Lab2 实验报告

雷雨轩 PB18111791 计算机学院

实验目标

- 用verilog实现RV32I 流水线CPU
- 熟悉RV32I 指令集的功能和各个bit位作用，以及其需要的数据通路，并在在RV32I 流水线CPU中实现所有指令的功能
- 理解各类数据相关和控制相关，实现Hazard模块，并注意对特殊情况处理(如需要插入气泡的数据相关以及控制相关)
- 学会设计数据通路，完成CSR类指令的功能实现
- 学会阅读和实现汇编代码，来完成CPU的测试
- 重新熟悉Vivado的使用，以及利用仿真文件来调试实现的CPU功能是否正确

实验环境和工具

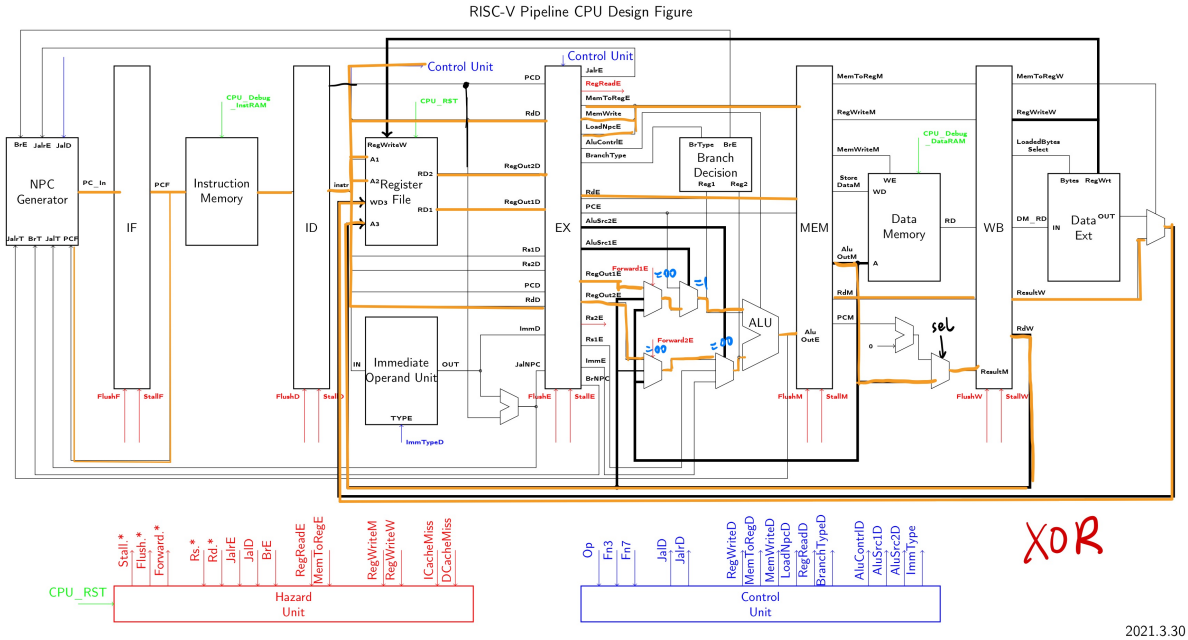
- Vscode 2021
- Vivado 2019.1

实验内容和过程

（总结自己所做的三个阶段工作）

阶段1、2

数据通路



整体实现思路

- 因为考虑到阶段1、2都是在实现各类基本指令，以及处理相关，所以为了构思的流畅性，将两个阶段合并在一起实现。
- 首先是理解清楚所给的数据通路的每段的含义，然后对比所给的代码框架，大致了解清楚每一部分代码在干些什么，哪些模块是需要重点实现的。

这里自己发现，因为RV32Core.v已经被实现好了，所以可以从中了解到各个信号的名称以及含义，从而明白其他模块的参数的具体含义

- 在具体实现方面，自己则是照着数据通路的图，按从左到右的顺序依次完成的各个部件，下面将对每个模块做实现思路的讲解

NPC_Generator模块

- 主要是根据各个跳转指令的信号值决定下一个PC值

  在lab1里确定了优先级： jalr = br > jal > pc+4

- 所以只需在always@(*)语句块里按照优先级进行条件语句判断并给PC_In赋值

IFSegReg模块：

- 不用改，实现很简单，就是传递一个PC值到PCF变量

IDSegReg模块

- 是IF-ID段寄存器模块，主要功能是从Inst mem里读出指令，但这里还需注意stall、flush等指令，不过这部分代码已被补全
- 需补全的两行代码比较简单，但需要注意地址A[31:2]，取31:2而忽略后两位是因为字对齐（因为数据通路是32bit的，所以访问时直接按字访问)

