

# 体系结构LAB2报告

---

裴启智

PB18111793

## 实验目标

---

- 阶段1
  - 自己手写合适的测试用汇编代码，通过提供的工具生成.inst和.data文件，用于初始化指令和数据的Block Memory，或者直接手写二进制测试代码
  - 测试用的指令流中需要包含的指令包括SLLI、SRLI、SRAI、ADD、SUB、SLL、SLT、SLTU、XOR、SRL、SRA、OR、AND、ADDI、SLTI、SLTIU、XORI、ORI、ANDI、LUI、AUIPC
  - 测试例（汇编和对应的.inst .data）可以用其他同学提供的，但是需要自己知道对应的指令逻辑，需要能清楚的向助教表达这个测试例如何验证CPU功能正确，即正确运行后寄存器值应该是多少
  - CPU执行后，各寄存器值符合预期
  - 此时不需要处理数据相关。可以令Harzard模块始终输出stall、flush恒为0，forward恒为不转发，每两条指令之间间隔四条空指令。
- 阶段2
  - 我们提供了 1testAll.data、1testAll.inst、2testAll.data、2testAll.data、3testAll.data、3testAll.inst 三个测试样例的.inst 和.data 文件，用于初始化指令和数据的 Block Memory。
  - 对于任意一个测试样例，CPU 开始执行后 3 号寄存器的值会从 2 一直累增，该数字正在进行第多少项测试，执行结束后 3 号寄存器值变为 1
  - CPU 执行后，各寄存器值符合预期
  - 测试用的指令流中，除了阶段一的测试指令，还需要包含的指令包括 JALR、LB、LH、LW、LBU、LHU、SB、SH、SW、BEQ、BNE、BLT、BLTU、BGE、BGEU、JAL
  - 此时需要处理数据相关，实现 Harzard 模块内部逻辑。
- 阶段3
  - 自己手写合适的测试用汇编代码，通过提供的工具生成.inst 和.data 文件，用于初始化指令和数据的 Block Memory
  - 在我们给的代码框架上添加你设计好的 CSR 数据通路
  - 测试用的指令流中需要包含的指令包括: CSRRW、CSRRS、CSRRC、CSRRWI、CSRRSI、CSRRCI
  - CPU 执行后，各寄存器值符合预期
  - 阶段二已经处理好数据相关，这里不再特别考察(不代表不用实现流水线相关)

## 实验环境和工具

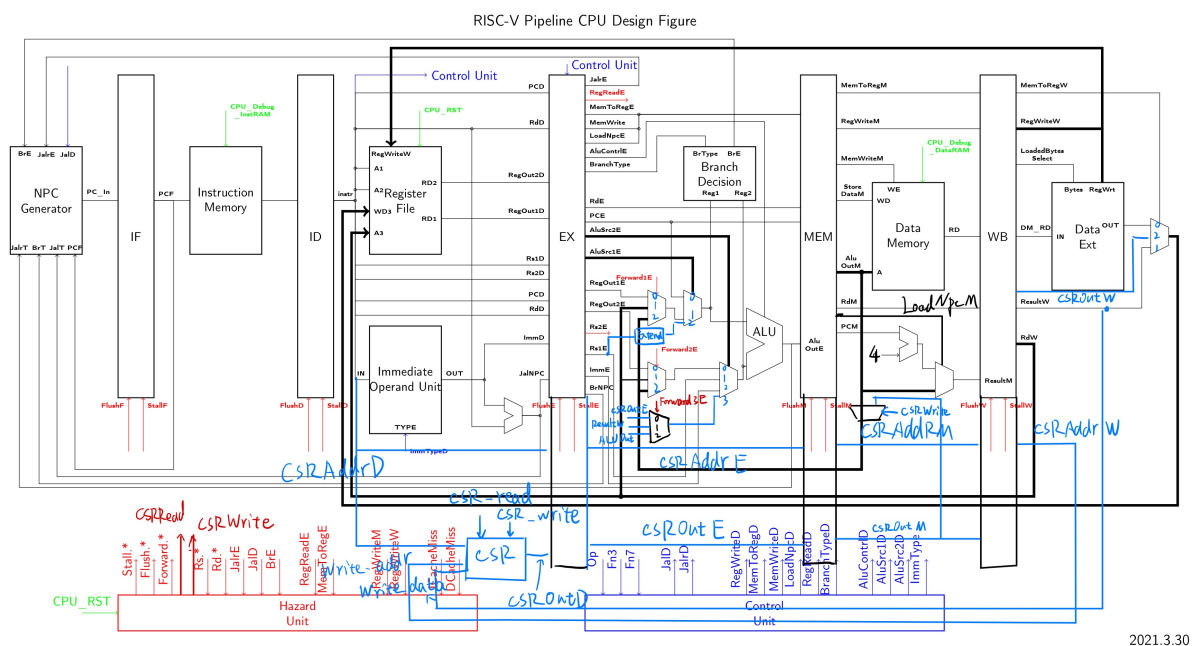
---

- Windows10 Professional
- Vivado 2019.2

## 实验内容和过程

---

# 设计图



2021.3.30

## 阶段1 2

- 这两部分主要是对助教给出的代码框架进行补全，下面对补全的模块进行——说明
- 代码中关于 CSR 的部分会在 阶段3 中说明
- 需要参考 RISC-V 指令集以及 insts.md 搞懂每个指令的细节工作

