lab02 实验报告

实验目标

在实验一中我们初步了解了RISCV指令集和数据通路，接下来需要用verilog去实现我们的RV32I流水线CPU。根据助教提供的代码框架， 实现并完善RV32I流水线CPU，需要完成基本的计算功能，进行冒险处理，添加CSR指令数据通路。

实验环境和工具

实验环境： 浏览器和VNC远程桌面的方式来使用虚拟机

实验工具： vivado , VSCode

实验要求

实验分三阶段要求

第一阶段：

- 自己编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块内存，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

第二阶段：

- 此时需要处理数据相关，实现Harzard模块内部逻辑。
- 测试用的指令流中，除阶段一的测试指令，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。

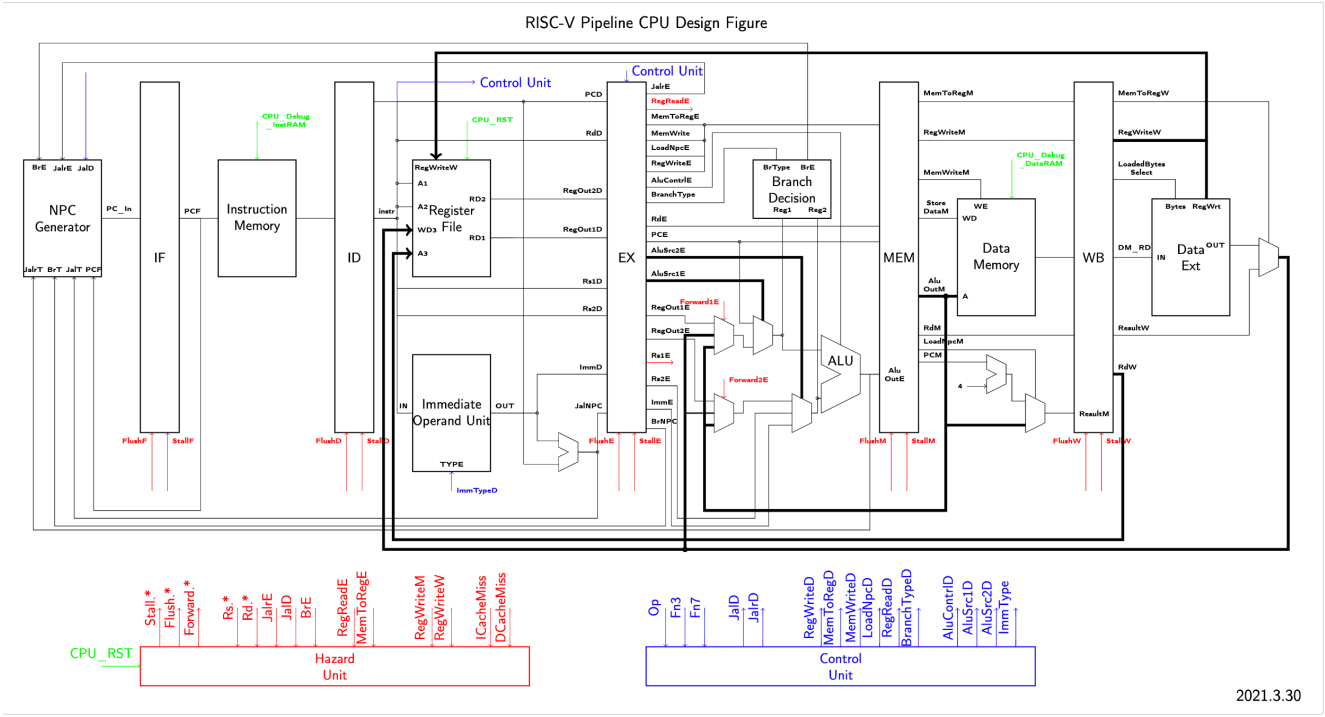
第三阶段：

- 在提供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
- 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)

