实验 6 报告

学号: 2018K8009929043 2018K8009929035
姓名: 曾冕 张翔雨
箱子号： 06

一、实验任务（10%）

本实验主要在阻塞-数据前递流水线的基础上添加了算术逻辑类指令和乘除运算类指令及其配套的数据搬运指令，其中实现乘除运算类指令主要调用了 Xilinx IP 实现，同时对阻塞的状况进行了更新。

二、实验设计（40%）

（一）总体设计思路

本次的 CPU 在 lab5（阻塞和数据前递）的基础上加入了对新的指令的支持。本次主要涉及的都是运算类指令，所以对于流水线总体来说变动不大，主要集中于个别模块内部信号的更新和对应的总线位宽更新。涉及到更新的模块有：ID 模块，EXE 模块，ALU 模块和总线位宽头文件

1. ID 模块设计思路：首先根据指令特征译码，因为使用 one-hot 编码方式所以需要更改 alu 位宽以增加新增添的指令，对于 mfhi,mflo,mthi,mtho 等指令布局出单独的总线信号，传递到 EXE 模块，具体实现方式在分模块中叙述
2. EXE 模块设计思路：根据新的 ds_to_es_bus 拆分出相应信号，在 src2 中单独进行 0 拓展。在本模块中定义了 cp0 的两个寄存器，完成寄存器的更新，并同时在 EXE 模块中调用了 ip_catalog 的除法器和除法器中的握手信号处理工作。
3. ALU 模块：根据额外的 one-hot 编码增加乘法指令和 ip_catalog 中乘法器的简单调用，同时使用了小技巧只调用一个乘法器就完成了有符号无符号两种乘法的实现。

