Lab2实验报告

PB17111623

范睿

实验目标

使用verilog 去实现RV32I 流水线 CPU 。

实现的指令集：

SLLI、SRLI、SRAI、ADD、SUB、SLL、SLT、 SLTU、XOR、SRL、SRA、OR、AND、ADDI、SLTI、SLTIU、XORI、ORI、ANDI、LUI、 AUIPC、JALR、LB、LH、 LW、LBU、LHU、SB、SH、SW、BEQ、BNE、BLT、BLTU、BGE、BGEU、JAL、CSRRW、CSRRS、CSRRC、CSRRWI、CSRRSI、CSRRCI

处理数据相关，flush和bubble

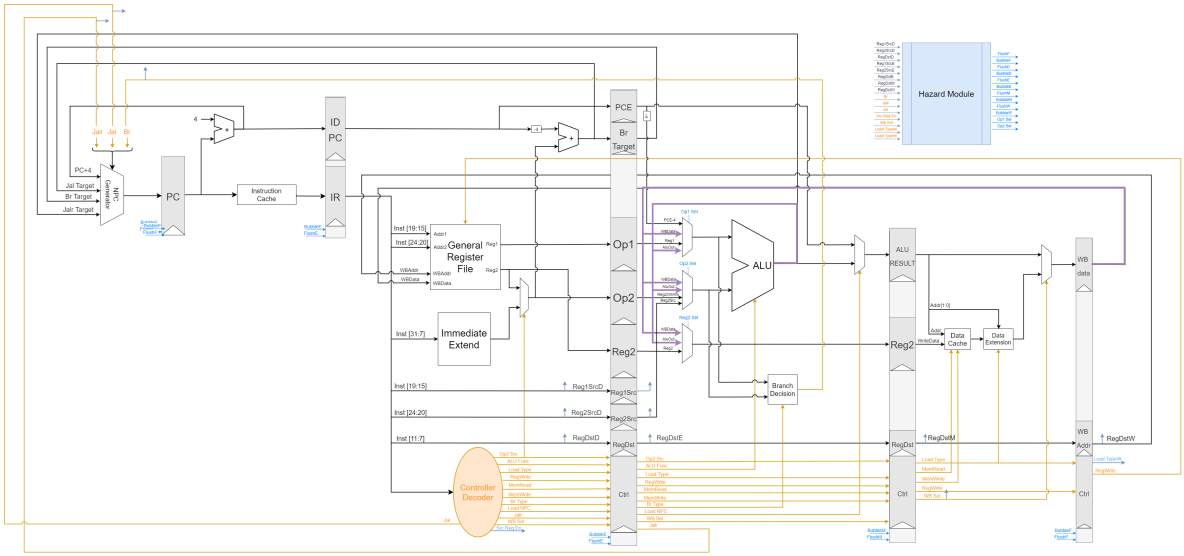
实验环境和工具

vivado 2018.2

实验内容和过程

阶段一

按照数据通路连线。



但是不连Hazard Module。

这一阶段的主要工作是写好ALU的逻辑和译码的逻辑。

Controller Decoder

Controller Decoder的作用是根据取到的指令inst生成各个阶段的控制信号。

- **jal**：当指令是jal指令时，jal为1，否则为0

- **jalr**：当指令是jalr指令是，jalr为1，否则为0
- **op2_src**：op2的选择信号，当指令是R类的运算指令时，op2_src为0，选择reg2；否则为1，选择立即数
- **ALU_func**：ALU的运算类型
  - 若指令是运算指令，ALU_func根据指令的运算类型变化
  - LUI指令的ALU_func是LUI，AUIPC指令的ALU_func是ADD
  - jalr指令的ALU_func是ADD
- **br_type**：分支类型
  - 指令是分支指令时，br_type置为对应类型
  - 指令不是分支指令是，br_type置为NOBRANCH
- **load_npc**：判断存入result寄存器的值是ALU_result还是npc
  - 当指令是jal和jalr时，load_npc为1，因为他们要将PC+4保存到rd中，其他指令load_npc都为0
- **wb_select**：判断存入WB data寄存器的值是result还是存储器的值
  - 当指令是LOAD类指令时，wb_select为1，因为他们要将读取的存储器的值写回到寄存器，其他指令的wb_select为0
- **load_type**：读存储器时读的数据类型
  - 当指令为LOAD类指令时，load_type根据LOAD类型置为相应值
  - 指令不为LOAD类指令是，load_type为NOREGWRITE
- **src_reg_en**：两位，分别表示reg1和reg2是否被读取，1为被用到，0位没有
  - R类运算指令和branch指令，两个寄存器都被读取，为2'b11
  - I类运算指令和jalr、LOAD类、STORE类指令，只用到了第一个寄存器，为2'b10
  - 其余指令都为2'b00
- **reg_write_en**：寄存器写信号
  - 指令是branch类或STORE类或rd=0的指令时，写信号为0，其余都为1
- **cache_write_en**：4位，每一位分别对应32位的数据中4个字节，表示该字节是否要被写
  - 指令不是STORE类时，值为0
  - sb: 4’b0001
  - sh: 4'b0011
  - sw: 4'b1111
- **alu_src1**：ALU操作数1的来源
  - 指令为jal、LUI、AUIPC时，值为1，选择PC-4；其他情况都为0，选择reg1
- **alu_src2**：ALU操作数1的来源
  - 指令为R类运算指令、branch类指令时，值为0，选择reg2
  - 指令为ssli，srli，srai时为2，选择reg2地址
  - 其余情况都为1，选择立即数
