Lab2 实验报告

【实验目标】

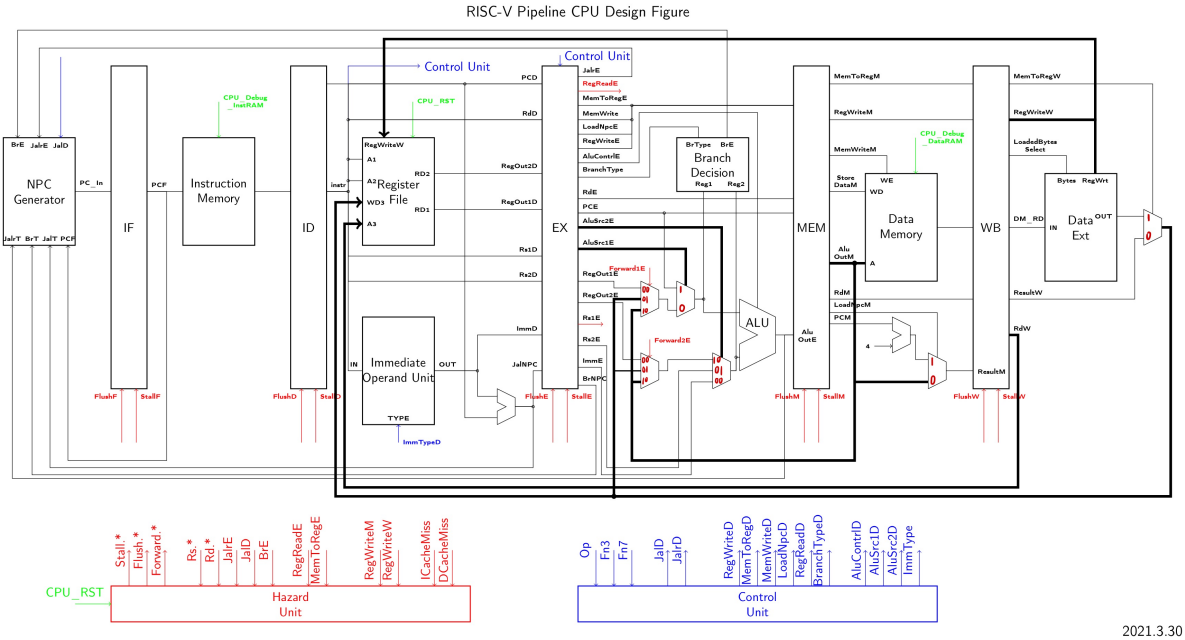
1. 完善流水线的代码，实现指令SLLI、SRLI、SRAI、ADD、SUB、SLL、SLT、SLTU、XOR、SRL、SRA、OR、AND、ADDI、SLTI、SLTIU、XORI、ORI、ANDI、LUI、AUIPC和 JALR、LB、LH、LW、LBU、LHU、SB、SH、SW、BEQ、BNE、BLT、BLTU、BGE、BGEU、JAL的运行，使得CPU执行后，各寄存器值符合预期；
2. 根据上述指令确定数据相关类型，并完善Harzard模块使得CPU能够处理数据相关的指令的正常运行，此时能够正常实现三个test中所有指令的正常运行，并且最终gp寄存器的值为1（每个test的样例成功运行后，gp+1直到最后置为1视为全部完成）；
3. 设计CSR的数据通路并添加代码，处理其中可能涉及的数据相关，并使用自行编写的测试代码进行验证，完成CSR相关指令的执行。

