实验 3 报告

学号：2017K8009922026
姓名：康齐瀚
箱子号：63

一、实验任务（10%）

调试提供的基于 mips 指令集的五级流水线 CPU 代码，找出一共 7 处 BUG 并修改，要求最终能仿真通过和 golden trace 的对比测试以及上板测试

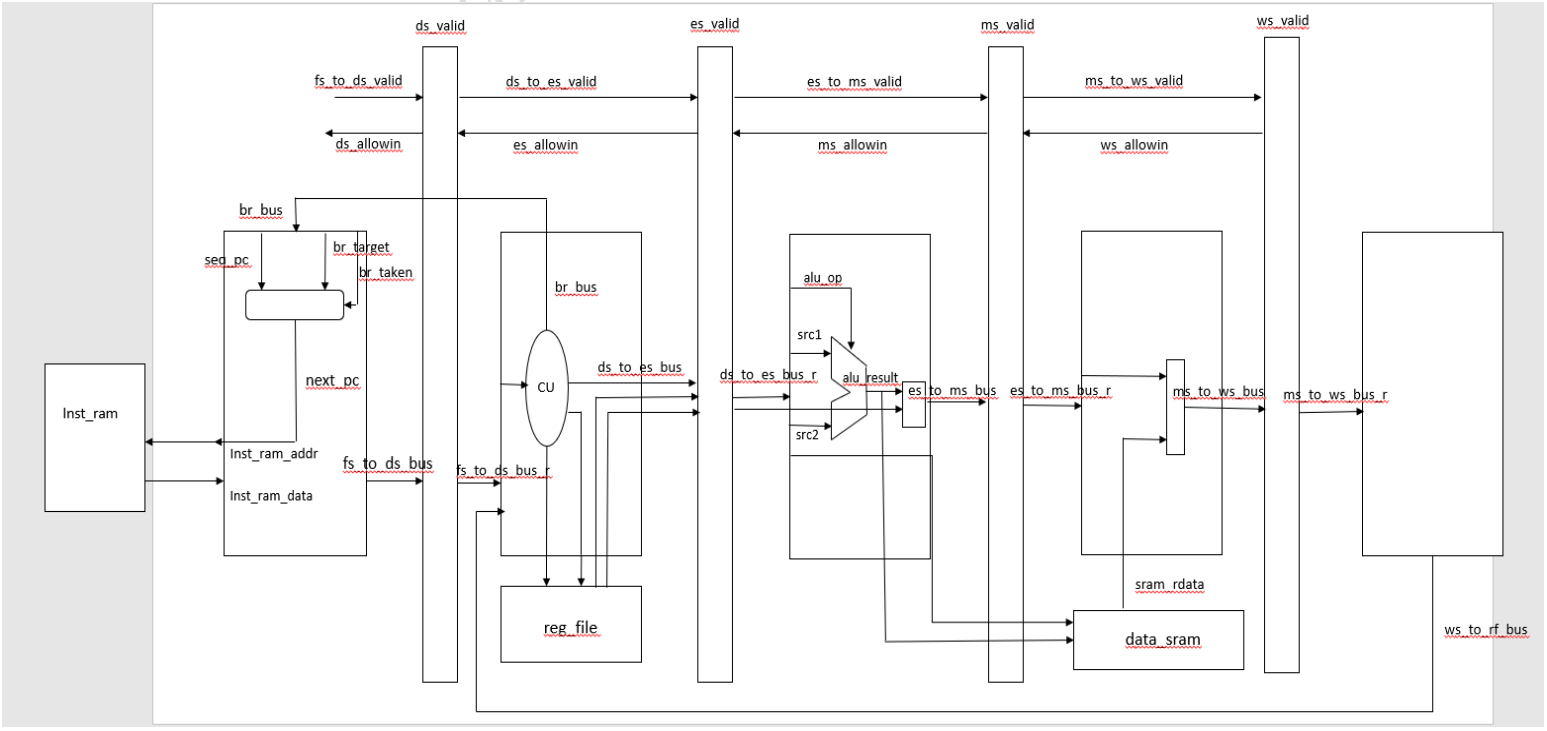
二、实验设计（40%）

（一）总体设计思路

本次设计总体上设计了一个具有基础功能的 MIPS 5 级流水线 CPU，总体上看 CPU 被划分为了 IF，ID，EXE，MEM, WB 五个阶段，分别完成取指令，译码指令并取操作数，执行，访存，写回寄存器堆等任务，每两个阶段之间使用流水线寄存器分隔开，EXE 阶段中调用 ALU 模块完成算术逻辑运算。ID 模块中调用寄存器堆模块读取数据。

流水线寄存器在时钟上升沿被赋值，获得的值在当前周期就能被下一阶段的相关信号使用，从而完成了指令在五个阶段中的流水。

结构设计图如下所示：



（二）重要模块 1 设计：IF 模块

1、工作原理

由于本次实验采用的是同步 RAM，发出读 RAM 指令后要等到下一拍才会有数据返回，因此在 PC 完成更新后就立刻对 inst_ram 发出读指令并给出读地址，然后将传回的指令通过流水线寄存器传递到下一流水级。

2、接口定义

	方向	位宽	功能描述
clk	IN	1	时钟信号
reset	OUT	1	控制复位
ds_allowin	IN	1	流水线传递控制信号
br_bus	IN	33	更新 PC
fs_to_ds_valid	OUT	1	流水线传递的控制信号
fs_to_ds_bus	OUT	64	流水线数据信息传递
Inst_sram_en	OUT	1	指令 RAM 的读使能信号
Inst_sram_wen	OUT	32	指令 RAM 的写使能信号
Inst_sram_addr	OUT	32	指令 RAM 的读地址
Inst_sram_wdata	OUT	32	指令 RAM 的写数据
Inst_sram_rdata	IN	32	指令 RAM 的读数据

3、功能描述

在 IF 模块内部，采用流水线传递控制信号的标准模板产生传递到下一级的控制信号 fs_to_ds_valid。通过将 nextPC 作为 inst_sram 的读地址来获取对应地址的指令。nextPC 的产生来源于本次 seqPC 和跳转 PC 的选择，前者将寄存器中保存的 fs_pc 加 4 即可，后者通过解析由 ID 模块传回的 br_bus 信号的 br_taken 和 br_Target 信号获得。同时将收到的指令信号 inst_sram_rdata 和当时的 PC 值通过 fs_to_ds_bus 信号传递到下一流水级

重要模块 2 设计：ID 模块

1、 工作原理

ID 模块对来自于 IF 模块的信号进行解析，尤其是对 fs_inst 信号解析，从而得到该条指令的所有信息，同时 ID 模块要完成对寄存器堆的读操作，由于寄存器堆的读操作是由组合逻辑完成的，译码指令和读寄存器堆都能完全在该流水级完成。

2、 接口定义

	方向	位宽	功能描述
clk	IN	1	时钟信号
reset	OUT	1	控制复位
es_allowin	IN	1	流水线传递控制信号
ds_allowin	OUT	1	流水线传递控制信号
fs_to_ds_valid	IN	1	流水线传递的控制信号
fs_to_ds_bus	IN	64	流水线数据信息传递
ds_to_es_valid	OUT	1	流水线传递控制信号
ds_to_es_bus	OUT	136	流水线数据信息传递信号
br_bus	OUT	33	更新 PC

	方向	位宽	功能描述
ws_to_rf_bus	IN	38	寄存器堆写相关数据

3、 功能描述

```
assign {ds_inst,
        ds_pc  } = fs_to_ds_bus_r;
```

ID 模块通过对来自 IF 模块的 fs_to_ds_bus 信号进行解析，获得当前欲执行的指令以及 PC，接下来对指令 ds_inst 进行译码，获得指令格式，立即数，寄存器编号，是否访存等信息，同时将这些必要的信息存入将要传递到下一流水级的 ds_to_es_bus 信号中。由于译码过程代码太长，这里不贴出代码，仅贴出一些必要信号的生成：

```
assign src1_is_sa   = inst_sll   | inst_srl | inst_sra;
assign src1_is_pc   = inst_jal;
assign src2_is_imm  = inst_addiu | inst_lui | inst_lw | inst_sw;
assign src2_is_8    = inst_jal;
assign res_from_mem = inst_lw;
assign dst_is_r31   = inst_jal;
assign dst_is_rt    = inst_addiu | inst_lui | inst_lw;
assign gr_we        = ~inst_sw & ~inst_beq & ~inst_bne & ~inst_jr;
assign mem_we       = inst_sw;
assign load_op      = inst_lw;
assign dest         = dst_is_r31 ? 5'd31 :
                      dst_is_rt  ? rt    :
                                   rd;
```