## ALU

这一部分主要是根据 RISC-V 指令集中各个指令的运算方式，根据译码得到的 AluControl 信号进行相应的运算

需要注意的是

- 区分有符号和无符号运算，使用 `wire signed [31:0] operand1_Signed = $signed(Operand1)` 的操作将无符号转化为有符号
- 位移运算，由于左移运算都是低位补0，所以不用区分算术左移和逻辑左移，而右移指令区分算术左移和逻辑右移，算术右移用 `>>>` 表示，逻辑右移用 `>>` 表示

```
module ALU(
    input wire [31:0] Operand1,
    input wire [31:0] Operand2,
    input wire [3:0] AluControl,
    output reg [31:0] AluOut
);

//请补全此处代码

//将输入转化为有符号数
wire signed [31:0] operand1_Signed = $signed(Operand1);
wire signed [31:0] operand2_Signed = $signed(Operand2);

always@(*)
begin
    case (AluControl)
        `ADD: AluOut <= operand1 + operand2;
        `SUB: AluOut <= operand1 - operand2;
```

```

`XOR: AluOut <= Operand1 ^ Operand2;
`OR: AluOut <= Operand1 | Operand2;
`AND: AluOut <= Operand1 & Operand2;
//逻辑左移
`SLL: AluOut <= Operand1 << (Operand2[4:0]);
//逻辑右移
`SRL: AluOut <= Operand1 >> (Operand2[4:0]);
//算术右移
`SRA: AluOut <= Operand1_Signed >>> (Operand2[4:0]);
//有符号数比较
`SLT: AluOut <= Operand1_Signed < Operand2_Signed ? 32'd1 : 32'd0;
//无符号数比较
`SLTU: AluOut <= Operand1 < Operand2 ? 32'd1 : 32'd0;
//构建32位常数，使用U类格式
`LUI: AluOut <= {Operand2[31:12], 12'b0};
//CSR
`CLR: AluOut <= ~Operand1 & Operand2;
`REG1: AluOut <= Operand1;
default: AluOut <= 32'hxxxxxxxx;
endcase
end
endmodule

```

## BranchDecision

这里主要是根据不同类型的 Branch 指令以及操作数来判断是否需要跳转

需要注意的是

- 跳转指令操作数仍需区分有符号和无符号，同 ALU
- 等于和不等对于有符号和无符号都是英语的

```

module BranchDecisionMaking(
    input wire [2:0] BranchTypeE,
    input wire [31:0] Operand1, Operand2,
    output reg BranchE
);

//请补全此处代码
wire signed [31:0] Operand1_Signed = $signed(Operand1);
wire signed [31:0] Operand2_Signed = $signed(Operand2);
always@(*)
begin
    case(BranchTypeE)

        `BEQ:
        if(Operand1 == Operand2)
            BranchE <= 1'b1;
        else
            BranchE <= 1'b0;

        `BNE:
        if(Operand1 != Operand2)
            BranchE <= 1'b1;
        else
            BranchE <= 1'b0;

        `BLT:

```

```

    if(Operand1_Signed < Operand2_Signed)
        BranchE <= 1'b1;
    else
        BranchE <= 1'b0;

    `BLTU:
    if(Operand1 < Operand2)
        BranchE <= 1'b1;
    else
        BranchE <= 1'b0;

    `BGE:
    if(Operand1_Signed >= Operand2_Signed)
        BranchE <= 1'b1;
    else
        BranchE <= 1'b0;

    `BGEU:
    if(Operand1 >= Operand2)
        BranchE <= 1'b1;
    else
        BranchE <= 1'b0;

    //NOBRANCH, 默认为0
    default: BranchE <= 1'b0;

endcase
end
endmodule

```

## DataExt

主要是根据不同类型的 Load 指令，根据 LoadedBytesSelect 选取从 Mem 读出的数据的某些位，并扩展为32位

需要注意的是

- `assign IN_Sel_8 = (IN >> (32'h8 * LoadedBytesSelect)) & 32'h000000ff` 操作根据 LoadedBytesSelect对IN进行右移，将要取的部分放到低8位，然后在后面的 always 块中取对应的位，并作有符号或无符号的位扩展，其余同理

```

module DataExt(
    input wire [31:0] IN,
    input wire [1:0] LoadedBytesSelect,
    input wire [2:0] Regwritew,
    output reg [31:0] OUT
);

// 请补全此处代码
wire [31:0] IN_Sel_8;
wire [31:0] IN_Sel_16;
//取低8位，根据LoadedBytesSelect对IN进行右移，将要取的部分放到低8位，下同
assign IN_Sel_8 = (IN >> (32'h8 * LoadedBytesSelect)) & 32'h000000ff;
assign IN_Sel_16 = (IN >> (32'h8 * LoadedBytesSelect)) & 32'h0000ffff;

```

```

always@(*)
begin
    case (RegWriteW)
        //8bits, 有符号
        `LB: OUT <= {{24{IN_Sel_8[7]}}, IN_Sel_8[7:0]};
        //16bits, 有符号
        `LH: OUT <= {{16{IN_Sel_16[15]}}, IN_Sel_16[15:0]};

        `LW: OUT <= IN;
        //8bits, 无符号
        `LBU: OUT <= {24'b0, IN_Sel_8[7:0]};
        //16bits, 无符号
        `LHU: OUT <= {16'b0, IN_Sel_16[15:0]};

        default: OUT = 32'hxxxxxxx;
    endcase
end

endmodule

```

## IDSegReg

主要是完善 InstructionRam,

需要注意的是

- 指令存储器按字寻址，所以 addra 信号应该连接 A 的高30位

```

InstructionRam InstructionRamInst (
    .clk      ( clk ),                      //请完善代码
    //按字寻址
    .addra    ( A[31:2] ),                  //请完善代码
    .douta    ( RD_raw ),
    .web      ( |WE2 ),
    .addrb    ( A2[31:2] ),
    .dinb     ( WD2 ),
    .doutb    ( RD2 )
);

```

## WBSegReg

主要是完善 DataRam

需要注意的是

- wea 接口和 WE相连，需要处理非字对齐 store。但二者位数不同，WE共4bit，为1的部分表示有效，而 wea 只接收 1 位信号，所以通过 `WE << A[1:0]` 来判断是否可写
- 数据存储器按字寻址，所以 addra 信号应该连接 A 的高30位
- dina 接口也需要处理非字对齐store，通过 `WD << (32'h8 * A[1:0])` 来从 32bit大小的 WD 中挑选出我们需要写入的数据

```

DataRam DataRamInst (
    .clk      ( clk ),                      //请完善代码

```

```

//WE接口和MemWriteE相连，对应非字对齐store，但二者位数不同
//MemWriteE共4bit，为1的部分表示有效，而wea只接收一位信号，所以要取出来1

.wea      ( WE << A[1:0] ),                      //请完善代码
.addra    ( A[31:2] ),                          //请完善代码
.dina     ( WD << (32'h8 * A[1:0]) ),            //请完善代码
.douta    ( RD_raw          ),
.web      ( WE2             ),
.addrb    ( A2[31:2]       ),
.dinb     ( WD2             ),
.doutb    ( RD2             )
);