【实验环境和工具】

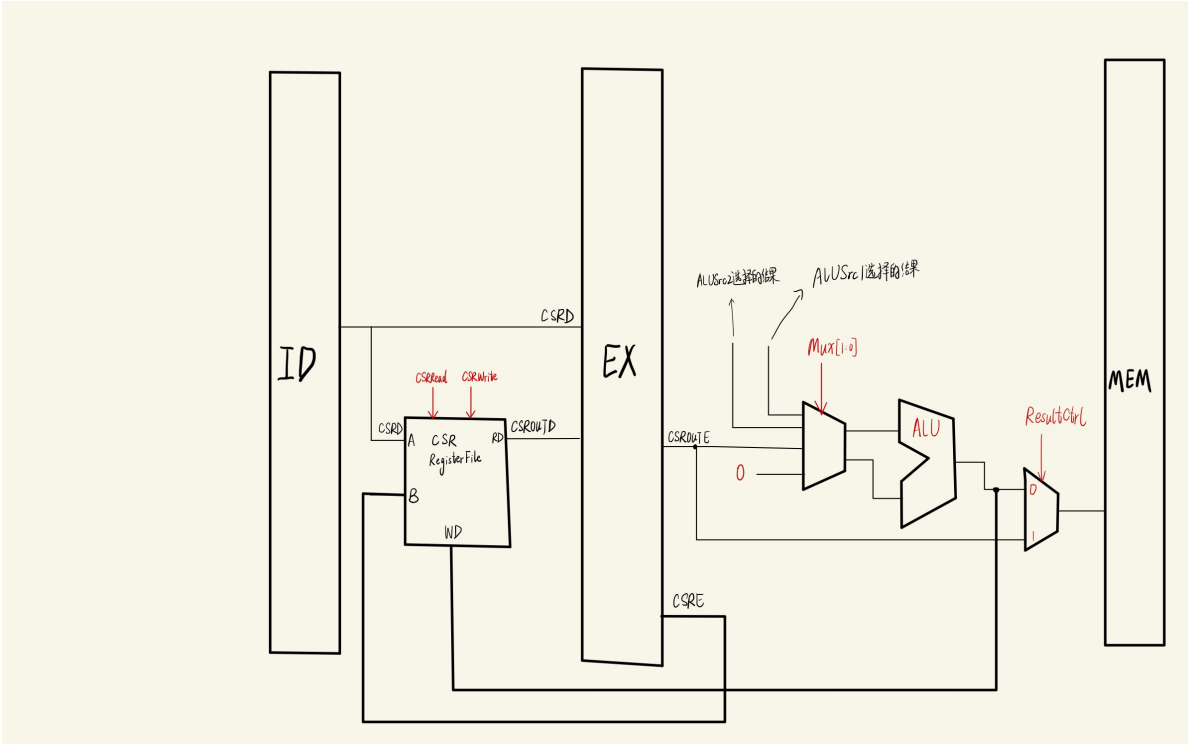
- 计算机一台，windows操作系统
- Vivado 2019.1软件

【实验内容和过程】

1、实验通路



数据通路如上所示，实验代码以经完成了数据通路的连接，需要完成的部分主要是PC的更新模块、立即数扩展模块、Branch的决定模块、ALU、WBSeg中数据存储器的连接（用于实现非字节写）、Data Ext模块（用于实现非字节毒）以及控制模块和冒险模块。数据通路中主要注意的是各个控制信号的通路及选择器的控制信号和选择的通路。



上为实现CSR相关指令所需额外添加的数据通路（控制模块未画出）。增加了一个CSR寄存器堆，用于读取和写入CSR寄存器的值，写和读均有控制信号控制，其根据相关的寄存器或者立即数和CSR指令决定。ALU前增加了一个四选二的选择器，非CSR指令时选取原来ALUSrc1和ALUSrc2选择的结果，CSR指令时根据其类型选择相应的操作数用于计算CSR寄存器值的相应变化，并通过数据通路引入CSR寄存器堆的WD端口，从而在EX段将CSR寄存器的写入值计算完成并写入，相对应的，CSR寄存器堆设置为先读后写，从而解决了CSR指令中关于CSR寄存器的数据相关，而对于CSR指令中关于rs1和rd寄存器的数据相关，由原有的冒险模块中的转发即可实现，无需再添加新的内容。

2、源代码及分析

(1) NPC Generator模块

```
always@(*)
    begin
        if(JalrE)
            PC_In <= JalrTarget;
        else if(BranchE)
            PC_In <= BranchTarget;
        else if(JalD)
            PC_In <= JalTarget;
        else
            PC_In <= PCF + 4;
    end
```

PC的更新模块主要是新的PC值的生成，主要更改的地方在于三类跳转指令的优先级，由于JALR和Branch类指令在EX段跳转，JAL指令在ID段跳转，因而在出现JALR或者Branch指令下一条为JAL指令时，要先进行前者的判断，再判断JAL指令的跳转。

(2) IDSegReg

```
InstructionRam InstructionRamInst (
         .clk    ( clk        ),
         .addra  ( A[31:2]    ),
        ......
     );
```

