实验 7 报告

蔡润泽、付轶凡
箱子号：45

一、实验任务（10%）

本次实验任务是要在实验六的 CPU 代码基础上添加更多的指令，具体包括转移类指令 BGEZ、BGTZ、BLEZ、BLTZ、J、 BLTZAL、BGEZAL、JALR，乘除运算类指令 MULT、MULTU、DIV、DIVU，以及访存指令 LB、LBU、 LH、LHU、LWL、LWR、SB、SH、SWL、SWR。运行 func_lab7，要求成功通过仿真和上板验证。

二、实验设计（40%）

（一）总体设计思路

硬件设计图如下：

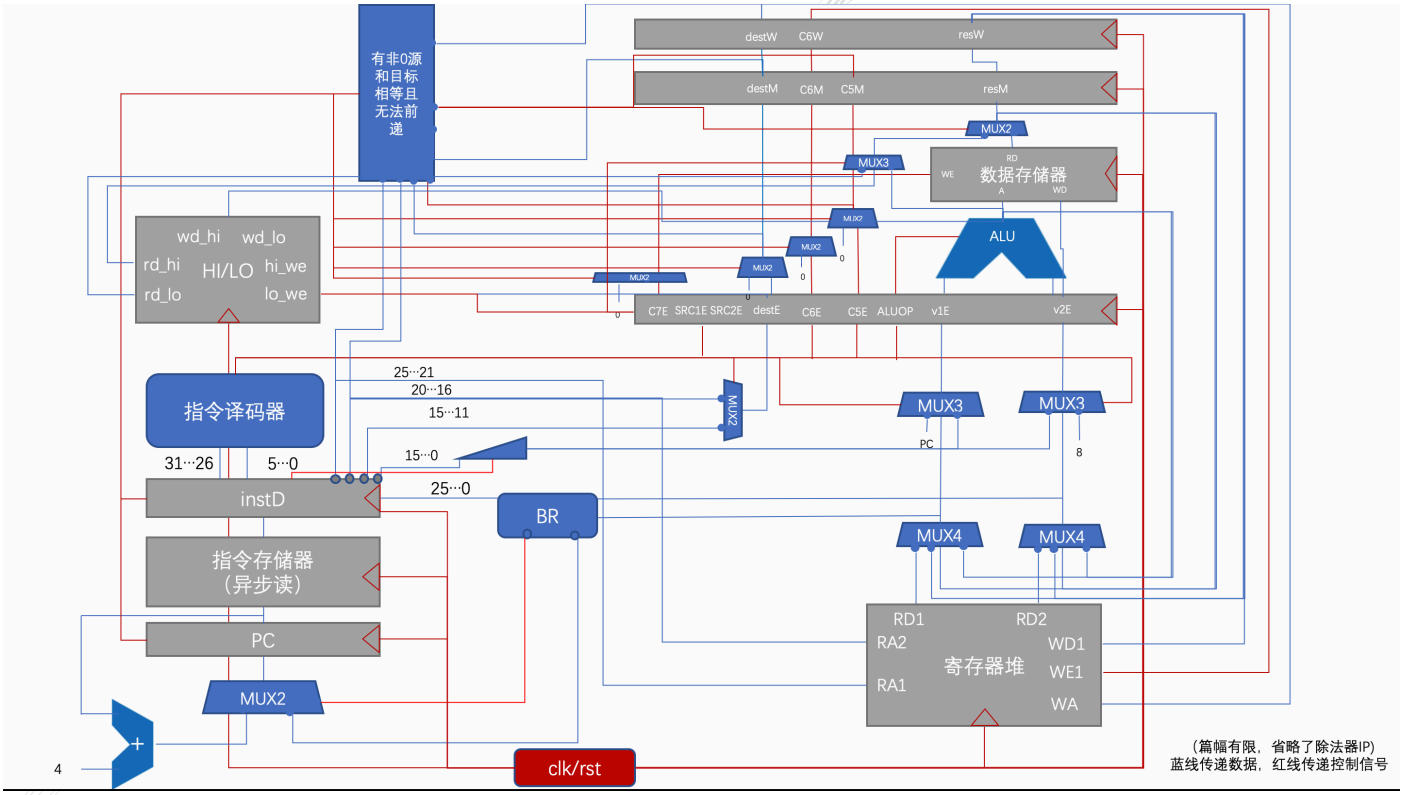

图 1.1 硬件结构设计图

如图 1.2 的流水示意图，在代码设计中，主要有 8 个模块，包括五级流水、ALU、寄存器堆以及 cp0 协处理器（用来储存乘法指令的高 32 位、低 32 位，以及除法指令的余数、商）。该设计使用了四个 IP，Inst_RAM 和 Data_RAM（采用同步 RAM），mydiv 和 mydivu（分别计算有符号、无符号除法）。并且采用 Tools 模块进行译码。