同时，ID 模块的寄存器还需要完成读寄存器操作，以及接受来自 WB 阶段的写寄存器相关信号并完成写寄存器操作：

```
assign {rf_we   ,  //37:37
        rf_waddr,  //36:32
        rf_wdata   //31:0
       } = ws_to_rf_bus;
```

```
regfile u_regfile(
    .clk    (clk      ),
    .raddr1 (rf_raddr1),
    .rdata1 (rf_rdata1),
    .raddr2 (rf_raddr2),
    .rdata2 (rf_rdata2),
    .we     (rf_we    ),
    .waddr  (rf_waddr ),
    .wdata  (rf_wdata )
    );
```

ID 模块还需要对诸如条件分支等指令做判断获得更新的 PC 值，然后将该 PC 值立马回传给 IF 模块以避免引起流水线阻塞等情况产生。这是通过 IF 和 ID 模块之间的一个 br_bus 信号实现的。

```
assign br_taken = (   inst_beq  &&  rs_eq_rt
                   || inst_bne  && !rs_eq_rt
                   || inst_jal
                   || inst_jr
                  ) && ds_valid;
assign br_target = (inst_beq || inst_bne) ? (fs_pc + {{14{imm[15]}}, imm[15:0], 2'b0}) :
                   (inst_jr)              ? rs_value :
                  /*inst_jal*/              {fs_pc[31:28], jidx[25:0], 2'b0};

assign br_bus       = {br_taken,br_target};
```

最终，将必要的信号，例如 rs，rt 寄存器的值，以及选择 alu 操作数是来自于寄存器还是立即数，是否需要访问存储器等指令合并为 ds_to_es_bus 信号传递向下一流水级。

重要模块 3 设计：EXE 模块

1、 工作原理

EXE 模块全程是标准的组合逻辑设计，只需注意将对应信号连接至 ALU 模块的对应接口即可。此外，由于数据 RAM 采用同步 RAM，本拍发出的读指令要下一拍才能到达，所以在 EXE 阶段就需要发出数据 RAM 的读操作，在 MEM 阶段就能正好获得读数据。写操作也是类似的

2、 接口定义

	方向	位宽	功能描述
clk	IN	1	时钟信号
reset	OUT	1	控制复位
ms_allowin	IN	1	流水线传递控制信号
es_allowin	OUT	1	流水线传递控制信号
ds_to_ds_valid	IN	1	流水线传递的控制信号
ds_to_es_bus	IN	136	流水线数据信息传递
es_to_ms_valid	OUT	1	流水线传递控制信号
es_to_ms_bus	OUT	71	流水线数据信息传递信号
data_sram_en	OUT	1	数据 RAM 读使能信号
data_sram_wen	OUT	4	数据 RAM 写使能信号
data_sram_addr	OUT	32	数据 RAM 读地址
data_sram_wdata	OUT	32	数据 RAM 写数据

3、 功能描述

EXE 模块对数据进行算术逻辑运算，产生的数据有两种可能，一种是作为 WB 阶段最终写回寄存器的数据，另一种是作为访存阶段用于访问存储器(取数据或存数据)的 RAM 的地址。这两种情况的选择信号由来自于 ID 模块的 ds_to_es_bus 信号中的 load_op 信号解析得到。并传递给下一流水级

```
assign es_res_from_mem = es_load_op;
assign es_to_ms_bus = {es_res_from_mem,  //70:70
                       es_gr_we       ,  //69:69
                       es_dest        ,  //68:64
                       es_alu_result  ,  //63:32
                       es_pc             //31:0
                      };
```

EXE 模块中调用 ALU 模块进行运算，同时需要对运算的源操作数进行选择，考虑是来自寄存器的操作数还是来自立即数的操作数，还是 PC，选择信号同样通过解析 ds_to_es_bus_r 寄存器中的对应段得到。

```
assign {es_alu_op      ,  //135:124
        es_load_op     ,  //123:123
        es_src1_is_sa  ,  //122:122
        es_src1_is_pc  ,  //121:121
        es_src2_is_imm ,  //120:120
        es_src2_is_8   ,  //119:119
        es_gr_we       ,  //118:118
        es_mem_we      ,  //117:117
        es_dest        ,  //116:112
        es_imm         ,  //111:96
        es_rs_value    ,  //95 :64
        es_rt_value    ,  //63 :32
        es_pc             //31 :0
       } = ds_to_es_bus_r;
```

```
assign es_alu_src1 = es_src1_is_sa  ? {27'b0, es_imm[10:6]} :
                     es_src1_is_pc  ? es_pc[31:0] :
                                      es_rs_value;
assign es_alu_src2 = es_src2_is_imm ? {{16{es_imm[15]}}, es_imm[15:0]} :
                     es_src2_is_8   ? 32'd8 :
                                      es_rt_value;
```

最后，在 EXE 阶段发出对数据 RAM 的读，写命令，以便在 MEM 阶段能获得数据或者写入数据

```
assign data_sram_en    = 1'b1;
assign data_sram_wen   = es_mem_we&&es_valid ? 4'hf : 4'h0;
assign data_sram_addr  = es_alu_result;
assign data_sram_wdata = es_rt_value;
```

重要模块 4 设计：MEM 模块

1、 工作原理

MEM 模块用于访问存储器，不论是写入数据还是读出数据，都已经在 EXE 模块的最后发出了对应的指令，因此 MEM 模块可以直接在当前时钟周期获得读数据或进行写操作

2、 接口定义

	方向	位宽	功能描述
clk	IN	1	时钟信号
reset	OUT	1	控制复位
ws_allowin	IN	1	流水线传递控制信号
ms_allowin	OUT	1	流水线传递控制信号
ms_to_ws_valid	OUT	1	流水线传递的控制信号
ms_to_ws_bus	OUT	70	流水线数据信息传递
es_to_ms_valid	IN	1	流水线传递控制信号
es_to_ms_bus	IN	71	流水线数据信息传递信号
data_sram_rdata	IN	32	数据 RAM 读数据

3、功能描述

MEM 模块读取来自数据 RAM 的读数据，并且将该数据与 EXE 阶段得到的 ALU 计算结果进行选择，选择信号来自于 EXE 模块流水线寄存器传递过来的 ms_res_from_mem：

```
assign {ms_res_from_mem,  //70:70
        ms_gr_we       ,  //69:69
        ms_dest        ,  //68:64
        ms_alu_result  ,  //63:32
        ms_pc             //31:0
       } = es_to_ms_bus_r;

assign mem_result = data_sram_rdata;

assign ms_final_result = ms_res_from_mem ? mem_result
                                         : ms_alu_result;
```

ms_final_result 信号表明最终将要写回寄存器的值，并连同 ws_dst, ws_pc 等信号一同形成 ms_to_ws_bus 信号传递至下一流水线寄存器

重要模块 5 设计：WB 模块

1、 工作原理

WB 模块向寄存器中写回数据，通过一个 ws_to_rf_bus 信号送回 ID 阶段进行写入，尽管 WB 和 ID 同时使用寄存器堆，但这并不意味着 WB 和 ID 阶段同时执行，二者在流水级上仍具有先后次序

