计算机体系结构实验报告

LAB02

题目:　RISC-V 32I 的实现

姓名: 魏钊

学号:　PB18111699

实验目的：

设计一个 RISC-V 32I 指令集的五段流水线 CPU，实现转发、冒险检测和 CSR 指令。

实验环境和工具：

Vivado

实验设计与过程：

ALU 模块：

```
module ALU(
    input wire [31:0] Operand1,
    input wire [31:0] Operand2,
    input wire [3:0] AluContrl,
    output reg [31:0] AluOut
    );
    always@(*)
        case (AluContrl)
            `SLL: AluOut = Operand1 << Operand2[4:0];
            `SRL: AluOut = Operand1 >> Operand2[4:0];
            `SRA: AluOut = $signed(Operand1) >>> Operand2[4:0];

            `ADD: AluOut = Operand1 + Operand2;
            `SUB: AluOut = Operand1 - Operand2;

            `XOR: AluOut = Operand1 ^ Operand2;
            `OR : AluOut = Operand1 | Operand2;
            `AND: AluOut = Operand1 & Operand2;

            `SLT: AluOut = ($signed(Operand1) < $signed(Operand2)) ? 32'b1 : 32'b0;
            `SLTU: AluOut = (Operand1 < Operand2) ? 32'b1 : 32'b0;

            `LUI: AluOut = Operand2;
            `CSR: AluOut = Operand1;
            default: AluOut = 32'hxxxxxxxx;
        endcase
endmodule
```

ALU 默认将所有操作数视为无符号数，但实际上部分运算是有符号数，这里我们使用 $signed 函数将其转换为有符号数。CSR 指令只需将操作数 1 输出即可。

立即数模块：

```
`include "Parameters.v"
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] };
            `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{1'b0}} };
            `JTYPE: Out <= { {12{In[31]}}, In[19:12], In[20], In[30:21], 1'b0 };
            default:Out<=32'hxxxxxxxx;
        endcase
    end

endmodule
```

按照手册中格式拼接即可：

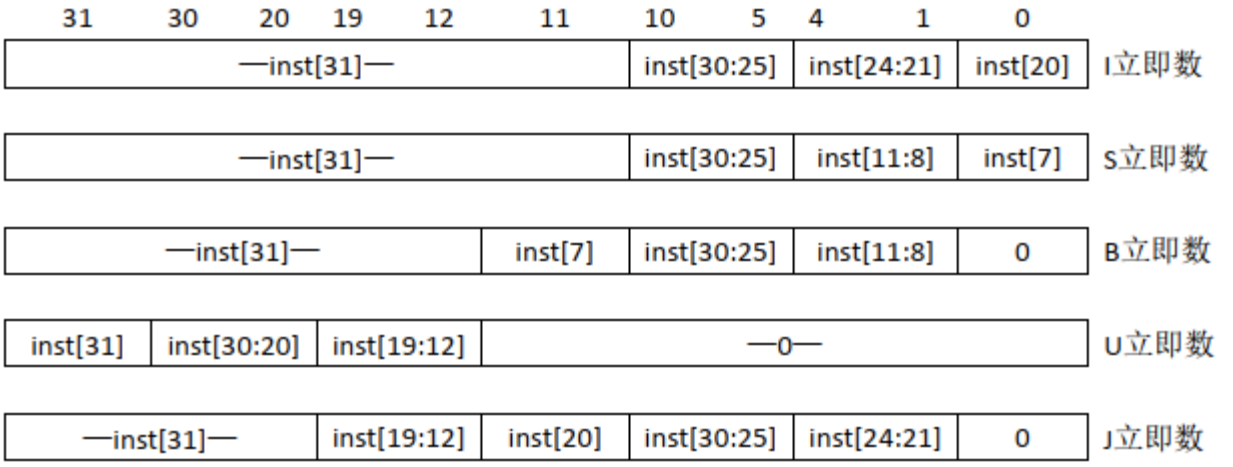


图 2.4：RISC-V 指令生成的立即数。用指令的位标注了用于构成立即数的字段。符号扩展总是使用 inst[31]。

NPC 模块：

```
module NPC_Generator(
    input wire [31:0] PCF,JalrTarget, BranchTarget, JalTarget,
    input wire BranchE,JalD,JalrE,
    output reg [31:0] PC_In
    );
    always@(*)
        if(BranchE)
            PC_In = BranchTarget;
        else if(JalrE)
            PC_In = JalrTarget;
        else if(JalD)
            PC_In = JalTarget;
        else
            PC_In = PCF + 4;
    // 璇疯 - 鍏ㄥ鍙?

endmodule
```

因为 Branch 和 Jarl 指令在 EX 段跳转，而 Jal 指令在 ID 段跳转，所以前两者优先度大于后者。

跳转决定单元：

```
`include "Parameters.v"
module BranchDecisionMaking(
    input wire [2:0] BranchTypeE,
    input wire [31:0] Operand1,Operand2,
    output reg BranchE
    );
    always@(*)
        case (BranchTypeE)
            `NOBRANCH: BranchE = 1'b0;
            `BEQ: BranchE = Operand1 == Operand2 ? 1'b1 : 1'b0;
            `BNE: BranchE = Operand1 != Operand2 ? 1'b1 : 1'b0;
            `BLT: BranchE = $signed(Operand1) < $signed(Operand2) ? 1'b1 : 1'b0;
            `BLTU: BranchE = Operand1 < Operand2 ? 1'b1 : 1'b0;
            `BGE: BranchE = $signed(Operand1) >= $signed(Operand2) ? 1'b1 : 1'b0;
            `BGEU: BranchE = Operand1 >= Operand2 ? 1'b1 : 1'b0;
            default: BranchE = 1'b0;
        endcase
    // 璇疯 - 鍏ㄥ鍙?
endmodule
```

