CompArch Lab2 Report

PB20111686 黄瑞轩

1 工作总结

本实验是基于助教给出的 RV32I Core 的 Verilog 代码框架补全缺失的模块，以实现具有五级流水功能的 RISC-V CPU，并通过给定的测试样例。

助教的框架已经完成了大部分的数据通路，接下来只需要按阶段完成模块内部设计即可。

1.1 第二阶段

由于第一阶段比较简单，故跳过。

1.1.1 ALU 模块

ALU 模块只需要根据 `parameters.v` 中给出的 `ALU_func` 列表进行编写。值得注意的有以下几点：

- 左右移指令的移动位数是 op2 的低 5 位
- 算术右移在 Verilog 中使用 `>>>` 来实现
- 使用 `wire signed [31:0]` 声明有符号线，在有符号比较时使用有符号线
- `NAND` 运算用于 CSR 指令，是 ~op1 & op2，而不是 ~(op1 & op2)

1.1.2 BranchDecision 模块

此模块主要根据比较结果决定是否分支，生成 br 信号。

值得注意的是，使用 `wire signed [31:0]` 声明有符号线，在有符号比较时需要使用有符号线。

1.1.3 DataExtend 模块

这个模块是在非字对齐的 Load 部分使用的（没有 ld, ldu 指令），`load_type` 指定 Load 的宽度，`addr` 指定 Load 部分在字中的位置。

值得注意的是，在有符号扩展中，我们使用的是 data 的最高位（符号位）；在无符号扩展中，直接补充 0。

1.1.4 ImmExtend 模块

这个模块是在 ID 阶段生成立即数所需的，只需要根据 `parameters.v` 中给出的 `imm_type` 列表进行编写。

1.1.5 NPCGenerator 模块

这个模块根据跳转信号产生下一个 PC 取值。

值得注意的是，因为 jal 信号在 ID 段就产生，br 和 jalr 信号在 EX 段才产生，所以 br、jalr 信号的优先级需要比 jal 高，这体现在 if-else 结构的顺序上。

1.1.6 ControlDecoder 模块

译码模块非常重要，虽然其写起来比较简单，就是根据不同的指令填写不同的信号。值得注意的有以下几点：

- JALR 是 I 型指令，其 `imm_type` 应该为 `ITYPE` 而不是 `JTYPE`
- 在译码 Branch、Load 等指令时，采用多级译码方式，简化译码模块编写
- 在译码 Store 指令时，`cache_write_en` 信号需要根据 DataCache 的逻辑来写，这个 4 位信号为 1 的位置指示了数据宽度
- 所有的信号在每一种情况下都要完备地赋值，否则可能会导致不同指令之间控制信号重叠，导致出错

1.1.7 Hazard 模块

Hazard 模块处理数据相关和控制相关，根据 Lab1 中的相关处理来实现。值得注意的有以下几点：

- 需要根据 `rst` 复位信号处理段间寄存器的 `bubble` 和 `flush`
- 采用静态预测分支不成功，如果实际分支成功，需要清空 ID 和 EX 段的指令，据此设置 `flush`
- 数据相关通过前递处理，load-use 型数据相关需要给流水线设置一个气泡

前递判断逻辑实现：

```
// generate op1_sel
always@(*) begin
    if (reg1_srcE == reg_dstM &&
        reg_write_en_MEM == 1 &&
        reg1_srcE != 0)
    begin
        // mem to ex forwarding, mem forwarding first
        op1_sel = 2'b01;
    end
    else if (
        reg1_srcE == reg_dstW &&
        reg_write_en_WB == 1 &&
        reg1_srcE != 0)
    begin
```

```
        // wb to ex forwarding
        op1_sel = 2'b10;
    end
    else begin
        op1_sel = 2'b00;
    end
end
```

load-use 产生气泡实现：

```
// load-use, 选择 cache
else if (
    wb_select &&
    (reg_dstE == reg1_srcD || reg_dstE == reg2_srcD))
begin
    bubbleD = 1;
    flushD = 0;
end
```

1.2 第三阶段

这一阶段实现的是 CSR 指令，通过阅读框架给出的数据通路可知，CSR 处理逻辑如下：

- 译码内容传到 CSR_EX 模块
- 在 EX 阶段完成 CSR 计算及写回

CSR_EX 是一个段间寄存器，很好写。CSR 寄存器文件可以仿照 General 寄存器文件编写。剩下的工作就是在译码模块加入 CSR 指令的控制信号。

值得注意的有以下几点：

- 由于在 EX 阶段要完成 CSR 寄存器读出、计算、写回，所以不能和 General 寄存器文件一样是下降沿写入，要改成上升沿写入，这样可以保证 read 的结果是 CSR 的旧值
- 需要根据已有数据通路的设定填写 CSR 译码信号

添加了 CSR 指令相关数据通路和部件的 pdf 随附。

2 问题与收获

2.1 遇到的问题

- 在写 NPCGenerator 时，忽略了输入信号 PC 已经是加过 4 的了，导致每次 PC 都 +8，出现问题
- 在写 ControllerDecoder 时，没有注意信号的完备性，导致有的指令旧信号被锁存，传递到了下一个指令，出现问题
- 没有仔细观察提供的数据通路，在译码时搞反了 `CSR_imm_or_reg` 的 0 和 1，出现问题
- 因为提供的 CSR 数据通路（EX 一段完成）和我的 Lab1 设想（和 R 型指令一样需要五段）差别较大，忘记寄存器的下降沿写入会导致 result 无法得到 CSR 旧值
- typo，将 funct3 写成了 opcode（Vivado 不检查位宽不匹配）

2.2 实验收获

- 第一次使用 Verilog 框架，学到了很多新的用法，主要是常量定义、文件读写等操作
- 在 COD 的基础上完善了相关处理和 CSR 指令，加深了对数据通路的认识
- 学会了使用 DrawIO 网站绘制数据通路
- 提升了使用 Vivado 进行硬件 debug 的能力

2.3 时间耗费

- 第二阶段：约一个下午
- 第三阶段：约一个晚上
- 报告：约 1 h

3 实验改进意见

- 段间寄存器可以写成一个模块，这样方便找信号、找连线
- 对 CSR 指令的处理方式（既计算又写回）可能会给 EX 阶段带来很大的时延，建议将 CSR 的计算和写回分段处理（这样又会导致数据相关，是要复杂些）
- 可以显式给 `imm_type` 指定一个 NOIMM 信号，有助于在没有立即数扩展的指令译码时增强语义信息
- 可以提供类似 iverilog、verilator 等轻量级仿真工具的框架或教程，只用仿真的话用 Vivado 显得比较臃肿