2、 接口定义

	方向	位宽	功能描述
clk	IN	1	时钟信号
reset	OUT	1	控制复位
ws_allowin	OUT	1	流水线传递控制信号
ms_to_ws_valid	IN	1	流水线传递的控制信号
ms_to_ws_bus	IN	70	流水线数据信息传递
ws_to_rf_bus	OUT	1	流水线传递控制信号
debug_wb_pc	OUT	32	用于调试的 PC 值
debug_wb_rf_wnum	OUT	5	用于调试的写回寄存器编号
debug_wb_rf_wen	OUT	4	用于调试的写回寄存器使能信号
debug_wb_rf_wdata	OUT	32	用于调试的写回寄存器数据

3、功能描述

WB 阶段将 MEM 阶段传递过来的 ms_final_result 数据写回寄存器堆，其中写回寄存器的编号由 ms_to_ws_bus 中的 ws_gr_we 信号与表示该位数据有效的 ws_valid 信号做逻辑并运算得到。这些数据形成 ws_to_rf_bus 信号一同作为输出送回到 ID 阶段的对应输入，再由 ID 阶段译码得到写回寄存器的编号，写使能信号以及数据，完成写回操作：

```
assign {ws_gr_we       ,  //69:69
        ws_dest        ,  //68:64
        ws_final_result,  //63:32
        ws_pc             //31:0
       } = ms_to_ws_bus_r;

assign rf_we    = ws_gr_we&&ws_valid;
assign rf_waddr = ws_dest;
assign rf_wdata = ws_final_result;
assign ws_to_rf_bus = {rf_we    ,  //37:37
                       rf_waddr,  //36:32
                       rf_wdata   //31:0
                      };
```

WB 模块中还提供了对应的 debug 信号和 trace 中的对应值进行比对，方便找出错误，debug 信号的值与其对应的 ws_to_ref_bus 中的值相同，此处不再赘述

（一）实验流水账

2019/9/13: 14:00 ~ 16:30 搭建环境以及调试，最终找出所有 bug，但综合失败

2019/9/14: 19:00 ~ 20:00 综合成功并且成功上板

2019/9/15: 8:00 ~ 16:00　写实验报告

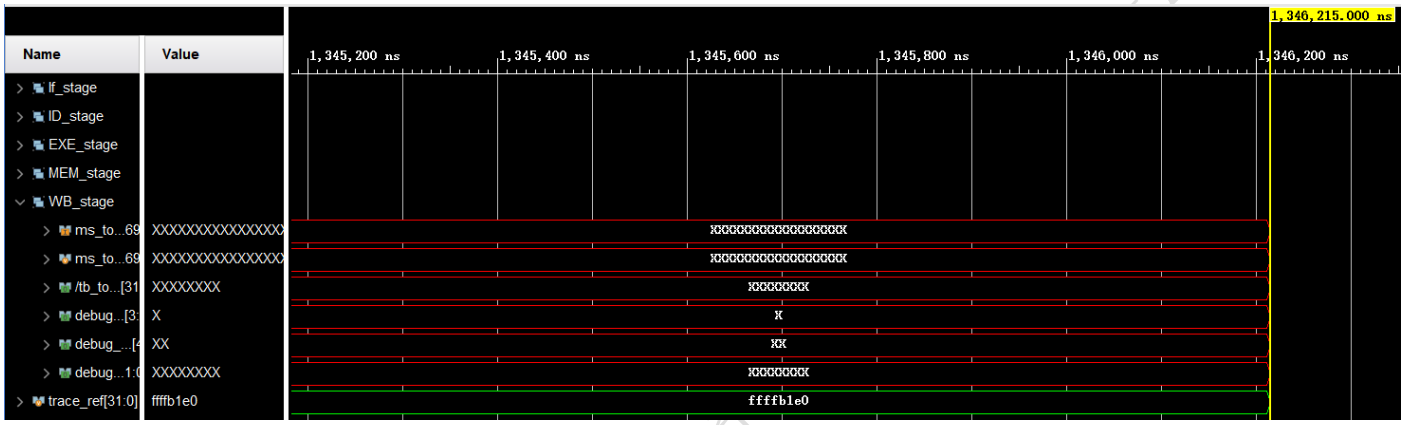
（二）错误记录

1、错误 1：大量出现 X 信号

（1）错误现象

仿真初始 1000ns 内 debug_ref_wdata 等信号显示为 X，在使用 run all 命令后仍然为 X,且仿真始终无法停止

（2）分析定位过程

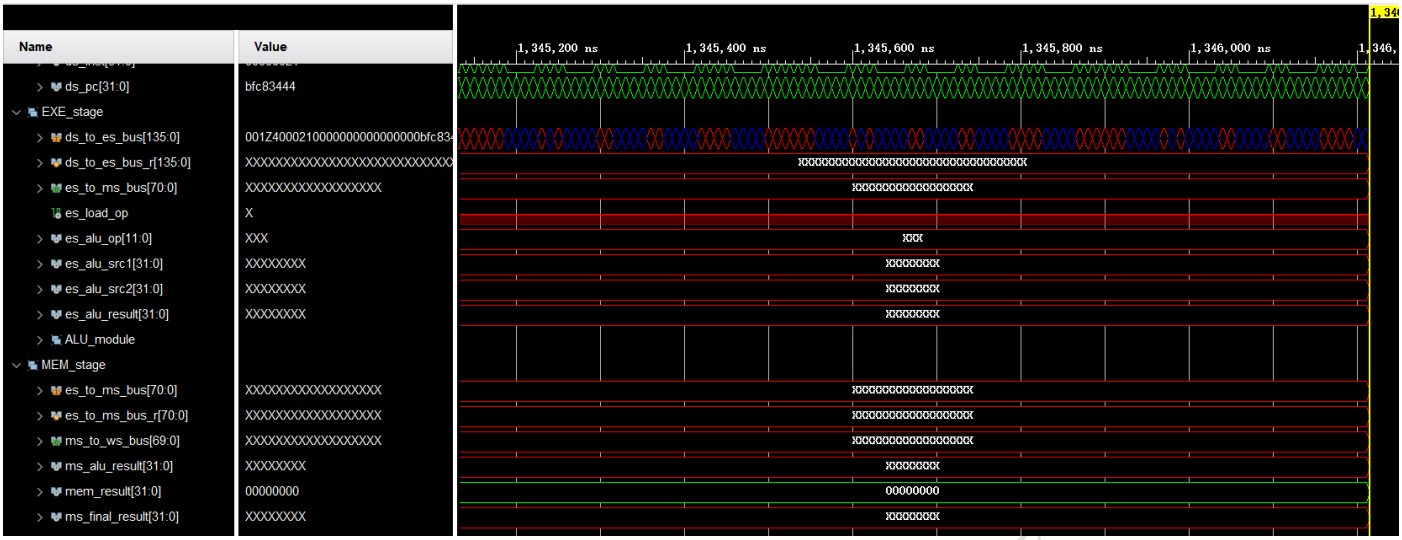


可以看到即使是在仿真到 1346200ns 时这些信号仍然为 X，联想到产生 X 信号的原因，可能是由于中间某处 reg 型变量未赋初值。

以 debug_wb_rf_wdata 信号为例，在代码中寻找产生它的源头。

分析代码可知，debug_wb_rf_wdata 信号来源于 ws_final_result，而 ws_final_result 信号来源于信号 ms_to_ws_bus_r,后者又在时钟上升沿处由输入信号 ms_to_ws_bus 赋值。

观察图中这几个信号，发现都是 X，因此 debug_wb_rf_wdata 出现问题的原因应该来自于上一个模块，即 MEM 模块，以此类推，层层向上回溯可以发现，EXE 阶段与 MEM 阶段的连接信号 es_to_ms_bus 为 X，EXE 模块到 MEM 模块的输出信号 es_to_ms_bus 也为 X，es_to_ms_bus 由 es_res_from_mem 等信号组成，而这些信号又来源于 ds_to_es_bus_r 寄存器，因此问题出在 ds_to_es_bus_r 信号上：