使用上文中提到的$signed 函数完成涉及有符号数的运算。

DataExt 模块：

```
`include "Parameters.v"
module DataExt(
    input wire [31:0] IN,
    input wire [1:0] LoadedBytesSelect,
    input wire [2:0] RegWriteW,
    output reg [31:0] OUT
    );
    always@(*)
        case(RegWriteW)
            `NOREGWRITE: OUT = 32'hxxxxxxxx;
            `LB:
                case(LoadedBytesSelect)
                    2'b00: OUT = {{24{IN[ 7]}}, IN[ 7: 0]};
                    2'b01: OUT = {{24{IN[15]}}, IN[15: 8]};
                    2'b10: OUT = {{24{IN[23]}}, IN[23:16]};
                    2'b11: OUT = {{24{IN[31]}}, IN[31:24]};
                    default: OUT = 32'hxxxxxxxx;
                endcase
            `LH:
                case(LoadedBytesSelect)
                    2'b00: OUT = {{16{IN[15]}}, IN[15: 0]};
                    2'b10: OUT = {{16{IN[31]}}, IN[31:16]};
                    default: OUT = 32'hxxxxxxxx;
                endcase
            `LW:
                OUT = IN;
            `LBU:
                case(LoadedBytesSelect)
                    2'b00: OUT = {{24{1'b0}}, IN[ 7: 0]};
                    2'b01: OUT = {{24{1'b0}}, IN[15: 8]};
                    2'b10: OUT = {{24{1'b0}}, IN[23:16]};
                    2'b11: OUT = {{24{1'b0}}, IN[31:24]};
                    default: OUT = 32'hxxxxxxxx;
                endcase
            `LHU:
                case(LoadedBytesSelect)
                    2'b00: OUT = {{16{1'b0}}, IN[15: 0]};
                    2'b10: OUT = {{16{1'b0}}, IN[31:16]};
                    default: OUT = 32'hxxxxxxxx;
                endcase
            default: OUT = 32'hxxxxxxxx;
        endcase
endmodule
```

这个模块处理 Load 类型指令及访存地址非字对齐的情况。

由于 Data Memory 是按字访问的，内存地址的低 2 位未使用，要在访存后根据寄存器写入模式（已定义在 Parameters.v 中）及低 2 位地址进行进一步的选择：

对于 LW 模式，无需处理， OUT = IN.

对于 LH 模式，可以按半字读取， 低 2 位地址可能是 00 或者 10，则从访存结果截取低 2 字节或者高 2 字节作为寄存器写入值的低 2 字节，高 2 字节进行符号拓展.

对于 LB 模式，可以按字节读取，低 2 位地址可能是 00/01/10/11，则从访存结果截取第 0/1/2/3 字节作为寄存器写入值的第 0 字节，高 3 字节进行符号拓展.

对于 LHU 模式，与 LH 模式类似，只是高 2 字节位无符号拓展

对于 LBU 模式，与 LB 模式类似，只是高 3 字节位无符号拓展

IDSegReg 模块：

```
    wire [31:0] RD_raw;
    InstructionRam InstructionRamInst (
        .clk    (clk),
        .addra  (A[31:2]),
        .douta  ( RD_raw     ),
        .web    ( |WE2       ),
        .addrb  ( A2[31:2]   ),
        .dinb   ( WD2        ),
        .doutb  ( RD2        )
     );
```

和时钟上升沿同步。另外，指令存储器的读地址应该传入 A[31:2]，按字节读取。

WBSegReg 模块：

```
wire [31:0] RD_raw;
DataRam DataRamInst (
    .clk    (clk),                          //璇峰畬
    .wea    (WE << A[1:0]),
    .addra  (A[31:2]),                          //璇
    .dina   (WD << ({3'b000, A[1:0]} << 3'd3)),
    .douta  ( RD_raw        ),
    .web    ( WE2           ),
    .addrb  ( A2[31:2]      ),
    .dinb   ( WD2           ),
    .doutb  ( RD2           )
);
```

SW、 SH、 SB 指令分别将从 rs2 寄存器低位开始的 32 位、 16 位、 8 位数值保存到存储器中，则 ControlUnit 中 MemWriteD 信号分别置为 4'b1111、 4'b0011、 4'b0001（让存储器按字节写入， 1bit 对应写入一字节）。

在例化 DataMem 模块时，并不能将 WB 段寄存器传入的 WE 和 WD 信号直接接入，因为写入地址的低 2 位不一定是 2'b00，应该需要进行移位，这样就可以实现非字对齐的写入。

设地址低 2 位为 A[1:0]，则写使能应该接入 WE << A[1:0]，写数据应为 WD << ({3'b000, A[1:0]} << 3'd3)。

控制单元模块：

首先定义：

```
`define ControlOut {{JalD,JalrD},{MemToRegD},{RegWriteD},{MemWriteD},{LoadNpcD},{RegReadD},{BranchTypeD},{AluContrlD},{AluSrc1D,AluSrc2D},{ImmType}}
```

