Lab2实验报告

PB18111757 陈金宝

实验目标

实现RV32I 流水线CPU。

实现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指令，处理数据相关。

同时实现CSR指令:CSRRW、CSRRS、CSRRC、CSRRWI、CSRRSI、CSRRCI，处理相关。

实验环境和工具

```
实验环境:
Windows 10 20H2
Linux arch 5.11.16-arch1-1 x86_64 GNU/Linux(用于生成测试样例)
工具:
vivado 2019.2(仿真)
vscode 1.56.0(代码编辑)
```

实验内容和过程

阶段一

第一阶段没有处理相关，每两条指令直接间隔4个nop，主要需要实现Control Unit，ALU和ImmUnit

在Control Unit中使用case语句，对不同类型的指令分情况赋值信号。一阶段实现了R类、I类和U类指令。对每种类型的指令需要区分ImmType，RAluSrc1D，RAluSrc2D，RegWriteD，RegReadD，AluContrlD等信号。

ALU根据ALuControl对操作数进行计算。vivado默认是无符号数。涉及到有符号数的计算或比较时，需要在操作数前加上`$signed`修饰符。比如:

```
`SRA:       AluOut <= ($signed(Operand1)) >>> Operand2[4:0];
```

ImmUnit根据对应的ImmType进行立即数扩展。比如:

```
`ITYPE: Out <= { {21{In[31]}}, In[30:20] };
```

一阶段部分测试样例如下:

```
#test1.S (部分)
.org 0x0
    .global _start
_start:
    lui x1, 0x1
    nop
    nop
    nop
```

```
    nop
    lui x2, 0x2
    nop
    nop
    nop
    nop
    add x3,x1,x2
    nop
    nop
    nop
    nop
    addi x3,x2,0x3
    nop
    nop
    nop
    nop
    sub x4,x2,x3
    nop
    nop
    nop
    nop
    lui x1, 0x1
    nop
    nop
    nop
    nop
    lui x2, 0x2
    nop
    nop
    nop
    nop
    srli x2,x2,0xc
    nop
    nop
    nop
    nop
    sll x3,x1,x2
    nop
    nop
    nop
    nop
    slli x3,x1,0x2
    ...
```

测试后各个寄存器的值符合预期

阶段二

阶段2在阶段1基础上实现load，store，branch和jump指令。且需要实现数据相关。

在control unit中实现对应的RJalD,RJalrD,RMemToRegD,RLoadNpcD,BranchType,DMemWriteD等信号

BranchDecisionMaking根据branchtype和两个操作数判断branch是否成功。

```
`include "Parameters.v"
module BranchDecisionMaking(
```

```
    input wire [2:0] BranchTypeE,
    input wire [31:0] Operand1,Operand2,
    output reg BranchE
);
always @(*)
begin
    case (BranchTypeE)
    `NOBRANCH:  BranchE <= 1'b0;
    `BEQ:   begin
        if(Operand1==Operand2)
            BranchE <= 1'b1;
        else
            BranchE <= 1'b0;
    end
    `BNE:   begin
        if(Operand1!=Operand2)
            BranchE <= 1'b1;
        else
            BranchE <= 1'b0;
    end
    `BLT:   begin
        if($signed(Operand1) < $signed(Operand2))
            BranchE <= 1'b1;
        else
            BranchE <= 1'b0;
    end
    `BLTU:  begin
        if(Operand1 < Operand2)
            BranchE <= 1'b1;
        else
            BranchE <= 1'b0;
    end
    `BGE:   begin
        if($signed(Operand1) >= $signed(Operand2))
            BranchE <= 1'b1;
        else
            BranchE <= 1'b0;
    end
    `BGEU:  begin
        if(Operand1 >= Operand2)
            BranchE <= 1'b1;
        else
            BranchE <= 1'b0;
    end
    default:BranchE <= 1'b0;
    endcase
end
endmodule
```

