Verilog-lab2实验报告

梁峻滔 PB19051175

实验内容

阶段一

因完成了阶段二，跳过阶段一。

阶段二

基于助教所提供的代码框架，补全缺失的模块：

1. ALU

   根据`ALUfunc`的值选择对应的运算/操作执行，除ADD、SUB、AND、OR等常规运算外，添加两个操作用于后面CSR指令的执行. 对于有符号数的处理, 输入的 wire 型操作数默认是无符号的, 可以声明两个有符号的 wire 型变量:

   ```
   wire signed [31:0] op1_signed, op2_signed;
   assign op1_signed = op1;
   assign op2_signed = op2;
   ```

   在进行有符号数运算时使用`op1_signed`和`op2_signed`作为操作数即可.

   为执行CSR指令增加的两个操作是:

   ```
   `OP1: ALU_out = op1; // CSRRW和CSRRWI需要直接使用rs1寄存器的值
   `NAND: ALU_out = ~op1 & op2; // !!!注意这里不是真正的"与非"运算，而是对操作数1取反后再跟操作数2相与，因为要实现CSRRC指令要用到
   // 实现CSRRS的操作用OR运算即可
   ```

2. BranchDecision

   根据`br_type`的值选择对应的跳转条件, 根据跳转条件和`reg1`、`reg2`的值输出跳转控制信号. 对于有符号数的处理类似ALU中的处理.

3. ControllerDecoder

   根据取到的指令生成各个控制信号: jal, jalr, op1_src, op2_src, ALU_func, br_type, load_npc, wb_select, load_type, reg_write_en, cache_write_en, imm_type, CSR_write_en, CSR_zimm_or_reg.

4. DataExtend

   load指令有多种类型: LB、LH、LW、LBU、LHU，这些指令从Data Cache中取出的数据位宽不同，而寄存器都是32位的，所以在这些指令将数据取出来后需要再经过适当的位扩展再存到寄存器中. 这个模块就是根据指令译码时输出的 `load_type` 信号, 对从Data Cache中取出的数据进行相应的位扩展. 而输入 `addr` 则指示取出的有效数据是哪一段.

5. Hazard

   识别流水线中的数据相关、控制相关，控制数据转发(实现bypass和forwarding), 输出对应的bubble和flush信号来控制停顿和清空段间寄存器.

6. ImmExtend

   根据指令译码得到的`imm_type`对立即数进行扩展.

7. NPCGenerator

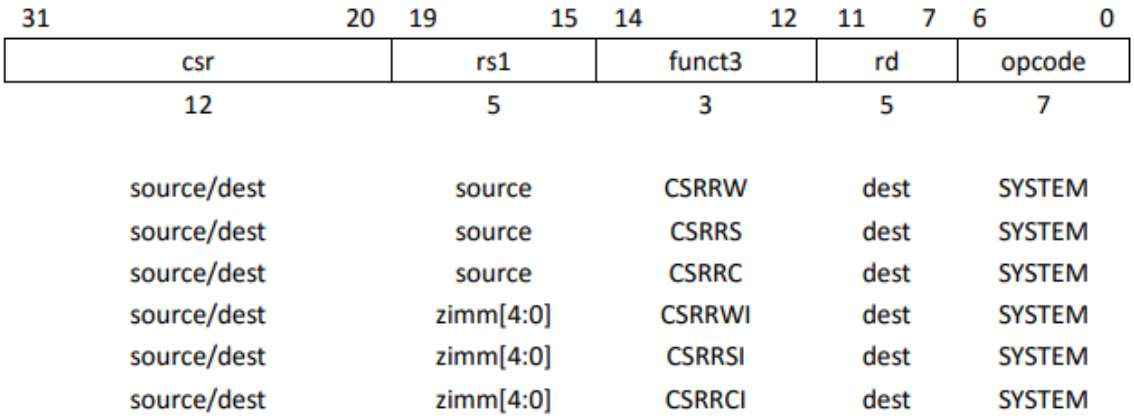
   根据跳转信号, 决定下一条将执行的指令的地址.

阶段三

在代码框架上添加CSR数据通路，补全模块内部代码。

理解CSR指令的功能

CSR指令格式：



`csr`字段指示要操作的CSR寄存器, 其他字段的含义类似普通指令.

