# 计算机体系结构Lab2实验报告V2

廖洲洲 PB17081504

## 实验目标

使用Verilog实现RV32I流水线CPU,并仿真测试通过

## 实验环境和工具

实验环境：Vivado 2016.2, VScode

语言:Verilog

## 实验内容和过程

RV32I指令格式

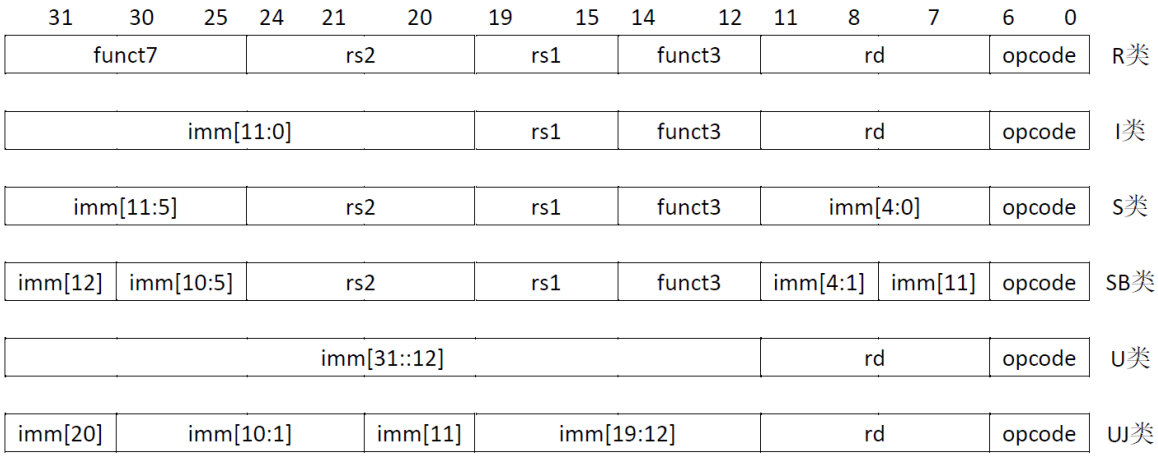



图 2.3：RISC-V 显示了立即数的基本指令格式

RISC-V指令生成的立即数

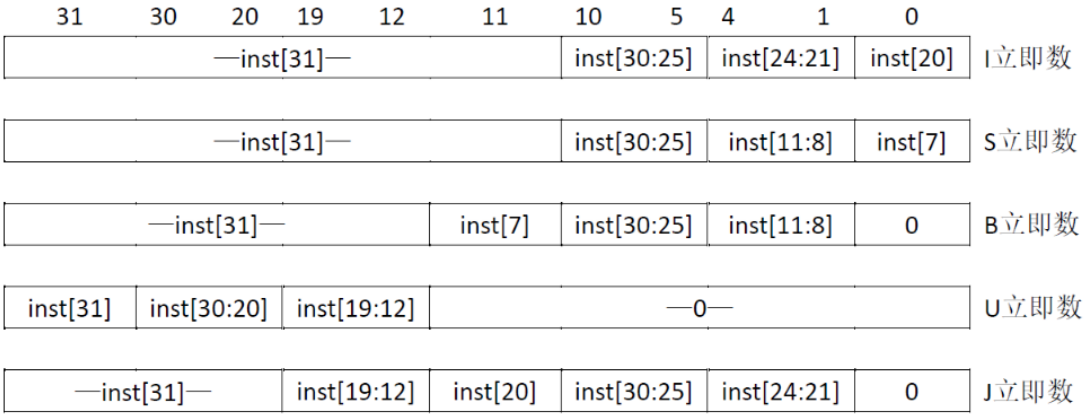



图 2.4：RISC-V 指令生成的立即数。用指令的位标注了用于构成立即数的字段。符号扩展总是使用 inst[31]。

I类指令：ADDI,SLTI,SLTIU,ANDI,ORI,XORI,SLLI,SRLI,SRAI,LB,LH,LW,LBU,LHU,`JALR`

R类指令：ADD,SLT,SLTU,AND,OR,XOR,SLL,SRL,SUB,SRA

S类指令：SB,SH,SW

B类指令：BEQ,BNE,BLT,BGE,BLTU,BGEU

U类指令：LUI,AUIPC

J类指令：JAL

# 阶段一

## NPC_Generator

- 根据跳转信号，决定执行的下一条指令地址
- 对于不同跳转target 的选择有优先级，比如Branch是否跳转需要再EX阶段才能出结果，Jalr指令需要在EX阶段计算出目标地址，而Jal指令是无条件跳转，在ID阶段就能计算出目标地址，这样当Branch指令后跟一条Jal指令时，会发现它们的跳转使能信号在同一周期有效，故需要划分优先级。因此Jal的优先级更低。
- 故优先级`Jalr=Br>Jal`

