实验 6 报告

组员 1 ：袁欣怡　2018K8009929021
组员 2 ：郑旭舟　2018K8009908047
箱子号：10

一、实验任务

实验目的:

在已有简单的流水线 CPU 上添加运算类指令，包括算数逻辑运算类指令 (ADD, ADDI, SUB)，乘除运算类指令 (MULT, MULTU, DIV)，和配套的数据搬运指令 (MFHI, MFLO, MTHI, MTLO) .

主要操作:

配合 CPU 流水级的设计，根据需要调整已有的数据通路。

二、实验设计

（一）总体设计思路

1. 硬件结构设计

和之前的项目的不同之处在于：为了将乘除法的结果赋给HI/LO寄存器，增加了新的数据通路，同时增加了新的ip核，来实现有无符号的乘除法。具体设计框图如下图所示：

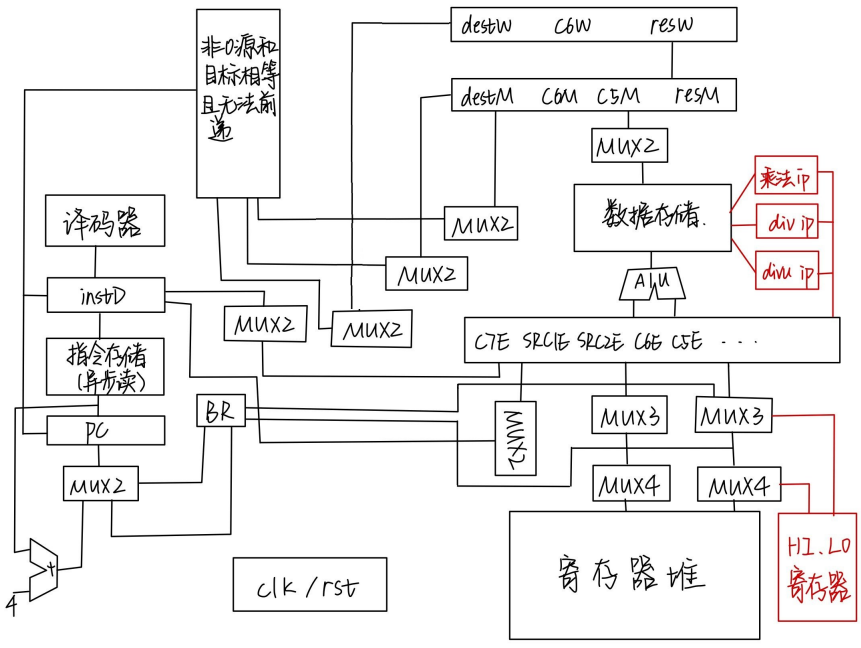

图 1

2. 五级流水线

如下图，本次实验没有对流水线进行改动，和之前保持一致。

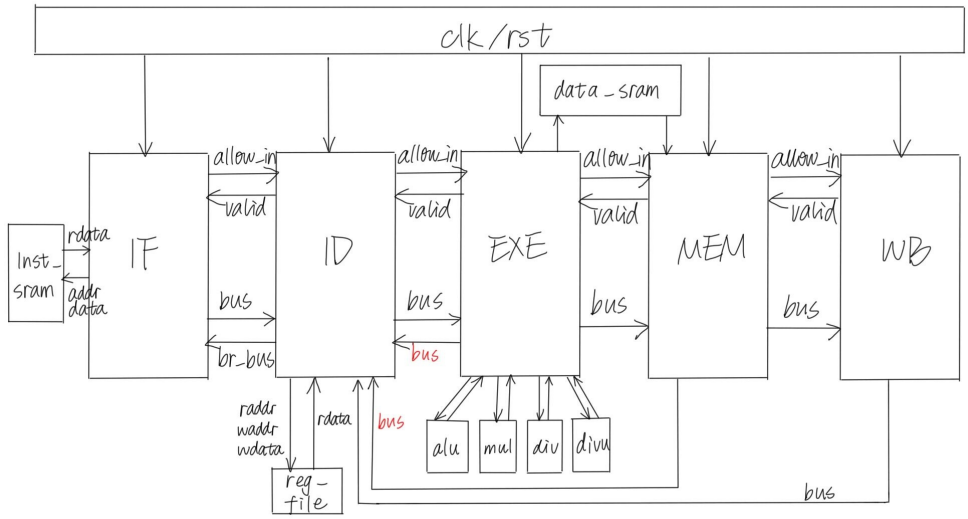

图 2

（二）重要模块 1： 算术逻辑单元ALU

1. 工作原理

将CPU中的运算处理进行模块化，方便外界调用。

2. 接口定义

表 1

名称	方向	位宽	功能描述
`alu_op`	IN	16	输入运算符，控制 ALU 中进行的运算种类
`alu_src1`	IN	32	运算数据 1
`alu_src2`	IN	32	运算数据 2
`alu_result`	OUT	32	运算结果