方便后续对信号的操作。

然后根据相应的 Op 和 Fn3 设置对应信号即可。

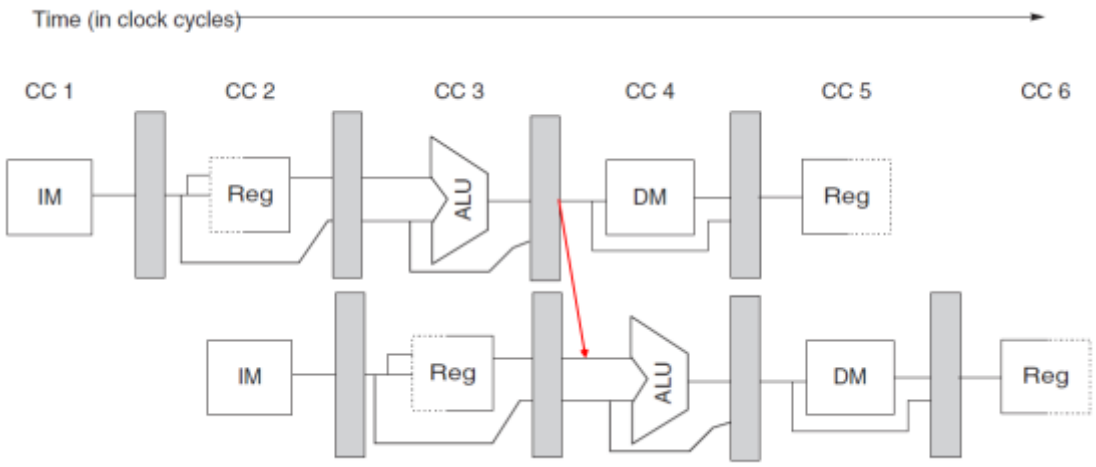
```verilog
always@(*)
    case(Op)
        7'b0010011: //REG-IMM
            case(Fn3)
                3'b000:  `ControlOut = {{2'b0_0}, {1'b0,`LW}, {4'b0000}, {1'b0}, {2'b10}, {`NOBRANCH}, {`ADD}, {3'b0_10}, {`ITYPE}};/* ADDI */
                3'b001:  `ControlOut = {{2'b0_0}, {1'b0,`LW}, {4'b0000}, {1'b0}, {2'b10}, {`NOBRANCH}, {`SLL}, {3'b0_01}, {`ITYPE}};/* SLLI */
                3'b010:  `ControlOut = {{2'b0_0}, {1'b0,`LW}, {4'b0000}, {1'b0}, {2'b10}, {`NOBRANCH}, {`SLT}, {3'b0_10}, {`ITYPE}};/* SLTI */
                3'b011:  `ControlOut = {{2'b0_0}, {1'b0,`LW}, {4'b0000}, {1'b0}, {2'b10}, {`NOBRANCH}, {`SLTU}, {3'b0_10}, {`ITYPE}};/* SLTIU */
                3'b100:  `ControlOut = {{2'b0_0}, {1'b0,`LW}, {4'b0000}, {1'b0}, {2'b10}, {`NOBRANCH}, {`XOR}, {3'b0_10}, {`ITYPE}};/* XORI */
                3'b101:
                    case(Fn7)
                        7'b0000000:  `ControlOut = {{2'b0_0}, {1'b0,`LW}, {4'b0000}, {1'b0}, {2'b10}, {`NOBRANCH}, {`SRL}, {3'b0_01}, {`ITYPE}};/* SRLI */
                        7'b0100000:  `ControlOut = {{2'b0_0}, {1'b0,`LW}, {4'b0000}, {1'b0}, {2'b10}, {`NOBRANCH}, {`SRA}, {3'b0_01}, {`ITYPE}};/* SRAI */
                    endcase
                3'b110:  `ControlOut = {{2'b0_0}, {1'b0,`LW}, {4'b0000}, {1'b0}, {2'b10}, {`NOBRANCH}, {`OR}, {3'b0_10}, {`ITYPE}};/* ORI */
                3'b111:  `ControlOut = {{2'b0_0}, {1'b0,`LW}, {4'b0000}, {1'b0}, {2'b10}, {`NOBRANCH}, {`AND}, {3'b0_10}, {`ITYPE}};/* ANDI */
            endcase
        7'b0110011: //REG-REG
            case(Fn3)
                3'b000: /*  */
                    case(Fn7)
                        7'b0000000:  `ControlOut = {{2'b0_0}, {1'b0,`LW}, {4'b0000}, {1'b0}, {2'b11}, {`NOBRANCH}, {`ADD}, {3'b0_00}, {`RTYPE}};/* ADD */
                        7'b0100000:  `ControlOut = {{2'b0_0}, {1'b0,`LW}, {4'b0000}, {1'b0}, {2'b11}, {`NOBRANCH}, {`SUB}, {3'b0_00}, {`RTYPE}};/* SUB */
                    endcase
                3'b001:  `ControlOut = {{2'b0_0}, {1'b0,`LW}, {4'b0000}, {1'b0}, {2'b11}, {`NOBRANCH}, {`SLL}, {3'b0_00}, {`RTYPE}};/* SLL */
                3'b010:  `ControlOut = {{2'b0_0}, {1'b0,`LW}, {4'b0000}, {1'b0}, {2'b11}, {`NOBRANCH}, {`SLT}, {3'b0_00}, {`RTYPE}};/* SLT */
                3'b011:  `ControlOut = {{2'b0_0}, {1'b0,`LW}, {4'b0000}, {1'b0}, {2'b11}, {`NOBRANCH}, {`SLTU}, {3'b0_00}, {`RTYPE}};/* SLTU */
                3'b100:  `ControlOut = {{2'b0_0}, {1'b0,`LW}, {4'b0000}, {1'b0}, {2'b11}, {`NOBRANCH}, {`XOR}, {3'b0_00}, {`RTYPE}};/* XOR */
                3'b101:
                    case(Fn7)
                        7'b0000000:  `ControlOut = {{2'b0_0}, {1'b0,`LW}, {4'b0000}, {1'b0}, {2'b11}, {`NOBRANCH}, {`SRL}, {3'b0_00}, {`RTYPE}};/* SRL */
                        7'b0100000:  `ControlOut = {{2'b0_0}, {1'b0,`LW}, {4'b0000}, {1'b0}, {2'b11}, {`NOBRANCH}, {`SRA}, {3'b0_00}, {`RTYPE}};/* SRA */
                    endcase
                3'b110:  `ControlOut = {{2'b0_0}, {1'b0,`LW}, {4'b0000}, {1'b0}, {2'b11}, {`NOBRANCH}, {`OR}, {3'b0_00}, {`RTYPE}};/* OR */
                3'b111:  `ControlOut = {{2'b0_0}, {1'b0,`LW}, {4'b0000}, {1'b0}, {2'b11}, {`NOBRANCH}, {`AND}, {3'b0_00}, {`RTYPE}};/* AND */
            endcase
        7'b0110111:  `ControlOut = {{2'b0_0}, {1'b0,`LW}, {4'b0000}, {1'b0}, {2'b00}, {`NOBRANCH}, {`LUI}, {3'b0_10}, {`UTYPE}};/* LUI */
        7'b0010111:  `ControlOut = {{2'b0_0}, {1'b0,`LW}, {4'b0000}, {1'b0}, {2'b00}, {`NOBRANCH}, {`ADD}, {3'b1_10}, {`UTYPE}};/* AUIPC */