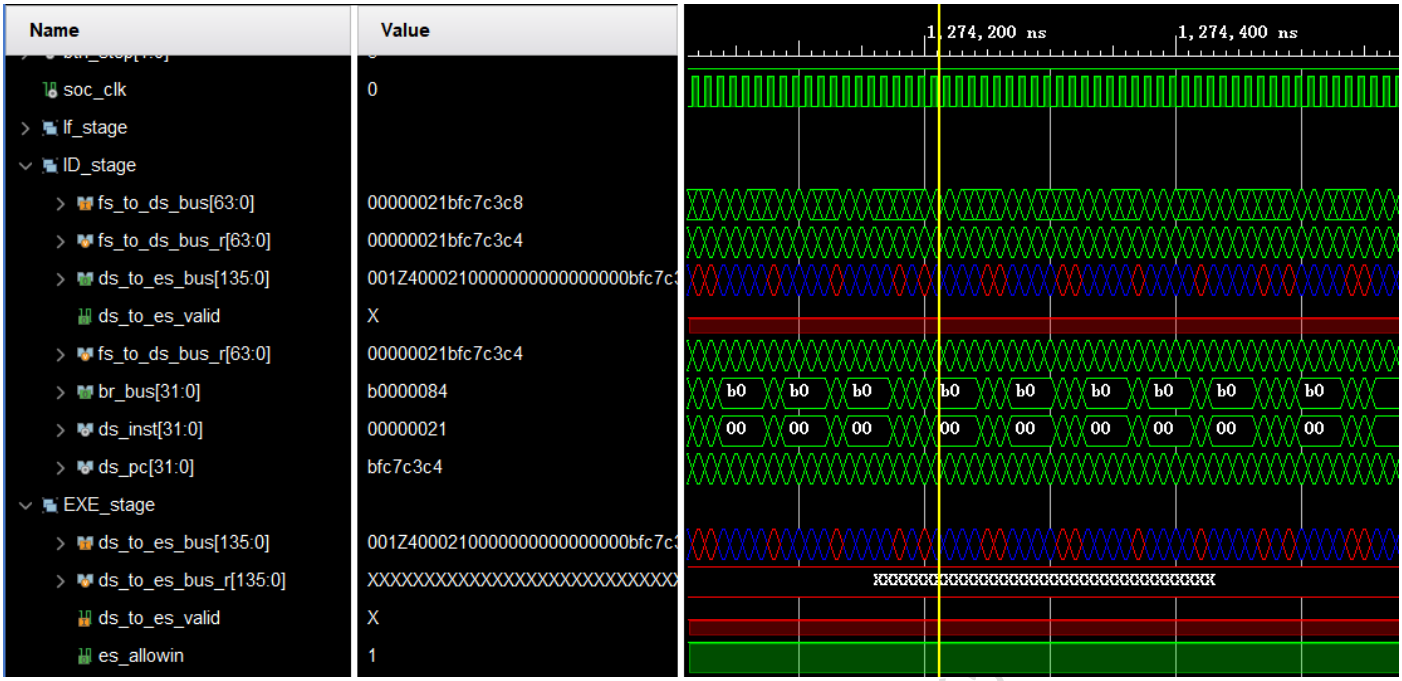


从图中可以看出 ds_to_es_bus 信号和 ds_to_es_bus_r 信号完全不同，但根据代码，ds_to_es_bus_r 信号应该在时钟上升沿被前者赋值:

```
always @(posedge clk) begin
    if (reset) begin
        es_valid <= 1'b0;
    end
    else if (es_allowin) begin
        es_valid <= ds_to_es_valid;
    end

    if (ds_to_es_valid && es_allowin) begin
        ds_to_es_bus_r <= ds_to_es_bus;
    end
end
```

出现错误唯一可能的原因是 ds_to_es_valid 信号和 es_allowin 信号的问题:



从波形图可以看出是 ds_to_es_valid 信号的问题，而且是在 ID 阶段出现的问题。在 ID 模块中 ds_to_es_valid 信号的代码如下：

```
assign ds_ready_go    = 1'b1;
assign ds_allowin     = !ds_valid || ds_ready_go && es_allowin;
assign ds_to_es_valid = ds_valid && ds_ready_go;
always @(posedge clk) begin
    if (fs_to_ds_valid && ds_allowin) begin
        fs_to_ds_bus_r <= fs_to_ds_bus;
    end
end
```

看得出 ds_to_es_valid 信号是由 ds_valid 和 ds_ready_go 共同决定的，而 ds_ready_go 恒为 1，问题就处在 da_valid 信号上，观察代码可以发现 ds_valid 信号作为一个 reg 信号时钟没有被赋值。

（3）错误原因

ID 模块中的 ds_valid 信号没有被赋值，导致该值一直为 X，从而引起后续的一系列信号也为 X

（4）修正效果

ds_valid 信号是在流水级之间传递流水信息的一个信号，其赋值由来完全固定，添加以下的代码即可

```
assign ds_ready_go    = 1'b1;
assign ds_allowin     = !ds_valid || ds_ready_go && es_allowin;
assign ds_to_es_valid = ds_valid && ds_ready_go;
always @(posedge clk) begin
    if (reset) begin
        ds_valid <= 1'b0;
    end
    else if(ds_allowin) begin
        ds_valid <= fs_to_ds_valid;
    end
    if (fs_to_ds_valid && ds_allowin) begin
        fs_to_ds_bus_r <= fs_to_ds_bus;
    end
end
```