3. 功能描述

`alu_op` 采用 12 位 one-hot 编码方式，控制算术逻辑单元进行的运算操作。

此外，此次实验在 ID 模块中对 `alu_op` 赋值时考虑了新加指令的情况。

（三）重要模块 2： 译码模块 ID_stage

1. 工作原理

对取指模块 `IF_stage` 中取到的指令进行译码，分析指令的类型、 ALU 的运算类型、是否需要进行阻塞、PC是否需要跳转等。

同时，译码模块 `ID_stage` 还需要将译码后的数据传递给执行模块 `EXE_stage` ，和将写回模块 `WB_stage` 传来的数据写入寄存器堆。

2. 接口定义

表 2

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 模块允许接受来自 ID 的数据
`ds_allowin`	OUT	1	ID 模块允许接受来自 IF 的数据
`fs_to_ds_valid`	IN	1	IF 模块向 ID 模块传递的数据有效信号
`fs_to_ds_bus`	IN	64	从 IF 模块到 ID 模块的数据总线
`es_to_ds_bus`	IN	38	从 EXE 模块到 ID 模块的数据总线（前递）
`ms_to_ds_bus`	IN	37	从 MEM 模块到 ID 模块的数据总线（前递）
`ds_to_es_valid`	OUT	1	ID 模块向 EXE 模块传递的数据有效信号
`ds_to_es_bus`	OUT	143	从 ID 模块到 EXE 模块的数据总线
`br_bus`	OUT	34	从 ID 模块到 IF 模块的跳转信息数据总线
`ws_to_rf_bus`	IN	38	从 WB 模块到 ID 模块的写寄存器数据总线

3. 功能描述

译码模块 ID_stage 最主要的功能是对指令进行译码，并且判断属于何种指令。

对于数据冲突的解决，根据指令的操作将它们分类成 type_st 、type_rs 、type_rt 、type_nr 四类，方便根据指令需要的源寄存器情况判断是否需要进行阻塞。

本次实验添加的部分新指令在执行模块 EXE_stage 中不能复用已有的控制通路，所以我们修改了 `ds_to_es_bus` ，让新加的乘除法指令和数据搬运指令独立于已有的控制信号（如 `alu_op` 等）传到执行模块 EXE_stage ，以方便后续进行对 HI/LO 寄存器赋值等操作。

（四）重要模块 3： 执行模块 EXE_stage

1. 工作原理

执行从译码模块 ID_stage 获得的指令，并将得到的结果传递给访存模块 MEM_stage .

EXE_stage 在进行一般的算术逻辑运算时需要调用 ALU 模块，在进行乘除法时需要调用乘法器和除法器。

> 乘除法器模块的具体内容将在后文内容中详细介绍。

2. 接口定义

表 3

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	MS 模块允许接受来自 EXE 模块的数据
`es_allowin`	OUT	1	EXE 模块允许接受来自 ID 模块的数据
`ds_to_es_valid`	IN	1	ID 模块向 EXE 模块传递的数据有效信号
`ds_to_es_bus`	IN	145	从 ID 模块到 EXE 模块的数据总线
`es_to_ms_valid`	OUT	1	EXE 模块向 MEM 模块传递的数据有效信号
`es_to_ms_bus`	OUT	71	从 EXE 模块到 MS 模块的数据总线
`es_to_ds_bus`	OUT	38	从 EXE 模块到 ID 模块的数据总线（前递）
`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

3. 功能描述

EXE_stage 接收译码模块 ID_stage 传来的指令信号，执行对应的功能。

在本次实验中，我们新增了一些运算指令：

- 扩展的常规的算术逻辑运算：接着使用 ALU 模块处理；
- 乘除法指令：分别调用乘法器和除法器 IP 核处理；
- 和 HI / LO 寄存器相关的数据搬运指令：独立于通用寄存器的读写，单独为特殊寄存器设计该阶段内的读写逻辑。

值得注意的是，由于乘法结果的位数比其他运算多一倍，而除法运算需要输出余数和商，因此在 MIPS32 架构中，有 HI 和 LO 两个独立于通用寄存器的 32 位特殊寄存器来存储乘除法的运算结果，这和其他算术运算写回通用寄存器的操作是有区别的。因此，在 EXE_stage 中需要构造新的数据通路，以实现读写 HI / LO 这两个特殊寄存器。

（五）重要模块 4： 33 位乘法器

1. 工作原理

该乘法器可以计算 33 位有符号数的乘法，而我们的需求是计算 32 位的有符号数乘法和无符号数乘法。

通过符号位扩展，32 位有符号数乘法和 32 位无符号数乘法都可以用这个 33 位有符号乘法运算器实现，而不需要实例化两个 32 位乘法器，节省了硬件资源。

2. 接口定义

表 4

名称	方向	位宽	功能描述
`es_mult_a`	IN	33	乘数 a
`es_mult_b`	IN	33	乘数 b
`es_mult_result`	OUT	66	乘法器的运算结果

3. 功能描述

```
//符号位扩展
assign es_mult_a = {es_op_mult & es_alu_src1[32], es_alu_src1};
assign es_mult_b = {es_op_mult & es_alu_src2[32], es_alu_src2};
//进行乘法运算
assign es_mult_result = $signed(es_multer_a) * $signed(es_multer_b);
```

首先对两个乘数做对应的符号位扩展，然后用 `*` 符号调用乘法器 IP 核运算，乘法结果输出到 `es_mult_reslt` .