由于dataram只处理末二位地址是00的情况，所以实现load和store指令需要额外处理结尾非00的情况。在WBSegReg和DataExt根据LoadedByteSelect以及WE信号进行对应的实现。实现数据存储在结尾非00地址以及结尾非00地址中内容的读取。

WBSegReg.v的处理:

```
//WBSegReg.v
DataRam DataRamInst (
        .clk    (clk),
        .wea    (WE << A[1:0]),
        .addra  (A[31:2]),
        .dina   (WD << {A[1:0],3'b0}),
        .douta  ( RD_raw         ),
        .web    ( WE2            ),
        .addrb  ( A2[31:2]       ),
        .dinb   ( WD2            ),
        .doutb  ( RD2            )
    );
```

DataExt.v进行类似的处理

NPC_GENERATOR中，branch和jalrd均在EX段跳转，jal在ID段跳转。可能出现冲突的情况。出现冲突时需要Branch和jalr优先。

```
module NPC_Generator(
    input wire [31:0] PCF,JalrTarget, BranchTarget, JalTarget,
    input wire BranchE,JalD,JalrE,
    output reg [31:0] PC_In
    );
wire [31:0] PC_raw;
assign PC_raw = PCF + 4;

always @(*)
begin
    if(BranchE) begin
    PC_In <= BranchTarget;
    end
    else if(JalrE) begin
    PC_In <= JalrTarget;
    end
    else if(JalD) begin
    PC_In <= JalTarget;
    end
    else begin
    PC_In <= PC_raw;
    end
end
endmodule
```

在hazard unit中检测到branch和jalrd成功时需要清空ID和EX段。检测到jal成功时清空ID段，优先级同上。检测到hazard时（load+use），需要使得IF,ID段进行stall。并flush掉EX段（防止use指令的信号向后传）。控制forward信号时，根据RegReadE，RegWriteM, RegWriteW信号以及Rs1E,Rs2E和RdM,RdW是否相等判断是否有相关。若和Mem段，WB段均产生相关，则forward Mem段。否则forward对应的段。但当Rd是x0(5'b00000)时不进行转发。

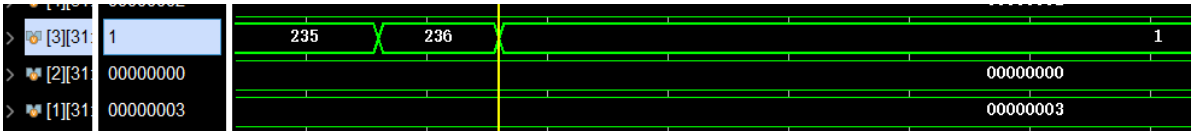
hazard unit处理相关的部分代码:

```
//HarzardUnit.v(部分)
assign regwem = |RegWriteM;
assign regwew = |RegWriteW;

assign rs1hitm = (regwem)&&(Rs1E==RdM)&&(RdM!=5'b00000)&&RegReadE[1];
```

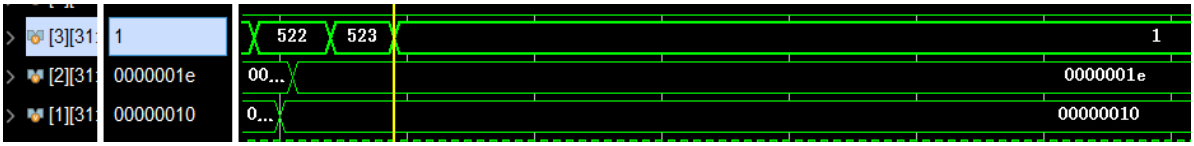
```
assign rs1hitw = (regwew)&&(Rs1E==RdW)&&(RdW!=5'b00000)&&RegReadE[1];
assign rs2hitm = (regwem)&&(Rs2E==RdM)&&(RdM!=5'b00000)&&RegReadE[0];
assign rs2hitw = (regwew)&&(Rs2E==RdW)&&(RdW!=5'b00000)&&RegReadE[0];

always@(*)begin
    if(CpuRst) begin
        Forward1E <= 2'b00;
        Forward2E <= 2'b00;
    end
    else begin
    case({rs1hitm,rs1hitw})
        2'b00:begin
            Forward1E <= 2'b00;
        end
        2'b01:begin
            Forward1E <= 2'b01;
        end
        2'b10:begin
            Forward1E <= 2'b10;
        end
        2'b11:begin
            Forward1E <= 2'b10;
        end
    endcase
    case({rs2hitm,rs2hitw})
        2'b00:begin
            Forward2E <= 2'b00;
        end
        2'b01:begin
            Forward2E <= 2'b01;
        end
        2'b10:begin
            Forward2E <= 2'b10;
        end
        2'b11:begin
            Forward2E <= 2'b10;
        end
    endcase
    end
end
```