        7'b0000011: //Load
            case(Fn3)
                3'b000:  `ControlOut = {{2'b0_0}, {1'b1,`LB}, {4'b0000}, {1'b0}, {2'b10}, {`NOBRANCH}, {`ADD}, {3'b0_10}, {`ITYPE}};/* LB */
                3'b001:  `ControlOut = {{2'b0_0}, {1'b1,`LH}, {4'b0000}, {1'b0}, {2'b10}, {`NOBRANCH}, {`ADD}, {3'b0_10}, {`ITYPE}};/* LH */
                3'b010:  `ControlOut = {{2'b0_0}, {1'b1,`LW}, {4'b0000}, {1'b0}, {2'b10}, {`NOBRANCH}, {`ADD}, {3'b0_10}, {`ITYPE}};/* LW */
                3'b100:  `ControlOut = {{2'b0_0}, {1'b1,`LBU}, {4'b0000}, {1'b0}, {2'b10}, {`NOBRANCH}, {`ADD}, {3'b0_10}, {`ITYPE}};/* LBU */
                3'b101:  `ControlOut = {{2'b0_0}, {1'b1,`LHU}, {4'b0000}, {1'b0}, {2'b10}, {`NOBRANCH}, {`ADD}, {3'b0_10}, {`ITYPE}};/* LHU */
            endcase
        7'b0100011: //Store
            case(Fn3)
                3'b000:  `ControlOut = {{2'b0_0}, {1'b0,`NOREGWRITE}, {4'b0001}, {1'b0}, {2'b11}, {`NOBRANCH}, {`ADD}, {3'b0_10}, {`STYPE}};/* SB */
                3'b001:  `ControlOut = {{2'b0_0}, {1'b0,`NOREGWRITE}, {4'b0011}, {1'b0}, {2'b11}, {`NOBRANCH}, {`ADD}, {3'b0_10}, {`STYPE}};/* SH */
                3'b010:  `ControlOut = {{2'b0_0}, {1'b0,`NOREGWRITE}, {4'b1111}, {1'b0}, {2'b11}, {`NOBRANCH}, {`ADD}, {3'b0_10}, {`STYPE}};/* SW */
            endcase
        7'b1100011: //Branch
            case(Fn3)
                3'b000:  `ControlOut = {{2'b0_0}, {1'b0,`NOREGWRITE}, {4'b0000}, {1'b0}, {2'b11}, {`BEQ}, {4'bxxxx}, {3'b0_00}, {`BTYPE}};/* BEQ */
                3'b001:  `ControlOut = {{2'b0_0}, {1'b0,`NOREGWRITE}, {4'b0000}, {1'b0}, {2'b11}, {`BNE}, {4'bxxxx}, {3'b0_00}, {`BTYPE}};/* BNE */
                3'b100:  `ControlOut = {{2'b0_0}, {1'b0,`NOREGWRITE}, {4'b0000}, {1'b0}, {2'b11}, {`BLT}, {4'bxxxx}, {3'b0_00}, {`BTYPE}};/* BLT */
                3'b101:  `ControlOut = {{2'b0_0}, {1'b0,`NOREGWRITE}, {4'b0000}, {1'b0}, {2'b11}, {`BGE}, {4'bxxxx}, {3'b0_00}, {`BTYPE}};/* BGE */
                3'b110:  `ControlOut = {{2'b0_0}, {1'b0,`NOREGWRITE}, {4'b0000}, {1'b0}, {2'b11}, {`BLTU}, {4'bxxxx}, {3'b0_00}, {`BTYPE}};/* BLTU */
                3'b111:  `ControlOut = {{2'b0_0}, {1'b0,`NOREGWRITE}, {4'b0000}, {1'b0}, {2'b11}, {`BGEU}, {4'bxxxx}, {3'b0_00}, {`BTYPE}};/* BGEU */
            endcase
        7'b1101111:  `ControlOut = {{2'b1_0}, {1'b0,`LW}, {4'b0000}, {1'b1}, {2'b00}, {`NOBRANCH}, {4'bxxxx}, {3'bx_xx}, {`JTYPE}};/* JAL */
        7'b1100111:  `ControlOut = {{2'b0_1}, {1'b0,`LW}, {4'b0000}, {1'b1}, {2'b10}, {`NOBRANCH}, {`ADD}, {3'b0_10}, {`ITYPE}};/* JALR */
        7'b1110011://CSR
            case(Fn3)
                 3'b001: `ControlOut = {{2'b0_0}, {1'b0,`LW}, {4'b0000}, {1'b0}, {2'b00}, {`NOBRANCH}, {`CSR}, {3'b0_00}, {`RTYPE}};/*CSRRW*/
                 3'b010: `ControlOut = {{2'b0_0}, {1'b0,`LW}, {4'b0000}, {1'b0}, {2'b00}, {`NOBRANCH}, {`CSR}, {3'b0_00}, {`RTYPE}};/*CSRRS*/
                 3'b011: `ControlOut = {{2'b0_0}, {1'b0,`LW}, {4'b0000}, {1'b0}, {2'b00}, {`NOBRANCH}, {`CSR}, {3'b0_00}, {`RTYPE}};/*CSRRC*/
                 3'b101: `ControlOut = {{2'b0_0}, {1'b0,`LW}, {4'b0000}, {1'b0}, {2'b00}, {`NOBRANCH}, {`CSR}, {3'b0_00}, {`RTYPE}};/*CSRRWI*/
                 3'b110: `ControlOut = {{2'b0_0}, {1'b0,`LW}, {4'b0000}, {1'b0}, {2'b00}, {`NOBRANCH}, {`CSR}, {3'b0_00}, {`RTYPE}};/*CSRRSI*/
                 3'b111: `ControlOut = {{2'b0_0}, {1'b0,`LW}, {4'b0000}, {1'b0}, {2'b00}, {`NOBRANCH}, {`CSR}, {3'b0_00}, {`RTYPE}};/*CSRRCI*/
            endcase
        default: `ControlOut = 26'b0;
    endcase
// 璇疯¯鍏儿😀澧勫晩鐮?
odule
```