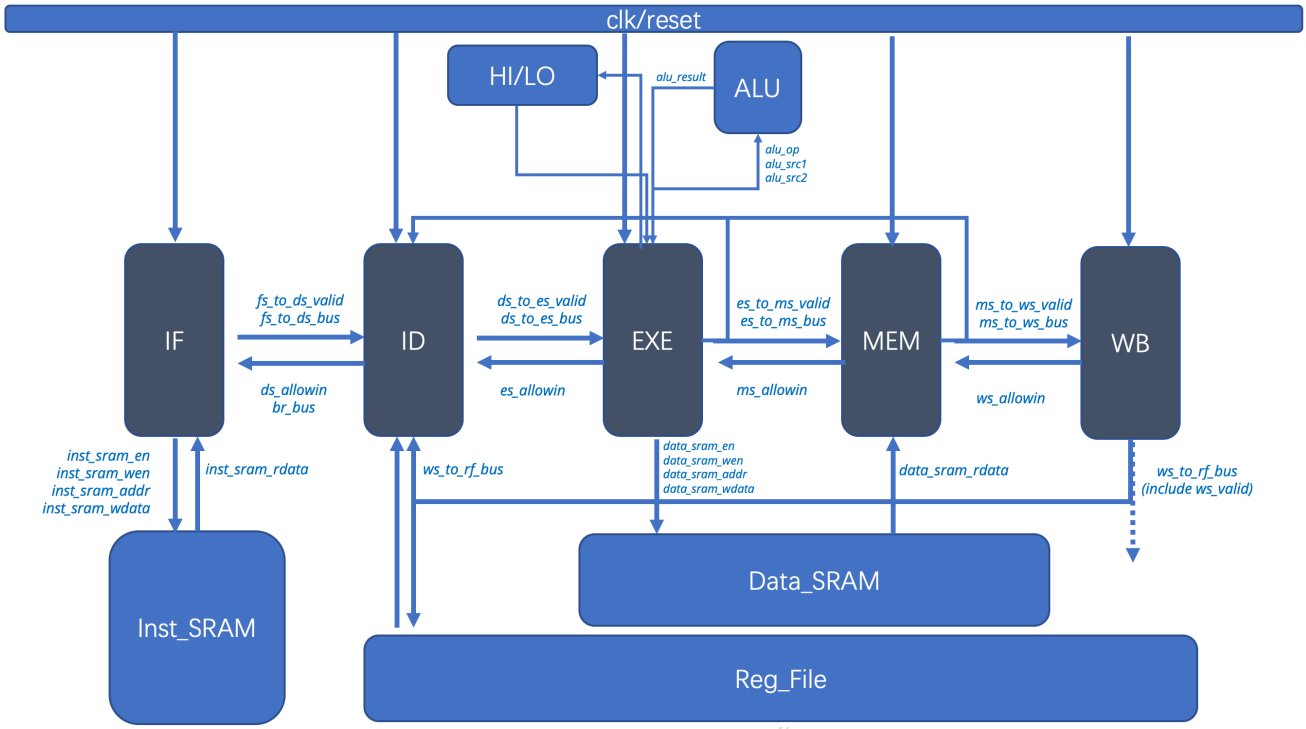

图 1.2 流水示意图

（二）重要模块 1 设计：算数逻辑单元（ALU）模块

1、工作原理

将 CPU 中的运算处理进行模块化，方便外界调用。同时模块化的 ALU 设计便于在其中增加新的运算功能，提高代码的扩展性。

2、接口定义

```
input  [15:0] alu_op,        //输入运算符
input  [31:0] alu_src1,      //输入数据 1
input  [31:0] alu_src2,      //输入数据 2
output [31:0] alu_result     //输出结果
```

3、功能描述

采用 16 位的独热码对 ALU 进行控制，根据独热码进行 16 项（相较之前的实验增添了有符号、无符号乘除法）不同的算数逻辑运算操作，并将结果传回给 exe 阶段。

（三）重要模块 2 设计：寄存器堆（Reg_File）模块

1、工作原理

将 32 个 32 位宽的寄存器堆模块化，以实现两读一写，同步读异步写的操作。

2、接口定义

```
input        clk,

// READ PORT 1
```

```
input  [ 4:0] raddr1,
output [31:0] rdata1,
// READ PORT 2
input  [ 4:0] raddr2,
output [31:0] rdata2,
// WRITE PORT
input         we,      //write enable, HIGH valid
input  [ 4:0] waddr,
input  [31:0] wdata
```

3、功能描述

当写使能信号为1时，在写回阶段对寄存器堆进行写入。同时，对于两个读端口信号，进行异步读取，将输出结果传递给ID阶段。

（四）重要模块3设计：取指阶段（IF_stage）模块

1、工作原理