ControlUnit模块

- 即CPU的指令译码器，是组合逻辑电路，只需根据IF-ID段寄存器里读出的指令的各个字段来设置各个逻辑信号即可，具体细节需参考每条指令的功能，以及各个信号的含义。这一部分的实现比较繁琐，因为需要去查找每个指令的各个字段的含义。注意到，因为模块的参数已经全部给出，所以只需要根据Op和funct3等字段，为每个信号赋值即可
- 需要完成的指令
  - R类指令：ADD、SUB、SLL、SLT、SLTU、XOR、SRL、SRA、OR、AND
  - I类指令 ADDI、SLTI、SLTIU、XORI、ORI、ANDI、SLLI、SRLI、SRAI
    - LOAD指令：LB、LH、LW、LBU、LHU
  - S型装载指令：SB、SH、SW、
  - U型指令： LUI、AUIPC
  - B型跳转指令： BEQ、BNE、BLT、BLTU、BGE、BGEU
  - J型指令： JAL、 JALR
- 其中注意到，有一些比较特定的选择器信号等，只需要根据Op能简单判断出来的，就直接利用assign语句完成

```
//首先把能快速确定的信号先直接确定下来
assign LoadNpcD = JalD | JalrD;
assign JalrD = (Op==7'b1100111)?1'b1:1'b0;
assign JalD = (Op==7'b1101111)?1'b1:1'b0;

assign MemToRegD = (Op==7'b0000011)?2'b01:((Op==7'b1110011)? 2'b10:2'b00);
//load指令的Op都一样，而且只有load指令会有mem to reg操作；新增的CSR指令使得MemToReg片选多一一项

assign AluSrc1D = (Op==7'b1110011 && Fn3[2]==1)?2'b10 :((Op==7'b0010111)?2'b01:2'b00);
//只有AUIPC指令才会为01；仅当有CSR指令，且是CSRRWI，CSRRSI，CSRRCi指令时，才会选择10

assign AluSrc2D = ( (Op==7'b0010011)&&(Fn3[1:0]==2'b01) )?(2'b01):(((Op==7'b0110011)||(Op==7'b1100011))?2'b00:((Op==7'b1110011)?2'b11:2'b10));
//AluSrc2D: 取01时，对应图里的Rs1E，即SRAI的这条算术右移指令； 正常R类指令或者B类型指令则需要reg2参与算术运算，所以信号为00； 若是CSR类指令，则为11；否则为10，对应ImmE
```

- 当然，还有一些信号的赋值情况比较复杂，通过always@(*)语句块里进行case来选择

```
//此处展示case语句的一个分支
7'b0010011:begin    //除去load之外的I类指令
    CSRReadD <= 0;
    CSRwrenD <= 0;
    RegWriteD<=`LW; //I类指令都需要32bit写回reg
    MemWriteD<=4'b0000;
    ImmType<=`ITYPE;
    BranchTypeD <= `NOBRANCH;
    case(Fn3)
        3'b000:AluContrlD<=`ADD;  //ADDI
        3'b001:AluContrlD<=`SLL;  //SLLI
        3'b010:AluContrlD<=`SLT;  //SLTI
        3'b011:AluContrlD<=`SLTU;   //SLTIU
        3'b100:AluContrlD<=`XOR;    //XORI
        3'b101:
            if(Fn7[5]==1)
                AluContrlD<=`SRA;   //SRAI
        else
            AluContrlD<=`SRL;   //SRLI
        3'b110:AluContrlD<=`OR;   //ORI
        default:AluContrlD<=`AND;    //ANDI     3'b111

    endcase
end
```

- 具体实现参考代码里的注释以及insts.md，是一个比较抠细节的工作
- 当然，还有一点需要注意，在为信号分配值之前需要把Parameters.v的各个信号含义弄清楚
    - 比如写寄存器时，根据写回的位数、以及有无符号就已经给出了多种值情况

```
//RegWrite[2:0]  six kind of ways to save values to Register
`define NOREGWRITE  3'b0    //  Do not write Register
`define LB  3'd1            //  load 8bit from Mem then signed extended
to 32bit
`define LH  3'd2            //  load 16bit from Mem then signed extended
to 32bit
`define LW  3'd3            //  write 32bit to Register
`define LBU  3'd4           //  load 8bit from Mem then unsigned
extended to 32bit
`define LHU  3'd5           //load 16bit from Mem then unsigned extended
to 32bit
```

- 其余的参数也是事先已经定义好的，所以读懂这个模块后，再根据每条指令的具体功能赋以对应的信号即可。

ALU模块

- 此模块的实现也较为简单，根据AluContrl信号来对操作数做+,-，且，或，移位等运算即可
- 这里需注意到，SRA和SLT指令，涉及到有符号数的运算。而因为wire类型默认是无符号类型，所以在计算时会有区别，需要利用`$signed(Operand1)`来做数据类型转换

BranchDecisionMaking

- 注意wire默认是无符号类型，所以在进行有符号的大小比较时需作转换，而=和!=有符号无符号比较都一样
- 此外对于6条B类指令，只需判断两操作数是否满足跳转条件即可，若满足则BranchE=1'b1

