体系结构LAB2实验报告

JL19110004 徐语林

一.实验目标

体系结构实验二是建立在实验一的基础上，实验一是对RV32I进行了初步的熟悉，以及对于常见的指令有了进一步的了解，实验二便开始用vivado进行RV32I流水线CPU的实现。实验二一共分为了三个阶段，第一阶段是完成SLLI、SRLI、SRAI、ADD、SUB、SLL、SLT、SLTU、XOR、SRL、SRA、OR、AND、ADDI、SLTI、SLTIU、XORI、ORI、ANDI、LUI、AUIPC指令的设计并进行验证；第二阶段是完成JALR、LB、LH、LW、LBU、LHU、SB、SH、SW、BEQ、BNE、BLT、BLTU、BGE、BGEU、JAL指令以及对于Harzard模块进行实现；第三阶段是添加CSR类指令，实现CSRRW、CSRRS、CSRRC、CSRRWI、CSRRSI、 CSRRCI指令。

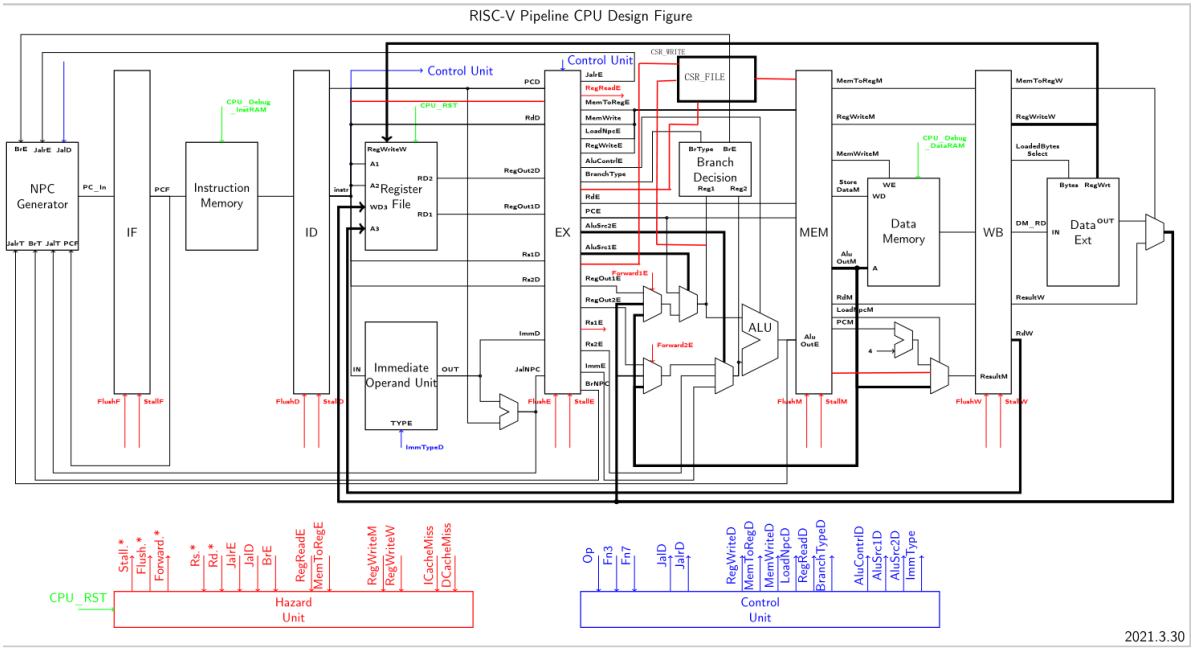
二.实验环境和工具

Vivado,Window操作系统

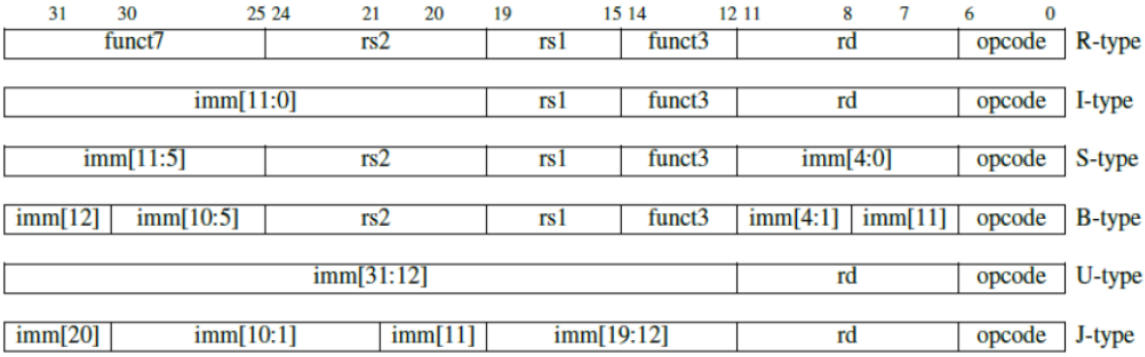
语言：verilog

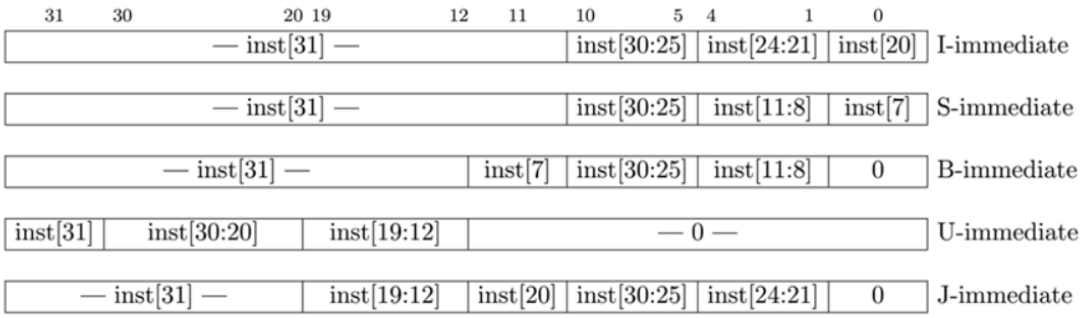
三.实验内容和过程

实验用图(总用图，包括了CSR类指令):



实验参考的资料(RV32I指令的结构)：





实验过程：

阶段一，阶段二

(实验代码中写出了无需修改的部分便不再赘述，只需要理解即可)

IDSegReg:

取[31:2]位即可

ImmOperandUnit:

立即数扩展按照的是上述实验参考资料中的第二个图来写

NPC_Generator:

此部分相当于一个选择器，选择下一条指令的地址；而对于跳转指令而言，Br指令和Jalr指令是在EX段计算出的跳转地址，而Jal指令是在ID段计算出的地址，因此优先级应当是Br=Jalr>Jal。因此在模块中对于跳转指令的处理要遵守这样一个优先级的顺序进行处理。

ALU:

根据指令的操作来完成操作数的计算；值得注意的是算数运算要使用带符号的数，而verilog中的wire型是无符号的，因此要做一个转换。

BranchDecisionMaking:

此部分是对于Br指令的处理，还是要根据Br指令的种类，对每一种指令的结果进行处理，值得注意的是BGE和BLT也是进行的有符号的运算，而BGEU和BLTU进行的是无符号的运算。

DataExt:

此部分主要是对load类指令进行处理；LoadedByteSelect信号根据A[1:0]来赋值，从而决定有效数据的具体位置;而RegWrite指令决定了是进行零扩展还是符号扩展，例如：

```
case (RegWriteW)NOREGWRITE: OUT <= 32'b0;
      LB: begin
                case (LoadedBytesSelect)
                    2'b00: OUT <= {{25{IN[7]}},IN[6:0]};
                    2'b01: OUT <= {{25{IN[15]}},IN[14:8]};
                    2'b10: OUT <= {{25{IN[23]}},IN[22:16]};
                    2'b11: OUT <= {{25{IN[31]}},IN[30:24]};
                    default: OUT <= 32'bx;
                endcase
```