最后的总体涉及思路如下图 1 所示。

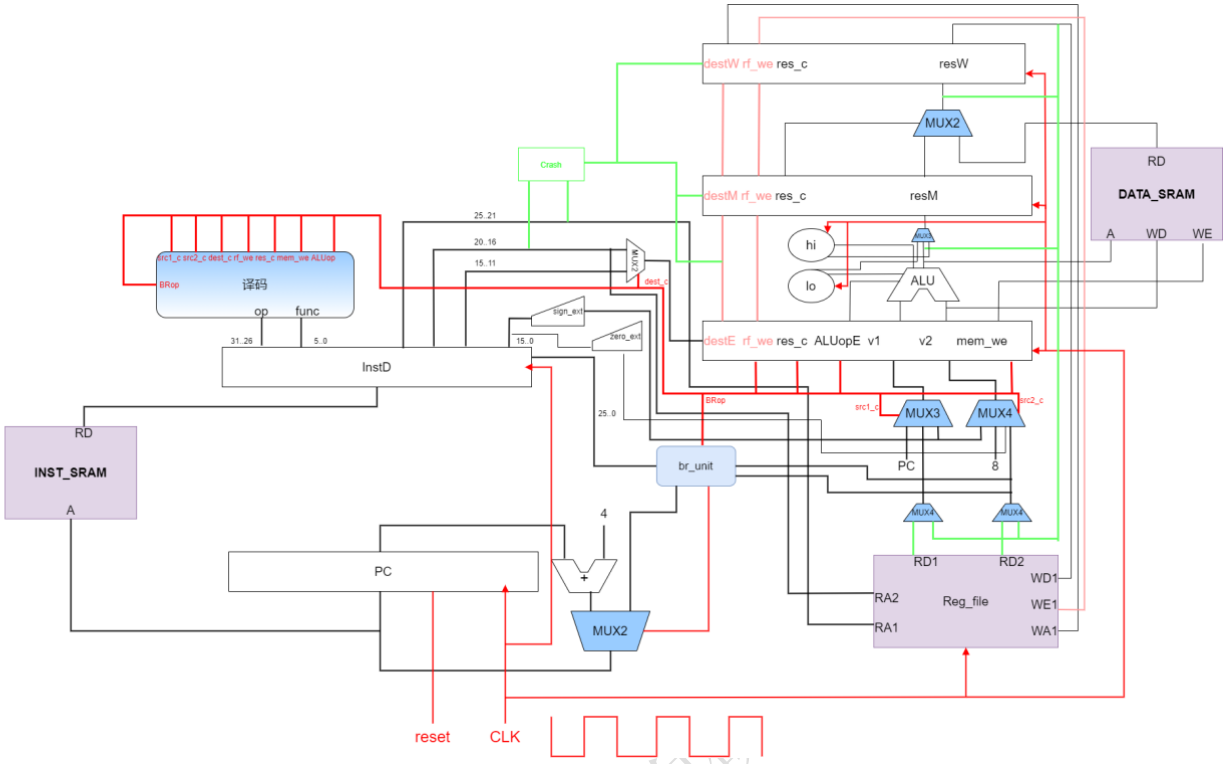

图 01: 整体数据通路设计

（二）重要模块 1 设计：ID 模块

1、工作原理

根据输入指令，通过复杂的内部信号处理输出对应的信号，控制下一步各分模块的控制逻辑。

Lab6 中，更改了 ds_to_es_bus 的位宽，传输了更多的内容，具体内容见功能描述

2、接口定义（仅展示 lab6 更新）

名称	方向	位宽	功能描述
ds_to_es_bus	OUT	145	到 exe_stage 的数据（包含增添的乘除法信号等）

3、功能描述

总线修改的信号如下图 3 所示：

```
assign {es_dest_lo,       //144:144 Lab6 Update
        es_dest_hi,       //143:143 ..
        es_res_from_hi,   //142:142 ..
        es_res_from_lo,   //141:141 ..
        es_src2_is_zimm,  //140:140 ..
        es_alu_op      ,  //139:124 Lab6 Update end here
```

图 02 ： 总线中新增/修改的信号

其中， es_dest_lo/hi 对应的是 mtlo/mthi 指令的信号， es_res_from_lo/hi 对应的是 mflo/mfhi 指令的信号， es_src2_is_zimm 是进行 0 扩展的信号（对应 ANDI、ORI 和 XORI 指令）

Es_alu_op 是根据额外的乘除法指令重新定义为 16 位宽的的 one-hot 编码指令

（三）重要模块 2 设计：EXE 模块（仅描述 lab6 更新）

1、工作原理

Lab6 中，EXE 内部完成了除法器的调用和 ALU 的调用（除法器的调用独立于 ALU），除法器握手信号的生成以及 cp0 寄存器的更新

2、接口定义

名称	方向	位宽	功能描述
ds_to_es_bus	IN	145	到 exe_stage 的数据（包含增添的乘除法信号等）

3、功能描述

外部信号输入输出变化不大，但是由于需要完成 cp0 寄存器的更新和除法器的调用，内部接口变化较大，下面分别介绍除法器握手信号的处理和 cp0 寄存器的处理。

除法器握手信号的处理代码以及对应的例化关系如下图 4 和 5 所示（仅展示有符号除法部分）：

```
always @(posedge clk)
begin
    if(reset)
    begin
        div_src_valid  <= 1'b0;
    end
    else if (div_src_valid && div_divisor_tready & div_dividend_tready)
    begin
        div_src_valid  <= 1'b0;
    end
    else if (es_allowin & ds_to_es_valid )
    begin
        div_src_valid  <= ds_to_es_bus[138];
    end
end
```

图 03：除法器的握手信号

```
mydiv u_mydiv(
      .aclk (clk),
      .s_axis_divisor_tvalid (div_src_valid),
      .s_axis_divisor_tready  (div_divisor_tready),
      .s_axis_divisor_tdata  (es_rt_value),
      .s_axis_dividend_tvalid (div_src_valid),
      .s_axis_dividend_tready (div_dividend_tready),
      .s_axis_dividend_tdata (es_rs_value),
      .m_axis_dout_tvalid (div_dout_tvalid),
      .m_axis_dout_tdata (div_result)
    );
```