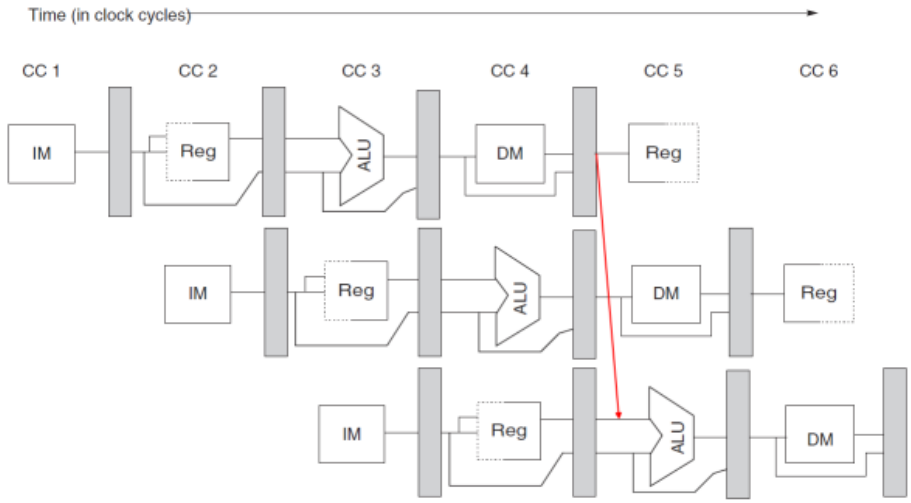
冒险模块：

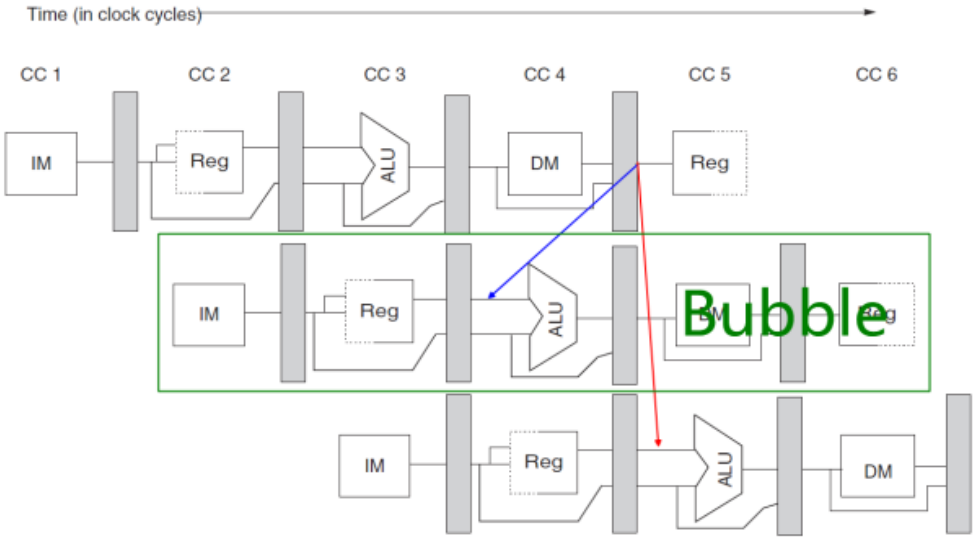
```
always@(*) begin
    FlushF <= CpuRst;//IF寄存器（PC寄存器）只有初始化时需要清空
    FlushD <= CpuRst || (BranchE || JalrE || JalD);//ID寄存器（处于IF/ID之间的寄存器）在发生3种跳转时清空
    FlushE <= CpuRst || (MemToRegE && (RdE == Rs1D || RdE == Rs2D)) || (BranchE || JalrE);//EX寄存器在发生2种跳转和无法转发的数据相关时清空
    FlushM <= CpuRst;//MEM寄存器（处于EX/MEM之间的寄存器）只有初始化时需要清空
    FlushW <= CpuRst;//WB寄存器（处于MEM/WB之间的寄存器）只有初始化时需要清空
    StallF <= `CpuRst && (MemToRegE && (RdE == Rs1D || RdE == Rs2D));
    StallD <= `CpuRst && (MemToRegE && (RdE == Rs1D || RdE == Rs2D));
    StallE <= 1'b0;
    StallM <= 1'b0;
    StallW <= 1'b0;

    //考虑当前指令在ID阶段
    //上一条指令访存并写回 且 当前指令ID阶段读的是同一个寄存器，则停顿（插入bubble）
    //这里并不像Forward那样判断RegWriteE非0，因为写入寄存器的数据可能是ALU的结果也可能是访存的结果
    //只有上一条指令写回寄存器的结果是访存的结果----情况3，才需要停顿，因此用MemToRegE判断