ImmOperandUnit

- 立即数都是有符号扩展（不能和Data Ext中符号扩展类型记混了）
- 不同类型指令扩展时的区别
  - 立即数各个位分布在指令的不同位置
  - BTYPE和JTYPE立即数要补0
- 总体而言，就是对照着每条指令的指令各个位的含义，把属于立即数的各个位拼接起来，并扩展到32bit。（这一部分也在lab1问题里问了verilog如何实现拼接操作）

```
always@(*)
begin
    case(Type)
        `ITYPE: Out<={ {21{In[31]}}, In[30:20] };//有符号扩展
        `JTYPE: Out<={ {12{In[31]}}, In[19:12], In[20], In[30:21], 1'b0 };//
有符号扩展
        `STYPE: Out<={ {21{In[31]}}, In[30:25], In[11:7] }; //有符号扩展
        `UTYPE: Out<={ In[31:12], 12'b0 };//针对LUI和AUIPC指令，将20位的U立即数放
到31-12位，低12位填0
        `BTYPE: Out<={ {20{In[31]}}, In[7], In[30:25], In[11:8], 1'b0 };//有
符号扩展，而且注意有末尾有0是省略了，未存储在指令中
       default:Out<=32'hxxxxxxxx;//RTYPE，不需要立即数扩展
    endcase
end
```

WBSegReg

- ```
  DataRam DataRamInst (
      .clk    (clk),
      .wea    (WE<<A[1:0]),
      .addra  (A[31:2]),
      .dina   (WD<<(8*A[1:0])),          //此处根据A[1:0]的值来选择左移多少位，对应
  SW,SH,SB存储的不同字节数
      .douta  ( RD_raw          ),
      .web    ( WE2             ),
      .addrb  ( A2[31:2]        ),
      .dinb   ( WD2             ),
      .doutb  ( RD2             )
  );
  ```

  这里需要特别说明

  - 首先DataRam是按字节写的，并且WE信号连接的是MemWriteM，其赋值是

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

    相当于WE信号只决定要写多少位数据进去

  - 所以用A[1:0]信号来决定到底写到RAM对应字的哪个byte上去，比如WE信号为4'b0001, 若按A[1:0]的值左移成了4'b0100,则相当于SB指令把rs2中的值的低8位送到Data mem对应字地址A[31:2]的[23:16]位 ,这个可以结合现成的DataRam.v模块来理解

  - 然后因为SB，SH写入的值都是一个字的后8it/16bit， 所以为了写入与WE<<A[1:0]对应的位置，数据WD也需要跟着左移

DataExt

- 这个模块的关键在于如何根据LoadedBytesSelect的值，来从Data mem里读出的数据里筛选需要的Byte。可以根据简单的位运算得到，也可以通过case语句来做。取到对应的位数后，再根据指令进行无符号/有符号扩展即可
- 而且注意：LB要取的字节，是32bit里[31:24],[23:16],[15:8],[7:0]中的任意一个，由LoadedBytesSelect决定。 LH同理
- 可以与WBSegReg模块的思想类比一下

HarzardUnit

- ICacheMiss和DcacheMiss信号因为本次实验不需要，所以也没有使用
- 处理上 分为数据相关以及控制相关的逻辑
  - 数据相关的话需要对Rs1和Rs2进行定向路径处理