图 04：除法器的例化关系

在处理握手信号的时候，由于必须要保证两个操作数有效信号同时变化，所以这里使用了同一个信号作为 tvaid 的例化输入，保证成功握手的同时减少了内部信号，这里还可以看到用的是总线中的数据，而不是对应的

aluop 的除法信号，详细原因在错误原因中解释。

当复位信号有效时，操作数 valid 信号拉低，为保证当除法指令信号以及两个操作数到达时 valid 信号能及时拉高，这里采用与 ds_to_es_bus_r 相同的时序赋值逻辑，当信号握手成功时 valid 信号拉低。

Cp0 寄存器的处理代码如下图 6 所示：

```
always @(posedge clk)
begin
    if (reset)
    begin
        cp0_hi <= 32'd0;
        cp0_lo <= 32'd0;
    end
    else if (es_alu_op[12]||es_alu_op[13])
    begin
        cp0_hi <= es_alu_hi_result;
        cp0_lo <= es_alu_lo_result;
    end
    else if (div_dout_tvalid && es_alu_op[14])
    begin
        cp0_hi <= div_result[31:0];
        cp0_lo <= div_result[63:32];
    end
    else if (divu_dout_tvalid && es_alu_op[15])
    begin
        cp0_hi <= divu_result[31:0];
        cp0_lo <= divu_result[63:32];
    end
    else if (es_dest_hi)
    begin
        cp0_hi <= es_rs_value;
    end
    else if (es_dest_lo)
    begin
        cp0_lo <= es_rs_value;
    end
end
```

图 05 ： cp0 寄存器的处理代码

当检测到乘法指令有效的时候将乘法结果写入，除法输出指令有效的时候完成除法结果的更新，当 mthi/lo 指令有效的时候将 rs 寄存器堆的内容写入 cp0_hi/lo 寄存器中。

相应的 当执行 mfhi/lo 指令的时候也要对 es_final_result 进行相应修改如下图 07 所示：

```
assign es_final_result = es_res_from_lo ? cp0_lo :
                         es_res_from_hi ? cp0_hi :
                         es_alu_result;
```

图 06 ： 读取 cp0 寄存器内容

（四）重要模块 3 设计：ALU 模块

1、工作原理

根据输入的 14 位编码进行相应运算（这里由于除法是在 EXE 模块调用的，所以传入 alu 模块的编码只有 14 位）

2、接口定义（仅展示 lab6 更新）

名称	方向	位宽	功能描述
ds_to_es_bus	IN	14	输入到 ALU 的控制代码
mult_hi_result	OUT	32	输出到 cp0_hi 寄存器的数据
mult_hi_result	OUT	32	输出到 cp0_lo 寄存器的数据

3、功能描述

重点描述 SRL/SRA 和 MULT/MULTU 指令的实现方式，简单的位运算部分略去。

SRL/SRA 指令实现方式如下图 08 所示：

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

图 07：SRL/SRA 实现方式

既先扩展为 64 位的数，再取后 32 位的方式，避免了选择器的加入。

MULT/MULTU 指令实现方式如下图 09 所示：

```
// Mult Result
assign mult_src1 = op_mult ? {{alu_src1[31]},alu_src1} : {{1'b0},alu_src1} ;
assign mult_src2 = op_mult ? {{alu_src2[31]},alu_src2} : {{1'b0},alu_src2} ;
assign mult_result = $signed(mult_src1) * $signed(mult_src2);
assign mult_hi_result = mult_result[63:32];
assign mult_lo_result = mult_result[31:0];
```

图 08：MULT/MULTU 实现方式

这里用了个小 ruick，先对立即数进行判断，如果为有符号乘法，则符号位扩展一位，无符号乘法则进行 0 扩展，最后用这两个 33 位的数进行运算，取乘法结果的前 32 位和中 32 位作为结果，避免了调用两个乘法器，减少了硬件开销。