- **imm_type**：立即数类型
  - 根据指令类型和func字段判断

ALU

ALU需要根据ALU_func的值判断运算方法，将op1与op2做完相应运算后输出到ALU_out

其中需要注意的是有符号数的处理方法：

```
ALU_out = ($signed(op1)) < ($signed(op2)) ? 1 : 0;  //有符号数比较
ALU_out = op1 < op2 ? 1 : 0;    //无符号数比较
```

阶段二

这一阶段主要是写好Hazard模块解决数据相关，实现分支和跳转指令。

Hazard

Hazard模块主要需要输出op1_sel，op2_sel，reg2_sel三个信号

- **op1_sel**：选择ALU第一个运算数据的来源

```
always@ (*) begin
    if (src_reg_en[1]//如果reg1被读了（才有可能发生数据相关）
        && reg_write_en_MEM //且在MEM阶段的指令会写寄存器
        && reg_dstM == reg1_srcE//且被写的寄存器=被读的寄存器
        && reg_dstM != 0)//且被写的寄存器不是0号寄存器
        op1_sel <= 2'd0;//此时发生数据相关，选择result寄存器

    else if (src_reg_en[1]
             && reg_write_en_WB//在WB阶段的指令会写寄存器
             && reg_dstW == reg1_srcE
             && reg_dstW != 0)
        op1_sel <= 2'd1;//此时发生数据相关，选择WB寄存器

    //否则不发生数据相关，按照本来的操作数进行选择
    else if (src_reg_en[1] && alu_src1 == 0)
        op1_sel <= 2'd3;
    else if (!src_reg_en[1] && alu_src1 == 1)
        op1_sel <= 2'd2;
end
```

- **op2_sel**：选择ALU第二个运算数据的来源

```
always@ (*) begin
    if (src_reg_en[0] //如果reg2被读了（才有可能发生数据相关）
        && reg_write_en_MEM //且在MEM阶段的指令会写寄存器
        && reg_dstM == reg2_srcE //且被写的寄存器=被读的寄存器
        && reg_dstM != 0) //且被写的寄存器不是0号寄存器
        op2_sel <= 2'd0; //此时发生数据相关，选择result寄存器

    else if (src_reg_en[0]
             && reg_write_en_WB //且在WB阶段的指令会写寄存器
             && reg_dstW == reg2_srcE
             && reg_dstW != 0)
        op2_sel <= 2'd1; //此时发生数据相关，选择WB寄存器

    //否则不发生数据相关，按照本来的操作数进行选择
    else if (src_reg_en[0] && alu_src2 == 0) op2_sel <= 2'd3;
    else if (!src_reg_en[0] && alu_src2 == 1) op2_sel <= 2'd3;
    else if (!src_reg_en[0] && alu_src2 == 2) op2_sel <= 2'd2;
     end
```