## ImmExtend

- 立即数扩展，将指令中的立即数部分扩展为完整立即数
- 根据图2.4即可

## ALU

- 算数运算和逻辑运算功能部件，根据ALU_func进行不同的运算
- $signed()函数返回有符号的值

```
    always@(*)
    begin
        case(ALU_func)
            `ADD: ALU_out <= op1 + op2;
            `SUB: ALU_out <= op1 - op2;
            `XOR: ALU_out <= op1 ^ op2;
            `OR:  ALU_out <= op1 | op2;
            `AND: ALU_out <= op1 & op2;
            `SRL: ALU_out <= (op1 >> op2[4:0]);
            `SLL: ALU_out <= (op1 << op2[4:0]);
```

```
            `SRA: ALU_out <= (op1_signed >>> op2[4:0]);
            `SLT: ALU_out <= (op1_signed < op2_signed )? 32'b1 : 32'b0;
            `SLTU: ALU_out <= (op1 < op2) ? 32'b1 : 32'b0;
            `LUI: ALU_out <= { op2[31:12],12'b0 };
           default: ALU_out <= 32'hxxxxxxxxx;
        endcase
    end
```

1. 操作数是默认为无符号数的，立即数类型都在op2中
2. 移位的次数被编码到 I 立即数字段的低 5 位。
3. 逻辑右移为>>,算数右移为>>>，算数右移要使用有符号数，因为要将符号位复制到空出的高位中
4. LUI将U立即数放到目标寄存器rd的高20位，将rd的低12位填0

## BranchDecision

- 根据Reg1、Reg2和br_type判断是否branch
- BLT和BGE是有符号运算

## DataCache

- Load有效字节地址是通过将寄存器 rs1 与符号扩展的 12 位偏移量相加而获得的。
- RV32I是按字节寻址，所有load 和 store 指令的有效地址，应该与该指令对应的数据类型相对齐（32 位访问在 4 字节边界对齐， 16 位访问在 2 字节边界对齐）
- 输入是地址的前30位，因此输出是32位数据，之后在DataExtend对数据进行选择，小端的

## DataExtend

- 将Cache中取出的数据根据Load Type进行扩展
- 首先根据Load Type决定进行符号扩展还是零扩展，再根据addr决定有效数据的具体位置

## ControllerDecoder

- 对指令进行译码，将其翻译成控制信号，传送给各个部件（依据opcode，funct3和funct7字段）
    - jal:jal跳转指令
    - jalr:jalr跳转指令
    - op2_src :ALU的第二个操作数来源。为1时，op2选择imm，为0时，op2选择reg2，只有R类
    - ALU_func:ALU执行的运算类型
    - br_type:branch的判断条件，可以是不进行branch
    - load_npc :写回寄存器的值的来源（PC或者ALU计算结果）, load_npc == 1时选择PC，jal或jalr，因为jal和jalr会将pc+4写入rd
    - wb_select :写回寄存器的值的来源（Cache内容或者ALU计算结果），wb_select == 1时选择cache内容,仅lw指令
    - load_type:load类型
    - src_reg_en:指令中src reg的地址是否有效，src_reg_en[1] == 1表示reg1被使用到了，src_reg_en[0]==1表示reg2被使用到了，I型为2'b10,R/S/B型为2'b11,其它为2'b00
    - reg_write_en:通用寄存器写使能，reg_write_en == 1表示需要写回reg,jal、jalr、AUIPC、LUI、R型、Load指令
    - cache_write_en:按字节写入data cache，SW 4'b1111, SH 4'b0011, SB 4'b0001，其他为4'd0
    - imm_type:指令中立即数类型
    - alu_src1:alu操作数1来源，alu_src1 == 0表示来自reg1，alu_src1 == 1表示来自PC，仅AUIPC
    - alu_src2:alu操作数2来源，alu_src2 == 2'b00表示来自reg2（R，B类），alu_src2 == 2'b01表示来自reg2地址(SLLI、SRLI、SRAI），alu_src2 == 2'b10表示来自立即数（I\U\S)