    //如果上一条指令写回寄存器的结果是ALU的结果，那么这就等价于情况1，会用Forward处理
end
```

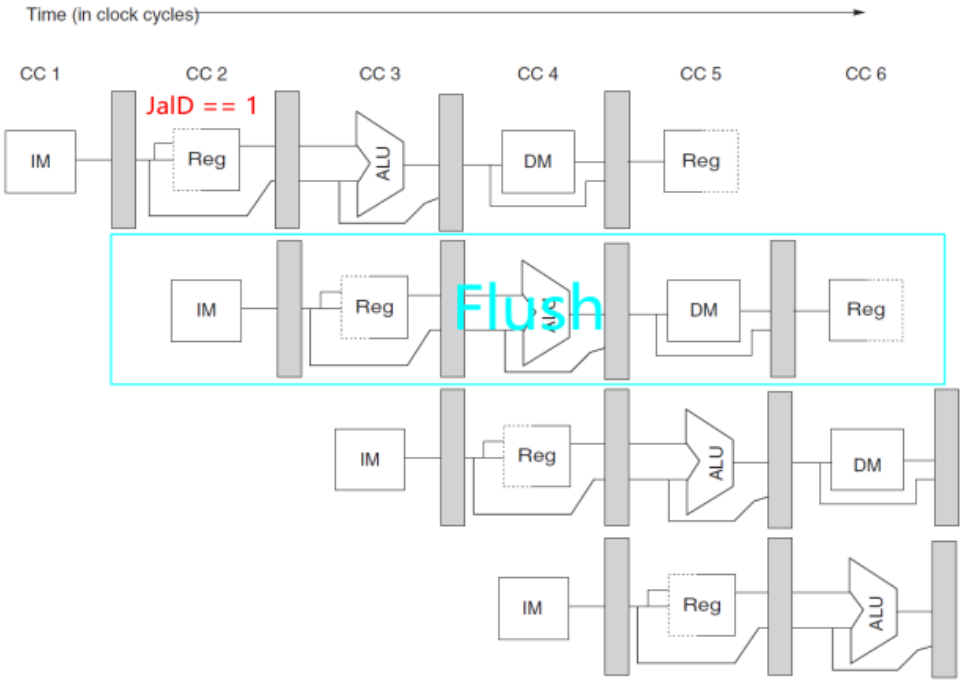
```
always@(*) begin
    //当前指令在EX阶段
    //默认forward=2'b00
    //如果RegWriteM不为0，说明上一条指令（此时在MEM阶段）的ALU结果要写回寄存器----情况1----forward=2'b01
    //如果RegWriteW不为0，说明上上一条指令（此时在WB阶段）的访存结果要写回寄存器----情况2----forward=2'b11
    //应该注意，某些指令写0号寄存器，这是不起作用的，也就无需forward
    //Forward Register Source 1
    Forward1E[0] <= RdW != 0 && |RegWriteW && RegReadE[1] && (RdW == Rs1E) && ~(|RegWriteM && RegReadE[1] && (RdM == Rs1E));//如果上上条指令写回位置是Rs1E，上条指令也是，则应该取上条指令写的值
    Forward1E[1] <= RdM != 0 && |RegWriteM && RegReadE[1] && (RdM == Rs1E);
    //Forward Register Source 2
    Forward2E[0] <= RdW != 0 && |RegWriteW && RegReadE[0] && (RdW == Rs2E) && ~(|RegWriteM && RegReadE[0] && (RdM == Rs2E));//如果上上条指令写回位置是Rs2E，上条指令也是，则应该取上条指令写的值
    Forward2E[1] <= RdM != 0 && |RegWriteM && RegReadE[0] && (RdM == Rs2E);
end
```