    并且对特殊的load后接寄存器使用指令做一个周期的停顿处理（因为没法用定向路径解决）

  - 控制相关的话，对BranchE，JalrE以及JalD，只需在跳转成立时把对应段寄存器清空，因为那些段里的数据是不跳转时的指令运行，这个也已经在lab1里做过分析了

- ```
  always @ (*)
      if(CpuRst)//当CpuRst==1时CPU全局复位清零（所有段寄存器flush）

  {StallF,FlushF,StallD,FlushD,StallE,FlushE,StallM,FlushM,StallW,FlushW} <=
  10'b0101010101;
  ```

```
    else if(BranchE | JalrE)//当在EX段检测到跳转信号时，根据lab1里分析，应该把ID和EX
段寄存器清空

{StallF,FlushF,StallD,FlushD,StallE,FlushE,StallM,FlushM,StallW,FlushW} <=
10'b0001010000;
    else if(MemToRegE & ((RdE==Rs1D)||(RdE==Rs2D)) )//当EX段执行的是Load指令时，
且写回的寄存器会被ID段代码用到，则需要插入气泡停顿一个周期

{StallF,FlushF,StallD,FlushD,StallE,FlushE,StallM,FlushM,StallW,FlushW} <=
10'b1010010000;
    else if(JalD)//ID段检测到跳转信号，需清空IF-ID段寄存器

{StallF,FlushF,StallD,FlushD,StallE,FlushE,StallM,FlushM,StallW,FlushW} <=
10'b0001000000;
    else

{StallF,FlushF,StallD,FlushD,StallE,FlushE,StallM,FlushM,StallW,FlushW} <=
10'b0000000000;

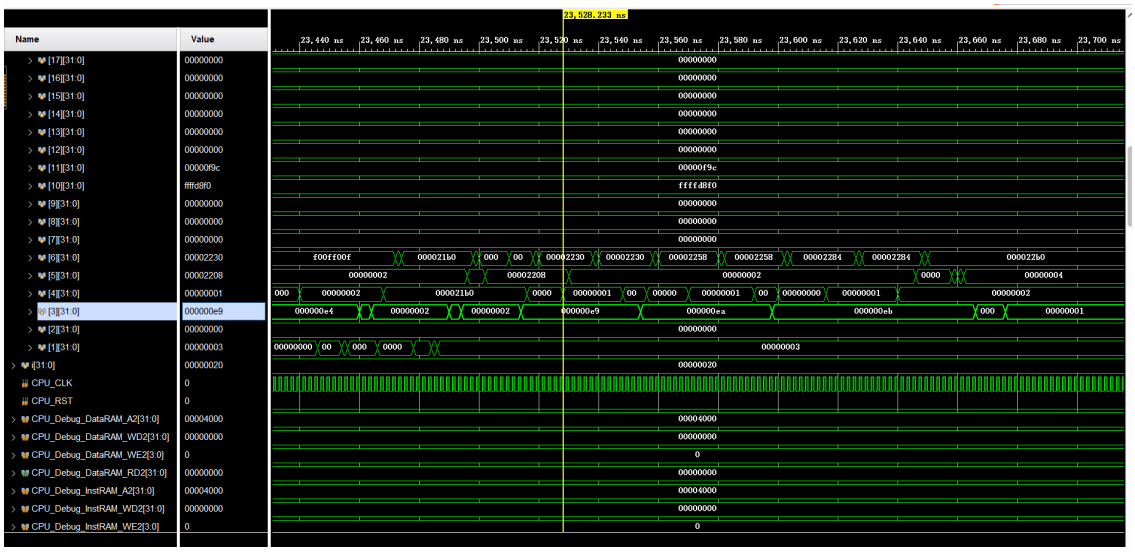
//对寄存器1的定向路径。处理的情况是EX段计算时需要用到Mem或WB段的数据；并且注意0号寄存器的
特殊性
always@(*)begin
    if( (RegWriteM!=3'b0) && (RegReadE[1]!=0) && (RdM==Rs1E) &&(RdM!=5'b0) )
        //先考虑Mem段的数据相关
        Forward1E<=2'b10;
    else if( (RegWriteW!=3'b0) && (RegReadE[1]!=0) && (RdW==Rs1E) &&
(RdW!=5'b0) )
        //再考虑WB段的数据相关
        Forward1E<=2'b01;
    else
        Forward1E<=2'b00;
end

//对寄存器2的定向路径，处理方式和寄存器1一样
always@(*)begin
    if( (RegWriteM!=3'b0) && (RegReadE[0]!=0) && (RdM==Rs2E) &&(RdM!=5'b0) )
        Forward2E<=2'b10;
    else if( (RegWriteW!=3'b0) && (RegReadE[0]!=0) && (RdW==Rs2E) &&
(RdW!=5'b0) )
        Forward2E<=2'b01;
    else
        Forward2E<=2'b00;
end
```