实验过程

第一阶段：

1. 完善助教提供的代码框架

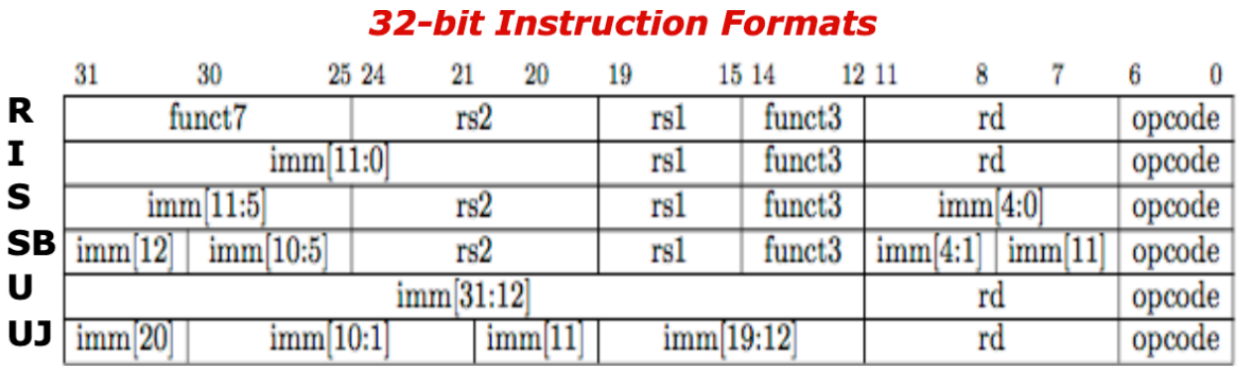


数据通路如图。

这一阶段有几个关键的部分。

首先是对控制单元的实现。也就是译码阶段针对不同的指令，产生相应的控制信号。这一部分实现的逻辑是根据指令不同的类型，按照不同格式解释指令 。再根据 `Op ,fun3, funct7,  rs1,rs2,rd,imm,` 字段的具体内容产生具体的控制信号 。

指令格式如下图。



代码框架如下：

```
always @(*) begin
    case (Op)
        7'b1101111: //Jal
        begin
            csr_wenD = 0;
            csr_immorregD = 0;
            RegWriteD = `LW;
            MemToRegD = 1'b0;
            MemWriteD = 4'b0000;
            LoadNpcD = 1'b1;
            RegReadD = 2'b00;
            BranchTypeD = 3'b000;
            AluContrlD = `ADD;
            AluSrc2D = 2'b00;
            AluSrc1D = 1'b0;
            ImmType = `JTYPE;
        end
        7'b1100111: //Jalr
        7'b1100011: //Branch
        7'b0110111: //LUI
        7'b0010111: //AUIPC
        7'b0010011: //alu imm
        7'b0110011: //alu reg
        7'b0000011: //load
        7'b0100011: //store
        7'b1110011: //csr
        default
    endcase
    end
endmodule
```

如上面的框架，在所有的Op后面，给控制信号赋值。具体的赋值内容根据具体的指令操作即可。需要注意的是`Op`可以决定指令的类型，但不能唯一决定指令，有时还需要根据不同的`fun3, funct7`产生具体的信号。

2. 编写测试样例

在第一阶段，还没有处理相关。所以对于流水线结构，为了验证基本数据通路的正确性，在两条测试指令之间插入四条nop指令。设计思路就是需要验证的21条指令均写一遍 。截取部分测试样例如下。

```
10054: f000f0b7            lui ra,0xf000f
10058: 00000013            nop
1005c: 00000013            nop
10060: 00000013            nop
10064: 00000013            nop
10068: 00409113            slli  sp,ra,0x4
1006c: 00000013            nop
10070: 00000013            nop
10074: 00000013            nop
10078: 00000013            nop
1007c: 0040d193            srli  gp,ra,0x4
```

说明正确性的方法是检查每一步指令结束后的寄存器值是否符合正常逻辑计算的值。

3. 仿真测试

   仿真文件是助教已经写好的， 这一部分需要先理解仿真文件的具体操作 。 浏览文件后发现，仿真文件只是读入instr , data 后 reset一下cpu，开始执行指令， 最后足够长时间后停止仿真，所以只需修改输入的文件即可。