ID段寄存器中补充了指令寄存器的数据通路，主要是addra，由于PC给的是字地址，而指令按照字节地址存储，因而取[31:2]保证了正常的取指。

（3）控制模块

```
    assign AluSrc1D = (Op == 7'b0010111) ? 1:0;     //1：PC（AUIPC指令选取） 0：ForwardData1
    assign AluSrc2D = ((Op == 7'b0010011)&&(Fn3[1:0] == 2'b01)) ? (2'b01) : (((Op == 7'b0110011)||(Op == 7'b1100011)) ? 2'b00 : 2'b10); //10：IMM 01:Rs2E(主要是SLLI、SRLI和SRAI三条移位指令，Rs2相应的位数为其移位的次数) 00:ForwardData2（使用到Rs2寄存器值的相关指令）
    assign JalD = (Op == 7'b1101111) ? 1:0;  //Jal跳转指令的使能信号
    assign JalrD = (Op == 7'b1100111) ? 1:0;  //Jalr跳转指令的使能信号
    assign MemToRegD = (Op == 7'b0000011) ? 1:0;   //LOAD指令用来选取从MEM读取的值写入寄存器堆中
    assign LoadNpcD = ((Op == 7'b1101111)||(Op == 7'b1100111)) ? 1:0;  //JAL和JALR指令选取PC+4为最终写入寄存器堆的值
    assign ResultCtrlD = (Op == 7'b1110011) ? 1:0;  //CSR指令选择写入rd寄存器的值为选取的CSR寄存器的值
    assign CSRWriteD = (Op == 7'b1110011) ? 1:0;    //CSR指令写使能
    assign CSRRead = (Op == 7'b1110011) ? 1:0;  //CSR指令读使能
```

上为部分控制信号，相关作用已注释。

```
    always@(*)
    begin
        case(Op)
            7'b0010011:  //立即数令
                begin
            7'b0110011: //R-TYPE
                begin
            7'b0110111: //LUI
                begin
            7'b0010111: //AUIPC
                begin
            7'b1101111: //JAL
                begin
            7'b1100111: //JALR
                begin
            7'b1100011: //Branch
                begin
            7'b0000011: //LOAD
                begin
            7'b0100011: //STORE
                begin
            7'b1110011: //CSR
                begin
            default:    //NOP
                begin
        endcase
    end
endmodule
```

```
7'b0010011:  //立即数指令
                begin
                    RegWriteD <= `LW;
                    MemWriteD <= 0;
                    ImmType <= `ITYPE;
                    RegReadD <= 2'b10;
                    BranchTypeD <= `NOBRANCH;
                    MuxcD <= 3'b000;
                    case(Fn3)
                        3'b000: //ADDI
                            AluContrlD <= `ADD;
                        3'b001: //SLLI
                            AluContrlD <= `SLL;
                        3'b010: //SLTI
                            AluContrlD <= `SLT;
                        3'b011: //SLTUI
                            AluContrlD <= `SLTU;
                        3'b100: //XORI
                            AluContrlD <= `XOR;
                        3'b101: //SRLI or SRAI
                            begin
                                if(Fn7 == 7'b0)
                                    AluContrlD <= `SRL;
                                else
                                    AluContrlD <= `SRA;
                            end
                        3'b110: //ORI
                            AluContrlD <= `OR;
                        3'b111: //ANDI
                            AluContrlD <= `AND;
                    endcase
                end
```

其余的根据Op的值设置相关的控制信号，以立即数指令为例：

RegWrite按位或为寄存器堆的写使能，其不同类型为LOAD类指令从MEM读取的数据的类型；

MEMWrite为数据存储器的写使能，主要是STORE指令将数据写入数据存储器，根据指令的不同其值控制写入8、16和32位；

ImmType控制立即数扩展模块根据对应指令的结构生成32位的立即数指令；

RegRead用于表示Rs1和Rrs2寄存器是否使用，从而用于冒险模块中的判断；

BranchType作用于Branch决定模块，用于控制其根据不同的Branch指令对两个操作数进行相应的比较生成BranchE信号；

Muxc信号用于四选二的数据选择器，主要用于CSR指令中选择送入ALU计算的值；

AluControl用于控制ALU模块进行不同的运算。

（4）立即数扩展模块

```
always@(*)
    begin
        case(Type)
            `ITYPE: Out <= { {21{In[31]}}, In[30:20] };
            `STYPE: Out <= { {21{In[31]}}, In[30:25], In[11:7] };
            `BTYPE: 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 };
            `CTYPE: Out <= { 27'b0, In[19:15] };
           default:Out<=32'hxxxxxxxx;
        endcase
    end
```