（六）重要模块 5： 32 位有符号除法器与 32 位无符号除法器

1. 工作原理

分别调用两个 IP 核 "divider" ，进行 32 位有/无符号数的除法。

2. 接口定义

表 5

名称	方向	位宽	功能描述
`aclk`	IN	1	时钟信号
`s_axis_dividend_tdata`	IN	32	被除数数据输入
`s_axis_dividend_tvalid`	IN	1	被除数输入有效信号
`s_axis_dividend_tready`	OUT	1	被除数输入应答信号
`s_axis_divisor_tdata`	IN	32	除数数据输入
`s_axis_divisor_tvalid`	IN	1	除数输入有效信号
`s_axis_divisor_tready`	OUT	1	被除数输入应答信号
`m_axis_dout_tdata`	IN	1	除法结果输出，高位为商，低位为余数
`m_axis_dout_tvalid`	OUT	1	除法结果数据有效信号

3. 功能描述

在 UI 界面添加有符号数除法器和无符号数除法器的 IP，然后在 EXE_stage 中按需求进行实例化。

该除法器在 8 个周期内完成除法。在除法指令进行到执行阶段时，进行运算的过程如下所列：

1. 如果除法运算还没开始，则同时输入 `除数` 和 `被除数` 数据，并且同时将 `输入数据对应的有效信号` 置为高电平；
2. 当两个 `输入数据对应的应答信号` 同时拉高时，说明握手成功，则将 `输入数据对应的有效信号` 清为低电平，以此保证一个除法操作只调用一次除法器，避免除法器和 CPU 对除法操作的认识出现分歧而出错；
3. 当 `除法结果数据有效信号` 拉高时，说明除法完成，此时 `m_axis_dout_tdata` 中输出有效的除法结果。

此外，在除法操作未完成时，将指令阻塞在当拍，避免产生 RAW 冲突。鉴于除法指令一般在程序中只占很小的一部分，这种阻塞的设计对整个 CPU 性能的影响并不大。

三、实验过程

（一）实验流水账

- 10 月 18 日上午，阅读讲义；
- 10 月 18 日下午和晚上，初步完成代码设计；
- 10 月 19 日下午，面向仿真波形调试；
- 10 月 9 日晚上，进一步完善设计并上板测试，撰写实验报告。

（二）错误记录

1. 错误 1 : PC 停滞在 0xbfc00004

(1) 错误现象

行为仿真时，`PC` 停在 0xbfc00004 处，tcl console 输出如下：

```
        [1082000 ns] Test is running, debug_wb_pc = 0xbfc00004
        [1092000 ns] Test is running, debug_wb_pc = 0xbfc00004
        [1102000 ns] Test is running, debug_wb_pc = 0xbfc00004
        [1112000 ns] Test is running, debug_wb_pc = 0xbfc00004
        [1122000 ns] Test is running, debug_wb_pc = 0xbfc00004
```