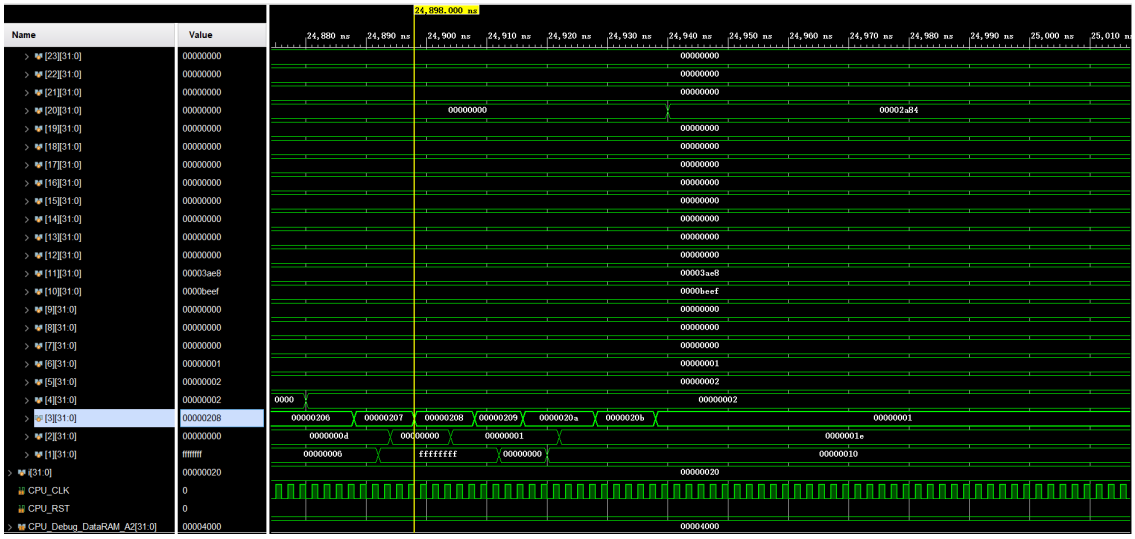
仿真运行结果

- 主要运行助教提供的3个测试例子，最后检查regfile文件的3号寄存器的变化情况，它的值代表执行到第几个test，全部测试通过，3号寄存器的值最终变为1。 所以实际变化应该是从第一条测试的编号（如2）开始递增到最后一条测试的编号(如200），最后每个测试例子跑完后, 寄存器的值稳定在1

