb_pc = 0x1c050b60
[14402000 ns] Test is running, debug_wb_pc = 0x1c050b60
[14412000 ns] Test is running, debug_wb_pc = 0x1c050b60
[14422000 ns] Test is running, debug_wb_pc = 0x1c050b60
[14432000 ns] Test is running, debug_wb_pc = 0x1c050b60
[14442000 ns] Test is running, debug_wb_pc = 0x1c050b60
[14452000 ns] Test is running, debug_wb_pc = 0x1c050b60
[14462000 ns] Test is running, debug_wb_pc = 0x1c050b60
[14472000 ns] Test is running, debug_wb_pc = 0x1c050b60
[14482000 ns] Test is running, debug_wb_pc = 0x1c050b60
[14492000 ns] Test is running, debug_wb_pc = 0x1c050b60

```

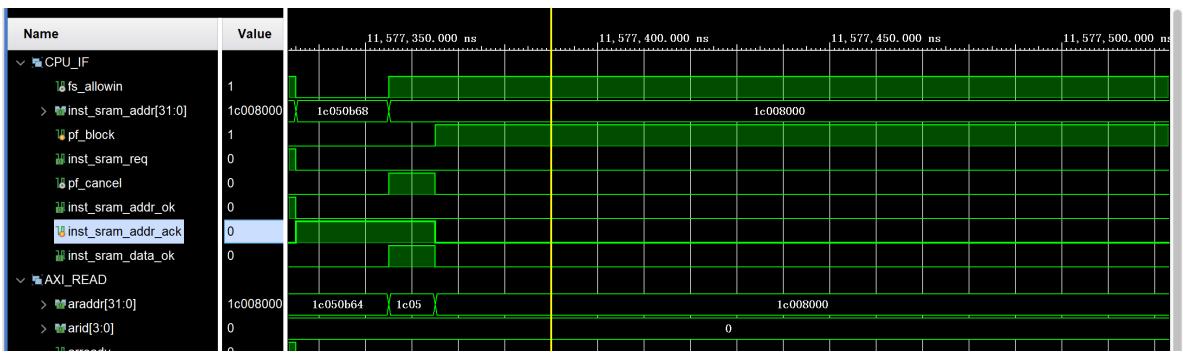
- 错误2：预取指级阻塞处理不当

(1) 错误现象

pc卡死。

(2) 分析定位过程

查看当前握手情况：



得知此时由于pf_block拉高无法进行下一次请求，原本pf_block实现如下：

```

1 always @(posedge clk) begin
2     if(~resetn)
3         pf_block <= 1'b0;
4     else if(pf_cancel & ~pf_block & ~axi_arid[0])
5         pf_block <= 1'b1;
6     else if(inst_sram_data_ok)
7         pf_block <= 1'b0;
8 end

```

意识到当前data_ok恰好已经拉高，无需block，同时注意到data_ok拉高的当拍就可以发出req，而无需等到下一拍，修改addr_ack信号如下：

```

1 // 判断当前地址是否已经握手成功，若成功则拉低req，避免重复申请
2 always @(posedge clk) begin
3     if(~resetn)
4         inst_sram_addr_ack_r <= 1'b0;
5     else if(pf_ready_go)
6         inst_sram_addr_ack_r <= 1'b1;
7     else if(inst_sram_data_ok)
8         inst_sram_addr_ack_r <= 1'b0;
9 end
10 assign inst_sram_addr_ack = inst_sram_addr_ack_r &
~inst_sram_data_ok;

```

这样设计可以使得请求发送更紧凑。

(3) 错误原因

预取指时阻塞信号处理不当。

(4) 修正效果

通过随机种子低位为 16'hFFFF (短延时) 的测试：

◀ ▶ ⏪ ⏩ ⏴ ⏵ ⏷ ⏸ ⏹ ⏺