将取指操作模块化，IF从inst_ram中读出指令，并且在该周期模块之内处理PC的值，并且将指令传递给bus传递给ID模块，ID模块在下一周期再接收 。

2、接口定义

表2.1 IF_stage接口定义

名称	方向	位宽	功能描述
clk	IN	1	时钟信号
reset	IN	1	复位信号
ds_allowin	IN	1	ID模块允许接受IF传值
br_bus	IN	33	输入是否跳转和branch的target
fs_to_ds_valid	OUT	1	IF模块可以向ID模块传值
fs_to_ds_bus	OUT	64	IF模块向ID模块传递数据（指令码和地址）
inst_sram_en	OUT	1	Inst_sram读使能
inst_sram_wen	OUT	4	Inst_sram写使能，此处恒为0
inst_sram_addr	OUT	32	Inst_sram目标地址
inst_sram_wdata	OUT	32	Inst_sram写数据
inst_sram_rdata	IN	32	Inst_sram读数据

3、功能描述

PC在收到reset信号时设为偏移量32'hbfbffffc，并且在该周期模块之内处理PC的值，PC值的变化根据br_bus取出来决定是否跳转还是加4。IF模块当inst_sram_en读使能信号为1时，将处理后的next_pc作为地址传递给inst_sram, 并从inst_ram中读出指令。IF模块将取出的指令和地址传递给ID模块，ID模块在下一周期再接收 。

（五）重要模块 4 设计：译码阶段（ID_stage）模块

1、工作原理

将从 IF 模块获取的指令进行译码，获得指令格式类型、ALU 操作类型、是否需要加载、写回内存和参与运算数据的值、是否需要加载协处理器 cp0 的 hi 和 lo 寄存器、是否需要写回协处理器 cp0 的 hi 和 lo 寄存器、跳转的目标 PC。判断 PC 是否需要跳转，并将结果返还给 IF 模块。将译码后的数据和控制信号传递给 EXE 模块,EXE 模块在下一时钟周期接受。另外，写回阶段的数据也通过该模块传递给寄存器堆。