第二阶段：

1. 分析几种相关的种类，和处理方法。

   数据相关： 数据相关可以通过转发处理 ， 对于`load-use` 型的数据相关处理方法是插入 bubble：

   > 第一个源操作数的转发条件：

   转发MEM段的条件

   - Rs1E = RdM ， 表示现在的源寄存器是上一条指令的目的寄存器
   - RegWriteM 要有效 ， 表示的是上一条指令是要写回 RdM 的 。
   - Regread[1] == 1 ，表示Rs1E是有效的。
   - RdM != 0 因为如果是zero 寄存器，写回是无效的。如果转发反而错误 。

     转发WE段的条件

   - Rs1E = RdW 。
   - RegWriteW 要有效 。 表示的是上面隔一条 指令的regwrite 信息 。
   - RegRead[1] == 1
   - RdM != 0

   > 第二个源操作数的转发条件： 逻辑同上。

   .....

   > Load use 的处理：

因为读出来的已经是在MEM阶段 。 后面一条指令已经处于EX阶段，需要使用。如果直接转发此时产生load-use 冒险 。 此时使用bubble。

条件判断：

- RdE = Rs1D || RdE = Rs2D .
- MemToRegE==1，表示上一条指令是load指令
- 不能是 Rs1E = RdM 因为这个阶段已经要使用 Rs1 。但是还没有处理

操作：

- 让 时钟边沿来到之后 。 EX 阶段不执行 后面的正常。即 Stall IF , stall ID , flush E。

控制相关： 分支预测，成功则继续执行， 失败则将分支语句后读入的指令flush 掉

- Branch

  检测阶段: EX段结束出结果 。

  有效信号： BranchE 。

  操作： 后面读入的指令清除 在下一个周期 即flushD 和 flushE 。

- Jalr

  同上

- Jal

  检测阶段: ID段 结束出结果 。

  有效信号： Jal。

  操作：后面读入的指令 在下一个周期 。 即 flushD 。

2. 代码实现

具体实现只要将上面几种情况列举一下处理即可, 代码框架如下：

```
always @(*) begin
    if(!CpuRst)
    begin
        begin //数据相关 。
        if(Rs1E == RdM && RegWriteM != 3'b0 && RegReadE[1] == 1)
            Forward1E <= 2'b10;
        else if(Rs1E == RdW && RegWriteW != 3'b0 && RegReadE[1] == 1)
             Forward1E <= 2'b01;
        else  Forward1E <= 2'b00;
        end
          //转发处理  Forward1
        begin
        if(Rs2E == RdM && RegWriteM != 3'b0 && RegReadE[0] == 1)
            Forward2E <= 2'b10;
        else if(Rs2E == RdW && RegWriteW != 3'b0 && RegReadE[0] == 1)
             Forward2E <= 2'b01;
        else  Forward2E <= 2'b00;
        end
        begin//控制相关
```

```
            //load-use 插入bubble
            if((Rs1D == RdE || Rs2D = RdE) && MemToRegE == 1 )
            else if(BranchE || JalrE)
            else if(JalD)
            else
        end
    end
    else

end
endmodule
```

3. 仿真测试

   这一阶段只需使用助教提供的测试样例 。 仿真后三号寄存器是1即可

第三阶段：

1. 分析csr的实现逻辑 ， 思考可能的实现方式

   需要实现的CSR指令是把csr寄存器的值写入rd寄存器 。 把 rs寄存器的值 或 zimm 经过运算后写回CSR。

   所以分两个部分实现 。

   第一个部分是往`RegisterFile` 里写 。

   这一部分我的实现是利用 原本的写回数据通路，分5个周期写回 。 我需要做的就是在MEM阶段 ， 让`ResultM` 选择的结果是 `CSRFile` 读出的值， 而不是正常的AluOut的值即可。

   第二部分是把Alu的运算结果写回CSRFile 。

   这一阶段需要处理ALU的操作数选择 ， 和操作的运算类型。这个直接在`ControlUnit` 实现。

```
            7'b1110011:
            begin
                csr_wenD = 1;
                MemWriteD = 4'b0000
                RegWriteD <= `LW;
                MemToRegD <= 1'b0;
                LoadNpcD <= 1'b0;
                BranchTypeD <= 3'b000;
                AluSrc2D <= 2'b00;
                AluSrc1D <= 1'b0;   //rs1
                ImmType <= 0;
                RegReadD <= 2'b11;
                case(Fn3)
                3'b001:
                    begin
                        AluContrlD = `OP1 ;
                        csr_immorregD = 0 ;
                    end
```

```
                3'b010:
                3'b011:
                3'b101:
                3'b110:
                3'b111:
                endcase
            end
```