波形如图 3 所示：

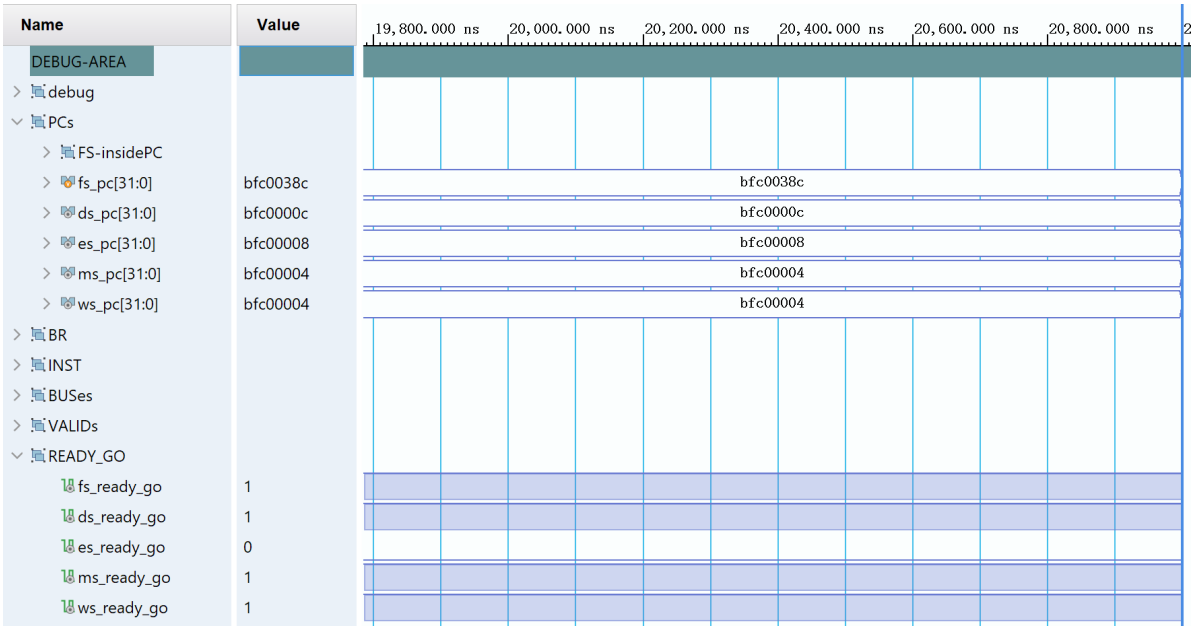

图 3

(2) 分析定位过程

如图 3，从波形可以推测出 `PC` 一直被阻塞是因为 `es_ready_go` 信号异常拉低。

检查 `es_ready_go` 的赋值逻辑：

```
assign es_ready_go = (|es_alu_op[11: 0])
                   | (|es_alu_op[15:12]) & es_hilo_we;
```

看起来没什么问题，但是实际上 `es_alu_op` 一共只有 12 位，在这里却出现了 `es_alu_op[15:12])`，这就是问题所在。

(3) 错误原因

一开始将乘除法集成在 ALU 内，但分析决定让 ALU 仍然保留为组合逻辑部件，因此优化设计时将乘除法调整到 ALU 外部。但在调整操作码的时候，误用了原设计的赋值。

(4) 修正效果

`es_ready_go` 信号的赋值修改如下：

```
assign es_ready_go = es_hilo_we | (~|{es_op_div,
                                      es_op_divu
                                     });
```

(5) 归纳总结

这是一个平凡的接线错误。

设计电路的过程中，如果对模块作了调整，一定要及时更新所有相关的接线和接口，否则非常容易出错。

2. 错误 2 : 接线缺失导致的逻辑运算出错

(1) 错误现象

行为仿真时，tcl console 报错输出如下：

```
----[ 696725 ns] Number 8'd28 Functional Test Point PASS!!!
--------------------------------------------------------------
[ 697077 ns] Error!!!
    reference: PC = 0xbfc53d28, wb_rf_wnum = 0x02, wb_rf_wdata = 0x00004c80
    mycpu    : PC = 0xbfc53d28, wb_rf_wnum = 0x02, wb_rf_wdata = 0x00000000
```

可以看到，这里是写回结果出了问题。

(2) 分析定位过程

查看反汇编文件 `test.s` ，得到出错当拍的指令是：

```
bfc53d28:   31025e89    andi    v0,t0,0x5e89
```

如图 4 ，检查该指令在执行阶段处的波形，发现：

1. 输入 ALU 模块的操作码正确，但输入 ALU 的操作数出错；
2. 操作数 2 `es_alu_src2` 明明应该是立即数，但 `es_src2_is_imm` 却错误地置为 0 .

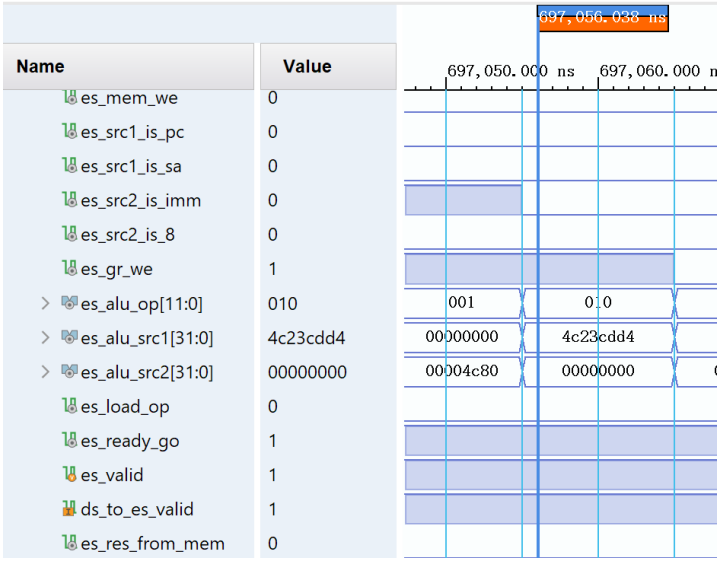

图 4

回顾指令集，发现算术运算和逻辑运算的立即数扩展分别是有符号扩展和无符号扩展，但是代码中仅在 `ID_stage.v` 有一行 `src2_is_uimm` 的赋值，且这个 wire 并没有被使用：

```
assign src2_is_uimm = inst_andi  | inst_ori | inst_xori;
```