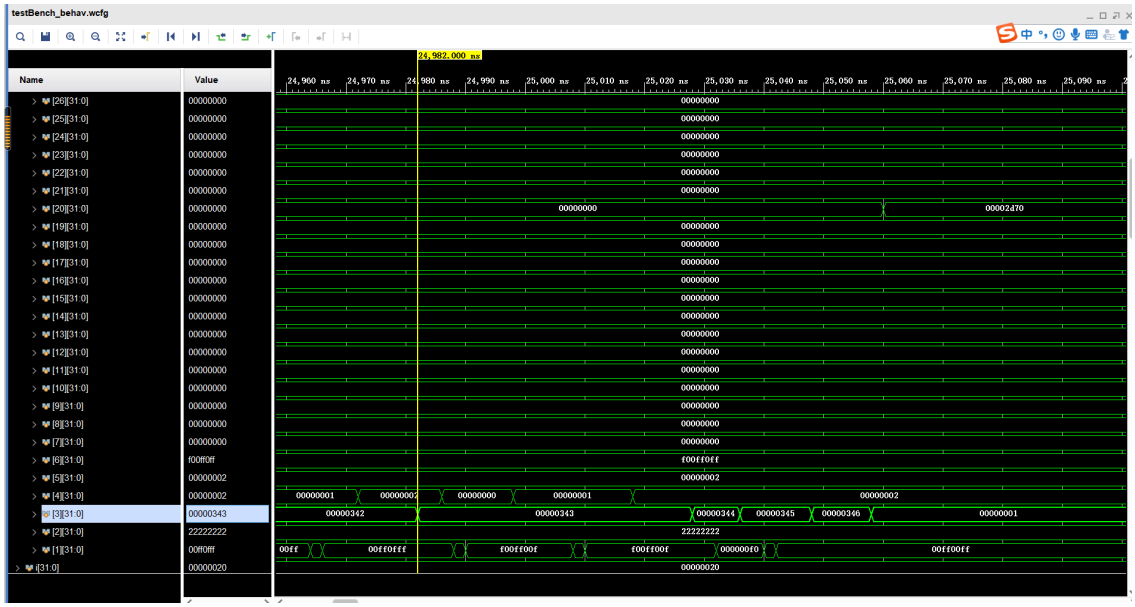
- 1testAll



- 2testAll

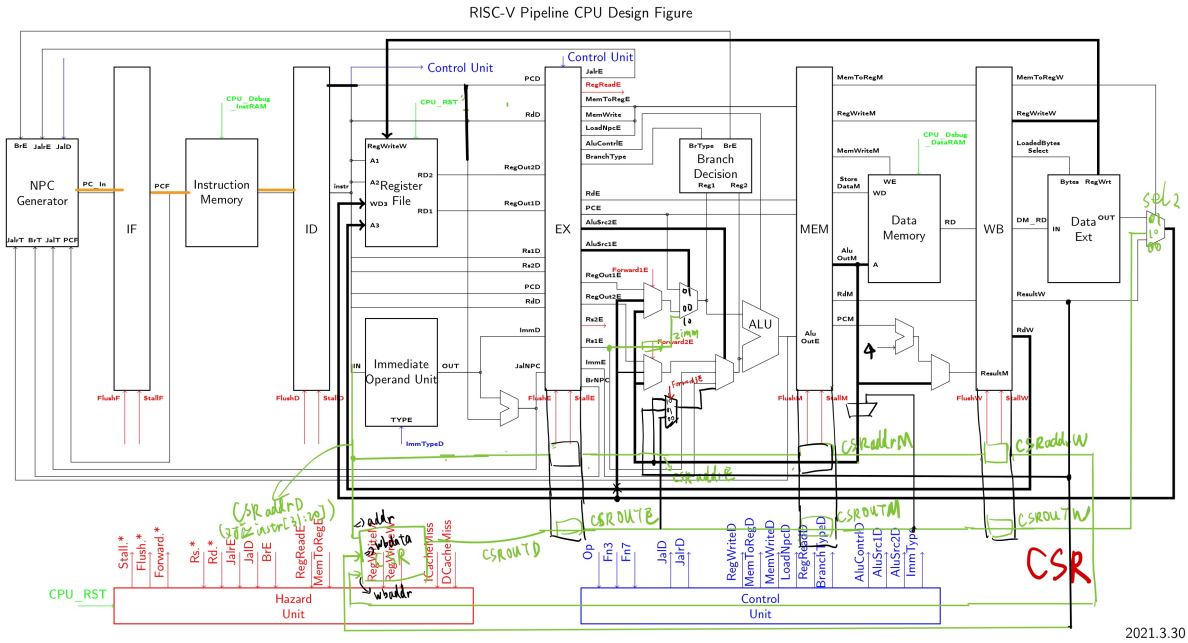


- 3testAll



阶段3

数据通路



整体实现思路

- 首先根据助教提供的指令集文档，理解清楚每条CSR指令的含义

  这里列出来：

  - CSRRW（Atomic Read/Write CSR）指令原子性的交换 CSR 和整数寄存器中的值。 CSRRW 指令读取在 CSR 中的旧值，将其零扩展到 XLEN 位，然后写入整数寄存器 rd 中。 rs1 寄存器中的值将被写入 CSR 中。

    注意，若目的通用寄存器为x0，则不会执行读取操作；若源通用寄存器 rs1 为x0也不会进行写入CSR操作

  - CSRRS（Atomic Read and Set Bit in CSR）指令读取 CSR 的值，将其零扩展到 XLEN 位，然后写入整数寄存器 rd 中。整数寄存器 rs1 中的值指明了哪些 CSR 中的位被置为 1。 rs1 中的任何为 1 的位，将导致 CSR 中对应位被置为 1，如果 CSR 中该位是可以写的话。

  - CSRRC（Atomic Read and Clear Bit in CSR）指令读取 CSR 的值，将其零扩展到 XLEN 位，然后写入整数寄存器 rd 中。整数寄存器 rs1 中的值指明了哪些 CSR 中的位被置为 0。 rs1 中的任何为 1 的位，将导致 CSR 中对应位被置为 0

  - 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32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。

- 然后根据lab1里设计的CSR通路，来在代码里增加一些信号，修改一些信号的计算，完成CSR指令功能的实现。但是lab1里设计的CSR通路还不够完善，所以又在自己实现代码的过程中做了一些补充

RV32Core.v模块

- 首先需要根据lab1里搭建的数据通路，在代码里做一个全局的概览实现，把一些信号名字确定，然后加入新的CSR寄存器模块，在初步实现时，这个模块也只是有个大概的雏形，具体的完善是在后面其他模块的实现里逐步完成的。

- 这里需要注意，有一些信号诸如MemToRegD, MemToRegE，MemToRegM，MemToRegW，AluSrc1D，AluSrc1E 的位数应修改为2bit,因为新增的CSR通路使得选择器的选择变多了
- CSR模块新增的一些信号的说明：

```
wire [31:0] CSROutD; //CSROut是从CSR寄存器读出的数据
wire [31:0] CSROutE;
wire [31:0] CSROutM;
wire [31:0] CSROutW;
wire [31:0] CSRaddrD; //CSRaddr是根据CSR指令的31:20位得到的
wire [11:0] CSRaddrE;
wire [11:0] CSRaddrM;
wire [11:0] CSRaddrW;
//下面的指令的一些作用会在HarzardUnit.v 以及 ControlUnit.v 里详细解释
wire [31:0] ForwardDataMem;//用于处理数据相关问题
wire CSRwrenD,CSRwrenE,CSRwrenM,CSRwrenW; //CSRWren是CSR指令的写使能信号
wire CSRReadD,CSRReadE;//CSRRead是CSR指令的读使能信号
```

CSR.v模块

- 新加入的CSR寄存器，其书写方式比较类似RegisterFile.v ,不过加入了初始化的代码
- 主要是利用instr[31:20]作为地址来从CSR reg里读数据，然后根据rs1的值情况对CSR同一位置进行写操作
- ```
  always@(negedge clk or posedge rst)
  begin
      if (rst)
          for (i = 0; i < 4096; i = i + 1)
              reg_file[i] <= 32'b0;
      else if(write_en)
          reg_file[w_addr] <= w_data;
  end

  assign reg_out = reg_file[r_addr];
  ```