CPU 数据通路增加旁路设计，来让前面的指令直接把已经生成出来的结果直接转给后面的指令。在本设计中，采用了“流水级组合逻辑的结果传递到译码级寄存器读出处”的方案。并通过后续阶段的 valid 信号和 gr_we 信号来控制 ID 模块中，rs_value 和 rt_value 的值。

另外对于 ready_go 信号，当译码级的指令和处在执行级的 LW 指令相关时，需要设置成“0”。

2、接口定义

表 2.2 ID_stage 接口定义

名称	方向	位宽	功能描述
clk	IN	1	时钟信号
reset	IN	1	复位信号
es_allowin	IN	1	EXE 模块允许接受 ID 传值
ds_allowin	OUT	1	允许 IF 模块向 ID 模块传递数据
fs_to_ds_valid	IN	1	IF 模块可以向 ID 模块传值
fs_to_ds_bus	IN	64	IF 模块向 ID 模块传递数据（指令码及地址）
ds_to_es_valid	OUT	1	允许 ID 模块向 ES 模块传递数据
ds_to_es_bus	OUT	153	ID 模块向 EXE 模块传递数据
br_bus	OUT	33	输出是否跳转和 branch 的 target 给 IF 模块
ws_to_rf_bus	IN	40	WB 模块向 ID 模块传递的需要写回 REG FILE 的信息
es_to_ms_bus	IN	108	EXE 模块向 MEM 模块传递数据
ms_to_ws_bus	IN	70	MEM 模块向 WB 模块传递数据
es_to_ms_valid	IN	1	EXE 模块可以向 MEM 模块传值
ms_to_ws_valid	IN	1	MEM 模块可以向 WB 模块传值
out_es_valid	IN	1	接收 es_valid 是否为 1
out_ms_valid	IN	1	接收 ms_valid 是否为 1

相较上次实验，本次实验的代码设计中在 ds_to_es_bus 中增添了 load_type、vaddr_2、store_type 三个信号，用来描述 load、store 指令类型以及虚拟地址，所以 ds_to_es_bus 总线的位宽扩充至 153。

3、功能描述

将从 IF 模块获取的指令进行译码，获得指令格式类型、ALU 操作类型、是否需要加载、写回内存和参与运算数据的值、是否需要加载协处理器 cp0 的 hi 和 lo 寄存器、是否需要写回协处理器 cp0 的 hi 和 lo 寄存器、跳转的目标 PC。判断 PC 是否需要跳转，并将结果返还给 IF 模块。将译码后的数据和控制信号在传递给 EXE 模块,EXE 模块在下一时钟周期接受。另外，写回阶段的数据也通过该模块传递给寄存器堆。并且采用前递的

方式来减少 CPU 的阻塞，缩短运行时间。

（六）重要模块 5 设计：执行阶段（EXE_stage）模块

1、工作原理

将从 ID 模块获取的指令相应的执行。将执行后的 ALU 结果和前阶段传递的通用寄存器写使能、写地址控制信号、PC 传通过总线传递给 MEM 模块,MEM 模块在下一周期接收新值。另外，load 指令的发出读信号处理也在 EXE 阶段完成，EXE 模块将数据传递给 MEM 模块，在下一周期 WB 阶段进行写回。输出数据 RAM 的写信号和数据。

此阶段在进行算数逻辑运算时，需要调用 ALU 模块。

在本次任务中，除了对 ALU 模块需要新增有符号、无符号乘法外，还需要在此阶段调用我们生成的 mydiv 和 mydivu 两个除法器 IP 完成新增的有符号、无符号除法运算，以及对除法器中的控制信号进行相应的设置。