这可能是一个设计时想要用到但忘记加入的信号，错误就出在这个未完成的设计上。

(3) 错误原因

缺少对操作数 2 的类型判断、扩展和立即数接入，导致操作数接入异常，进而使运算结果异常。

(4) 修正效果

在 ID_stage 中更改 `src2_is_imm` 的赋值，使它支持逻辑运算的立即数扩展：

```
assign src2_is_imm  = |{inst_addiu, inst_lui, inst_lw, inst_sw,
                        inst_addi, inst_slti, inst_sltiu,
                        inst_andi, inst_xori, inst_ori // uimm supported
                        };
```

在 EX_stage 中添加 `es_src2_is_uimm` 信号，并使用 `es_src2_is_imm` 和 `es_alu_op` 来判断 `src2` 是否使用无符号扩展立即数。

```
wire es_src2_is_uimm;
assign es_src2_is_uimm = es_src2_is_imm & (es_alu_op[4] // andi
                                          |es_alu_op[6] // ori
                                          |es_alu_op[7] // xori
                                          );
```

(5) 归纳总结

这个错误来自设计中未充分考虑的部分，因此以后进行功能添加时一定要做到“谋定而后动”，对每个功能都保证数据通路的完备性。

3. 错误 3 : 除法器相关的多处问题导致该组测试无法通过

(1) 错误现象

行为仿真时，tcl console 报错输出如下：

```
----[ 843365 ns] Number 8'd34 Functional Test Point PASS!!!
--------------------------------------------------------------
[ 843727 ns] Error!!!
    reference: PC = 0xbfc2c9ec, wb_rf_wnum = 0x15, wb_rf_wdata = 0x00000002
    mycpu    : PC = 0xbfc2c9ec, wb_rf_wnum = 0x15, wb_rf_wdata = 0x00000000
```

(2) 分析定位过程

查看反汇编文件 `test.s` ，得到出错当拍的指令是：

```
bfc2c9ec:   0000a812    mflo    s5
```

且该指令位于 `bfc2c9d0 <n35_div_test>` 段中。它的前一条指令是：

```
bfc2c9e8:   0109001a    div zero,t0,t1
```

如下图，观察波形，从 `es_op_div` 的波形可以看出，该除法指令只在 EXE_stage 被阻塞了一拍，这是不合理的：

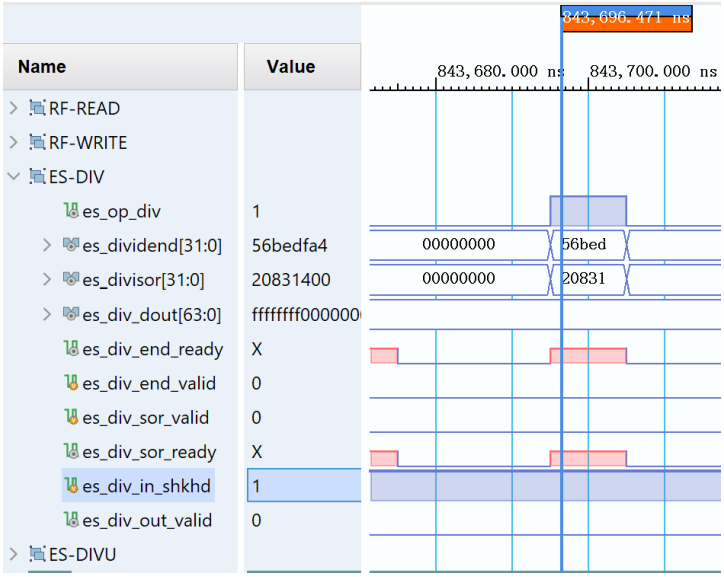

图 4

检查代码后，发现 `es_ready_go` 的阻塞条件写错了：（似乎是键盘无意识的误操导致的 typo ）

```
assign es_ready_go = es_hilo_we | (~{es_op_div,  // typo: "~" 后少了一个 "|"
                                    es_op_divu
                                   });
```