```
[12772000 ns] Test is running, debug_wb_pc = 0x1c00f070
----[12779685 ns] Number 8'd57 Functional Test Point PASS!!!
    [12782000 ns] Test is running, debug_wb_pc = 0x1c00f34c
    [12792000 ns] Test is running, debug_wb_pc = 0x1c042ebc
    [12802000 ns] Test is running, debug_wb_pc = 0x1c043050
----[12804615 ns] Number 8'd58 Functional Test Point PASS!!!
=====
Test end!
----PASS!!!
$finish called at time : 12808155 ns : File "D:/Desktop/cdp_edc_local/mycpu_env_16/soc_verify/soc_
run: Time (s): cpu = 00:02:51 ; elapsed = 00:02:47 . Memory (MB): peak = 968.375 ; gain = 0.000
```

- 错误3：指令buffer有效位清零时机不对

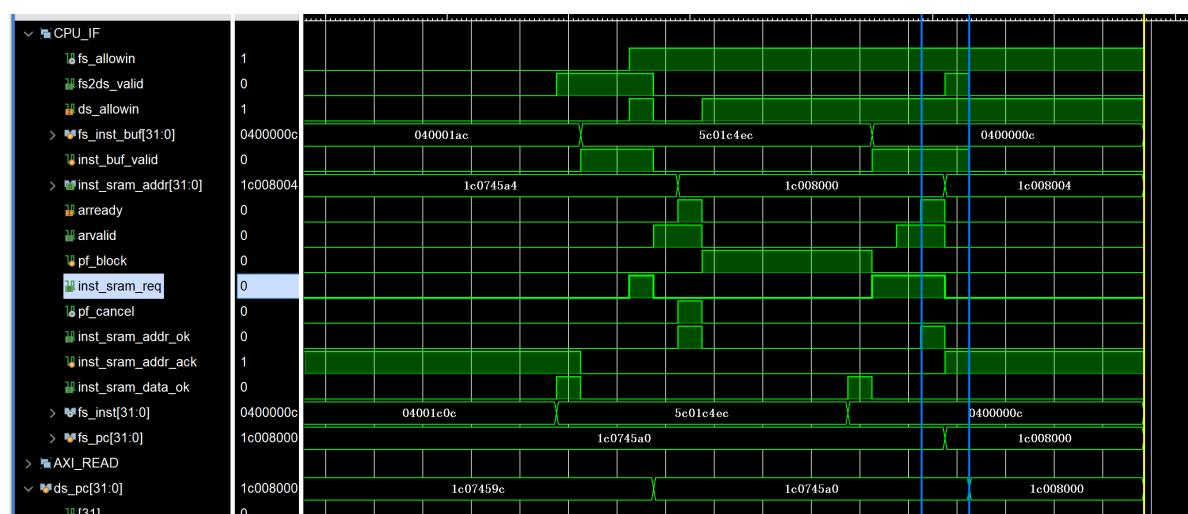
(1) 错误现象

再修改随机种子为 16h'FF00 (长延时) :

```
----[19145425 ns] Number 8'd51 Functional Test Point PASS!!!
    [19152000 ns] Test is running, debug_wb_pc = 0x227f9789
-----
[19158897 ns] Error!!!
    reference: PC = 0x1c008000, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x001d0000
    mycpu   : PC = 0x1c008000, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x00000008
-----
$finish called at time : 19158937 ns : File "D:/Desktop/cdp_edc_local/mycpu_env_16/soc_verify/soc_axi/testbench/
run: Time (s): cpu = 00:02:53 ; elapsed = 00:02:59 . Memory (MB): peak = 968.375 ; gain = 0.000
```

(2) 分析定位过程

查看当前IF级情况：



由于buffer中的数据有效信号在一拍才被拉低，当前buffer中存的还是上一条指令，导致出错。意识到此前处理buffer有误，以前将buffer有效位清零的条件是取到的指令可流入ID级（居然能狗住这么久……），应该修改为：