此段代码便是RegWrite信号若是匹配上了LB指令，那么便进行符号扩展(LBU,LHU为零扩展)，而LoadedByteSelect便根据赋值来选择有效数据的具体位置。

WBSegReg:

此部分重点是对非字对齐的写的考虑，是利用了MemWrite指令，4'b0001表示写入一个byte,4'b0011表示写入两个byte,4'b1111表示写入全部的byte.因此再根据信号，对写入的数据进行处理，如下：

```
(WE==4'b0001)? ({WD[7:0],WD[7:0],WD[7:0],WD[7:0]}) : ((WE==4'b0011)? ({WD[15:0],WD[15:0]}) : WD)
```

ControlUnit:

此部分应该说是实验的核心的部分，是对所有信号的一个处理单元。下面将针对每个信号做一个大致的说明：

JalD:要判断是否是Jal指令，只需对Op进行判断即可，JalD=1,则表示Jal指令到达了ID译码阶段

JalrD:要判断是否是Jalr指令，只需对Op进行判断即可，JalrD=1,则表示Jalr指令到达了ID译码阶段

MemToRegD:判断是否是Load类指令，若是则置为1；此信号使用在WB阶段写回的选择上

LoadNpcD：判断是将NextPC还是ALU运算结果送到ResultM中，只需根据指令的Op进行判断即可

AluSrc1D和AluSrc2D：这两个信号都是表示对ALU模块输入源的选择

RegReadD：此信号若为2'b10则表示A1端口被用到，若为2'b01则表示A2端口被用到

RegWriteD：此信号后面要被用到DataExt部件，来决定Load指令的扩展类型，因此对照着不同指令编码的不同来进行判定和选择即可

MemWriteD：此信号中4'b0001表示写入一个byte,4'b0011表示写入两个byte,4'b1111表示写入全部的byte，因此根据指令的类型来进行分类即可

BranchTypeD：此信号Br指令的类型，根据不同的分支指令Op以及Fn3编码的不同来对此信号进行赋值即可

AluContrlD：此信号表示ALU执行的计算功能，也是通过要进入ALU的指令的Op,Fn3,Fn7的编码的不同来进行分类

ImmType：此信号表示立即数的格式，根据不同类指令的Op来进行判断和赋值即可

下面是比较重要的Harzard模块：

HarzardUnit：

这一模块主要是处理stall和Flush信号，stall是冻结的功能而Flush则是清零；还有一个便是forward信号的处理

stall和Flush:这一部分便是针对load指令以及br,Jal,Jalr指令。其中br,Jal,Jalr是解决控制相关，Br和Jalr指令要在EX段才可确认跳转的地址，如果跳转成功那么进入流水线的其后两条指令便不能执行，因此应该将ID,EX段冲刷掉；而同理Jal指令是在ID段确定的跳转地址，如果跳转成功那么进入流水线的其后一条指令便不能执行，因此应该将IF段冲刷掉，而解决控制相关的时候Stall指令置为0不触发即可。而对于Load指令后紧跟一个需要读写相关寄存器的指令来说，旁路会失去作用，因此需要将IF,ID段的Stall信号置为1.

Forward:这一部分涉及的是旁路的建立，从而解决RAW数据相关。因此按照示例图对forward信号进行赋值即可。

```
Forward1E[0]=(|RegWriteW)&&(RdW!=0)&&(!((RdM==Rs1E)&&(|RegWriteM)))&&
(RdW==Rs1E)&&RegReadE[1];
```

```
Forward1E[1]=(|RegWriteM)&&(RdM!=0)&&(RdM==Rs1E)&&RegReadE[1];
```

Forward1E[0]=1,则表示取的数据来自于WB段写回的数据，因此Regwritew信号按位或为1，表示有写回，且是在WB段写回，那么RdW!=0,且要发生数据相关。