```

## ImmOperandUnit

主要是根据不同的指令类型进行立即数的扩展，需要仔细查看手册，否则易出错

需要注意的是

- 立即数都是有符号的
- 在拼接时要检查拼接的项的和是否为 32位

```

module ImmOperandUnit(
    input wire [31:7] In,
    input wire [2:0] Type,
    output reg [31:0] Out
);
//
always@(*)
begin
    case(Type)
        `ITYPE: Out <= { {21{In[31]}}, In[30:20] };
        //请完善代码
        `STYPE: Out <= { {21{In[31]}}, In[30:25], In[11:7] };
        `BTYP: Out <= { {20{In[31]}}, In[7], In[30:25], In[11:8], 1'b0 };
        `UTYPE: Out <= { In[31:12], 12'b0 };
        `JTYPE: Out <= { {12{In[31]}}, In[19:12], In[20], In[30:21], 1'b0 };
        default: Out <= 32'hxxxxxxxx;
    endcase
end

endmodule

```

## NPC\_Generator

这里主要是根据跳转信号，判断 PC 的值如何变化

需要注意的是

- 优先级问题，根据实验 1 中的分析，Jalr 和 Branch 的优先级应高于 Jal 的优先级，所以在 always 块中的判断中应先判断 Jalr 和 Branch 指令，再判断 Jal 指令。如果三者都不为真，则代表顺序执行，PC + 4即可

```

module NPC_Generator(

```

```

input wire [31:0] PCF, JalrTarget, BranchTarget, JalTarget,
input wire BranchE, JalD, JalrE,
output reg [31:0] PC_In
);

// 请补全此处代码
always@(*)
begin
    //注意这里判断的顺序体现了优先级，EX阶段的优先级高于ID段
    if (JalrE)
        PC_In <= JalrTarget;

    else if(BranchE)
        PC_In <= BranchTarget;

    else if(JalD)
        PC_In <= JalTarget;

    else
        PC_In <= PCF + 4;
end

endmodule

```

## ControlUnit

主要是根据不同的指令类型，在译码阶段产生对应的控制信号。这一部分了解各个指令的详细功能，然后根据设计图分析各个信号的作用

需要注意的是

- 首先通过 assign 确定简单的多路选择器的信号
- 再通过 always 块的组合逻辑，确定一些较为复杂的信号

```

module ControlUnit(
    input wire [6:0] Op,
    input wire [2:0] Fn3,
    input wire [6:0] Fn7,
    //CSR
    input wire [4:0] Rs1D,
    input wire [4:0] RdD,
    output wire JalD,
    output wire JalrD,
    output reg [2:0] RegWrited,
    //CSR modify
    output wire [1:0] MemToRegD,

    output reg [3:0] MemWrited,
    output wire LoadNpcD,
    output reg [1:0] RegReadD,
    output reg [2:0] BranchTypeD,
    output reg [3:0] AluContr1D,
    output wire [1:0] AluSrc2D,
    //CSR modify
    output wire [1:0] AluSrc1D,
    output reg [2:0] ImmType,
    //CSR

```

```

output reg CSRReadD,
output reg CSRWriteD
);

// 请补全此处代码
//Load类指令
assign MemToRegD = (Op == 7'b0000011) ? 2'b01 :
                    (Op == 7'b1110011) ? 2'b10 : 2'b00;
//将NextPC输出到ResultM
assign LoadNpcD = JalD | JalrD;

assign JalD = (Op==7'b1101111) ? 1'b1 : 1'b0;
assign JalrD = (Op==7'b1100111) ? 1'b1 : 1'b0;

// AUIPC指令对应01, CSRRWI、CSRRSI、CSRRCI指令对应10
assign AluSrc1D = (Op == 7'b1110011 && Fn3[2] == 1) ?
                  2'b10 :
                  ((Op == 7'b0010011) ? 2'b01 : 2'b00);

//若为立即数位移运算, 则Op == 7'b0010011, 且Fn3[1:0] == 2'b01, 则对应SLLI、SRLI、
SRAI
//若为R类指令或Branch指令, 则对应2'b00
//其他立即数类指令, 对应2'b10
assign AluSrc2D = ((Op == 7'b0110011) || (Op == 7'b1100011)) ? 2'b00 :
                  (((Op == 7'b0010011) && (Fn3[1:0] == 2'b01)) ? 2'b01 :
                  ((Op == 7'b1110011) ? 2'b11 : 2'b10));

always@(*)
begin
    if(Op == 7'b1100011) //判断是Branch指令
    begin
        case(Fn3)
            3'b000: BranchTypeD <= `BEQ;
            3'b001: BranchTypeD <= `BNE;
            3'b100: BranchTypeD <= `BLT;
            3'b101: BranchTypeD <= `BGE;
            3'b110: BranchTypeD <= `BLTU;
            3'b111: BranchTypeD <= `BGEU;

            endcase
        end
    else BranchTypeD <= `NOBRANCH;
end

//表示A1和A2对应的寄存器值是否被使用到了, 用于forward的处理
always@(*)
begin
    case(ImmType)
        `UTYPE: RegReadD = 2'b00;
        `JTYPE: RegReadD = 2'b00;
        //I类指令, 表示用到了A1, 没有用到A2
        `ITYPE: RegReadD = 2'b10;
        `RTYPE: RegReadD = 2'b11;
        `STYPE: RegReadD = 2'b11;
        `BTYP: RegReadD = 2'b11;
    endcase
end