- **reg2_sel**：选择reg2的来源

```
always@ (*) begin
    if (reg_write_en_MEM //在MEM阶段的指令会写寄存器
        && reg_dstM == reg2_srcE //且被写的寄存器=被读的寄存器
        && reg_dstM != 0) //且被写的寄存器不是0号寄存器
        reg2_sel <= 2'd0; //此时发生数据相关，选择result寄存器
```

```
6
7       else if (reg_write_en_WB //在WB阶段的指令会写寄存器
8               && reg_dstW == reg2_srcE
9               && reg_dstW != 0)
10          reg2_sel <= 2'd1; //此时发生数据相关，选择WB寄存器
11
12      //不发生数据相关，选择reg2
13      else reg2_sel <= 2'd2;
14  end
```

- **特殊的数据相关**：如果前一条指令是一个load，紧跟着一条指令需要用到load写入的寄存器，这样的数据相关需要插入一个bubble

```
always@ (*) begin
    if (wb_select == 1 //EX阶段是一个load指令
        //和后一条指令发生数据相关
        && (reg_dstE == reg1_srcD || reg_dstE == reg2_srcD))
    begin
        bubbleF <= 1;//IF阶段暂停一下
        bubbleD <= 1;//ID阶段暂停一下
        bubbleE <= 0;//EX正常
        bubbleM <= 0;//MEM正常
        bubbleW <= 0;//WB正常
    end
    else begin
        bubbleF <= 0;
        bubbleD <= 0;
        bubbleE <= 0;
        bubbleM <= 0;
        bubbleW <= 0;
    end
end
```

  - 这样两条指令间加入了一个bubble，继续执行时可以正常地转发数据

分支跳转

分支所需的信号在阶段一已经做完了，现在只需要在Hazard中在分支发生时做出一些相应

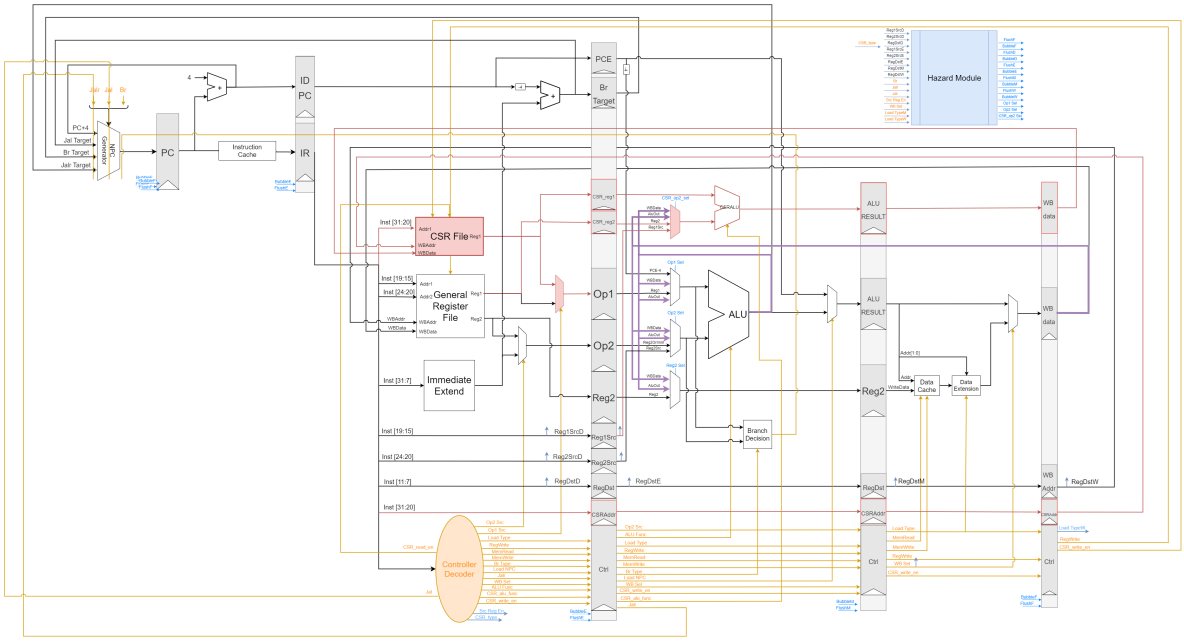
```
always@ (*) begin
    if (rst) begin
        flushF <= 1;
        flushD <= 1;
        flushE <= 1;
        flushM <= 1;
        flushW <= 1;
    end
    else if (br) begin//br,要冲刷掉ID,EX,MEM
        flushF <= 0;
        flushD <= 1;
        flushE <= 1;
        flushM <= 0;
        flushW <= 0;
    end
    else if (jalr) begin//jalr，要冲刷ID,EX
        flushF <= 0;
        flushD <= 1;
```

```
19          flushE <= 1;
20          flushM <= 0;//MEM不冲是因为jalr还要写回寄存器
21          flushW <= 0;
22       end
23       else if (jal) begin//jal，之冲刷ID
24          flushF <= 0;
25          flushD <= 1;
26          flushE <= 0;
27          flushM <= 0;
28          flushW <= 0;
29        end
30        else begin
31          flushF <= 0;
32          flushD <= 0;
33          flushE <= 0;
34          flushM <= 0;
35          flushW <= 0;
36        end
37 end
```