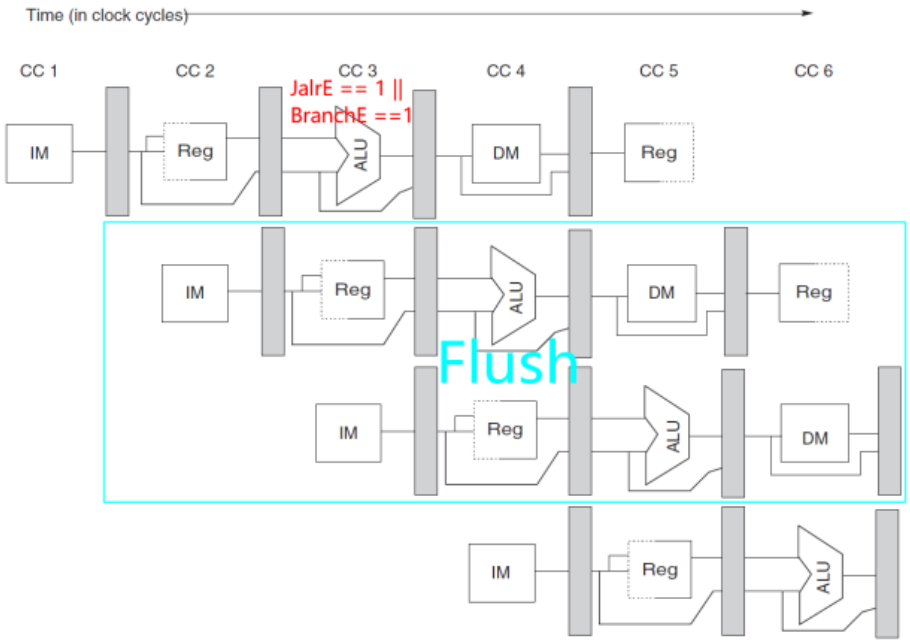
由于某些指令会对 0 号寄存器进行写入（比如不需要 JAL 的链接结果时，可以选择 0 号寄存器作为目标寄存器），但是实际是不会进行写入的。这时如果下一条指令用到了 0 号寄存器作为常量，转发会导致当前指令写入的无用数据被转发给下一条指令。 因此，再默认情况下或者要使用的寄存器是 0 号寄存器，Forward 为 2'b00，表示不使用转发。




Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
IM
Reg
ALU
DM
Reg


Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
IM
Reg
ALU
DM
Reg
Bubble


Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
JalD == 1
IM
Reg
ALU
DM
Reg
Flush