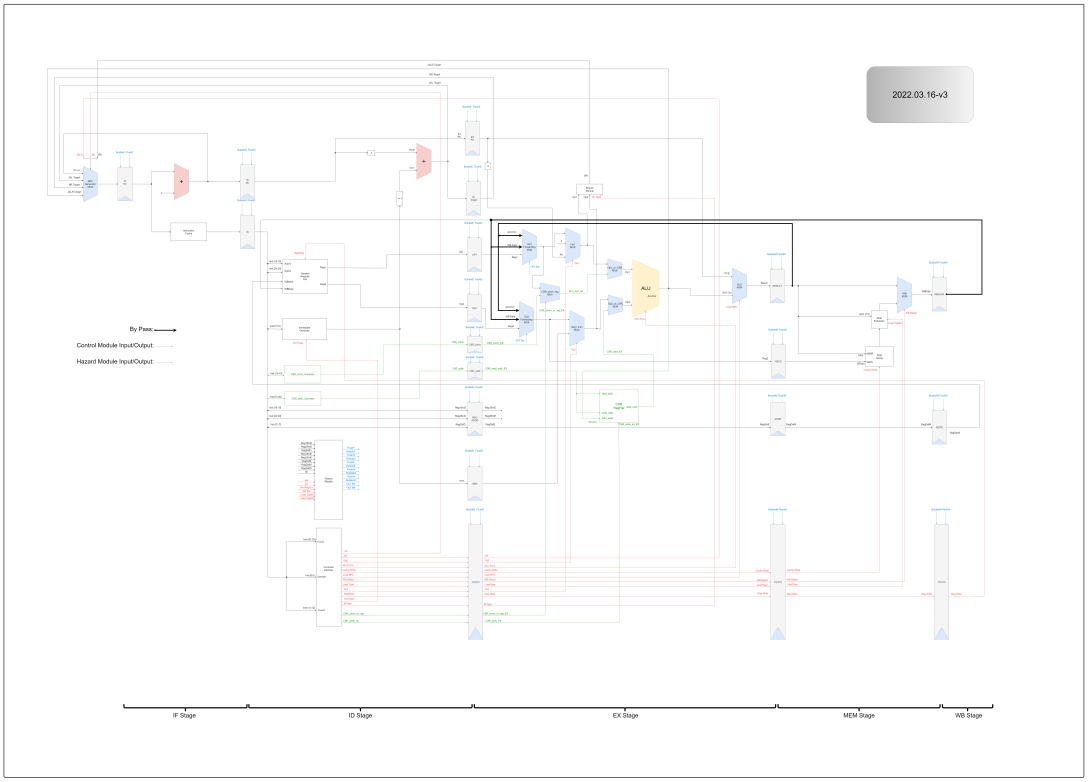
1. **CSRRW**(Atomic Read/Write CSR): 交换CSR寄存器和整数寄存器的值. 其中`csr`字段指定要交换值的CSR寄存器, 该指令读取指定CSR寄存器中的旧值, 将其零扩展到 XLEN 位(系统位数, RV32I中即32位), 然后写入`rd`中, 而`rs1`的值将被写入CSR寄存器中.
2. **CSRRS**(Atomic Read and Set Bit in CSR): 该指令读取指定CSR寄存器的旧值, 将其零扩展到 XLEN 位, 然后写入`rd`中. 同时要修改CSR寄存器中的值, `rs1`中的值指示CSR寄存器中的哪些位将被置为1: `rs1`中为1的位指示将CSR寄存器中对应位置为1, CSR寄存器中的其他位不受影响.
3. **CSRRC**(Atomic Read and Clear Bit in CSR): 该指令读取指定CSR寄存器中的旧值, 将其零扩展到 XLEN 位, 然后写入`rd`. 同时要修改CSR寄存器中的值, `rs1`的值指示CSR寄存器中哪些位将被置为0: `rs1`中为1的位指示将CSR寄存器中对应位置为0, 其他位不受影响.
4. **CSRRWI**, **CSRRSI**, **CSRRCI**的功能类似, 不同的是它们不是使用`rs1`寄存器中的值, 而是使用指令中位于`rs1`字段的、零扩展到 XLEN 位的5位立即数**zimm**.

CSR数据通路

CSR指令涉及的操作有:

1. 读和写通用寄存器: 可以直接使用原本的ID段和WB段实现.
2. 生成立即数zimm: 可以在ID段实现, 方便与`rs1`为寄存器时统一用同一个操作数传递寄存器传递给ALU.
3. 读和写CSR寄存器: 根据`Lab2/SourceCode/PART-code/CSR`目录下代码框架和`Lab2/SourceCode/PART-code/RV32ICore.v`的提示, CSR寄存器堆可以放在EX段. 在一条CSR指令中读和写的CSR是同一个, 在ID段译码时取出CSR寄存器地址, 传递到EX段用.
4. 根据整数寄存器或者立即数(rs1)的值将将CSR寄存器的某些位置1或置0: 可以在ALU增加该操作, 例如, 将`rs1`(寄存器值或扩展后的立即数)作为操作数1(这样的好处是可以直接在原来流水线的基础上实现, 不需要增加额外的寄存器部件), 将CSR寄存器值作为操作数2, 传递给ALU, ALU输出将CSR寄存器指定位修改后的结果.
5. 直接传递寄存器的值: **CSRRW**指令将`rs1`的值写入CSR寄存器中, 同时将CSR寄存器中原来的值写入`rd`中, 一种方案是加一个选择器来选择写回CSR寄存器的值是来自ALU计算结果还是来自ID段传递过来的寄存器1的值, 还有另一种方案是在ALU中新增一种直接传递操作数1的操作, 采用后者可以统一写回CSR寄存器的值的接口.

下面附上加上CSR数据通路后的RV32I-Core数据通路, 绿色部分为新添加的部分.



问题和建议总结

在代码框架的基础上补全模块相对容易很多, 很多模块的逻辑都比较简单, 但是需要注意很多细节和有一些地方要花一点时间理解, 比如考虑到所有指令, ALU需要支持哪些运算和操作; 每条指令的执行过程都需要理解, 才能写好控制器模块; DataExtend模块为什么需要一个2位的`addr`输入端口; CSR指令的功能和执行过程以及数据通路设计等等. 主要是ControllerDecoder、Hazard以及实现对CSR指令的支持会相对复杂. 实验过程比较花时间的也主要是这三部分代码的编写、画数据通路以及debug. 出现的bug不多, 都是变量声明方面的(低级)错误, 比如漏了声明位宽, 还有过于依赖代码自动补全插件导致变量名本身就写错了等等.