操作数的选择，我定义了`csr_Operand1 , nocsr_Operand1 ,csr_Operand2 , nocsr_Operand2` 然后使用`csr_wenE` 信号选择。

然后是写回CSRFile 。这里我的实现是在EX段出结果后直接写回 。 这样操作的好处是，可以避免一些数据相关。另外一点值得注意的是，写回是在时钟下降沿，这样可以在半个周期实现。不耽误下一个周期的读数据相关。

2. 代码实现

这部分的核心代码是上面的控制信号产生 。 和以下部分的信号选择。

```
    assign needforward = (csr_wenM == 0)? AluOutM : csr_rout_dataM ;
    assign ForwardData1 = Forward1E[1]?(needforward):( Forward1E[0]?
RegWriteData:RegOut1E );
    assign ForwardData2 = Forward2E[1]?(needforward):( Forward2E[0]?
RegWriteData:RegOut2E );
    assign csr_zimmextD = { {27{1'b0}} , Rs1D};
    assign nocsr_Operand1 = AluSrc1E? PCE:ForwardData1;
    assign csr_Operand1 = csr_immorregE? csr_zimmextE:ForwardData1;
    assign Operand1 = csr_wenE? csr_Operand1:nocsr_Operand1;
    assign nocsr_Operand2 = AluSrc2E[1]?(ImmE):( AluSrc2E[0]?Rs2E:ForwardData2 );
    assign Operand2 = csr_wenE? csr_rout_dataE:nocsr_Operand2;
    assign nocsr_ResultM = LoadNpcM ? (PCM+4) : AluOutM;
    assign ResultM = csr_wenM? csr_rout_dataM : nocsr_ResultM;
    assign RegWriteData = ~MemToRegW ? ResultW:DM_RD_Ext;
```

3. 仿真测试

编写测试样例包含 ，所有实现的的CSR指令。

```
00010054 <test_0>:
   10054: 00000193           li  gp,0
   10058: 00f00093           li  ra,15
   1005c: 00009073           csrw  ustatus,ra
   10060: 00003173           csrrc sp,ustatus,zero
   10064: 06111063           bne sp,ra,100c4 <failed>
   10068: 000c7073           csrci ustatus,24
   1006c: 00003173           csrrc sp,ustatus,zero
   10070: 00700093           li  ra,7
   10074: 04111863           bne sp,ra,100c4 <failed>
00010078 <test_2>:
```

```
   10078: 00200193            li  gp,2
   1007c: 00100093            li  ra,1
   10080: 00209073            fsrm  ra
   10084: 002c6173            csrrsi  sp,frm,24
   10088: 02111e63            bne sp,ra,100c4 <failed>
   1008c: 00201173            fsrm  sp,zero
   10090: 01900093            li  ra,25
   10094: 02111863            bne sp,ra,100c4 <failed>
00010098 <test_3>:
   10098: 00300193            li  gp,3
   1009c: 003c5073            csrwi fcsr,24
   100a0: 00700093            li  ra,7
   100a4: 0030a173            csrrs sp,fcsr,ra
   100a8: 01800093            li  ra,24
   100ac: 00111c63            bne sp,ra,100c4 <failed>
   100b0: 00301173            fssr  sp,zero
   100b4: 01f00093            li  ra,31
   100b8: 00111663            bne sp,ra,100c4 <failed>
000100bc <success>:
   100bc: 00100193            li  gp,1
   100c0: ffdff06f            j 100bc <success>
000100c4 <failed>:
   100c4: 0000006f            j 100c4 <failed>
```