# 阶段二

## HazardUnit

- 用来处理流水线冲突，通过插入气泡，使用forward以及冲刷流水段来解决数据相关和控制相关，同时在cpu重置时，利用flush来清空CPU
  - Bubble:当Bubble信号为1时，流水段寄存器中的内容不变
  - Flush：当Flush信号为1时，清空流水线段寄存器中的内容
- 指令顺序执行，数据相关只有写后读，部分写后读相关使用旁路解决，部分使用插入气泡解决
  - CPU重置，所有stall置0，flush置1
  - 控制相关
    1. 分支跳转或jalr指令要等到第三周期才能确认跳转地址，如果跳转预测错误则会导致其后两个周期，即IF和ID段的指令为不该执行的指令，因此对IF-ID和ID-EX的流水线寄存器冲刷。
    2. jal指令，其在ID阶段即能得到目标地址，跳转地址错误则会导致IF周期的指令为错误指令因此需要对IF-ID的流水线寄存器冲刷。
  - 数据相关（RAW）
    1. 因为通用寄存器同步写，异步读，在时钟下降沿写入，因此读寄存器先于写寄存器，一条指令最多与其后的两条指令产生数据相关
    2. 对于一般的数据相关，即写寄存器指令的后三条指令用到了其目标寄存器，而此时目标寄存器还没有更新，因此设置旁路，将要写回的结果直接送回到ALU源操作数端口。
       如：Reg1SrcE=RegDstW,Src_Reg_En[1]=1,Reg_write_en_WB=1,则Op1_Sel=2'b1

       ```
           //对src1
           always @(*)
           begin
               if(reg_write_en_MEM && src_reg_en[1] == 1 && reg1_srcE ==
       reg_dstM && reg_dstM != 5'b0)
               begin  //写寄存器和下一条指令的读寄存器相同，不能是0号寄存器
                   op1_sel = 2'b00;
               end
               else if(reg_write_en_WB && src_reg_en[1] == 1 && reg1_srcE
       == reg_dstW && reg_dstW != 5'b0)
               begin  //写寄存器和下下条指令的读寄存器相同
                   op1_sel = 2'b01;
               end
               else if(alu_src1 == 1)    //AUIPC指令
               begin
                   op1_sel = 2'b10;
               end
               else
               begin
                   op1_sel = 2'b11;
               end
           end
       ```

    3. 对于load指令后的读相关寄存器的操作，由于读数据在MEM阶段才能取得数据，而紧跟的指令在EX阶段需要用到源数据，在同一时间内，因此旁路失去作用，使用在load后增加一个气泡的方法解决

# 阶段三

- CSR指令为原子性读、修改、写控制和状态寄存器的指令
  - CSR指令

| csr | rs1 | funct3 | rd | opcode | 意义 |
|---|---|---|---|---|---|
| source/dest | source | CSRRW | dest | SYSTEM | rd=csr; csr=rs1 |
| source/dest | source | CSRRS | dest | SYSTEM | rd=csr; csr\|=rs1 |
| source/dest | source | CSRRC | dest | SYSTEM | rd=csr; csr &= !rs1 |
| source/dest | uimm[4:0] | CSRRWI | dest | SYSTEM | rd=csr; csr=uimm[4:0] |
| source/dest | uimm[4:0] | CSRRSI | dest | SYSTEM | rd=csr; csr\|=uimm[4:0] |
| source/dest | uimm[4:0] | CSRRCI | dest | SYSTEM | rd=csr; csr &= !uimm[4:0] |

  opcode=1110011

  func3:

  CSRRW:001

  CSRRS:010

  CSRRC:011

  CSRRWI:101

  CSRRSI:110

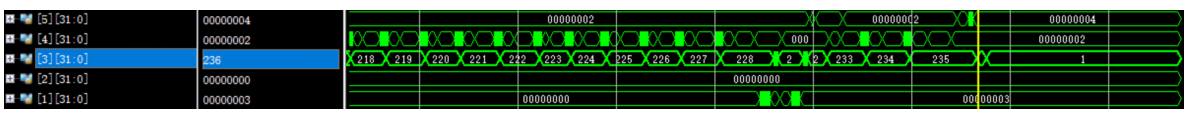
  CSRRCI:111

- 设计思路：
  - 在EX阶段增加CSR_File，使得可以读写CSR_File，输入端口有inst[31:12](包含了csr地址、zimm值和funct3），写使能,源数据，输出对应CSR中的旧值,之后将旧值写回rd；
  - 在CSR_File中增加按位掩码功能，以实现CSRRS、CSRRC、CSRRSI、CSRRCI的按位掩码功能，其输入是rs1中的值或零扩展到32位的5位立即数（zimm[4:0]），根据func3判断将CSR旧值对应位置0或置1
  - 增加流水线段寄存器
  - 修改COntroller Decoder和Load_NPC处的多选器