2、接口定义

表 2.3 EXE_stage 接口定义

名称	方向	位宽	功能描述
clk	IN	1	时钟信号
reset	IN	1	复位信号
ms_allowin	IN	1	MEM 模块允许接受 EXE 传值
es_allowin	OUT	1	EXE 模块允许接受 ID 传值
ds_to_es_valid	IN	1	ID 模块可以向 EXE 模块传值
ds_to_es_bus	IN	153	ID 模块向 EXE 模块传递数据
es_to_ms_valid	OUT	1	EXE 模块可以向 MEM 模块传值
es_to_ms_bus	OUT	108	EXE 模块向 MEM 模块传递数据
data_sram_en	OUT	1	data_sram 读使能
data_sram_wen	OUT	4	data_sram 写使能
data_sram_addr	OUT	32	data_sram 目标地址
data_sram_wdata	OUT	32	data_sram 写数据
out_es_valid	OUT	1	将 es_valid 的值传递给 ID 模块

mydiv 和 mydivu 除法器 IP 数据通道定义如下：

```
s_axis_divisor_tdata[31:0];        //除数
s_axis_divisor_tready;             //除数应答信号
s_axis_divisor_tvalid;             //除数请求信号
s_axis_dividend_tdata[31:0];       //被除数
s_axis_dividend_tready;            //被除数应答信号
s_axis_dividend_tvalid;            //被除数请求信号
m_axis_dout_tdata[63:0];           //商和余数，其中高 32 位为余数、低 32 位为商
m_axis_dout_tvalid;                //结果有效信号，表示除法已经算完，可以取得结果
```

aclk; //时钟信号

在除法指令处于执行流水级且没有对除法器成功输入数据的时候，同时将 s_axis_dividend_tvalid 和 s_axis_divisor_tvalid 置为 1，向除法器 IP 发送调用除法器的请求。当发现 s_axis_dividend_tready 和 s_axis_divisor_tready 反馈为 1 后，tvalid 和 tready 成功握手，并需要在下一拍将 s_axis_dividend_tvalid 和 s_axis_di vior_tvalid 清为 0，以此保证一个除法操作只调用一次除法器 IP，避免除法器给 CPU 送多个结果从而导致出错。成功握手后，数据传入除法器的各个数据通路，8 拍后，除法器产生结果并将 m_axis_dout_tvalid 信号置 1，表示可以取得除法结果。

为了避免除法指令与之后的指令产生“写后读”相关，考虑到除法器 IP 需要 8 拍才能拿到结果，所以如果执行流水级这一拍执行的是除法指令，需要将执行级阻塞住（即修改 es_ready_go 信号），拿到除法结果后再释放，因为通常程序中的除法指令较少，这样的阻塞设计不会太影响整体的性能。

另外 es_to_ms_bus 相较上次新增添了 es_load_type、es_vaddr_2、es_rt_value 三个信号，给 MEM 阶段传 load 指令类型、虚拟地址以及 rt 寄存器的值（用来完成 lwl 和 lwr 指令写 rt 寄存器值的拼接）。

3、功能描述

将从 ID 模块获取的指令相应的执行。将执行后和前阶段传递的数据控制信号通过数据总线在下一时钟周期更新传给 MEM 模块。另外，load 指令的发出读信号处理也在 EXE 阶段完成，EXE 模块将数据传递给 MEM 模块，在下一周期进行写回。输出数据 RAM 的写信号和数据。

（六）重要模块 6 设计：访存阶段（MEM_stage）模块

1、工作原理

将从 EXE 模块获取的访存指令相应的执行。根据 es_to_ms_bs 中的是否数据来自数据 RAM 信号确定是否有访存取出的数据，并将相应指令的最终结果和前阶段传递的通用寄存器写使能、写地址控制信号、PC 传通过总线在下一时钟周期更新给 WB 模块。