```



```

        default: RegReadD = 2'b00;
    endcase
end

//CSRRead and CSRWrite
//当rs=0时，不对CSR进行操作
always@(*)
begin
    // if(Op == 7'b11100110)
    // begin
    //     CSRReadD <= 1;
    //     CSRWriteD <= 1;
    // end
    if(Op == 7'b1110011 && Rs1D != 5'b0)
    begin
        CSRReadD <= 1;
        CSRWriteD <= 1;
    end
    else if(Op == 7'b1110011 && Rs1D == 5'b0 && Fn3[1] != 1'b0)
    begin
        CSRReadD <= 1;
        CSRWriteD <= 0;
    end
    else if(Op == 7'b1110011 && RdD == 5'b0 && Fn3[1] == 1'b0)
    begin
        CSRReadD <= 1;
        CSRWriteD <= 0;
    end
    else
    begin
        CSRReadD <= 1;
        CSRWriteD <= 0;
    end
end

always@(*)
begin
    case(Op)
        7'b0110011: //寄存器型运算
        begin
            ImmType <= `RTYPE;
            RegWriteD <= `LW;
            MemWriteD <= 4'b0000;

            case(Fn3)
                3'b000:
                begin
                    if(Fn7[5] == 0)
                        AluContr1D <= `ADD;
                    else
                        AluContr1D <= `SUB;
                end
                3'b001: AluContr1D <= `SLL;
                3'b010: AluContr1D <= `SLT;
                3'b011: AluContr1D <= `SLTU;
                3'b100: AluContr1D <= `XOR;
                3'b101:
            end
        end
    endcase
end

```

```

begin
    if(Fn7[5] == 0)
        AluContrlD <= `SRL;
    else
        AluContrlD <= `SRA;
    end
    3'b110: AluContrlD <= `OR;
    3'b111: AluContrlD <= `AND;
endcase
end

7'b0010011: //移位 立即数
begin
    ImmType <= `ITYPE;
    RegWritED <= `LW;
    MemWritED <= 4'b0000;

    case(Fn3)
        3'b000: AluContrlD <= `ADD;
        3'b001: AluContrlD <= `SLL;
        3'b010: AluContrlD <= `SLT;
        3'b011: AluContrlD <= `SLTU;
        3'b100: AluContrlD <= `XOR;
        3'b101:
            if(Fn7[5]==1)
                AluContrlD <= `SRA;
            else
                AluContrlD <= `SRL;
            3'b110: AluContrlD <= `OR;
            3'b111: AluContrlD <= `AND;
        endcase
    end

7'b0000011: //存储器读
begin
    ImmType <= `ITYPE;
    MemWritED <= 4'b0000;
    AluContrlD <= `ADD;

    case(Fn3)
        3'b000: RegWritED <= `LB;
        3'b001: RegWritED <= `LH;
        3'b010: RegWritED <= `LW;
        3'b100: RegWritED <= `LBU;
        3'b101: RegWritED <= `LHU;
        default: RegWritED <= `LW;
    endcase
end

7'b0100011: //存储器写
begin
    ImmType <= `STYPE;
    RegWritED <= `NOREGWRITE;
    AluContrlD <= `ADD;

    case(Fn3)

```

```

        3'b000: MemWritED <= 4'b0001;    //SB
        3'b001: MemWritED <= 4'b0011;    //SH
        3'b010: MemWritED <= 4'b1111;    //SW
        default: MemWritED <= 4'b1111;    //SW

    endcase
end

7'b1101111: //JAL
begin
    ImmType <= `JTYPE;
    RegWritED <= `LW;
    MemWritED <= 4'b0000;
    AluContrlD <= `ADD;
end

7'b1100111: //JALR
begin
    ImmType <= `ITYPE;
    RegWritED <= `LW;
    MemWritED <= 4'b0000;
    AluContrlD <= `ADD;

end

7'b1100011: //Branch
begin
    ImmType <= `BTYPE;
    RegWritED <= `NOREGWRITE;
    MemWritED <= 4'b0000;
    AluContrlD <= `ADD;
end

7'b0110111: //LUI
begin
    RegWritED <= `LW;
    MemWritED <= 4'b0000;
    AluContrlD <= `LUI;
    ImmType <= `UTYPE;
end

7'b0010111: //AUIPC, 通过ADD来实现
begin
    RegWritED <= `LW;
    MemWritED <= 4'b0000;
    AluContrlD <= `ADD;
    ImmType <= `UTYPE;
end

7'b1110011: //CSR
begin

    RegWritED <= `LW;
    ImmType <= `ITYPE;
    case (Fn3)
        3'b001: AluContrlD <= `REG1;
        3'b010: AluContrlD <= `OR;
        3'b011: AluContrlD <= `CLR;
        3'b101: AluContrlD <= `REG1;
    endcase
end

```

```

        3'b110: AluContrlD <= `OR;
        3'b111: AluContrlD <= `CLR;
        default: AluContrlD <= `CLR;
    endcase
end
default:
begin //无效指令 or 空指令
    ImmType <= `ITYPE;
    RegWritED <= `NOREGWRITE;
    MemWritED <= 4'b0000;
    AluContrlD <= `ADD;
end

endcase
end

endmodule

```

## HarzardUnit

主要是实现旁路以及控制 Stall 和 Flush，主要处理数据相关和控制相关

- 数据相关：通过设置旁路来解决
  - 如果无法通过旁路来解决，如 Load + R 类指令，需要设置 Stall 和 Flush 使流水线停顿，即IF和ID段停止，EX段被刷新

```

//无法通过旁路解决的RAW相关,插入一个bubble
//停止读指令，停止解码，刷新EX段，MEM和WB不变使得Load指令再执行一个cycle
else if (((RdE == Rs1D) || (RdE == Rs2D)) && MemToRegE[0])
begin
    StallF <= 1'b1;
    FlushF <= 1'b0;
    StallD <= 1'b1;
    FlushD <= 1'b0;
    StallE <= 1'b0;
    FlushE <= 1'b1;
    StallM <= 1'b0;
    FlushM <= 1'b0;
    StallW <= 1'b0;
    FlushW <= 1'b0;
end

```

- 控制相关：主要是处理跳转指令 Branch、Jal、Jalr，只需在跳转成功时将对应的段寄存器清空，即让后面的指令不再运行

```

module HarzardUnit(
    input wire CpuRst, ICacheMiss, DCacheMiss,
    input wire BranchE, JalrE, JalD,
    input wire [4:0] Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW,
    input wire [1:0] RegReadE,
    //CSR modify
    input wire [1:0] MemToRegE,
    input wire [2:0] RegwriteM, Regwritew,