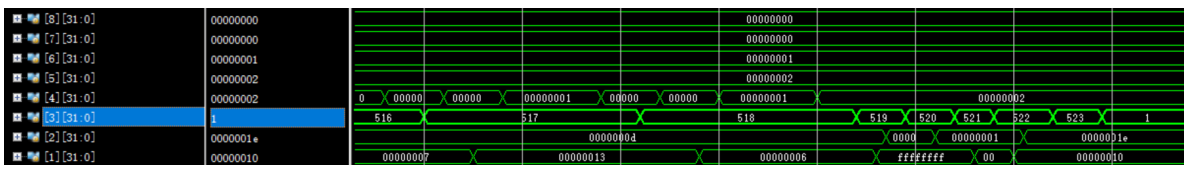
# 实验结果

## 阶段一、二：

### test1:



### test2:



### test3:

test1、2、3均测试通过

## 阶段三

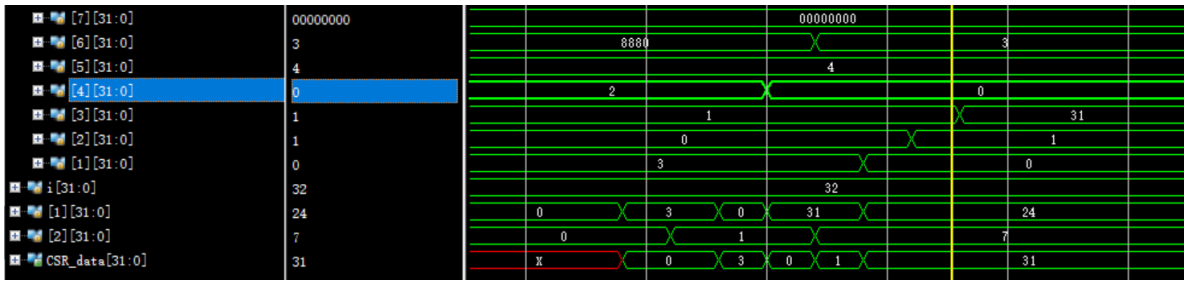
### test4:

直接在test1指令上进行修改，在test1测试全部通过的基础上测试CSR指令，即在test1的末尾pass部分加上CSR指令

```
0001234c <pass>:
   1234c:   00100193            li  gp,1
   12350:   00000013
   12354:   00000013
   12358:   00000013
   1235c:   00000013    //test1指令执行完后reg：[1]=3,[2]=0,[3]=1,[4]=2,[5]=4,
[6]=8800
   12360:   00109173            csrrw 0x2,0x1,0x1 //reg[2]=0,csr[1]=3
   12364:   0021a273            csrrs 0x4,0x2,0x3 //reg[4]=0,csr[2]=1
   12368:   0010b373            csrrc 0x6,0x1,0x1 //reg[6]=3,csr[1]=0
   1236c:   001fd0f3            csrrwi 0x1,0x1,0b11111 //reg[1]=0,csr[1]=31
   12370:   0023e173            csrrsi 0x2,0x2,0b00111 //reg[2]=1,csr[2]=7
   12374:   0013f1f3            csrrci 0x3,0x1,0b00111 //reg[3]=31,csr[1]=24
```

结果正确的话：csr[1]从0->3->0->31->24 csr[2]从0->1->7

reg[1]从3->0 reg[2]从0->0->1 reg[3]从1->31 reg[4]从2->0



故测试通过

# 实验总结

- 做这个实验一定要非常细心和有耐心,不然debug的时候要花很多时间
- 加深了对RV32I基本整数集指令的了解，比如
  - SLLI、SRLI、SRAI被移位常数次，是I类格式的特例，被移位的次数即为rs2的地址
  - jal和jalr会将pc+4保留再rd中
  - jalr是I类
  - 之前以为一条指令可能与的三条指令产生数据相关，但由于通用寄存器采用同步写，异步读，读是在下降沿的，当在WB阶段写数据时，在时钟下降沿会完成写，而如果在ID阶段读了写的寄存器，其不会产生相关，这是因为读是异步的，当写入新的数据时，读出的数据也发生改变。因此最多可能与其后两条指令产生数据相关。
  - Data Cache取出的数据总是32位，之后根据Load Type和地址后两位进行数据的选择，写也是同理

- CSR在实现时容易做成把新的csr寄存器中的值赋给目的寄存器
- 加深了对流水线cpu中的数据相关、控制相关及其解决方法的理解
- 更加熟悉了Verilog的使用
- 学习利用了仿真测试代码的正确性
- 时间安排：
    1. RV32I指令和代码框架的熟悉 约4h
    2. 阶段一的实现  约7h
    3. 阶段二的实现 约9h（阶段一二的debug）
    4. 阶段三的实现 约9h

# 意见

- 设计图和代码有部分端口名不同，希望能进行修改