2、接口定义

表 2.4 MEM_stage 接口定义

名称	方向	位宽	功能描述
clk	IN	1	时钟信号
reset	IN	1	复位信号
ws_allowin	IN	1	WB 模块允许接受 MEM 传值
ms_allowin	OUT	1	MEM 模块允许接受 EXE 传值
es_to_ms_valid	IN	1	EXE 模块可以向 MEM 模块传值
es_to_ms_bus	IN	108	EXE 模块向 MEM 模块传递数据
ms_to_ws_valid	OUT	1	MEM 模块可以向 EXE 模块传值
ms_to_ws_bus	OUT	70	MEM 模块向 WB 模块传递数据
data_sram_rdata	OUT	32	data_sram 读出的数据
out_ms_valid	OUT	1	将 ms_valid 的值传递给 ID 模块

3、功能描述

将从 EXE 模块获取的访存指令相应的执行。确定是否有访存指令，并将相应指令的数据和前阶段传递的数据控制信号传通过总线在下一时钟周期更新给 WB 模块。

（七）重要模块 7 设计：访存阶段（WB_stage）模块

1、工作原理

将从 MEM 模块获取的写回指令相应的执行。确定是否有写回指令，并进行相应的操作。同时，该模块将 PC、寄存器堆写使能、地址、和写回结果传递给 debug 模块，用于调试 CPU 的正确性。

表 2.5 WB_stage 接口定义

名称	方向	位宽	功能描述
clk	IN	1	时钟信号
reset	IN	1	复位信号
ws_allowin	OUT	1	WB 模块允许接受 MEM 传值
ms_to_ws_valid	IN	1	MEM 模块可以向 WB 模块传值
ms_to_ws_bus	IN	70	MEM 模块向 WB 模块传递数据
ws_to_rf_bus	OUT	40	WB 模块向寄存器堆模块（通过 ID 模块）传递数据（包括 ws_gr_we、ws_valid 等）
debug_wb_pc	OUT	32	debug 显示 PC
Debug_wb_rf_wen	OUT	4	debug 显示寄存器堆写使能
Debug_wb_rf_wnum	OUT	5	debug 显示寄存器堆写地址
Debug_wb_rf_wdata	OUT	32	debug 显示寄存器堆写数据

2、功能描述

将从 MEM 模块获取的写回指令相应的执行。确定是否有写回指令，并进行相应的操作。同时，该模块将 PC、寄存器堆写使能、地址、和写回结果传递给 debug 模块，用于调试 CPU 的正确性。

（八）重要模块 8 设计：HI/LO 寄存器模块

1、工作原理

记录乘法指令生成的完整的 64 位乘积，以及除法指令生成的各 32 位的商和余数。

表 2.6 hi/lo 寄存器模块接口定义

名称	方向	位宽	功能描述
clk	IN	1	时钟信号
rd_hi	OUT	32	从 cp0 模块中读取 hi 寄存器的值
rd_lo	OUT	32	从 cp0 模块中读取 lo 寄存器的值
hi_we	IN	1	写 hi 寄存器使能
lo_we	IN	1	写 lo 寄存器使能
wd_hi	IN	32	写 hi 寄存器数据，本实验中为乘法指令生成结果的高 32 位以及除法指令生成结果的余数
wd_lo	IN	32	写 lo 寄存器数据，本实验中为乘法指令生成结果的低 32 位以及除法指令生成结果的商

2、功能描述

类似于寄存器堆 regfile 模块，实现对 hi、lo 寄存器的同步写和异步读。

三、实验过程（50%）

（一）实验流水账

10 月 18 日 20：00-20：30 阅读讲义

10 月 18 日 20：30-22：00 设计代码

10 月 19 日 10：00-11：30 继续实现代码

10 月 20 日 14：00-15：30 调试 bug

10 月 20 日 19：30-22：30 撰写实验报告

（二）错误记录

1、错误 1：trace 比对节点出现错误

（1）错误现象

运行仿真，比对 trace，发现 PC、wb_rf_wnum、wb_rf_wdata 值全都不一样，如图 3.1 所示：

```
[1097357 ns] Error!!!
    reference: PC = 0xbfc98e88, wb_rf_wnum = 0x02, wb_rf_wdata = 0x42ea0000
    mycpu    : PC = 0xbfc98ea4, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x00000000
```