EXSegReg.v 模块

- 主要就是修改一下MemToReg, AluSrc1的长度； 以及把CSR相关的信号继续传递下去

MEMSegReg.v模块

- 处理方式同EXSegReg.v

WBSegReg.v 模块

- 处理方式同EXSegReg.v

HarzardUnit.v 模块

- 主要新增专门处理CSR的数据相关的定向路径通路
- CSR指令需要处理的数据相关
  - 首先，与rs1有关的相关已经在阶段2处理过了，即RAW这种相关。而且对于zimm 这种情况，通过把zimm的线放在forward1E 选择器之后来做片选，避免了与Rs1定向路径的冲突，即zimm的优先级更高 （因为可能存在CSRRWI指令的[19:15]字段的值对应前一条指令的rd，但实际上并不会读寄存器
  - 与rd的相关：即CSR指令之后又有指令要读取该寄存器，此时也可分为在MEM段以及在WB段的定向路径
  ```

        - 对于Mem段的CSRout数据，需要与AluOutM做选择，个人选择利用CSRwrenM使能信号判断，当前Mem段运行的是否是CSR指令，若是，那么定向路径连接的值应该为CSROutM，这里在代码里的实现则是新增了ForwardDataMem信号

          `assign ForwardDataMem = CSRwrenM? CSROutM:AluOutM;`
        - 对于WB段，因为已经有MemToregW这个片选信号，所以只需要把RegWriteData作为定向路径数据即可
        - 所以这一部分的处理变化很小，大部分都已经在阶段2完成了
        - 当然这里牵扯到若rd是x0寄存器的情况，但分析后发现，阶段2里已经处理好了，因为如果Mem段或者WB段的Rd寄存器是x0，那么则不会有定向路径。
    - CSR寄存器的数据相关：即前一条指令准备写回CSR某个位置，然后下一条指令又要用到该位置的数据

      考虑到对于CSR指令的写回数据的计算仍是由ALU模块来处理，即在AluSrc2E选择器处连上了CSROUTE信号，所以可以在此基础上新增定向路径，把Mem段的AluOutM值以及WB段的ResultW值连到该定向路径选择器ForwardData3上

      （因为若Mem段对应指令是CSR，那么AluOutM的值为要写回CSR寄存器的值； 若WB段对于指令是CSR，那么ResultW的值是写回CSR寄存器的值）

      ```
      //CSR专用定向路径
          always@(*)
          begin
              if( CSRWriteM && CSRRead && CSRSrcM==CSRSrcE )
                  //先看Mem段有无数据相关
                  Forward3E<=2'b10;
              else if( CSRWriteW && CSRRead && CSRSrcW==CSRSrcE )
                  //再看wb段有无数据相关
                  Forward3E<=2'b01;
              else
                  Forward3E<=2'b00;
          end
      ```

ControlUnit.v 模块

- 首先是对MemToRegD，AluSrc1D，AluSrc2D这三个信号的修改，根据数据通路，因为有了CSR类指令，所以需要做一定额外的判断
- 此外就是对其余信号的判断，都集成在了一个大的always语句块里
    - 首先是CSRRead和CSRwren信号，分别为读使能和写使能。这两个信号用于定向路径的判断（即标志当前指令为CSR类指令）以及CSR寄存器的写使能。当然仅当判断指令为CSR类指令时才会置为1
    - 对于不同的CSR指令（根据funct3字段区分），给予不同给的ALU计算信号
- 注意到CSR指令在面临x0寄存器时的要求（参考 整体实现思路 一节），需要额外处理信号:

  对CSRRW，CSRRS，CSRRC指令：

    - 根据Rs1D值和RdD判断
        - 若Rs1D为x0寄存器：CSRwrenD=0， 即不会写CSR寄存器