阶段三

阶段三主要是在阶段一二的基础上增加上最后六条CSR类指令，CSR类指令的格式如下所示：

31 20	19 15	14 12	11 7	6 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
source/dest	source	CSRRW	dest	SYSTEM
source/dest	source	CSRRS	dest	SYSTEM
source/dest	source	CSRRC	dest	SYSTEM
source/dest	uimm[4:0]	CSRRWI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRSI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRCI	dest	SYSTEM

设计数据通路思路：由于csr占据了31:20位，那么原有寄存器与之不匹配，所以应该新增一个寄存器，在我的设计里是将新增加的CSR寄存器放在了EX段，这样进入的数据端口可以直接连上已经进行了数据相关处理的一端即可。而在MEM段的选择器应该进行适当的扩充，详细参加总的设计图。

CSR_File：

```
always@(negedge clk or posedge rst)
    begin
        if(rst)                                    for(i=1;i<4096;i=i+1) RegFile[i]
[31:0]<=32'b0;
        else if( (csr_write==1'b1) && (csr_addr!=12'b0) )
        begin
           csr_data = (csr_addr == 12'b0) ? 32'h0 : RegFile[csr_addr];
           case(csr_func3) CSRRW:RegFile[csr_addr]<=csr_wd;
```

`CSRRS:RegFile[csr_addr]<= (RegFile[csr_addr]) | (csr_wd);//如果写回的数据的任何一位为1，则会导致csr中的对应位置为1` CSRRC:RegFile[csr_addr]<= (RegFile[csr_addr]) & (~csr_wd);//如果写回的数据任何一位为1，则会导致csr中的对应位置为0

`CSRRWI:RegFile[csr_addr]<={27'b0,zimm[4:0]};` CSRRSI:RegFile[csr_addr]<=

(RegFile[csr_addr]) | ({27'b0,zimm[4:0]});

```
        CSRRCI:RegFile[csr_addr]<=(RegFile[csr_addr]) & (~{27'b0,zimm[4:0]});
            default:RegFile[csr_addr]<=32'hxxxx;
            endcase
        end
    end
```

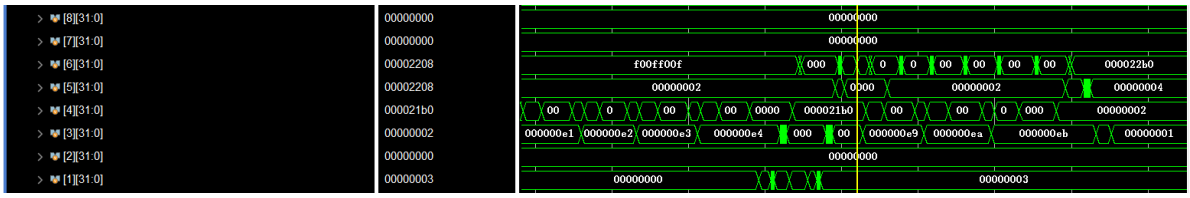
这一部分和前面寄存器堆的处理方式其实很相似，只是改了一下数据；此外便是根据CSR类指令的结构，对不同种类的CSR指令进行处理即可。

LoadNPCM:此信号进行了修改，进行了一定的扩充，增加了一个端口表示从CSR寄存器读出的数据写回到Regfile中。

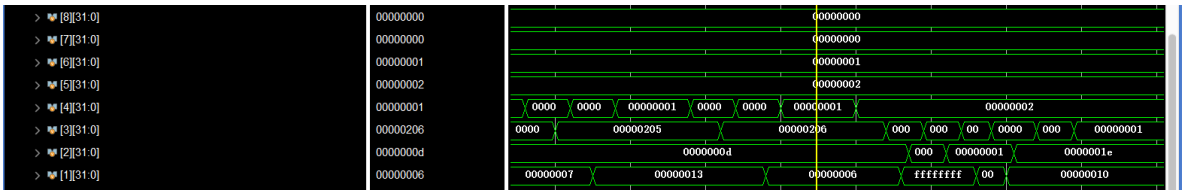
```
ResultM = (LoadNpcM==2'b01) ? (PCM+4) : ((LoadNpcM==2'b10)?csr_dataM:AluOutM);
```