该代码添加后，流水信号就能在各流水级之间顺利的传递了。

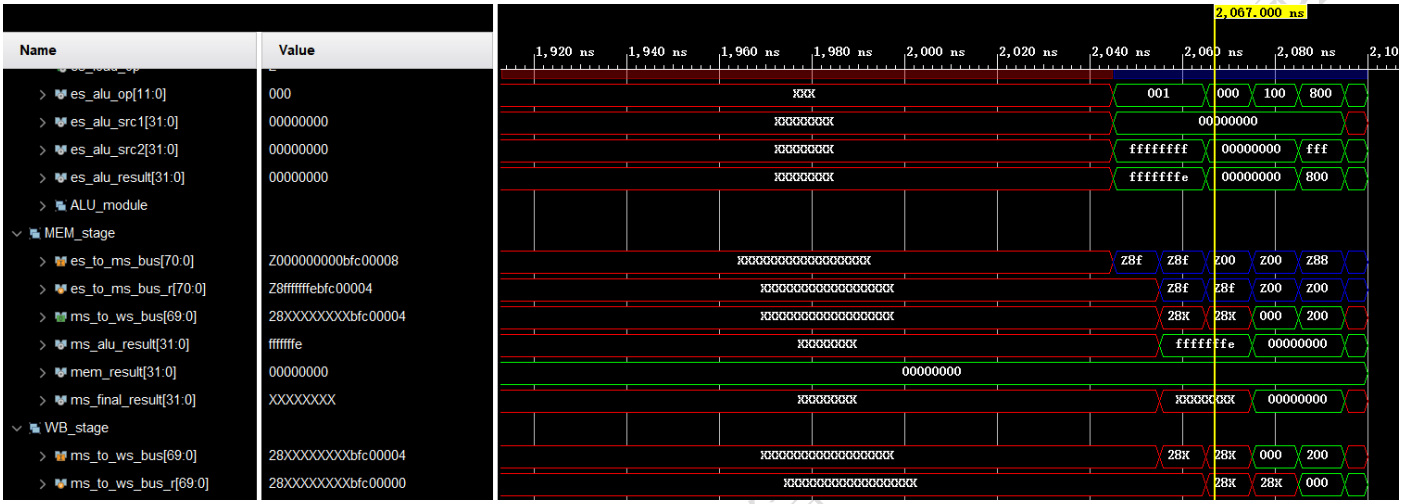
（5）归纳总结（可选）

2、错误 2：寄存器写回值错误

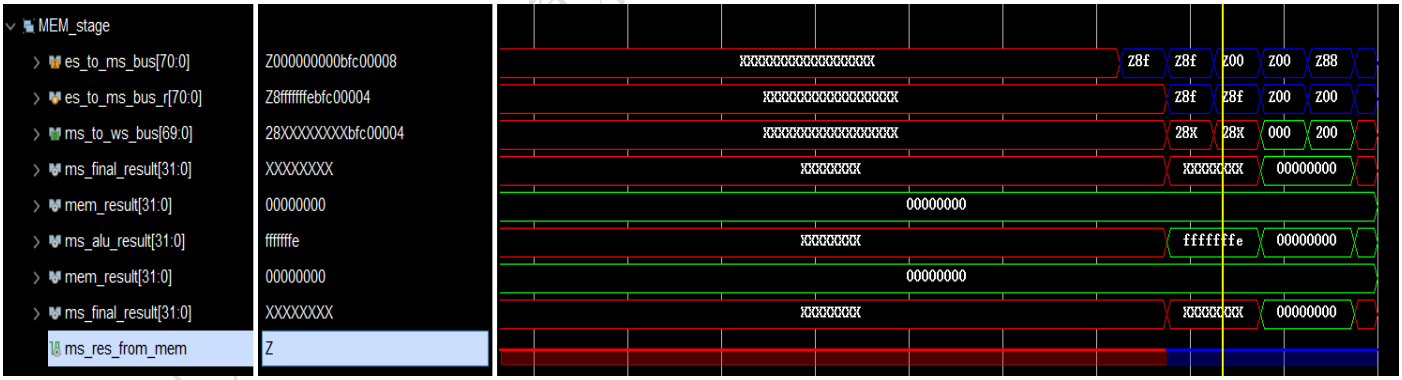
（1）错误现象

仿真到 2067ns 时，对比 goldentrace 可知写回寄存器的数据有误，写回了 0xXXXXXXXX, 导致仿真停止

（2）分析定位过程

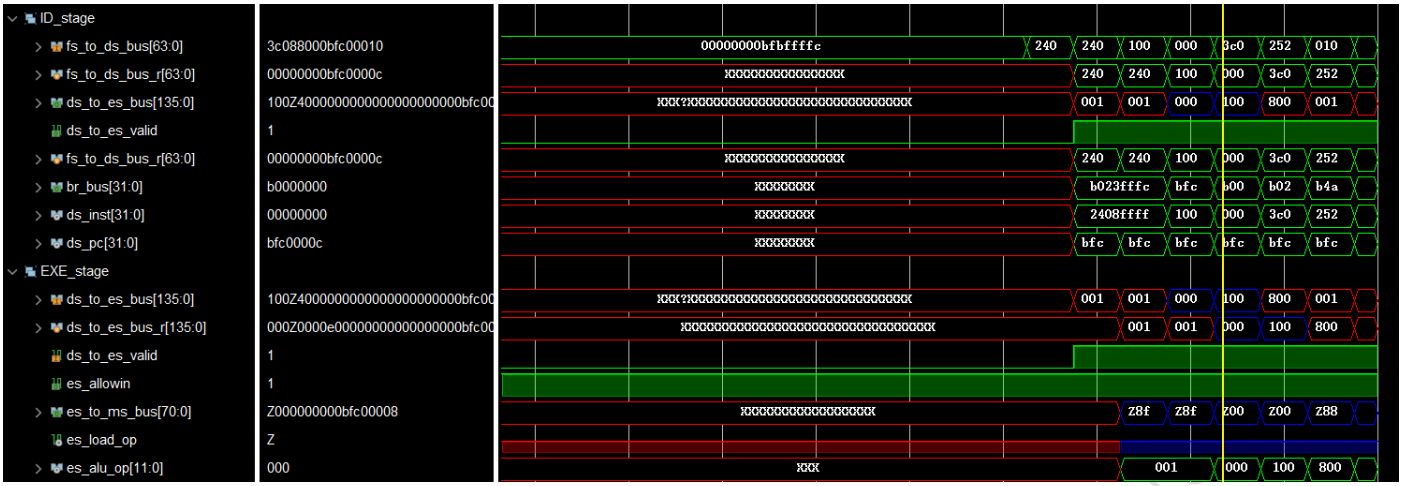


沿用上面的分析过程可以发现 debug_wb_wdata 实际上最终来源于 ms_to_ws_bus 的 63 到 32 位，而 ms_to_ws_bus 的 63 到 32 位信号来自于 ms_final_result 信号，ms_final_result 信号实际上是一个二选一信号，选择来自 alu 还是来自数据 RAM 的数据，查询反汇编指令可知该条指令实际上是一条加法指令，应该选择 alu_result,然而分析波形图可以发现，无论是 alu_result 还是 mem_result,两者均不为 X, 然而 ms_final_result 为 X，说明是选择信号的问题:

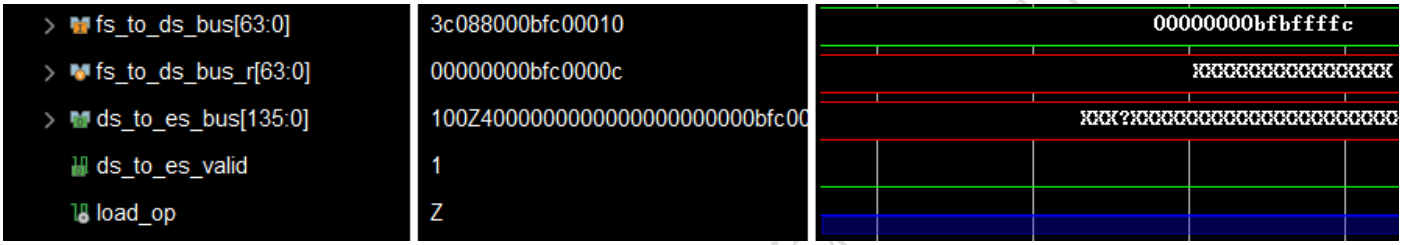



```
assign ms_final_result = ms_res_from_mem ? mem_result
                       : ms_alu_result;
```

从波形图中可以看出 me_res_from_mem 信号为 Z，说明该信号可能未赋值，或是在模块调用时未连接上。ms_res_from_mem 信号来自于 es_to_ms_bus_r 信号的第一位，而从波形图中看出 es_to_ms_bus_r 信号的第一位为 Z，因此是在 EXE 模块中生成该信号时就出错了。



从 EXE 模块直到找到 ID 模块中的对应信号 load_op:



发现该信号为 Z，作为一个内部信号，非常可能是因为从未对此信号赋值导致。在仔细检查了代码之后发现确实没有对 load_op 信号的相关赋值操作。

（3）错误原因

ID 模块中的 wire 信号 load_op 信号从未被赋值，导致高阻，从而引起后面的一系列信号均为 Z 或 X

（4）修正效果

load_op 信号表示该条指令是一条 load 型指令，在译码时赋值即可，增加以下的指令：

```
assign load_op      = inst_lw;
```