```

```

//CSR
input wire [11:0] CSRSrcE, CSRSrcM, CSRSrcW,
input wire CSRReadE, CSRWriteM, CSRWriteW,

output reg StallF, FlushF, StallD, FlushD, StallE, FlushE, StallM, FlushM,
StallW, FlushW,
output reg [1:0] Forward1E, Forward2E,
//CSR
output reg [1:0] Forward3E
);

//请补全此处代码
//这里需要注意转发优先级，应优先转发离当前指令近的
//Forward1E
always@(*)
begin
    //当前指令EX段需要用到之前指令WB段得到的值,且这个值要写回寄存器，且当前用到了这个值，
    且写回的不是0号寄存器
    if((RdM == Rs1E) && (RegWriteM != 3'b0) && (RegReadE[1] == 1) && (RdM !=
5'b0))

        Forward1E <= 2'b10;

    //当前指令EX段需要用到之前指令MEM段得到的值,且这个值要写回寄存器，且当前用到了这个值，
    且写回的不是0号寄存器
    else if((RdW == Rs1E) && (RegWriteW != 3'b0) && (RegReadE[1] == 1) &&
(RdW != 5'b0))
        Forward1E <= 2'b01;
    else
        Forward1E <= 2'b00;
end

//Forward2E
always@(*)
begin
    //当前指令EX段需要用到之前指令WB段得到的值,且这个值要写回寄存器，且当前用到了这个值，
    且写回的不是0号寄存器
    if((RdM == Rs2E) && (RegWriteM != 3'b0) && (RegReadE[0] == 1) && (RdM !=
5'b0))

        Forward2E <= 2'b10;

    //当前指令EX段需要用到之前指令MEM段得到的值,且这个值要写回寄存器，且当前用到了这个值，
    且写回的不是0号寄存器
    else if((RdW == Rs2E) && (RegWriteW != 3'b0) && (RegReadE[0] == 1) &&
(RdW != 5'b0))
        Forward2E <= 2'b01;
    else
        Forward2E <= 2'b00;
end

//Forward3E, 用以处理读写CSR带来的冲突 (RAW)
always @(*)
begin
    if(CSRReadE && CSRWriteM && CSRSrcM == CSRSrcE)
        Forward3E <= 2'b10;
    else if(CSRReadE && CSRWriteW && CSRSrcW == CSRSrcE)
        Forward3E <= 2'b01;
    else
        Forward3E <= 2'b00;
end

```

```

//Stall and Flush
always@(*)
begin
    //当CpuRst==1时CPU全局复位清零（所有段寄存器flush），Cpu_Rst==0时cpu开始执行指令
    if (CpuRst)
    begin
        StallF <= 1'b0;
        FlushF <= 1'b1;
        StallD <= 1'b0;
        FlushD <= 1'b1;
        StallE <= 1'b0;
        FlushE <= 1'b1;
        StallM <= 1'b0;
        FlushM <= 1'b1;
        StallW <= 1'b0;
        FlushW <= 1'b1;
    end
    //Jalr和Branch在EX段结束，需要刷新FlushD和FlushE
    else if (JalrE || BranchE)
    begin
        StallF <= 1'b0;
        FlushF <= 1'b0;
        StallD <= 1'b0;
        FlushD <= 1'b1;
        StallE <= 1'b0;
        FlushE <= 1'b1;
        StallM <= 1'b0;
        FlushM <= 1'b0;
        StallW <= 1'b0;
        FlushW <= 1'b0;
    end
    //无法通过旁路解决的RAW相关,插入一个bubble
    //停止读指令，停止解码，刷新EX段，MEM和WB不变使得Load指令再执行一个cycle
    else if (((RdE == Rs1D) || (RdE == Rs2D)) && MemToRegE[0])
    begin
        StallF <= 1'b1;
        FlushF <= 1'b0;
        StallD <= 1'b1;
        FlushD <= 1'b0;
        StallE <= 1'b0;
        FlushE <= 1'b1;
        StallM <= 1'b0;
        FlushM <= 1'b0;
        StallW <= 1'b0;
        FlushW <= 1'b0;
    end

    //Jal在ID段结束，只需刷新FlushD
    else if (JalD)
    begin
        StallF <= 1'b0;
        FlushF <= 1'b0;
        StallD <= 1'b0;
        FlushD <= 1'b1;
        StallE <= 1'b0;
        FlushE <= 1'b0;
        StallM <= 1'b0;
        FlushM <= 1'b0;
    end
end

```

```

        Stallw <= 1'b0;
        Flushw <= 1'b0;
    end

    //其他情况
    else
    begin
        StallF <= 1'b0;
        FlushF <= 1'b0;
        StallD <= 1'b0;
        FlushD <= 1'b0;
        StallE <= 1'b0;
        FlushE <= 1'b0;
        StallM <= 1'b0;
        FlushM <= 1'b0;
        Stallw <= 1'b0;
        Flushw <= 1'b0;
    end
end

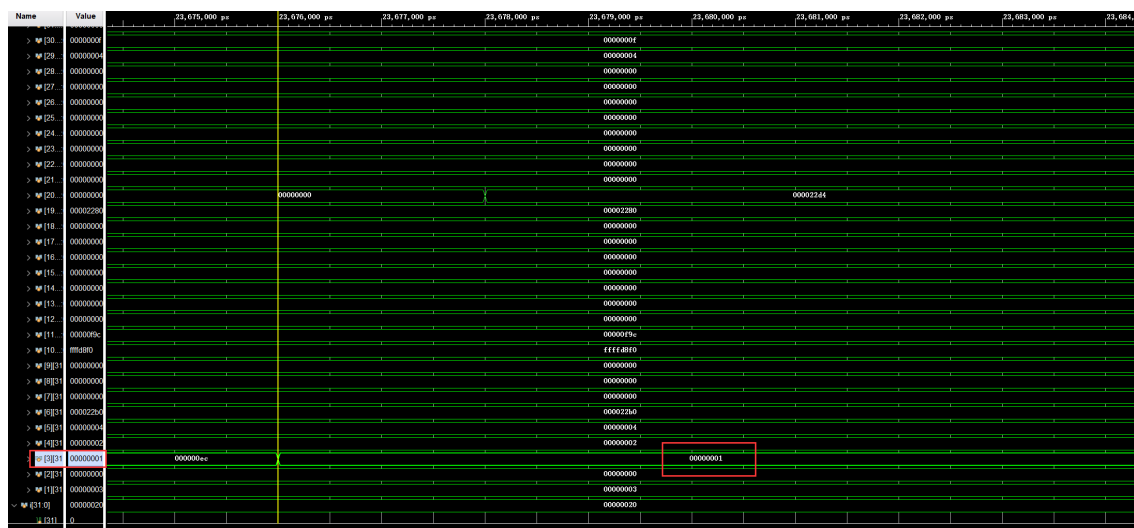
endmodule