图 3.1 当前对比节点不匹配

（2）分析定位过程

通过查看 trace 的对比节点，发现：在 PC=0XBFC98EA4 时，wb_rf_wnum、wb_rf_wdata 的值正确，而在 PC=0XBFC98E88 时，wb_rf_wnum、wb_rf_wdata 的值也正确。此时出现的错误应该为在不该进行 trace 对比时进行了 trace 对比，导致 reference 和 mycpu 出现了不匹配。通过查看波形，确认出错 PC 处对应的指令涉及到跳转指令，如图 3.2.

```
bfc98e80:   10000008    b    bfc98ea4 <n43_j_test+0x34>
bfc98e84:   00000000    nop
bfc98e88:   3c0242ea    lui v0,0x42ea
bfc98e8c:   24426edf    addiu   v0,v0,28383
bfc98e90:   0bf263ae    j    bfc98eb8 <n43_j_test+0x48>
bfc98e94:   00000000    nop
bfc98e98:   10000009    b    bfc98ec0 <n43_j_test+0x50>
bfc98e9c:   00000000    nop
bfc98ea0:   00000000    nop
bfc98ea4:   0bf263a2    j    bfc98e88 <n43_j_test+0x18>
```

图 3.2 相关汇编指令

分析指令，只有在通用寄存器写使能时，才会进行 trace 比对。因此通过查看通用寄存器写使能信号，如图 3.3，发现：在译码阶段 gr_we 信号缺少了~inst_j，同时发现 inst_bgez 写成了 isnt_bgez。

```
assign gr_we        = ~inst_sw & ~inst_beq & ~inst_bne & ~inst_jr & ~inst_sb & ~inst_sh
                    & ~inst_swl & ~inst_swr & ~isnt_bgez & ~inst_bgtz& ~inst_blez & ~inst_bltz ;
```

图 3.3 错误的 gr_we 代码

（3）错误原因

译码阶段 gr_we 信号缺少了~inst_j，同时发现 inst_bgez 写成了 isnt_bgez。

（4）修正效果

在 gr_we 信号的末尾加上“& ~inst_j”，并将“isnt_bgez”修改为 “inst_bgez”。修改后进行仿真，该问题被解决。

2、错误 2：PC 值出错

（1）错误现象

运行仿真，比对 trace，发现 PC 值出错，如图 3.4 所示：

```
    reference: PC = 0xbfc58948, wb_rf_wnum = 0x15, wb_rf_wdata = 0x00000000
    mycpu    : PC = 0xbfc58908, wb_rf_wnum = 0x02, wb_rf_wdata = 0xbe6e0000
------------------------------------------------------------------
```

图 3.4 PC 出错

（2）分析定位过程

如图 3.5，通过查看对应处的汇编指令，发现导致错误的原因应该是在 PC=0xbfc58934 处，应该实现跳转，而该设计的 CPU 没有出现跳转。

```
bfc5892c:   0501fff6    bgez    t0,bfc58908 <n44_bgez_test+0x18>
bfc58930:   00000000    nop
bfc58934:   10000004    b   bfc58948 <n44_bgez_test+0x58>
bfc58938:   00000000    nop
bfc5893c:   00000000    nop
bfc58940:   3c0328b8    lui v1,0x28b8
bfc58944:   346350c0    ori v1,v1,0x50c0
bfc58948:   24150000    li  s5,0
```

图 3.5 对应处的汇编指令

通过查看 ID 阶段对 BGEZ 指令的处理，发现：原来设计的代码中，对于 rs_value 和 0 作比较时，没有改成有符号数，导致比较结果全是 rs_value 恒大于等于 0。

```
assign rs_eq_rt    = (rs_value == rt_value);
assign rs_ge_zero  = (rs_value >= 0);
assign rs_gt_zero  = (rs_value >  0);
assign rs_le_zero  = (rs_value <= 0);
assign rs_lt_zero  = (rs_value <  0);
```