（5）归纳总结（可选）

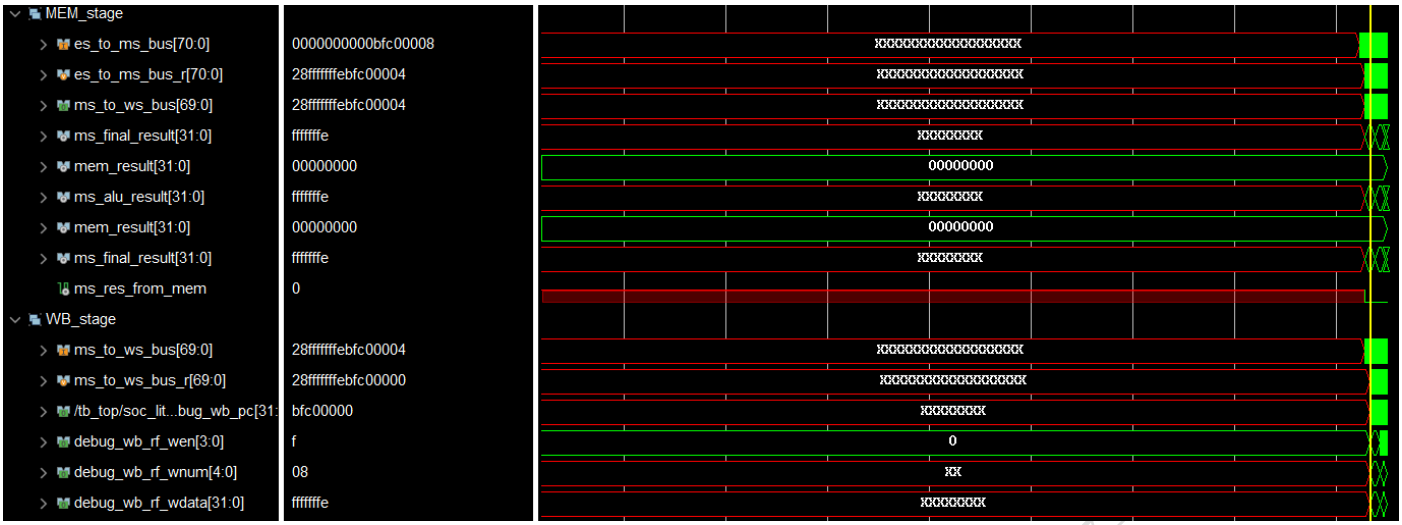
3、错误 3：寄存器写回值错误

（1）错误现象

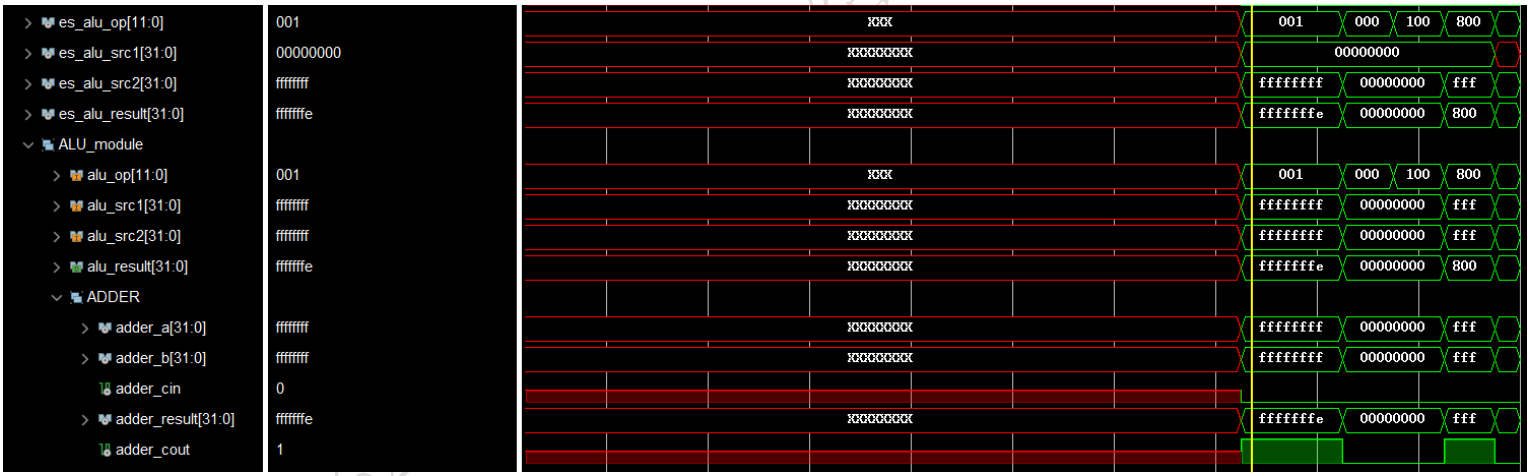
仿真到 2067ns 时，对比 goldentrace 可知写回寄存器的数据有误，应该写回 0xffffffff, 实际写回了 0xfffffffe, 导致仿真停止

（2）分析定位过程

重复上述步骤，追踪信号到 MEM 模块：



发现 ms_alu_result 和 ms_final_result 均为 0xfffffffe, 该条指令是 li 指令，实际上就是用 alu 算出结果再加载到寄存器中，因此这里选择寄存器结果是正确的。出错的原因在于 ALU 计算结果出错。由于是流水线，所以要回到两个时钟周期前 EXE 阶段计算结果时，此时得到的波形图如下所示：



分析此时的两个源操作数和运算 op 发现都是正确的，因此问题出在 ALU 模块，观察 ALU 模块的相关信号，发现 ALU 的两个操作数都是 0xffffffff, 与 EXE 阶段的模块不符合，因此是模块调用出现问题：

```
alu u_alu(
    .alu_op     (es_alu_op    ),
    .alu_src1   (es_alu_src2  ),
    .alu_src2   (es_alu_src2  ),
    .alu_result (es_alu_result)
    );
```

此处 ALU 模块的两个接口都被连接为了 es_alu_src2 端口

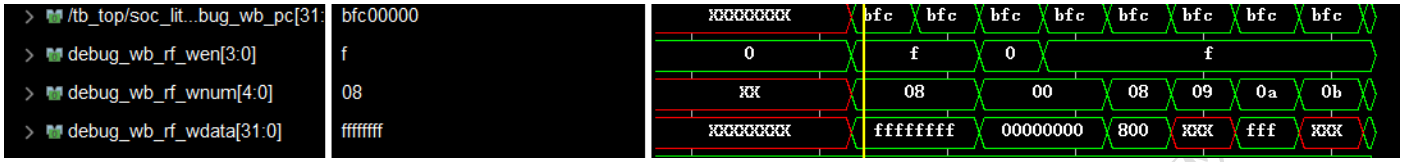
（3）错误原因

ALU 模块调用时接口连接信号错误，连接成了两个 src2

（4）修正效果

将 alu_src1 的连接端口改为 es_alu_src1 即可。

修改后正确的结果 0xffffffff 被写入了寄存器：



（5）归纳总结（可选）

4、错误 4：PC 跳转位置错误

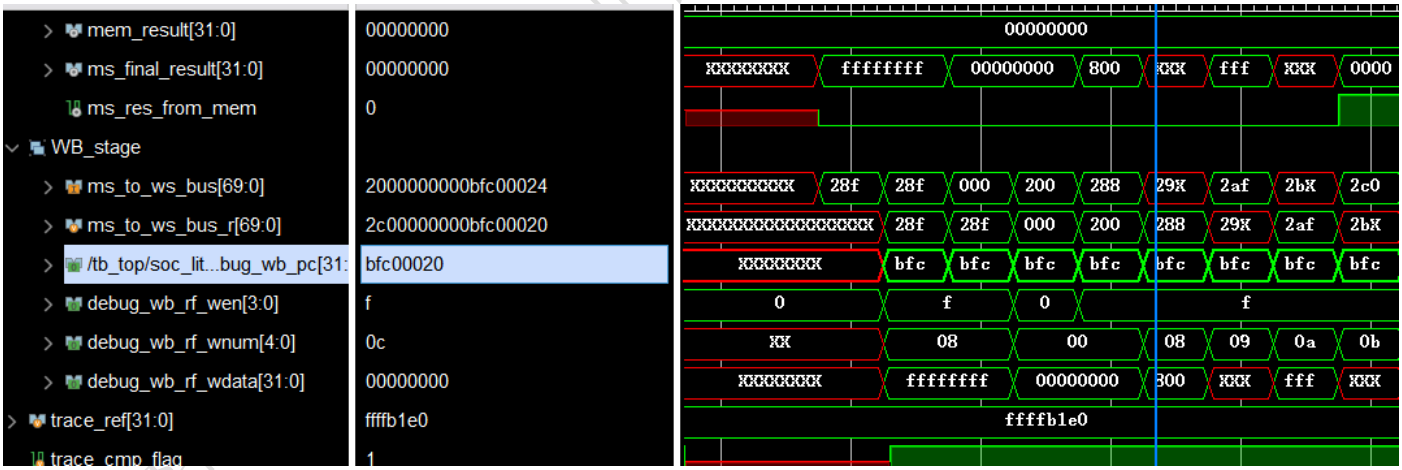
（1）错误现象

仿真到 2107ns 时，PC 的跳转位置发生错误：

```
[   2107 ns] Error!!!
    reference: PC = 0xbfc0038c, wb_rf_wnum = 0x04, wb_rf_wdata = 0xbfb00000
    mycpu    : PC = 0xbfc00010, wb_rf_wnum = 0x08, wb_rf_wdata = 0x80000000
```