```

## 仿真结果

- 运行助教给出的3个测试样例，检查 regfile[3] 的变化情况，其值代表运行到第几个 test。如果全部测试通过则其值变为 1

- 1testAll



- 2testAll





- CSRRWI指令、CSRRSI指令、CSRRCI指令分别于CSRRW指令、CSRRS指令、CSRRC指令相似，除了它们是使用一个处于rs1字段的、零扩展到XLEN位的5位立即数（zimm[4:0]）而不是使用rs1整数寄存器的值。如果zimm[4:0]字段是零，那么这些指令将不会写CSR，因此应该不会产生任何由于写CSR产生的副作用
- 在 lab1 中设计的 CSR 通路的基础上作一定的修改，实现各个 CSR 指令要求的功能

## Parameters

- 添加 CSR 相关的参数

```
`define CLR 4'd11
`define REG1 4'd12
```

## ALU

- 添加 CSR 指令相关的运算
- CSRRW, CSRRWI 直接将 RegFile[rs1] 或 zimm的零扩展存入 CSR[rd]，使用如下的 REG1信号，直接传输即可
- CSRRS, CSRRSI 根据 RegFile[rs1] 或 zimm的零扩展对 CSR[rd]的对应位置1，使用本身的或运算即可
- CSRRC, CSRRCI 根据 RegFile[rs1] 或 zimm的零扩展对 CSR[rd]的对应位置0，定义如下的 CLR 运算

```
//CSR
`CLR: AluOut <= ~Operand1 & Operand2;
`REG1:AluOut <= Operand1;
```

## CSR

CSR 本身就是一个寄存器文件，写法和 RegisterFile 相似

需要注意的是

- CSR指令的 CSR\_Addr 有12位，所以共有 4096 个寄存器
- CSR\_read信号主要用来控制：对于CSRRW和CSRRWI，如果 rd = x0，那么这条指令将不会读该CSR，且不会导致任何因为CSR 读而出现的副作用
- CSR\_write信号主要用来控制：对于CSRRS、CSRRSI、CSRRC、CSRRCI指令，如果 rs1 = x0或 zimm[4:0]字段是零，则这些指令将不会写 CSR，因此应该不会产生任何由于写 CSR 产生的副作用

```
module CSR_Reg(
    input wire clk,
    input wire rst,
    input wire CSR_write,
    input wire CSR_read,
    input wire [11:0] Read_addr,
    input wire [11:0] Write_addr,
    input wire [31:0] Write_data,
    output wire [31:0] Out
);
    integer i;
    //2^12次方个单元，每个单元12bit
    reg [31:0] CSR_file[4095:0];

    initial
    begin
```

```

        for(i = 0; i < 4096; i = i + 1)
            CSR_file[i][31:0] <= 32'b0;
    end

    always@(negedge clk or posedge rst)
    begin
        if (rst)
            for (i = 0; i < 4096; i = i + 1)
                CSR_file[i][31:0] <= 32'b0;
            else if(CSR_write)
                CSR_file[Write_addr] <= write_data;
        end

        //attention
        assign Out = CSR_read ? CSR_file[Read_addr] : 32'b0;
    endmodule

```

## EXSegReg

- 修改 AluSrc1和 MemToReg信号的长度，并将 CSR 的信号向后传递

## MEMSegReg

- 修改 AluSrc1和 MemToReg信号的长度，并将 CSR 的信号向后传递

## WBSegReg

- 修改 AluSrc1和 MemToReg信号的长度，并将 CSR 的信号向后传递

## ControlUnit

- 根据数据通路修改 MemToReg、AluSrc1、AluSrc2 信号，添加和 CSR 有关的判断
- 通过 Fn3 区分不同的 CSR 运算，赋予 ALU 不同的控制信号

```

7'b1110011: //CSR
begin

    RegWrited <= `LW;
    ImmType <= `ITYPE;
    case (Fn3)
        3'b001: AluContr1D <= `REG1;
        3'b010: AluContr1D <= `OR;
        3'b011: AluContr1D <= `CLR;
        3'b101: AluContr1D <= `REG1;
        3'b110: AluContr1D <= `OR;
        3'b111: AluContr1D <= `CLR;
        default: AluContr1D <= `CLR;
    endcase
end

```

- 添加 CSRRead 和 CSRWrite 信号，用于
  - 控制 CSR 的读写使能
  - HarzardUnit 中定向路径的判断
- CSR的特殊处理，同前所述
  - 对于CSRRead和CSRWrite，如果 rd = x0，那么这条指令将不会读该 CSR，且不会导致任何因为 CSR 读而出现的副作用，由CSR\_read信号控制

- CSR\_write信号主要用来控制：对于CSRRead、CSRWrite、CSRRC、CSRRCI指令，如果 rs1 = x0 或 zimm[4:0]字段是零，则这些指令将不会写 CSR，因此应该不会产生任何由于写 CSR 产生的副作用，由CSR\_write信号控制

```
//CSRRead and CSRWrite
//当rs=0时，不对CSR进行操作
always@(*)
begin
    if(Op == 7'b1110011 && Rs1D != 5'b0)
    begin
        CSRReadD <= 1;
        CSRWriteD <= 1;
    end
    else if(Op == 7'b1110011 && Rs1D == 5'b0 && Fn3[1] != 1'b0)
    begin
        CSRReadD <= 1;
        CSRWriteD <= 0;
    end
    else if(Op == 7'b1110011 && RdD == 5'b0 && Fn3[1] == 1'b0)
    begin
        CSRReadD <= 1;
        CSRWriteD <= 0;
    end
    else
    begin
        CSRReadD <= 1;
        CSRWriteD <= 0;
    end
end
end
```