三、实验过程（50%）

（一）实验流水账

10.13 晚上　小组成员完成了 ID 状态的指令译码以及简单运算指令控制信号赋值

10.14 晚上　小组成员完成了 EXE 及 ALU 状态的更新，添加了乘除法指令以及相关数据搬运指令。通过测

试。

10.17　　开始写实验报告，大约 3 小时半完成

（二）错误记录

1、错误 1：1.执行阶段除法操作数有效信号不拉高

（1）错误现象

仿真不停止，卡在执行阶段，如下图 10 所示：

```
[ 842000 ns] Test is running, debug_wb_pc = 0xbfc13a98
----[ 843375 ns] Number 8'd34 Functional Test Point PASS!!!
        [ 852000 ns] Test is running, debug_wb_pc = 0xbfc2c9e4
        [ 862000 ns] Test is running, debug_wb_pc = 0xbfc2c9e4
        [ 872000 ns] Test is running, debug_wb_pc = 0xbfc2c9e4
        [ 882000 ns] Test is running, debug_wb_pc = 0xbfc2c9e4
        [ 892000 ns] Test is running, debug_wb_pc = 0xbfc2c9e4
        [ 902000 ns] Test is running, debug_wb_pc = 0xbfc2c9e4
        [ 912000 ns] Test is running, debug_wb_pc = 0xbfc2c9e4
        [ 922000 ns] Test is running, debug_wb_pc = 0xbfc2c9e4
        [ 932000 ns] Test is running, debug_wb_pc = 0xbfc2c9e4
        [ 942000 ns] Test is running, debug_wb_pc = 0xbfc2c9e4
        [ 952000 ns] Test is running, debug_wb_pc = 0xbfc2c9e4
        [ 962000 ns] Test is running, debug_wb_pc = 0xbfc2c9e4
        [ 972000 ns] Test is running, debug_wb_pc = 0xbfc2c9e4
        [ 982000 ns] Test is running, debug_wb_pc = 0xbfc2c9e4
        [ 992000 ns] Test is running, debug_wb_pc = 0xbfc2c9e4
        [1002000 ns] Test is running, debug_wb_pc = 0xbfc2c9e4
        [1012000 ns] Test is running, debug_wb_pc = 0xbfc2c9e4
        [1022000 ns] Test is running, debug_wb_pc = 0xbfc2c9e4
        [1032000 ns] Test is running, debug_wb_pc = 0xbfc2c9e4
        [1042000 ns] Test is running, debug_wb_pc = 0xbfc2c9e4
        [1052000 ns] Test is running, debug_wb_pc = 0xbfc2c9e4
        [1062000 ns] Test is running, debug_wb_pc = 0xbfc2c9e4
```

图 09：仿真卡死在某个阶段

（2）分析定位过程

查找 lab CPU_CDE\soft\func_lab6\obj\test.S 反汇编文件发现这个 pc 执行党的是一条除法指令，且 PC 保持一个值大概率是握手信号处理不正确，对握手信号进行排查。

（3）错误原因

握手信号没有被正确处理，由于调用的 es_alu_op，这是个从 bus 过来的寄存器中读取的数据，会慢一拍，导致握手信号无法正确进行，如下图 11 所示：

```
always @(posedge clk)
begin
    if(reset)
    begin
        divu_src_valid  <= 1'b0;
    end
    else if (divu_src_valid && divu_divisor_tready & divu_dividend_tready)
    begin
        divu_src_valid  <= 1'b0;
    end
    else if (es_allowin & ds_to_es_valid)
    begin
        divu_src_valid  <= es_alu_op[15];
    end
end
```