正确的结果是在仿真后3号寄存器为1 。

实验结果

这里展示实验仿真的结果 。 仅截取说明正确性的部分

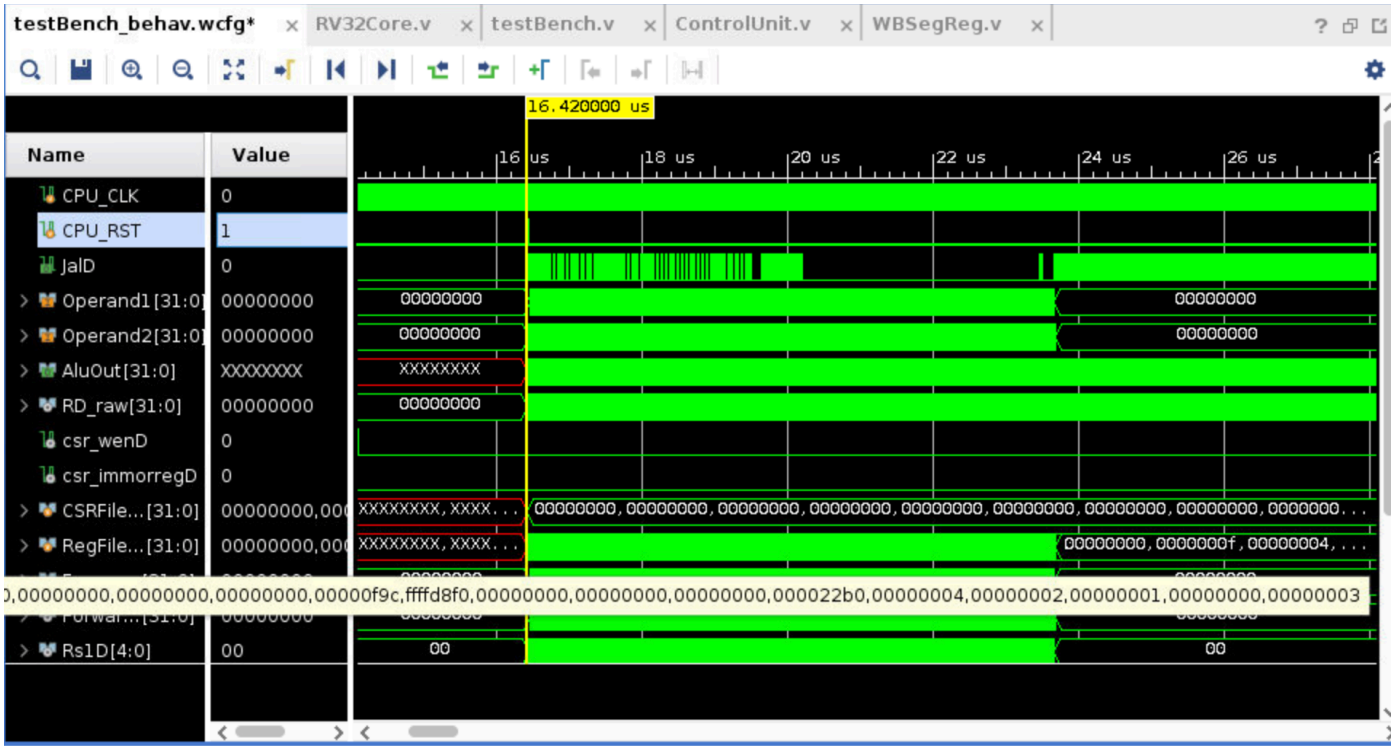
第一阶段：

由于第一阶段的正确性，是第二三阶段的基础，所以以下两个阶段的正确性，可以说明阶段一的正确性。

第二阶段：

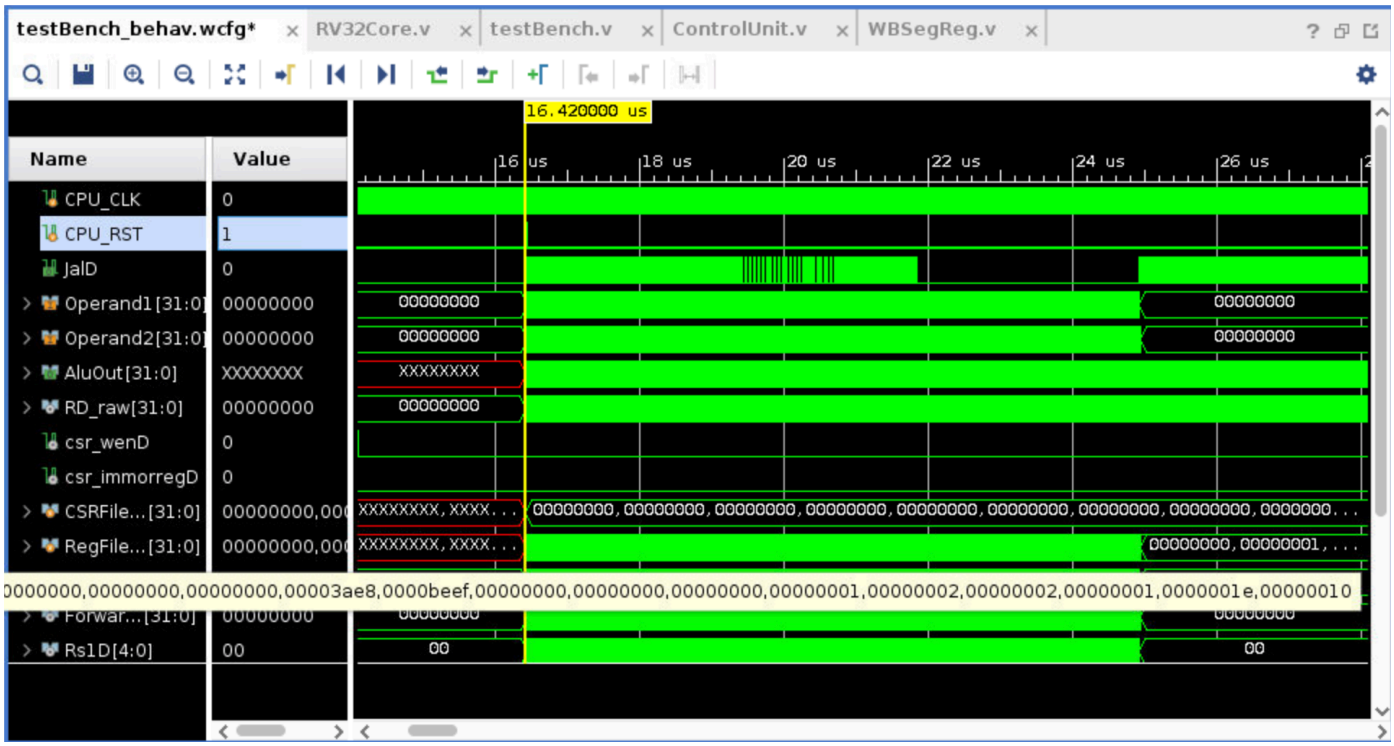
三仿真结果如图。

Test1



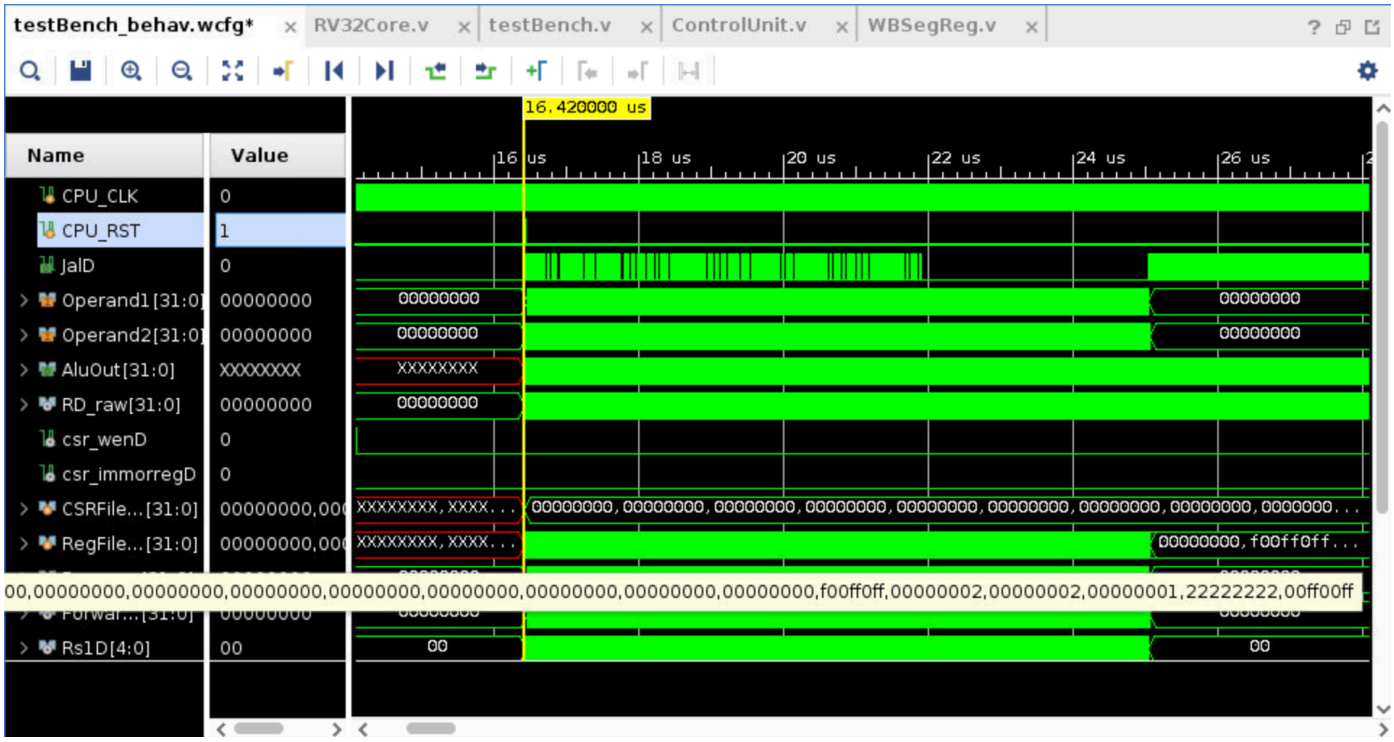
可以看到结束时三号寄存器的值是1 。

Test2



可以看到结束时三号寄存器的值是1 。

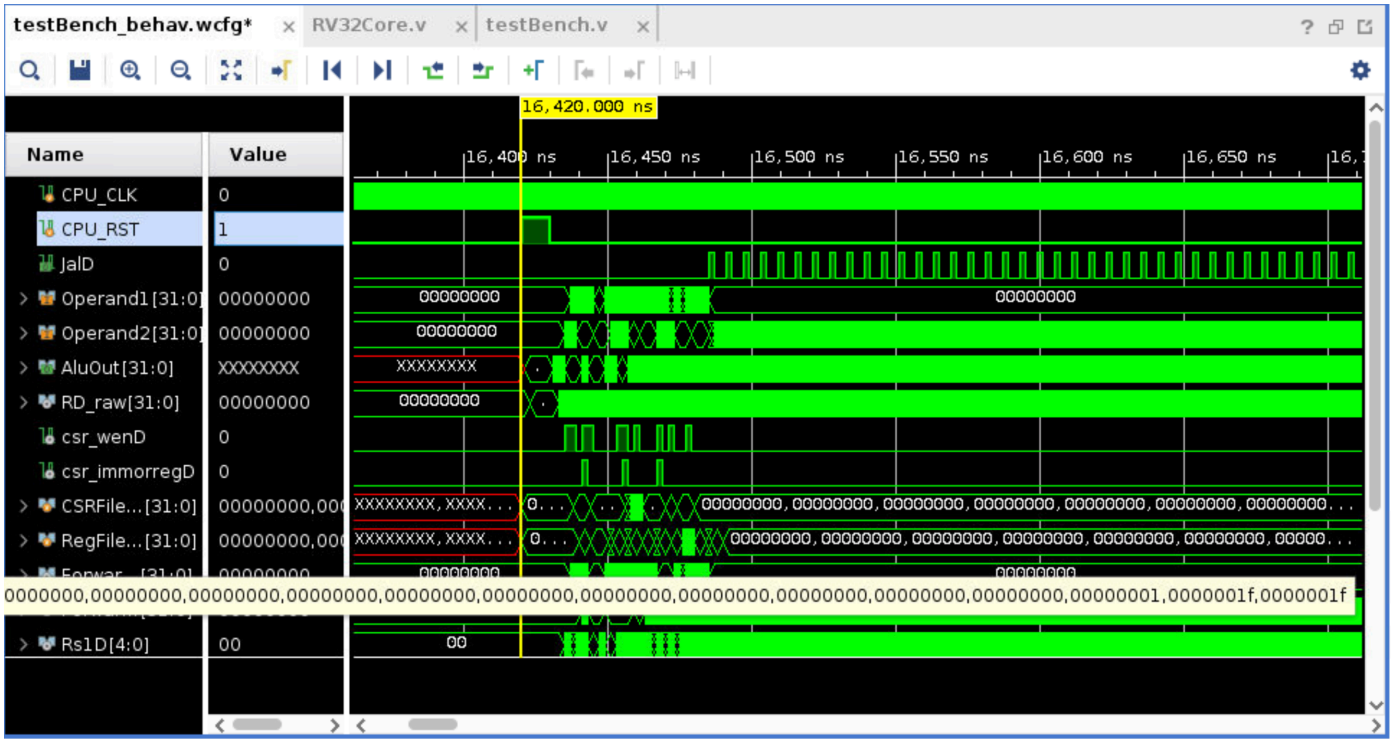
Test3



可以看到结束时三号寄存器的值是1 。

第三阶段：

仿真测试结果：



可以看到结束时三号寄存器的值是1 。

实验总结

遇到的问题及解决

1. 非字对齐