（2）分析定位过程

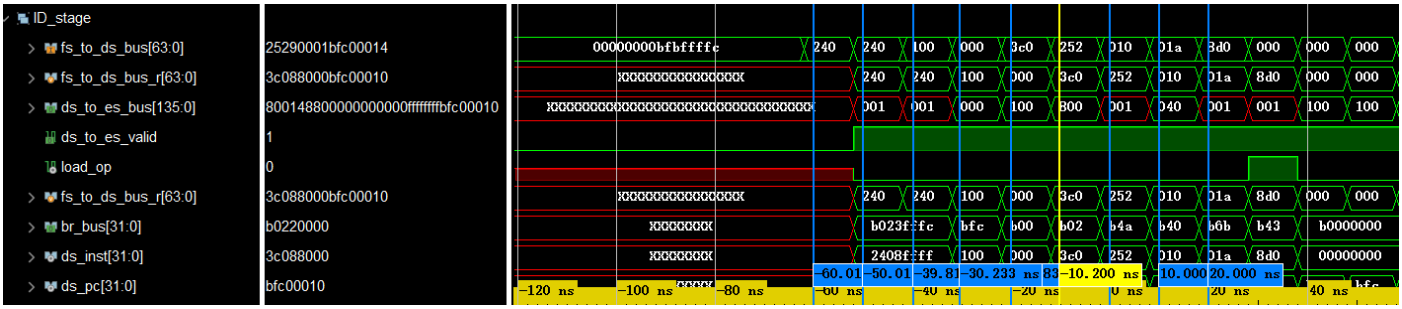
考察该条跳转指令，发现是一条 beq 指令，跳转位置是 0xbfc0038c，由于是跳转指令



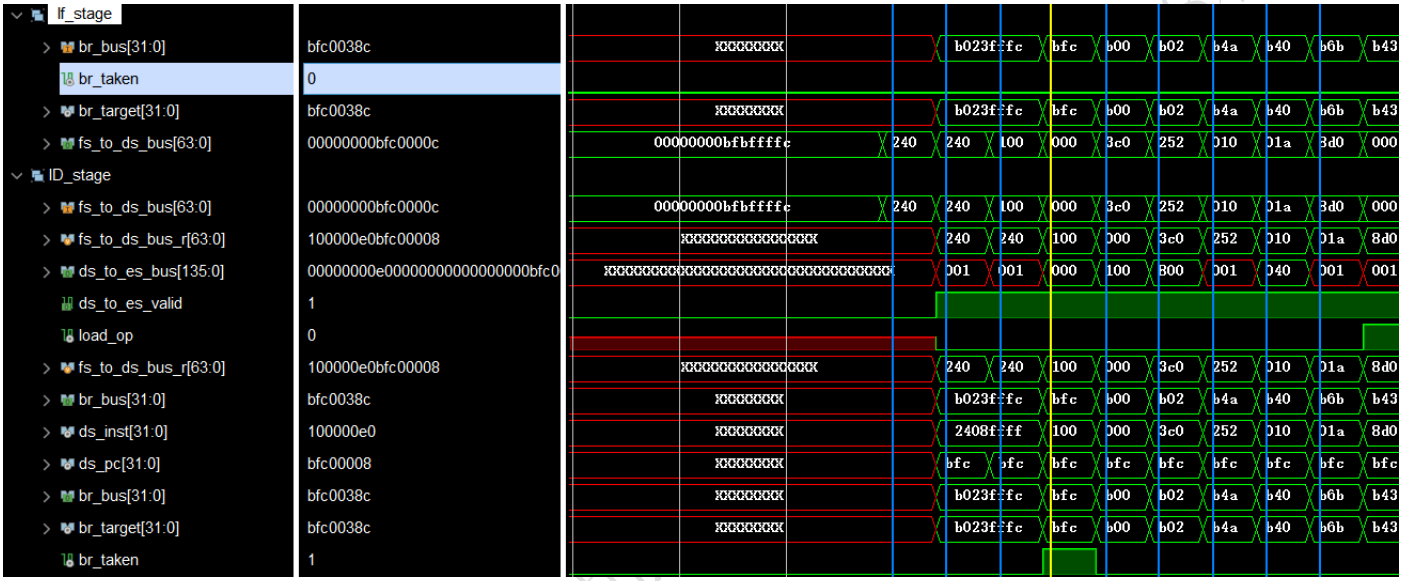
在发生错误的位置跳转写回的 PC 为 0xbfc00020，说明很大可能是顺序执行，没有发生跳转，因此可能为跳转判断出错。

```
assign debug_wb_pc       = ws_pc;
assign {ws_gr_we         ,  //69:69
        ws_dest          ,  //68:64
        ws_final_result  ,  //63:32
        ws_pc               //31:0
       } = ms_to_ws_bus_r;
```

根据分析可知，在该周期产生的写回 PC 在三个周期前的 ID 阶段就已经译码得到了：



可以看出此处 ds_to_es_bus 开始向下传递的数据就已经为 0xbfc00010 了，然而，由于这是一条分支指令，因此有延迟槽，beq 指令后的那条 nop 指令总会执行，所以我们需要到上一条指令的执行位置考察 beq 的执行。



从该波形图中我们可以看出，fs_to_ds_bus 的最后几位是 0xbfc0000c,但是实际上，在 IF 阶段接收到了来自 ID 极端的 br_bus 信号而应该对 PC 进行更新，然而却并没有，分析代码可知这是由于选择信号 br_taken 出错导致的。再观察波形图可以发现 ID 译码得到并发送出的 br_taken 和 IF 接受并解析的 br_taken 不同。仔细分析可以发现由 br_taken 和 br_target 拼接成的 br_bus 信号只有 32 位，因此 br_taken 信号被忽略了。

```
`define BR_BUS_WD       32
output [`BR_BUS_WD       -1:0] br_bus
assign br_bus           = {br_taken,br_target};
```

（3）错误原因

br_bus 位宽错误，导致 br_bus 信号被忽略，引起 PC 错误跳转

（4）修正效果

在 myCPU.h 中将 BR_BUS_WD 改为 33 即可

5、**错误 5：**仿真停止

（1）**错误现象**

当仿真进行到 717915ns 时，仿真不再进行，波形图不再前进

（2）**分析定位过程**

波形停止很有可能是由于出现了组合环导致的，观察波形停止处的 ALUop 信号：

此处信号为 0x020, 解析该信号可知此时 EXE 模块执行的操作是 nor 操作，查看 EXE 模块中实现 nor 的代码：

```
assign nor_result = ~or_result;
assign or_result  = alu_src1 | alu_src2 | alu_result;
assign alu_result = ({32{op_add|op_sub}} & add_sub_result)
                  | ({32{op_slt       }} & slt_result)
                  | ({32{op_sltu      }} & sltu_result)
                  | ({32{op_and       }} & and_result)
                  | ({32{op_nor       }} & nor_result)
                  | ({32{op_or        }} & or_result)
                  | ({32{op_xor       }} & xor_result)
                  | ({32{op_lui       }} & lui_result)
                  | ({32{op_sll       }} & sll_result)
                  | ({32{op_srl|op_sra}} & sr_result);
```

不难看出 alu_result 中含有 nor_result， 而 nor_result 中又有 alu_result, 形成了一个组合环

（3）**错误原因**

alu 模块实现中 alu_result 处存在组合环

（4）**修正效果**

将 or_result 的赋值语句中的 alu_result 去掉，如下图所示：

```
assign or_result  = alu_src1 | alu_src2;
```