修改以后还是出错，但是可以阻塞在当条指令了：

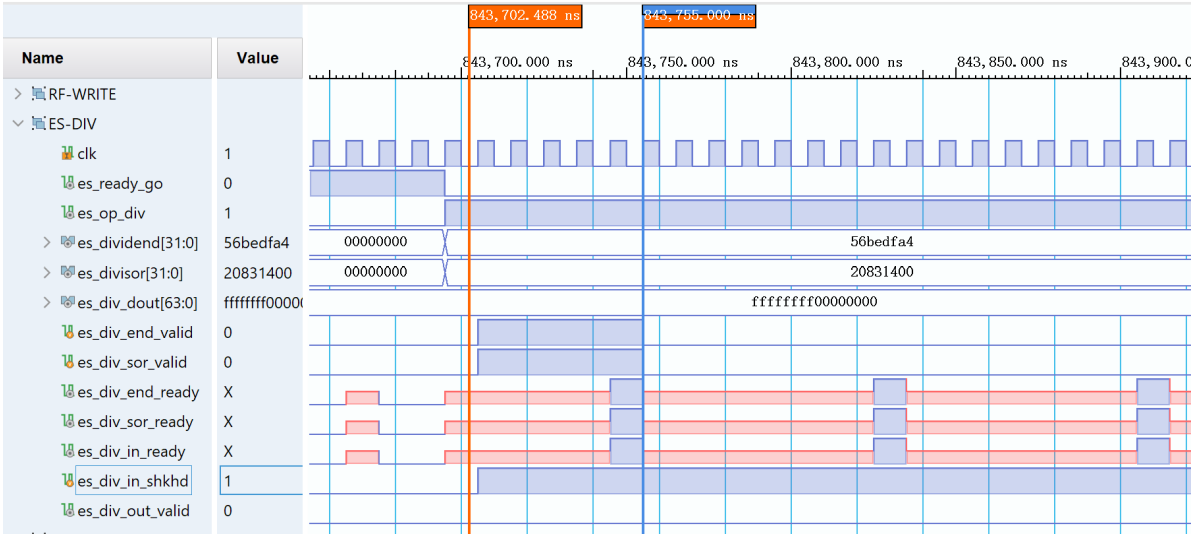

图 5

在波形比对的过程中，无意之中查对了指令集（如图 6 ），意外地发现商和余放反了：

- 除法器结果输出中，高位存商，低位存余；
- 寄存器 HI 中放余数，LO 中放商；

DIV **Divide Word**

31–26	25–21	20–16	15–6	5–0
SPECIAL 000000	rs	rt	0 00 0000 0000	DIV 011010
6	5	5	10	6

Format: `DIV rs, rt` **MIPS32, removed in Release 6**

Purpose: Divide Word

To divide a 32-bit signed integers.

Description: `(HI, LO) ← GPR[rs] / GPR[rt]`

The 32-bit word value in GPR *rs* is divided by the 32-bit value in GPR *rt*, treating both operands as signed values. The 32-bit quotient is placed into special register *LO* and the 32-bit remainder isplaced into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions:

If the divisor in GPR *rt* is zero, the arithmetic result value is **UNPREDICTABLE**.

Availability and Compatibility:

DIV has been removed in Release 6 and has been replaced by DIV and MOD instructions that produce only quotient and remainder, respectively. Refer to the Release 6 introduced 'DIV' and 'MOD' instructions in this manual for more information. This instruction remains current for all release levels lower than Release 6 of the MIPS architecture.

Operation:

```
q  ← GPR[rs]31..0 div GPR[rt]31..0
LO ← q
r  ← GPR[rs]31..0 mod GPR[rt]31..0
HI ← r
```

图 6

接着，又意外地发现用来暂存结果的 `es_alu_hi_res` 和 `es_alu_lo_res` 未声明，在仿真中自动置位宽为 1，但实际上这两个结果的位宽显然是 32：

```
assign es_alu_hi_res = {32{es_op_mult|es_op_multu}} & es_multer_result[63:32]
                     | {32{es_op_div              }} & es_div_dout[31:0]
                     | {32{es_op_divu             }} & es_divu_dout[31:0];
assign es_alu_lo_res = {32{es_op_mult|es_op_multu}} & es_multer_result[31:0]
                     | {32{es_op_div              }} & es_div_dout[63:32]
                     | {32{es_op_divu             }} & es_divu_dout[63:32];
```

修改之后，发现当拍读写正确，但是仍然报错，且 `PC` 跳到了 `bfc2c9bc <inst_error>`.

再次核对反汇编和波形行为，发现用来标识除法器输入信号握手成功的 `es_div_in_shkhd` 信号未在一次除法结束后清空，导致 CPU 运行全程只能运行一个有效的除法指令。

返回检查代码，发现确实没有为 `es_div_in_shkhd` 信号设计完备的清空条件：

```
// div input sending valid
always @(posedge clk ) begin
    ...
    else if (~es_div_in_ready & ~es_div_in_shkhd) begin
        es_div_end_valid <= es_op_div;
        es_div_sor_valid <= es_op_div;
        es_div_in_shkhd  <= 1'b1     ;  // not reset after div
    end
    ...
end
```

分析预期的 CPU 行为，选择 `es_div_dout_valid` 信号作为判定清空 `es_div_in_shkhd` 信号的条件：

```
always @(posedge clk ) begin
    ...
    else if (es_div_in_shkhd & es_div_out_valid) begin // [FIXED] set shkhd =
1'b0
        es_div_in_shkhd <= 1'b0;
    end
end
```