## HarzardUnit

添加 CSR 有关的数据相关的处理

- rs1有关的相关在阶段 2 中处理过了
- 对于 CSR 的 I 类指令，通过将 Rs1E 进行位扩展，然后通过 AluSrc1 进行选择，这里实际上没有相关，因为用到的立即数可以直接从指令本身得到
- rd 有关的相关：CSR指令写入 RegFile[rd]，而后面有指令要读 RegFile[rd]
  - 如果当前数据处于 Mem 段，直接修改数据通路，加一个选择器即可，相当于新增一个 ForwardCSR\_or\_Alu，并利用之前的定向路径

```
// RV32Core
assign ForwardCSR_or_Alu = CSRWriteM ? CSROutM : AluOutM;
```

- 如果当前数据处于 WB 段，只需连接 CSROutW 到 MemToRegW 对应的多路选择器即可
- CSR 寄存器的 RAW 数据相关
  - 添加一个如下的多路选择器，由信号 Forward3E控制

```
// RV32Core
assign ForwardData3 = Forward3E[1] ? AluOutM : (Forward3E[0] ? ResultW : CSROutE);
```

- 在 HarzardUnit 中添加如下

```
//Forward3E, 用以处理读写CSR带来的冲突 (RAW)
always @(*)
begin
    if(CSRReadE && CSRWriteM && CSRSrcM == CSRSrcE)
        Forward3E <= 2'b10;
    else if(CSRReadE && CSRWriteW && CSRSrcW == CSRSrcE)
        Forward3E <= 2'b01;
    else
        Forward3E <= 2'b00;
end
```

## RV32Core

- CSR 对该文件引入了一定的修改，如AluSrc1和 MemToReg信号的长度、Forward3的连接等等，并修改了一些线路的连接方式

```
//CSR modify
assign ForwardData1 = Forward1E[1]?(ForwardCSR_or_Alu):( Forward1E[0]?
RegWriteData:RegOut1E );

assign Operand1 = AluSrc1E[1]? zimm : (AluSrc1E[0]?PCE:ForwardData1);

assign ForwardData2 = Forward2E[1]?(ForwardCSR_or_Alu):( Forward2E[0]?
RegWriteData:RegOut2E );

assign Operand2 = AluSrc2E[1]?(AluSrc2E[0]?ForwardData3:Imme):(
AluSrc2E[0]?Rs2E:ForwardData2 );
assign ResultM = LoadNpcM ? (PCM+4) : AluOutM;
//assign RegWriteData = ~MemToRegW?Resultw:DM_RD_Ext;
assign RegWriteData = MemToRegW[1] ? CSROutW : (MemToRegW[0] ? DM_RD_Ext
: Resultw);

//CSR
assign CSRAAddrD = Instr[31:20];

assign ForwardData3 = Forward3E[1] ? AluOutM : (Forward3E[0] ? Resultw :
CSROutE);

assign ForwardCSR_or_Alu = CSRWriteM ? CSROutM : AluOutM;
```

- 后面每个模块的连接也有相应的变化，主要是传递 CSR 有关的控制信号以及连接 CSR 寄存器文件，这里只给出 CSR 的连接

```
//CSR
CSR_Reg RegisterFile2(
    .clk(CPU_CLK),
    .rst(CPU_RST),
    .CSR_write(CSRWriteW),
    .CSR_read(CSRReadD),
    .Read_addr(CSRAddrD),
    .Write_addr(CSRAddrW),
    .Write_data(ResultW),
    .Out(CSROutD)
);
```

## 仿真结果

- 利用其他同学写好的测试集，仿照助教的测试集，检查每条 CSR 指令。检查 regfile[3] 的变化情况，其值代表运行到第几个 test。如果全部测试通过则其值变为 1
- 测试集如下

```
4testAll.om:      file format elf32-littleriscv

Disassembly of section .text:

00010080 <_start>:
    10080:  00000013          nop

00010084 <test2>:
    10084:  01100093      li  ra,17
    10088:  01200113      li  sp,18
    1008c:  00a09173      csrrw sp,0xa,ra
    10090:  00200193      li  gp,2
    10094:  0e011263      bnez  sp,10178 <fail>

00010098 <test3>:
    10098:  00a09173      csrrw sp,0xa,ra
    1009c:  01100f13      li  t5,17
    100a0:  00300193      li  gp,3
    100a4:  0de11a63      bne  sp,t5,10178 <fail>

000100a8 <test4>:
    100a8:  01100093      li  ra,17
    100ac:  01200113      li  sp,18
    100b0:  00a09173      csrrw sp,0xa,ra
    100b4:  00a9d173      csrrwi sp,0xa,19
    100b8:  00400193      li  gp,4
    100bc:  0a209e63      bne  ra,sp,10178 <fail>

000100c0 <test5>:
    100c0:  00a09173      csrrw sp,0xa,ra
    100c4:  01300f13      li  t5,19
    100c8:  00500193      li  gp,5
    100cc:  0be11663      bne  sp,t5,10178 <fail>

000100d0 <test6>:
    100d0:  01300093      li  ra,19
```

```

100d4: 01400113      li  sp,20
100d8: 00b0a173      csrrs  sp,0xb,ra
100dc: 00600193      li  gp,6
100e0: 08011c63      bnez  sp,10178 <fail>

000100e4 <test7>:
100e4: 00b09173      csrrw  sp,0xb,ra
100e8: 01300f13      li  t5,19
100ec: 00700193      li  gp,7
100f0: 09e11463      bne  sp,t5,10178 <fail>

000100f4 <test8>:
100f4: 01300093      li  ra,19
100f8: 01400113      li  sp,20
100fc: 00dde173      csrrsi sp,0xd,27
10100: 00800193      li  gp,8
10104: 06011a63      bnez  sp,10178 <fail>

00010108 <test9>:
10108: 00d09173      csrrw  sp,0xd,ra
1010c: 01b00f13      li  t5,27
10110: 00900193      li  gp,9
10114: 07e11263      bne  sp,t5,10178 <fail>

00010118 <test10>:
10118: 01100093      li  ra,17
1011c: 01200113      li  sp,18
10120: 00a09173      csrrw  sp,0xa,ra
10124: 00100093      li  ra,1
10128: 00a0b173      csrrc  sp,0xa,ra
1012c: 01100f13      li  t5,17
10130: 00a00193      li  gp,10
10134: 042f1263      bne  t5,sp,10178 <fail>

00010138 <test11>:
10138: 00a09173      csrrw  sp,0xa,ra
1013c: 01000f13      li  t5,16
10140: 00b00193      li  gp,11
10144: 022f1a63      bne  t5,sp,10178 <fail>

00010148 <test12>:
10148: 01300093      li  ra,19
1014c: 01400113      li  sp,20
10150: 00b0a173      csrrs  sp,0xb,ra
10154: 00b17173      csrrci sp,0xb,2
10158: 01300f13      li  t5,19
1015c: 00c00193      li  gp,12
10160: 002f1c63      bne  t5,sp,10178 <fail>

00010164 <test13>:
10164: 00b09173      csrrw  sp,0xb,ra
10168: 01100f13      li  t5,17
1016c: 00d00193      li  gp,13
10170: 01e11463      bne  sp,t5,10178 <fail>
10174: 00301463      bne  zero,gp,1017c <pass>

00010178 <fail>:
10178: 00000a6f      jal  s4,10178 <fail>