```
1 // 设置寄存器，暂存指令，并用valid信号表示其内指令是否有效
```

```

2 always @(posedge clk) begin
3   if(~resetn) begin
4     fs_inst_buf <= 32'b0;
5     inst_buf_valid <= 1'b0;
6   end
7   else if(to_fs_valid & fs_allowin) // 缓存已经流入IF级
8     inst_buf_valid <= 1'b0;
9   else if(fs_cancel)           // IF取消后需要清空当前buffer
10    inst_buf_valid <= 1'b0;
11   else if(~inst_buf_valid & inst_sram_data_ok & ~inst_discard)
12     n
13     fs_inst_buf <= fs_inst;
14     inst_buf_valid <= 1'b1;
15   end
16 end

```

(3) 错误原因

指令buffer有效位清零的时机不对。

(4) 修正效果

通过当前测试点：

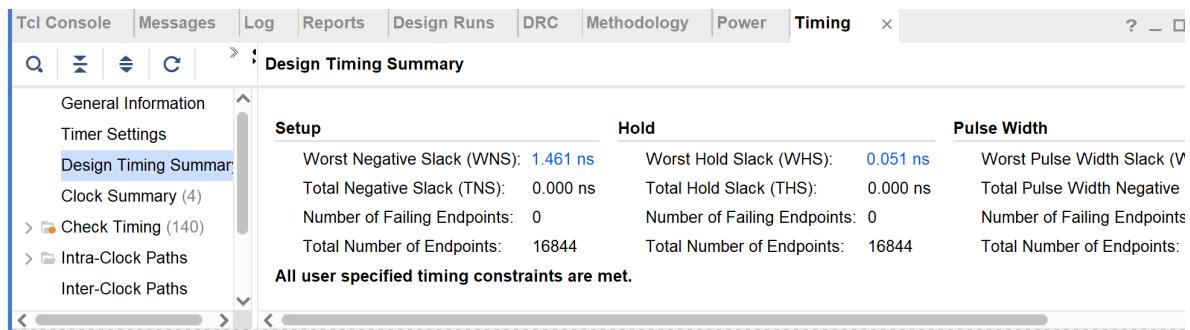


```

----[19707415 ns] Number 8'd58 Functional Test Point PASS!!!
[19712000 ns] Test is running, debug_wb_pc = 0x1c00f2ac
=====
Test end!
----PASS!!!
$finish called at time : 19713795 ns : File "D:/Desktop/cdp_ebe_local/mycpu_env_16/soc_verify/soc_axi/testbench/mycpu_tb.v" Li
run: Time (s): cpu = 00:03:20 ; elapsed = 00:03:18 . Memory (MB): peak = 1147.082 ; gain = 0.000
launch_runs impl_1 -to_step write_bitstream -jobs 6
[Wed Oct 26 15:13:37 2022] Launched synth_1...

```

照例看看布局布线后的时序报告（自我陶醉）：



Setup			Hold			Pulse Width		
Worst Negative Slack (WNS):	1.461 ns		Worst Hold Slack (WHS):	0.051 ns		Worst Pulse Width Slack (WPWS):		
Total Negative Slack (TNS):	0.000 ns		Total Hold Slack (THS):	0.000 ns		Total Pulse Width Negative (TPWN):		
Number of Failing Endpoints:	0		Number of Failing Endpoints:	0		Number of Failing Endpoints:		
Total Number of Endpoints:	16844		Total Number of Endpoints:	16844		Total Number of Endpoints:		

All user specified timing constraints are met.

上板测试也可通过。

四.实验总结：几个印象深刻debug的痛点

- exp14:

- 预取指阶段遇到非顺序取指的时候一定要考虑好各种信号的优先级；
 - 前一拍的例外信号>当前拍的例外信号>前一拍的中断返回信号>当前拍的中断返回信号>前一拍的分支跳转信号>当前拍的分支跳转信号
 - 例外or中断返回时记得处理子模块的复位信号（如alu的reset）；
 - 取指buffer的有效信号处理要多加小心。
- exp15：
 - 寄存器大法好！不知道什么时候会变，就一股脑存下来！**听讲义的话，否则会变得不幸** 😞（但不过我居然没遭受addr_ok和data_ok跟valid、ready信号挂钩的反击）；
 - 注意写地址和写数据同时握手成功的情况；
 - exp16：
 - 注意避免地址重复握手的情况！