注意：判断顺序要先判断br和jalr，后判断jal，因为如果jal和br或jal和jalr同时有效，都不选jal，因为肯定是另外一条指令在先。

阶段三

此阶段主要处理CSR相关的指令，先按照数据通路连线。



主要改动：

- 增加**CSR File**，输入有clk,rst,addr,wb_addr,wb_data，输入控制信号CSR_read_en,CSR_write_en，输出有reg1
  - addr[11:0]是译码时读取CSR时的地址
  - wb_addr[11:0]是写回时的地址
  - wb_data[31:0]是写回时的数据
  - CSR_read_en是CSR读使能
  - CSR_write_en是CSR写使能
  - reg1是读取的CSR的数据
- 增加**CSRALU**，输入有op1，op2，ALU_func，输出有ALU_out

    - op1是第一个操作数，来源于CSR File（一定）
    - op2是第二个操作数，可能来源于Register File的第一个读出来的结果reg1，还有可能来源于Reg1Src（Zimm）。由于可能来源于通用寄存器，所以有可能发生数据相关，那么就还可能来源于ALU_out和WB_data
    - ALU_func指示ALU的运算
    - ALU_out是运算结果
- 增加**CSRALU_RESULT**, **CSRWB_data**, **CSRAddr**，功能类似于ALU_RESULT, WB_data和RegDst
- Hazard模块增加输入**CSR_type**和输出**CSR_op2_sel**
    - CSR_type指示CSRALU第二个操作数的来源，1来源于reg1,2来源于Reg1Src
    - CSR_op2_sel是CSRALU第二个操作数的来源选择信号，Hazard根据CSR_type和数据相关的信号判断CSRALU第二个操作数来源于哪个地方
- 增加控制信号**Op1_Src**
    - Op1_Src选择进入Op1寄存器的值来源于哪里，0来源于CSR File，1来源于reg1
    - 它能够做到在原子交换指令时将CSR的数据通过普通的ALU写回到通用寄存器中，这样写通用寄存器的路和写CSR的路完全分开，互不相交。
- ALU_func增加**CSR类型**
    - 原子交换指令来到时，ALU_func被置为CSR类型，ALU会将第一个操作数直接传给ALU_out

实验总结

在有了助教提供的框架的前提下，实现所有的数据通路其实不困难。比较麻烦的在于对各种细节的处理上。一根线没有连上就跑不通。

在这次实验中，数据通路的阶段我写的很快，时间主要在于debug上。

- 第一阶段的bug集中在ALU_func中，尤其是I类指令的区分上需要小心
- 第二阶段的bug集中在Controller Decoder中的各种控制信号上。最大的bug在于我把STORE类的op搞错了，导致很多控制信号都不对。然后JAL指令需要将ALU_func置为ADD，这一点花了我一段时间才意识到。
- 第三阶段的工作首先要把详细的数据通路画出来，再连线。画数据通路不难，按照功能添加元件和连线就可以。但是把图转化成代码这一步费了一些力气。有很多线，连着连着就忘了那个连过了，那个没连。还要记着定义的变量名字，加端口，写定义语句...总之比较繁琐，我经常遗漏。

不管怎样，这次实验还是做完了，还是很有收获。我对于流水线的理解似乎更透彻。虽然组成原理的时候我没有选择写流水线，但是这次实验也没有很困难。本来以为很难写的Hazard模块几行代码也就没了。经验就是，写前一定要想好，把代码写完先检查一遍再去仿真，不要着急忙慌的去找bug，检查一遍可以排除掉一些可能会出现的问题，使代码过程更加高效。

改进意见

没啥，都挺好的。