```

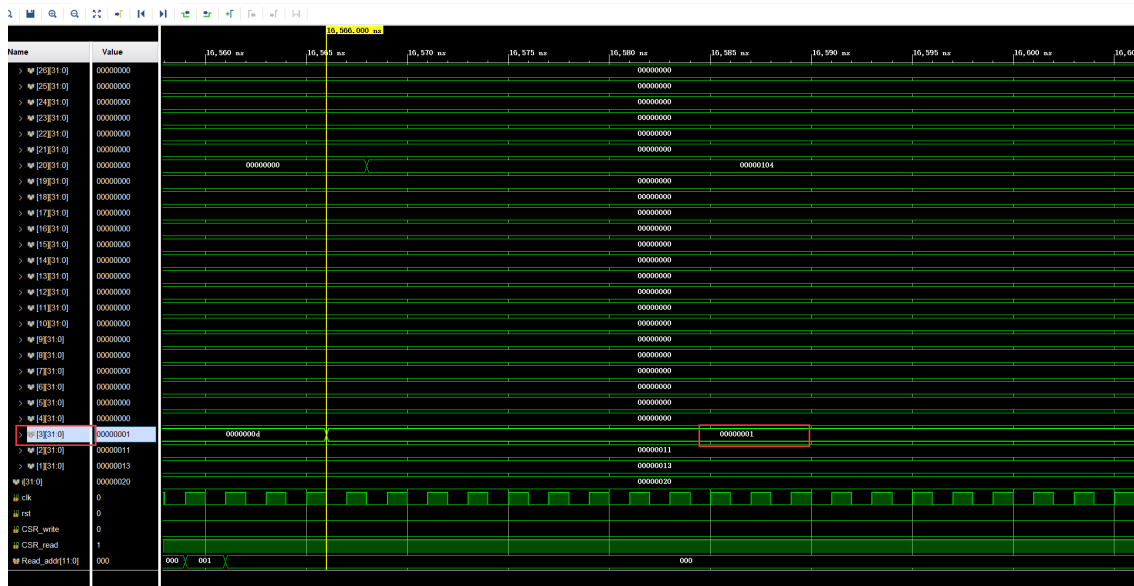
```

0001017c <pass>:
    1017c:    00100193          li    gp,1

00010180 <finish1>:
    10180:    00000a6f          jal   s4,10180 <finish1>
    10184:    c0001073          unimp
    . . .

```

## 仿真结果



## 实验总结

- 通过这次实验，熟悉了 RV32 流水线 CPU 整体的架构，并熟悉了各个指令运行的流程
- 一年没怎么用 Verilog 了，很多语法都差不多忘记了，花费了一定的时间来复习 Verilog 基础
- 对数据相关和控制相关有了更加深入的理解，大部分的数据相关可以通过旁路来解决，对于无法通过旁路解决的需要插入气泡，也就是控制 Flush 和 Stall 信号
- 在前两阶段中，刚开始写完之后，指令总是走到某个测试集就失败了。后面发现是 NPC\_generator 的优先级问题，不同跳转指令是有优先级的，对应 Verilog 中的判断顺序，不能随意指定
- 在第三阶段中对 LoadNpc 信号的用途不是很清楚，主要用在如下两处

```

assign LoadNpcD = JalD | JalrD
assign ResultM = LoadNpcM ? (PC+4) : AluOutM;

```

刚开始以为 Jal 和 Jalr 只是单纯的跳转，后续查阅手册才知道：

Jal 和 Jalr 将跳转指令后面指令的地址 (pc+4) 保存到寄存器 rd 中，所以说 Jal 和 Jalr 本质上也需要 5 个阶段才能完成

- 在写实验的时候是由简到繁补全模块的，ControlUnit 是我最后才完成的，里面有很多的信号，很容易弄错。自己在写的时候也非常谨慎，后面虽然还是出了少许的 bug，但通过仿真，再对照着指令手册很快就找到了 bug 所在。HarzardUnit 模块和去年 COD 实验中基本一致，当年自己花了很久才想明白 Harzard 模块到底应该怎么实现，本次实验中就省去了很多功夫
- 阶段3 的 CSR 本质上就是一个寄存器文件，所以这个模块本身不是很难。难的地方在于
  - CSR 指令的细节，主要是 rs1 = x0 或 rd = x0 或 zimm = 0 时的处理
  - CSR 指令的相关，会和哪些指令产生什么样的相关，旁路的设计等等

- 本次实验断断续续花费了4-5天的时间，主要是 debug 以及设计 CSR 的数据通路等较为耗时。前两阶段约3天，CSR阶段约2天。虽然做实验期间遇到 bug 却找不到在哪会让人很焦躁，但整个实验下来自己还是有不少的收获，让我对 RV32 流水线有了更加深刻的理解，同时熟悉了常用的 RISC-V指令

## 改进意见

---

- 希望可以在实验1完成后对实验1进行**详细**的讲解，这样可以减少实验2遇到的各种奇怪的问题