如上图修改后仿真能够正常进行

6、**错误 6：**寄存器写回错误

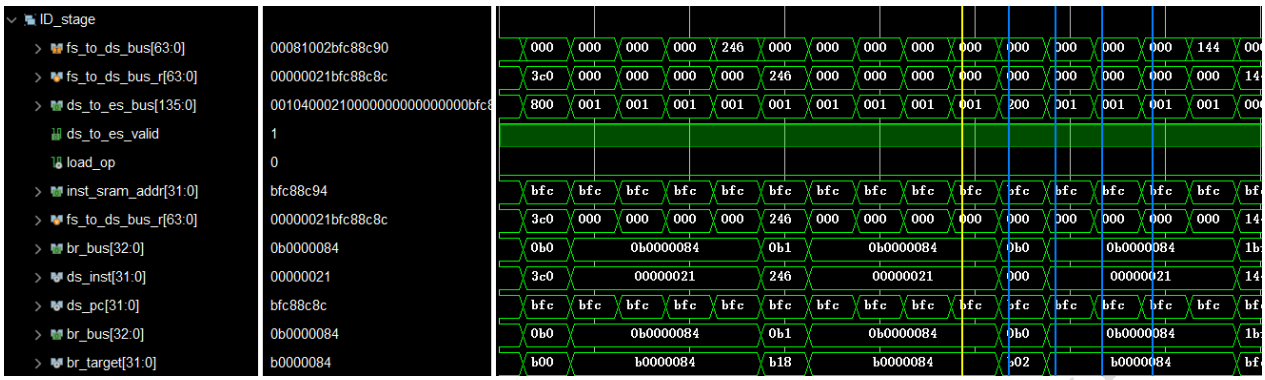
（1）**错误现象**

仿真进行到 888537ns 时显示此时写回寄存器的数据有误：

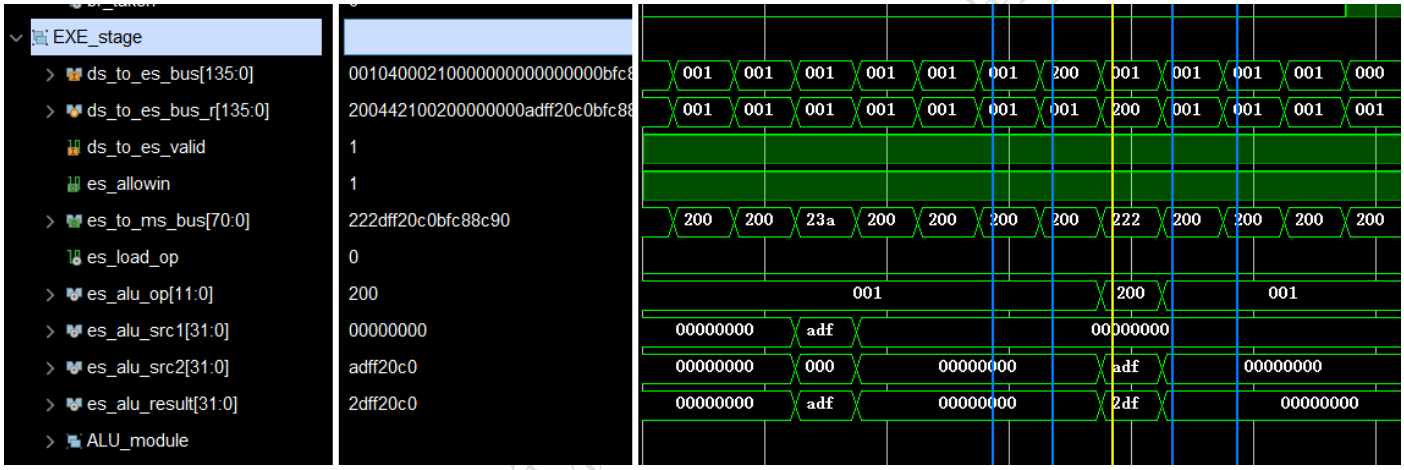
```
[ 888537 ns] Error!!!
    reference: PC = 0xbfc88c90, wb_rf_wnum = 0x02, wb_rf_wdata = 0xadff20c0
    mycpu    : PC = 0xbfc88c90, wb_rf_wnum = 0x02, wb_rf_wdata = 0x2dff20c0
```

（2）分析定位过程

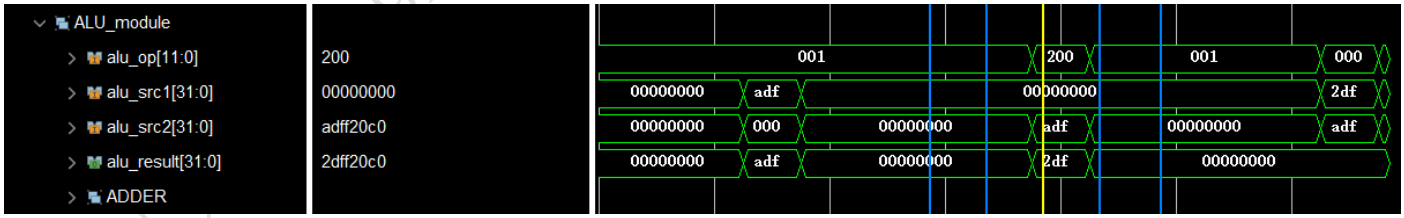
由于是在 888537ns 时发生的写回寄存器数据错误信息，回到五个周期之前的 IF 阶段就能得到当前指令的 PC 值，如下图所示：



可以看出当前指令执行的是 0xbfc88c90 位置的指令，查询 test.s 文件可知这是一条右移指令。且两个操作数分别为$t0 寄存器中的值和 0x0。到 EXE 阶段中观察 src1, src2, aluop 这三个信号。



看得出 es_alu_op, src1,src2 的值确实是正确的值，但计算结果却错误了，应该是 ALU 模块的错误，抓取 ALU 模块的内部信号，可以发现 ALU 模块传递的接口信号完全正确，但是计算结果却不对，应该是内部逻辑有误。



查看 ALU 的实现代码，尤其是关于右移的实现部分：

```
assign sr64_result = {{32{op_sra & alu_src2[31]}}, alu_src2[31:0]} >> alu_src1[4:0];

assign sr_result   = sr64_result[30:0];

// final result mux
assign alu_result = ({32{op_add|op_sub}} & add_sub_result)
                  | ({32{op_slt       }} & slt_result)
                  | ({32{op_sltu      }} & sltu_result)
                  | ({32{op_and       }} & and_result)
                  | ({32{op_nor       }} & nor_result)
                  | ({32{op_or        }} & or_result)
                  | ({32{op_xor       }} & xor_result)
                  | ({32{op_lui       }} & lui_result)
                  | ({32{op_sll       }} & sll_result)
                  | ({32{op_srl|op_sra}} & sr_result);
```

可以发现 sr_result 仅有 31 位。

（3）错误原因

sr_result 信号赋值时位宽错误，导致最高位的 1 被省略，因此写入数据错误

（4）修正效果

将 sr_result 信号赋值时改成赋值为 sr64_result 的 31 到 0 位，如下图所示：

```
assign sr_result    = sr64_result[31:0];
```

修改之后能正常通过该处测试

7、错误 7：

（1）错误现象

阅读代码时发现，没有发现有明显的波形或上板错误

（2）分析定位过程

阅读代码时发现该错误，因此没有分析定位过程

错误代码所示如下：

```
always @(posedge clk) begin
    if (reset) begin
        ms_valid <= 1'b0;
    end
    else if (ms_allowin) begin
        ms_valid <= es_to_ms_valid;
    end

    if (es_to_ms_valid && ms_allowin) begin
        es_to_ms_bus_r  = es_to_ms_bus;
    end
end
```

（3）错误原因

在时序逻辑中应该使用非阻塞赋值，而不应该使用阻塞赋值

（4）修正效果

将 es_to_ms_bus_r 处的阻塞赋值改为非阻塞赋值，如下图所示：

```
always @(posedge clk) begin
    if (reset) begin
        ms_valid <= 1'b0;
    end
    else if (ms_allowin) begin
        ms_valid <= es_to_ms_valid;
    end

    if (es_to_ms_valid && ms_allowin) begin
        es_to_ms_bus_r  <= es_to_ms_bus;
    end
end
```

四、实验总结（可选）

通过这次实验，我对五级流水线的工作原理有了更加深入的认识，同时再一次复习了 RAM 的相关知识并且锻炼了找 bug 的能力