根据指令的立即数类型生成相应的立即数，其中CTYPE是CSR类的指令的立即数类型。

（5）Branch决定模块

```
always@(*)
    begin
        case(BranchTypeE)
            `BEQ: BranchE <= (Operand1 == Operand2) ? 1'b1 : 1'b0;
            `BNE: BranchE <= (Operand1 != Operand2) ? 1'b1 : 1'b0;
            `BLT: BranchE <= (Operand1_Signed < Operand2_Signed) ? 1'b1 : 1'b0;
            `BLTU: BranchE <= (Operand1 < Operand2) ? 1'b1 : 1'b0;
            `BGE: BranchE <= (Operand1_Signed >= Operand2_Signed) ? 1'b1 : 1'b0;
            `BGEU: BranchE <= (Operand1 >= Operand2) ? 1'b1 : 1'b0;
           default: BranchE <= 1'b0;
        endcase
    end
```

根据Branch的类型，对两个操作数进行不同的比较产生BranchE信号。

（6）ALU模块

```
    wire signed [31:0] Operand1_Signed = $signed(Operand1);
    wire signed [31:0] Operand2_Signed = $signed(Operand2); //SLT语句使用时为有符号
数

    always@(*)
    begin
        case(AluContrl)
            `SLL:
                begin
                    AluOut <= Operand1 << (Operand2[4:0]);
                end
            `SRL:
                begin
                    AluOut <= Operand1 >> (Operand2[4:0]);
                end
            `SRA:
                begin
                    AluOut <= Operand1_Signed >>> (Operand2[4:0]);
                end
            `ADD:
                begin
                    AluOut <= Operand1 + Operand2;
```

```
                end
            `SUB:
                begin
                    AluOut <= Operand1 - Operand2;
                end
            ......
    end
```

由控制模块产生的AluControl信号决定不同的运算类型，部分需要有符号数的指令要进行转换。

（7）四选二数据选择器

```
always@(*)
    begin
        case(s)
            3'b000: //NOT CSR
                begin
                    out1 <= s1;//ForwardData1
                    out2 <= s2;//ForwardData2
                end
            3'b001: //CSRRW
                begin
                    out1 <= s1;
                    out2 <= s4;//0
                end
            3'b010: //CSRRWI
                begin
                    out1 <= s2;//CSR
                    out2 <= s4;
                end
            3'b011: //CSRRC,CSRRS
                begin
                    out1 <= s3;
                    out2 <= s1;
                end
            3'b100: //CSRRCI,CSRRSI
                begin
                    out1 <= s3;
                    out2 <= s2;
                end
            default:;
        endcase
    end
```

该数据选择器主要用于CSR指令的操作数选择，如CSRRW指令将Rs1的值写入CSR中，故第二个输出为0，同理CSRRC、CSRRS需要将CSR寄存器的值根据Rs1中1的位置进行相应的计算，故输出为CSR和Rs1。

（8）WBSegReg

```
    DataRam DataRamInst (
        .clk    ( clk           ),
        .wea    ( WE << A[1:0]  ),
        .addra  ( A[31:2]       ),
        .dina   ( WD << (A[1:0]*8) ),
        ......
    );
```

WB段寄存器中完善了数据存储器的数据通路，主要是wea和dina，WE根据地址的低两位进行移位主要是进行非字节写，同样的WD移位是将写入数据和写使能一致，完成非字节写。

（9）DataExt

```
    wire [31:0] IN_NA;
    assign IN_NA = IN >> (LoadedBytesSelect * 32'h8);

    always@(*)
    begin
        case(RegWriteW)
            `LB: OUT <= { {24{IN_NA[7]}}, IN_NA[7:0] };
            `LH: OUT <= { {16{IN_NA[15]}}, IN_NA[15:0] };
            `LW: OUT <= IN;
            `LBU: OUT <= { 24'b0 , IN_NA[7:0] };
            `LHU: OUT <= { 16'b0 , IN_NA[15:0] };
            default: OUT <= 32'hxxxxxxxx;
        endcase
    end
```