测试样例使用实验提供的三个样例(1testAll,2testAll,3testAll)。测试后3号寄存器的值均为1，通过测试。仿真截图如下:

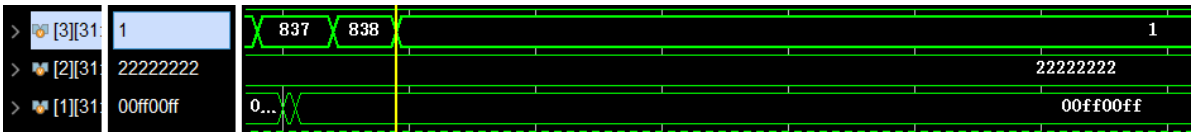
样例一:



样例二:



样例三:



阶段三

阶段3实现CSR。添加CSR数据通路以及相关处理。添加`CSRFile.v`，`CSRALU.v`。CSR也是五段流水，在WB段写入。MEM段向后传递对应信号。

在`CSRFile.v`中实现CSR寄存器，和RegieterFile类似。共12位地址，4096个。当高两位地址是`2'b11`时是只读的。读寄存器是异步的。

```
module CSRFile (
    input clk,
    input rst,
    input WE,
    input wire [11:0] A,
    input wire [11:0] A1,
    input wire [31:0] WD,
    output wire[31:0] RD
);

assign WE_VALID = (A1[11:10]!=2'b11)&&WE;
reg [31:0] CSRReg[4095:0];
integer i;

always@(negedge clk or posedge rst)
begin
    if(rst) begin
        for(i=0;i<4096;i=i+1)
            CSRReg[i][31:0] <= 32'b0;
    end
    else if(WE_VALID) begin
        CSRReg[A1] <=  WD;
    end
end
assign RD = CSRReg[A];
endmodule
```

`CSRAlu.v`中实现三种操作，分别是交换(csrrw),置位(csrrs)和清除(csrrc)。对应的CSRAluCtl信号共三种，分别是SWAP,SET,CLEAR。均在`Parameters.v`中添加了定义。

```
//Parameters.v(部分)
//CSRAlu
    `define SWAP  2'b01
    `define SET   2'b10
    `define CLEAR 2'b11
```

CSRAlu的两个操作数分别是Rs1对应的寄存器值和CSR寄存器的值。CSRAlu有两个输出，分别是AluOut和CSROut。CSROut是要写道CSR中的值，ALUout是要写入到RegisterFile中的值。

```
//CSRALU.v
module CSRALU(
    input wire [31:0] Operand1,
```

```
    input wire [31:0] Operand2,
    input wire [1:0] AluCTL,
    output reg [31:0] AluOut,
    output reg [31:0] CSROut
);
always@(*)
begin
    case (AluCTL)
        `SWAP: begin
            AluOut <= Operand2;
            CSROut <= Operand1;
        end
        `SET: begin
            AluOut <= Operand2;
            CSROut <= Operand2 | Operand1;
        end
        `CLEAR: begin
            AluOut <= Operand2;
            CSROut <= (~Operand1) & Operand2;
        end
        default:begin
            AluOut <= Operand1;
            CSROut <= Operand2;
        end
    endcase
end
endmodule
```