修改结束后，该组测试能够正常跑通。

(3) 错误原因

这个错误涉及以下问题：

1. 阻塞信号的赋值逻辑错误；
2. 除法指令数据输出接线错误；
3. 暂存结果的 wire 未声明/位宽错误；
4. 一次除法操作完成后，握手信号未及时清空；

(4) 修正效果

修正完成后，该组测试 PASS .

```
----[ 912035 ns] Number 8'd35 Functional Test Point PASS!!!
```

(5) 归纳总结

这个错误的类型是接线错误和设计不完备，今后在硬件设计时，尤其是涉及握手信号的设计中，一定要对电路的行为进行完备的设计。

4. 错误 4 : 乘法结果有误，出现波形为 X

(1) 错误现象

行为仿真时，tcl console 报错输出如下:

```
----[1011545 ns] Number 8'd36 Functional Test Point PASS!!!
--------------------------------------------------------------
[1011907 ns] Error!!!
    reference: PC = 0xbfc57a4c, wb_rf_wnum = 0x15, wb_rf_wdata = 0x0a20a480
    mycpu    : PC = 0xbfc57a4c, wb_rf_wnum = 0x15, wb_rf_wdata = 0xxxxxxxxx
```

(2) 分析定位过程

定位到该处波形，发现 wire 类型的变量赋值时接线错误:

```
assign es_mult_a    = {es_op_mult & es_alu_src1[32], es_alu_src1};
assign es_mult_b    = {es_op_mult & es_alu_src2[32], es_alu_src2};
```

(3) 错误原因

wire 类型的变量 `es_mult_a` 和 `es_mult_b` 赋值时接线错误。

(4) 修正效果

将上述部分改为:

```
assign es_mult_a    = {es_op_mult & es_alu_src1[31], es_alu_src1}; // 32->31
assign es_mult_b    = {es_op_mult & es_alu_src2[31], es_alu_src2};
```

修改后仿真，即通过该组测试:

```
----[1065945 ns] Number 8'd39 Functional Test Point PASS!!!
```

(5) 归纳总结

这个错误的类型是接线错误，接线时应该验证等号两边的信号的正确性。

5. 错误 5 : 总线位宽宏定义出错导致 MTLO 未拉起

(1) 错误现象

行为仿真时，tcl console 报错输出如下：

```
----[1065945 ns] Number 8'd39 Functional Test Point PASS!!!
--------------------------------------------------------------
[1066277 ns] Error!!!
    reference: PC = 0xbfc56bc0, wb_rf_wnum = 0x02, wb_rf_wdata = 0x08fc0000
    mycpu    : PC = 0xbfc56bc0, wb_rf_wnum = 0x02, wb_rf_wdata = 0x00000000
```

(2) 分析定位过程

查看反汇编文件，对应的指令是：

```
bfc56bbc:   01000013    mtlo    t0
bfc56bc0:   00001012    mflo    v0
```