此模块为读数据的扩展，IN_NA为读取的数据根据字地址的低两位进行移位，再补位完成数据的读取。

（10）CSR寄存器堆

```
always@(negedge clk or posedge rst)
    begin
        if(rst)                                 for(i=0;i<32;i=i+1) CSR[i][31:0]
<=32'b0;
        else if(WE)     CSR[B]<=WD;
    end
    //
    always@(*)
    begin
        if(!clk && CSRRead)
            RD <= CSR[A];
    end
```

主要实现了CSR寄存器堆的读和写，这里控制读在clk低电平时读，保证了先写后读，避免了CSR寄存器的数据相关。

（11）冒险模块

```
    always@(*)
    begin
        if(CpuRst)
            {StallF, FlushF, StallD, FlushD, StallE, FlushE, StallM, FlushM,
StallW, FlushW} <= 10'b0101010101;
```

```
        else if(BranchE)
            {StallF, FlushF, StallD, FlushD, StallE, FlushE, StallM, FlushM,
StallW, FlushW} <= 10'b0001010000;
        else if(JalrE)
            {StallF, FlushF, StallD, FlushD, StallE, FlushE, StallM, FlushM,
StallW, FlushW} <= 10'b0001010000;
        else if(JalD)
            {StallF, FlushF, StallD, FlushD, StallE, FlushE, StallM, FlushM,
StallW, FlushW} <= 10'b0001000000;
        else if( MemToRegE && ((RdE == Rs1D)||(RdE == Rs2D)))
            {StallF, FlushF, StallD, FlushD, StallE, FlushE, StallM, FlushM,
StallW, FlushW} <= 10'b1010010000;
        else
            {StallF, FlushF, StallD, FlushD, StallE, FlushE, StallM, FlushM,
StallW, FlushW} <= 10'b0000000000;
    end
```

段寄存器的Stall和flush主要存在这六种情况：

一是rst信号生效时，全部flush；

二是无冲突时，无stall，无flush；

三和四是Branch和jalr跳转成功时，要冲刷ID段寄存器和EX段寄存器；

五是Jal跳转成功时，冲刷ID段寄存器；

六是出现LOAD+use类冲突时，要stall流水线。

```
    always@(*)      //Rs1的转发
    begin
        if((RegReadE[1])&&(RegWriteM != 0)&&(RdM == Rs1E)&&(RdM != 0))
            Forward1E <= 2'b10;
        else if((RegReadE[1])&&(RegWriteW != 0)&&(RdW == Rs1E)&&(RdW != 0))
            Forward1E <= 2'b01;
        else
            Forward1E <= 2'b00;
    end
```

上为Rs1的转发，主要的决定信号有RegRead[1]表示使用了Rs1寄存器，RdM为MEM段的Rd寄存器号，RdW为WB段的Rd寄存器号，RegWrite表示对应段的Rd是否写入寄存器堆中，通过比较决定是否将相关值替换掉Rs1，同理Rs2也是如此。