在ControlUnit添加CSR的控制信号:CSRWriteD,CSRAluCtlD,CSRRead，CSRAlusrc1D,AluOutSrc。CSRWriteD、CSRRead控制CSR的读写。对于 CSRRS和CSRRC指令，如果 rs1 = x0，那么指令将根本不会去写CSR。对应CSRRW，如果 rd = x0，那么这条指令将不会读该CSR。当CSR是立即数指令（CSRRWI,CSRRSI,CSRRCI）时，设置立即数类型为Ztype（在parameters.v中添加定义）。CSRAluCtlD为CSRALu的控制信号，上面已经介绍。AluOutSrc用于在CSRALu和ALU之间选择输出。当指令是CSR指令时，该信号为1。CSRAlusrc1D则用于选择CSRALU的操作数1是立即数还是Rs1对应的寄存器值（考虑到相关，实际上是ForwardData1E，下面会提到相关）。

在ImmOperandUnit中添加Ztype的对应实现(即csrrwi,csrrci,csrrsi用到的立即数)

```
`ZTYPE: Out <= { 27'b0,In[19:15]};
```

在HazardUnit中实现对应CSR的相关处理。关于Rs1的相关处理，阶段2已经实现。这里直接复用。即在EX段CSRAlu的Operrand1应从Imm和ForwardData1E中选择，控制信号为CSRAlusrc1。

```
//RV32Core.v(部分)
assign CSROperand1 = CSRAlusrc1E?ImmE:ForwardData1;
```

关于CSR寄存器的相关处理与之前的类似，考虑CSRRwrite和CSRRead信号。以及当前EX段CSR地址于Mem段、WB段的CSR地址是否相同。

```
//HarzardUnit.v(部分)
wire csrrs2hitm;
wire csrrs2hitw;
assign csrrs2hitm = (CSRWriteM)&&(CSRRs2E==CSRRdM)&&(CSRRdM[11:10]!=2'b11)&&
(CSRReadE);
```

```
assign csrrs2hitw = (CSRWriteW)&&(CSRRs2E==CSRRdW)&&(CSRRdW[11:10]!=2'b11)&&
(CSRReadE);

always @(*) begin//csr
    if(CpuRst) begin
        CSRForwardE <= 2'b00;
    end
    else begin
    case({csrrs2hitm,csrrs2hitw})
    2'b00:begin
        CSRForwardE <= 2'b00;
    end
    2'b01:begin
        CSRForwardE <= 2'b01;
    end
    2'b10:begin
        CSRForwardE <= 2'b10;
    end
    2'b11:begin
        CSRForwardE <= 2'b10;
    end
    endcase
    end
end
```

```
//RV32Core.v(部分)
assign CSROperand2 = CSRForwardE[1]?(CSRWDM) : (CSRForwardE[0] ? CSRWDW :
CSROutE);
```

各个段寄存器添加传递对应的CSR信号

CSR的测试样例:

```
lui sp,0x1
lui ra,0x1
csrrw   sp,ustatus,ra
lui ra,0x3
csrrs   sp,ustatus,ra
csrrc   sp,ustatus,ra
csrrwi  sp,ustatus,1
csrrsi  sp,ustatus,3
csrrci  sp,ustatus,3
csrrsi  sp,ustatus,0
```

仿真截图:

实验总结

收获：对rv32i指令集、数据通路以及流水线的相关知识更加熟悉。

踩的坑：阶段2涉及到末尾地址非11的数据存储与读取。CSR的相关处理需要考虑rd与rs是x0的情况。

所花时间：control unit花费的时间、阶段2debug以及阶段3构思CSR数据通路的时间较长

改进意见

实验课上可以详细讲一讲数据通路