图 10：错误代码示范：

（4）修正效果

将代码修改为 `div_src_valid  <= ds_to_es_bus[138]`
仿真正确运行，得到结果。

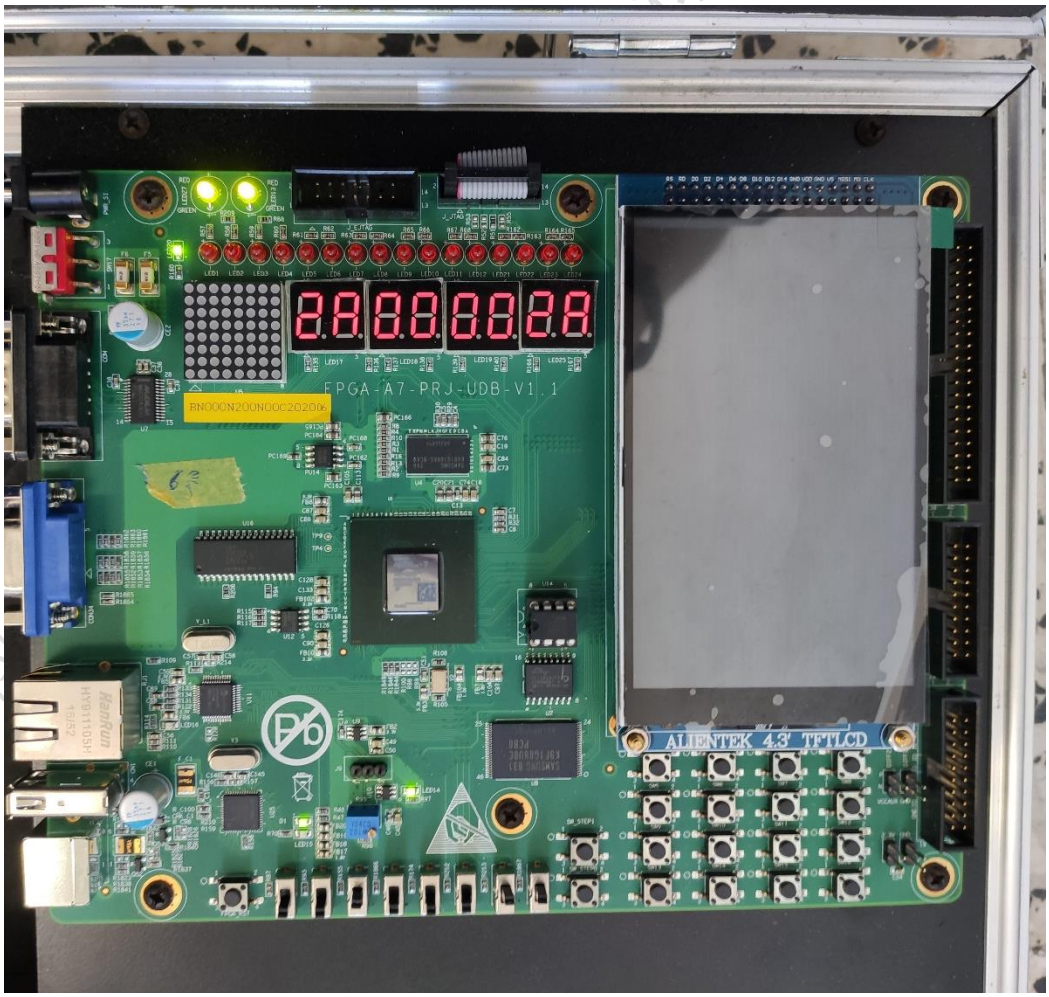

图 11 上板通过照片

四、实验总结（可选）

本次实验拿到任务后感觉要增加一倍的指令，还是比较令人头疼的，仔细阅读任务书后发现很多指令可以复用之前的数据通路，思路还是比较清晰的。难点主要在乘除法指令，乘法指令较为简单一些，只需在 ALU 中添加高位和低位结果的输出接口即可，内部添加乘法操作。除法要麻烦一点，需要理解除法 ip 各个接口的功能是什么，由于除法会导致 EXE 模块堵塞，也需要修改 ready_go 信号和结果数据，同时还需要添加 cp0 寄存器 hi 和 lo 的赋值逻辑，是本次实验的难点。不过经过这次实验之后，感觉对实验的代码框架理解更深了，也复习了时序逻辑的一些代码书写规范。