如图，观察波形，发现前一拍的 MTLO 指令操作码未拉起：

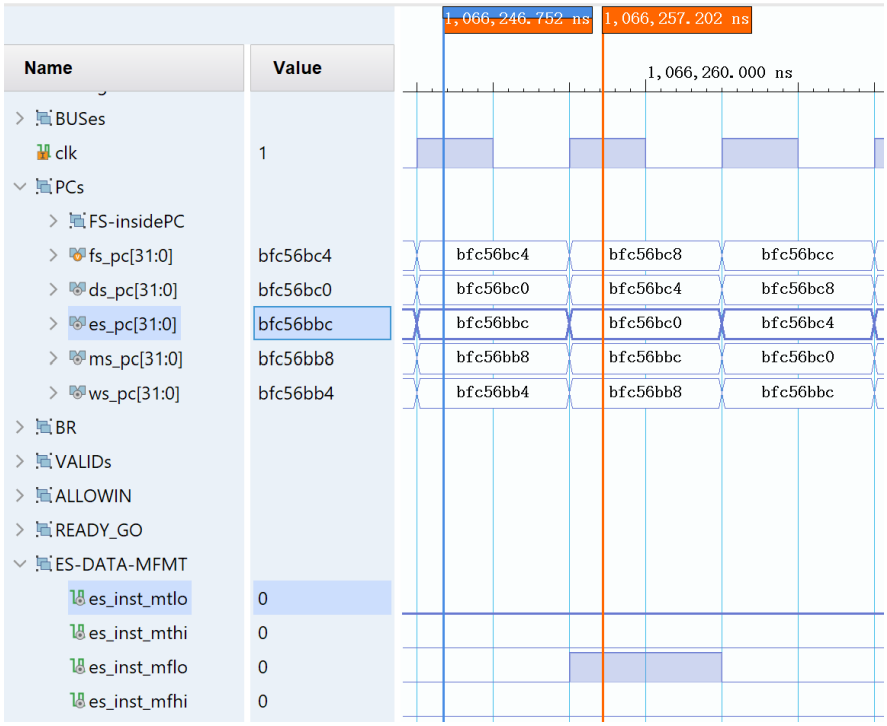

图 7

如图，检查对应数据总线的波形，发现它没有传递 MTLO 指令的第 143 位：

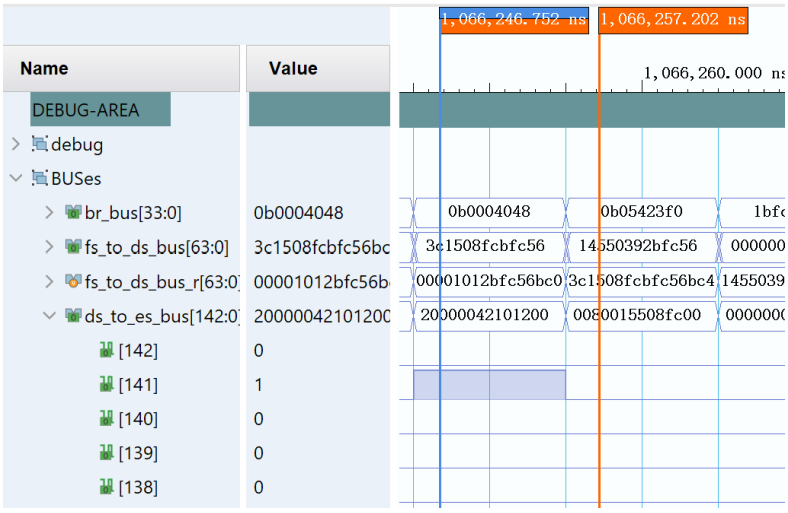

图 8

对照了注释，发现总线位宽宏定义出错：

```
    `define DS_TO_ES_BUS_WD 142
```

(3) 错误原因

总线位宽宏定义出错，导致 MTLO 指令的操作码未正常传进执行阶段。

(4) 修正效果

将宏定义位宽修改为正确的值

```
    `define DS_TO_ES_BUS_WD 143
```

然后进行仿真测试，显示 All PASS：

```
----[1089505 ns] Number 8'd42 Functional Test Point PASS!!!
==============================================================
Test end!
----PASS!!!
$finish called at time : 1089965 ns : File
"C:/labs_ca/ucas_ca_lab_2020/lab6/CPU_CDE/mycpu_verify/testbench/mycpu_tb.v"
Line 267
run: Time (s): cpu = 00:01:51 ; elapsed = 00:01:21 . Memory (MB): peak =
1680.363 ; gain = 6.301
```

(5) 归纳总结

接线定义不清晰；每次微调总线时，都应该对照注释仔细查对。

6. 错误 6 : 多驱动导致的 Implementation Failed

(1) 错误现象

预备生成 Bitstream 前 Implementation Failed，报错信息中提示 `es_div_end_ready` 和 `es_div_sor_ready` 等信号出现多驱动。

(2) 分析定位过程

检查代码中含 `es_div_end_ready` 和 `es_div_sor_ready` 等信号的部分，发现对这些从除法器接出的线，代码中错误且多余地用 `es_op_div` 等信号进行了赋值。

(3) 错误原因

重复对同一信号赋值，导致多驱动。

(4) 修正效果

删除错误且不必要的赋值逻辑 RTL 代码后，Implementation 可以正常进行了。

> 此外，在修改完成之后，之前的调试中看到的 `es_div_end_ready` 波形里莫名其妙的 X（见图 4 ）也不见了，想来也是多驱动的问题，只不过因为错得运气好，恰好巧妙地躲在了仿真测试的盲区。

(5) 归纳总结

对信号的赋值要在清醒并且清楚信号定义的情况下进行，多驱动这种低级错误不可取。

一般来说多驱动的错误分为两种，一种是 typo 造成错误赋值，另一种是设计有问题：

- 在设计构思阶段要完备地思考，规避后者；
- 在代码阶段要清醒地工作，规避前者。

7. 错误 7 : 赋值位宽

(1) 错误现象

> 不影响仿真测试。

在 Bitstream 生成过程中，注意到 warning 里面有一个地方显示“需要赋值的地方可能位宽不足”。

(2) 分析定位过程

根据 Warning 指示的路径，发现 `es_mult_res` 位宽出错，本应该是 `[65:0]` ，也就是 66 位，但代码中却写了 `[66:0]` .

(3) 错误原因

`es_mult_res` 信号的位宽出错。

(4) 修正效果

将位宽改正为正确的值，再次仿真后不再出现该 Warning .

(5) 归纳总结

在接线时，应先仔细对照接口宽度等信息再进行赋值。

四、实验总结

无。