   这一部分需要比较巧妙的设计来处理 。 并且结合已经给出的DataRam的读写逻辑 。 来实现 。 对不同的指令类型给出正确的结果 。 我的视线方法是。设计一个四位的 MemWrite信号 。 指定写的字节 。

```
case(Fn3)
       3'b000: MemWriteD <= 4'b0001;
       3'b001: MemWriteD <= 4'b0011;
       3'b010: MemWriteD <= 4'b1111;
endcase
```

   对应DataRam的写使能信号。并且传入的写数据是经过位移的 对应DataDam的写数据逻辑 。

```
        .wea    (WE<<A[1:0]),                       //?????
        .addra  (A[31:2]),                        //?????
        .dina   (WD<<(8*A[1:0])),
```

2. Load-use相关的处理

   Load-use 型的数据相关需要和控制相关一起处理。并且优先级高于Jar。一开始我的实现没有考虑到load-use的实现时机，把他和控制相关并列处理了， 这样导致电路产生问题 。因为这两个处理都是对Stall 和Flush系列信号的处理， 并列处理相当于一个信号有两个控制逻辑 。最终修改为

```
        begin//控制相关
            //load-use 插入bubble
            if((Rs1D == RdE || Rs2D = RdE) && MemToRegE == 1 )   //load-use
            else if(BranchE || JalrE)   //Branch , Jalr
            else if(JalD)               //Jal
            else
        end
```

   写到一个句子里即可 。 优先级提到了最前面 。 因为如果use是跳转指令，也是保证时输入正确，在判断是否跳转的。

3. csr的数据相关处理

   一开始实现是，并没有考虑相关，导致测试数据通不过。因位我的写回CSRFile是在EX阶段写回的，所以认为不会产生相关 。 但是值得注意的是，往RegFile写的方向需要5个周期 。 所以需要转发 。 如

```
   10060: 00003173            csrrc sp,ustatus,zero
   10064: 06111063            bne sp,ra,100c4 <failed>
```

   这里的转发信号，和之前是一样的，不需要对信号进行处理。需要的处理是，选择转发的数据，具体是AluOut的结果（non-csr 指令） ， 还是CSRFile读出的指令（CSR 指令） 。

```
    assign needforward = (csr_wenM == 0)? AluOutM : csr_rout_dataM ;
```

引发的思考

目前的实验仅停留在数据数通路的正确性要求上，对实现的CPU的效率，性能没有分析。使用仿真看到的结果也不会体现硬件的时延，依然是逻辑上的输出结果，如果需要上板子，可能就需要考虑更多的实际问题。