（12）数据通路（CSR指令添加的部分）

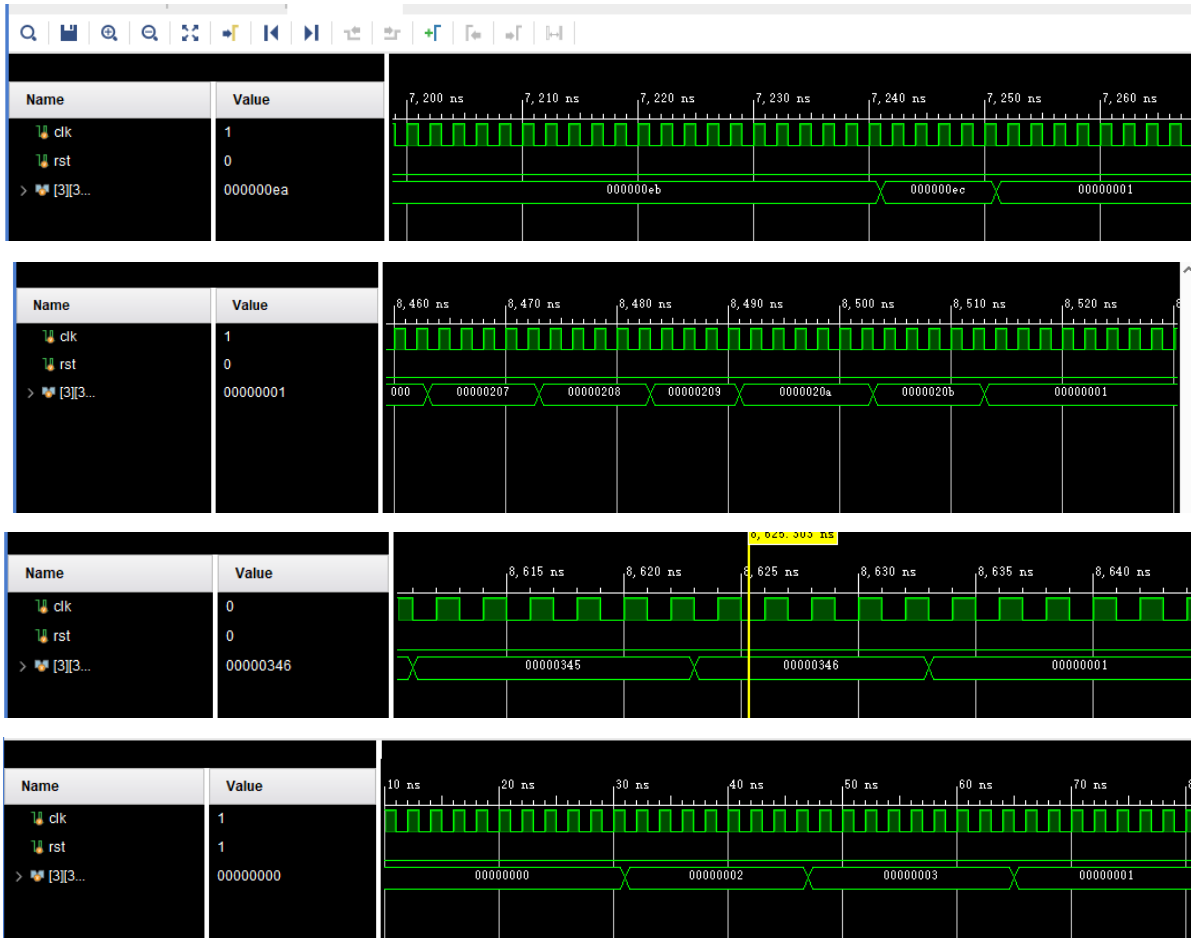
```
    assign CSRRead0 = ((CSRRead && ((MuxcD == 3'b001)||(MuxcD == 3'b0101))&&(RdD
== 5'b0))||(CSRRead == 0)) ? 0:1;
    assign CSRWriteEE = CSRWriteE?((((MuxcE == 3'b011)&&(Rs1E == 5'b0))||((MuxcE
== 3'b100)&&(ImmE == 32'b0)))? 0 :1):0;
    assign CSRD = Rs2D;
```

CSRRead0额外的判断主要是由于CSRRW和CSRRWI指令中Rd为$0是不读CSR寄存器；

CSRWriteEE额外的判断主要是CSRRC、CSRRS中Rs1为$0不写CSR寄存器，CSRRCI、CSRRSI中立即数为0时不写CSR寄存器；

CSRD在指令中对应的位数和Rs2D一致。

3、实验仿真结果



根据给出的测试文件和自行生成的CSR指令测试文件，最终gp寄存器的值均跳转至1，测试均通过。

【实验总结】

本次实验主要实现了RSICV的流水线CPU，由于是补全代码的形式，主要时间花费在了原本的数据通路的阅读和数据通路的追踪上，其中部分通路上与图中存在细微的差别，需要根据代码给出的信息来确定而非直接根据数据通路图来决定。其次大部分花费的时间主要是在控制模块的完成上，由于涉及到的指令较多，控制信号也同样较多，在开始编写的时候出现了部分失误导致控制信号出错，测试中主要出现的问题也来源于此，如果在代码阶段能够更加专注或者提前对全部的控制信号进行归纳的话，能节约大量的时间和debug的精力。其次出现问题的地方在数据存储器的连接部分，开始对非字节写的概念并不清楚，单纯的直接连接了MEMWrite和数据，导致test2的后半部分完全无法进行，最终通过比较了跳转指令比较的两个数理解了非字节写的部分，最后完成相关内容的修改。对本实验的建议主要是实验通路上可以更加清晰一些，能够使整个实验更加的清晰，也可以帮助流水线学习上存在困难的同学更好的理解整个实验的相关内容，使得实验更快更好的完成。