图 3.6 ID 阶段对应代码

（3）错误原因

rs_value 和 0 作比较时，没有改成有符号数

（4）修正效果

如图 3.7 代码所示，在源代码的基础上增加了$sigend（），修正完毕后，该问题得到解决。

```
assign rs_eq_rt   = (rs_value == rt_value);
assign rs_ge_zero = ($signed(rs_value) >= 0);
assign rs_gt_zero = ($signed(rs_value) >  0);
assign rs_le_zero = ($signed(rs_value) <= 0);
assign rs_lt_zero = ($signed(rs_value) <  0);
```

图 3.7 对 rs_value 增加了$sigend（）

3、错误 3：trace 比对写寄存器的值有误

（1）错误现象

运行仿真，比对 trace，在 PC 值为 0xbfc6e0a4 发生了写寄存器值错误，如图 3.8：

```
[1457007 ns] Error!!!
    reference: PC = 0xbfc6e0a4, wb_rf_wnum = 0x02, wb_rf_wdata = 0x0000000b
    mycpu    : PC = 0xbfc6e0a4, wb_rf_wnum = 0x02, wb_rf_wdata = 0xc83b0be0
```

图 3.8 wb_rf_wdata 出错

（2）分析定位过程

在 test.s 中找到对应指令为“lb v0,14241(t0)”，说明 lb 指令写值有误，观察 mycpu 的 wb_rf_wdata，发现这不可能是 lb 指令应该出现的结果，于是抓取 MEM 阶段的 load_type 信号到波形中，发现此时该信号为 9，在我们设置的宏定义中，load_type 只有 3bit，最多为 7，如下图：

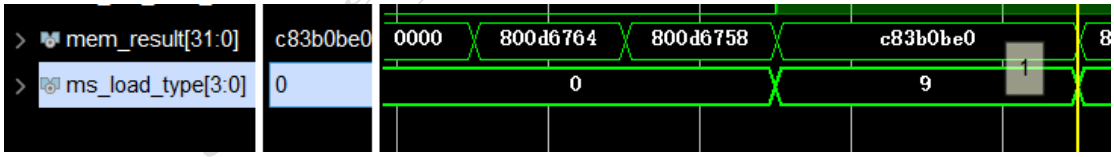

图 3.9 ms_load_type 值有误

```
`define LW_TYPE          3'b000
`define LB_TYPE          3'b001
`define LBU_TYPE         3'b010
`define LH_TYPE          3'b011
`define LHU_TYPE         3'b100
`define LWL_TYPE         3'b101
`define LWR_TYPE         3'b110
```

图 3.10 mycpu.h 中关于 load 指令类型的宏定义

立即想到是 ms_load_type 位宽错误，从波形图中也可以发现 ms_load_type 的位宽设置成了 4。

（3）错误原因

ms_load_type 位宽错误。

（4）修正效果

将 ms_load_type 位宽改为 3，再次仿真，如图 3.11，所有测试顺利通过。

```
----PASS!!!
$finish called at time : 1701755 ns : File "D:/share/UCAS_CDE/mycpu_verify/testbench/mycpu_tb.v" Line 267
run: Time (s): cpu = 00:01:49 ; elapsed = 00:01:38 . Memory (MB): peak = 954.410 ; gain = 148.707
```

图 3.11 测试通过

四、实验总结（可选）

本次实验也需要自己通过查看 MIPS 手册来添加一些新的指令。在设计代码的过程中，依旧需要保持细致，不然还是会出现例如变量名写错等问题。另外，由于本周时间有限，在 LWL/LWR 指令设计的选择上，只选择了可以复用上学期组成原理课数据通路的方案二，希望能在下次实验中，能够对这一块进行优化。