        - 若RdD为x0寄存器：CSRReadD=0， 即不会读CSR寄存器

  对CSRRWI，CSRRSI，CSRRCI指令：

    - 若RdD为x0寄存器：CSRReadD=0，即不会读CSR寄存器
    - 若Rs1D为x0寄存器: CSRwrenD=0，即不会写CSR寄存器

  所以以上两者实际上可以统一写在一起。

这样处理后，会导致遇到x0寄存器时实际上CSR专用的定向路径判断时会有所变化，这些也已经在harzard.v 模块里考虑到了。

ALU.v 模块

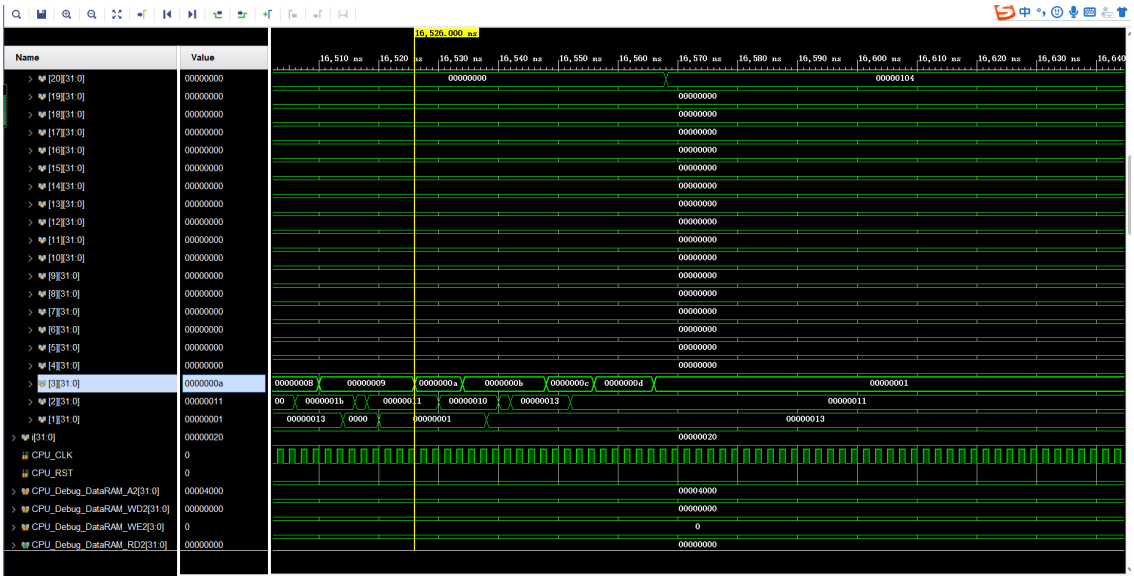
- CSRRW, CSRRWI的计算是直接把rs1的值或者zimm的0扩展值存入CSR寄存器，记此计算的ALU信号为`RW
- CSRRS，CSRRSI的计算是根据rs1或者zimm的0扩展后的1值来置1，所以直接用本来就有的`OR运算
- CSRRC，CSRRCI的计算是根据rs1或者zimm的0扩展后的1值来置0，新设计了一个`RC信号，来完成此功能（取反+且）

Parameters.v模块

- 主要增加了ALUControl信号，用于处理不同CSR指令的计算

仿真运行结果

- 实现了测试集4testAll.txt 以及对应的.inst, .data文件，并且是仿照助教提供的例子，是每完成一个测试样例就判断指令运行结果是否正确，若不正确，会跳到fail字段； 若正确，则开始运行下一个测试。 每个测试例子在运行时，会把3号寄存器置为测试例子的编号。当所有例子跑完后，会把3号寄存器置1。
- 所以最终仍然是看3号寄存器的变化情况



- 当然，已经在4testAll.txt为一些指令运行情况作了注释，并且在当面验收时验证了为正确结果(主要还是看的regfile里的sp和ra寄存器的值)，所以这里就不再展示。

实验总结

（说说自己踩的坑，总结收获，分析下自己花了多少时间，都用来做什么事情）

- 这次实验是COD后第一次用verilog，很多verilog相关语法有些遗忘，所以花了一定时间来复习语法，熟悉verilog
- 本次实验的前两个阶段是完全对照设计图实现的，是对各个模块细节的补充。
    - 感觉上ControlUnit.v是比较繁琐的一部分，一方面要把所有信号一个不落的全部赋值，另一方面需要与指令一一对照，并参考parameters.v里的参数定义，才能够把细节一点点抠出来。自己在实现的过程中也在这部分遇到一些bug(纯粹因为写错)，耽搁了一些时间，但debug的过程也是加深了我对流水线的实现细节的认知。
    - 此外，Harzard.v模块我认为是本次实验的精髓所在，需要对数据相关及其特殊情况(ld)，还有控制相关都有很好的了解后，才能写出代码逻辑，这里我也是借助于COD实验里的经验，在

把代码逻辑完全构思好后再下手，整体感觉良好。
    - 然后就是WBSegReg里那两行的代码补全，看似量很少，但实际也花了我不少时间来琢磨要怎么填。这里的技巧就是通过借助顶层模块RV32Core.v 的参数以及store类指令的存储位数来理解。理解到位了，实现就水到渠成。
- 在阶段三时，自己是先借助lab1的完成的数据通路，摸索着理清需要添加的一些东西，然后着手实现。但是却也落下了很大的隐患，因为一开始没有考虑到CSR的一些数据相关的处理，导致EX段的一些选择器处的实现有点无从下手。

  所以最后，自己又把相关这一块的思路捋清楚后，完善了数据通路，才最终实现成功。详细思路也在实验过程中记录下来，最后整理到了实验报告的阶段3部分。当然，对于CSR遇到x0寄存器的问题，也是最后意识到后才补充上去的，说明想要一次性把所有问题全部捋清楚后才实现不太现实，还是要边写，边想，边完善，先以最小的写的时间代价来找问题，然后完善思路，最后再整体实现。
- 为了完成本次实验，三个阶段我共花费了4天左右的时间。前2天时间完成前两个阶段，后 2天时间完成阶段三（因为还涉及到测试例子的阅读设计）。这个过程中也是花了不少时间debug，以及构思整个数据通路。虽然过程很艰难，在做实验前也觉得困难重重，但做完之后，还是觉得收获很大，也很有成就感。

提出改进实验的意见

- 感觉lab1的答案给的不是很好，可以考虑对lab1答案做详细的讲解，特别是CSR数据通路设计这一块，这样在本次实验阶段3就会少踩一些坑。