四.实验结果

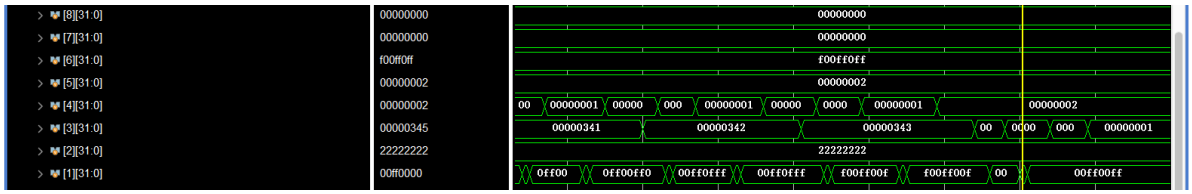
test1:



test2:



test3:



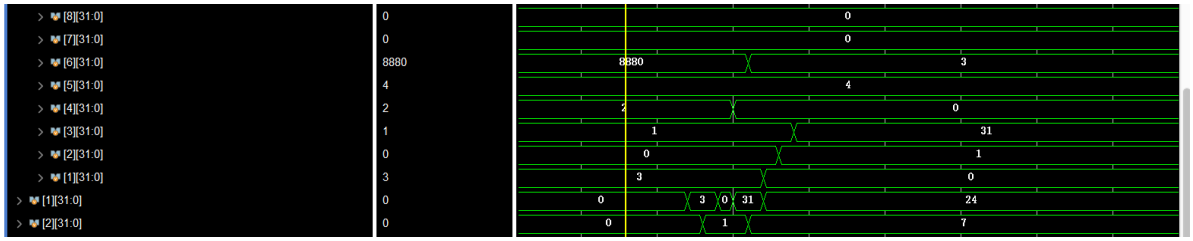
前三个test主要用于测试阶段一和阶段二，可见全部通过测试

test4:

测试代码如下：

```
csrrw $2,$1,$1          //reg[2]=0 csr[1]=3
csrrs $4,$2,$3       //reg[4]=0 csr[2]=1
csrrc $6,$1,$1 // reg[6]=3 csr[1]=0
csrrwi $1,$1,0b11111  //reg[1]=0 csr[1]=31
csrrsi $2,$2,0b00111 //reg[2]=1 csr[2]=7
csrrci $3,$1,0b00111 //reg[3]=31 csr[1]=24
```

将上述几行代码加入到了test1的中，由此如果通过测试，regfile中的数据：Reg[1]将从3变为0；Reg[2]将从0变为1；Reg[3]将从1变为31，Reg[4]将从2变为0；Csr[1]从0变到3到0到31再到24，Csr[2]从0到1到7



测试通过

五.实验总结

本次实验主要是建立在实验一的基础上，因此在写实验之前要对指令的结构熟悉，才不会导致在写代码的时候出错。流水线部分应该不陌生，做过组成原理实验后写体系结构的实验总体来说还是挺轻松的，踩的坑主要还是来自于非字对齐处理那一块对于写入数据以及使能信号的处理以及CSR的设计部分，在写代码过程中倒没遇到太大的麻烦，只是有时候会将编码输错，导致一条一条的去寻找还是挺崩溃的。通过本次实验还是加深了对于RV32I的指令的理解以及对流水线的工作有了进一步的认识，锻炼了verilog的能力，第一阶段和第二阶段花了一天时间来理清思路，两天时间来写代码，一天时间来测试和debug;第三阶段花了一上午的时间来设计CSR数据通路，以及一下午的时间进行实现和验证。

最后关于实验所有代码以及相关注释将在附件中给出，实验报告只对实验中的重点部分以及个人所花时间多的部分进行了重点介绍。

六.建议

无，体验不错。