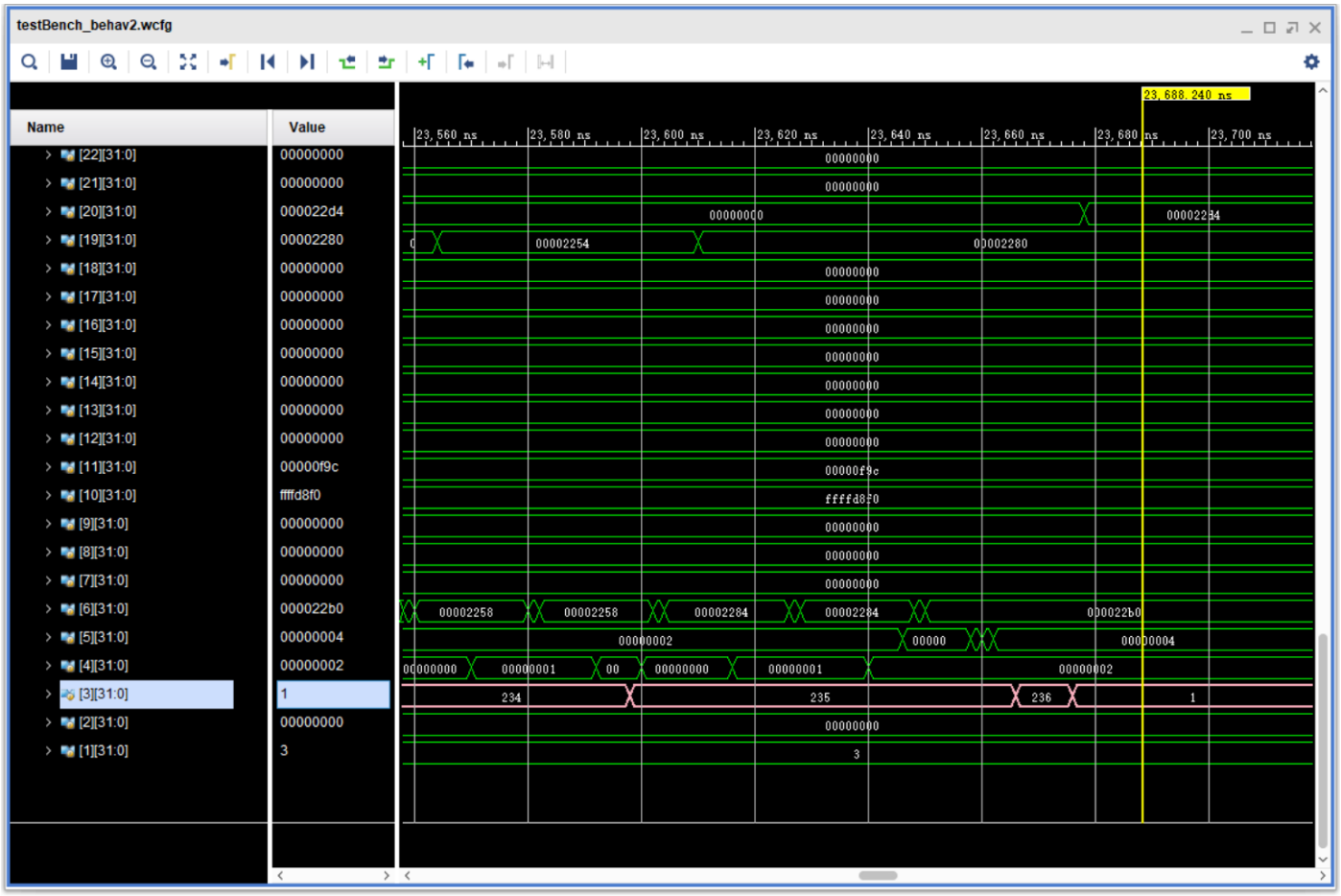


以上为需要转发和 Stall 以及 Flush 的情况。

前两阶段的实验结果：

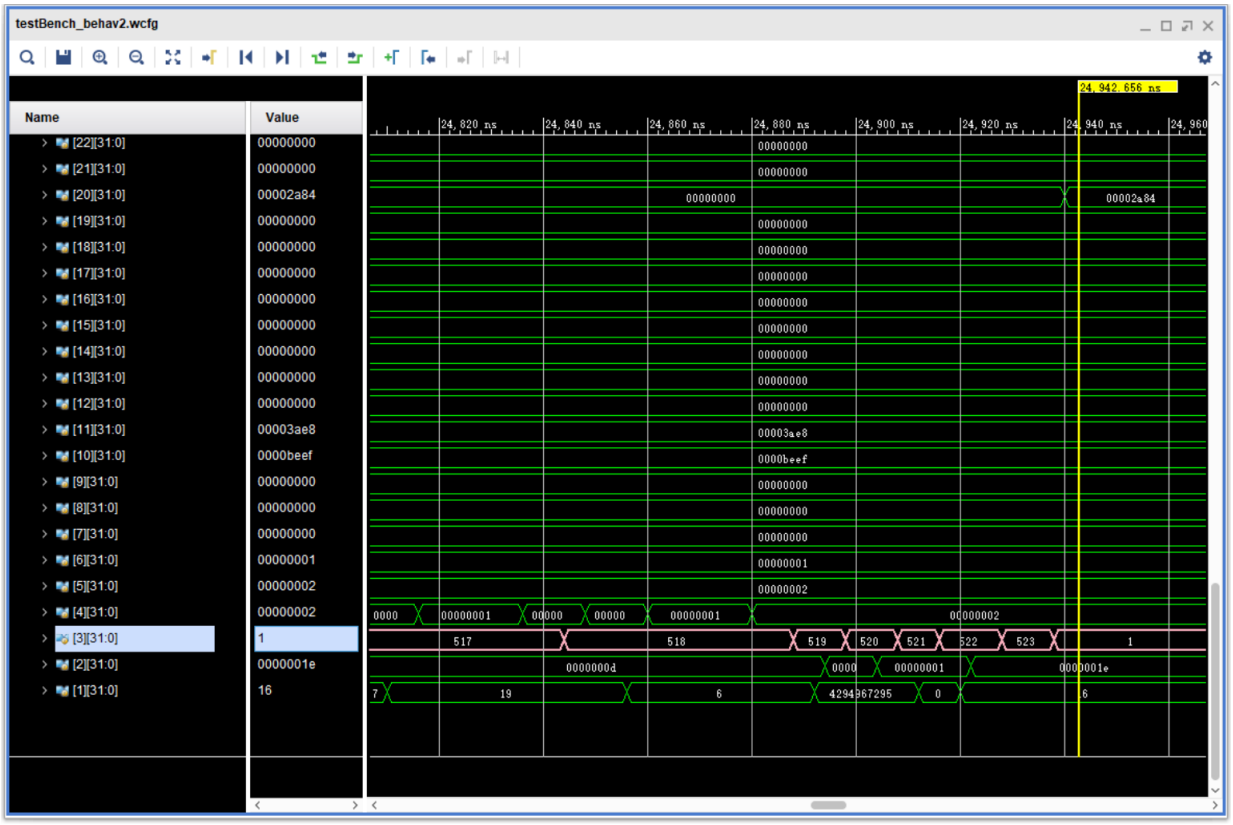
1testAll.inst

仿真结果， 236 号测试后 3 号寄存器的值变为 1， 且不再改变



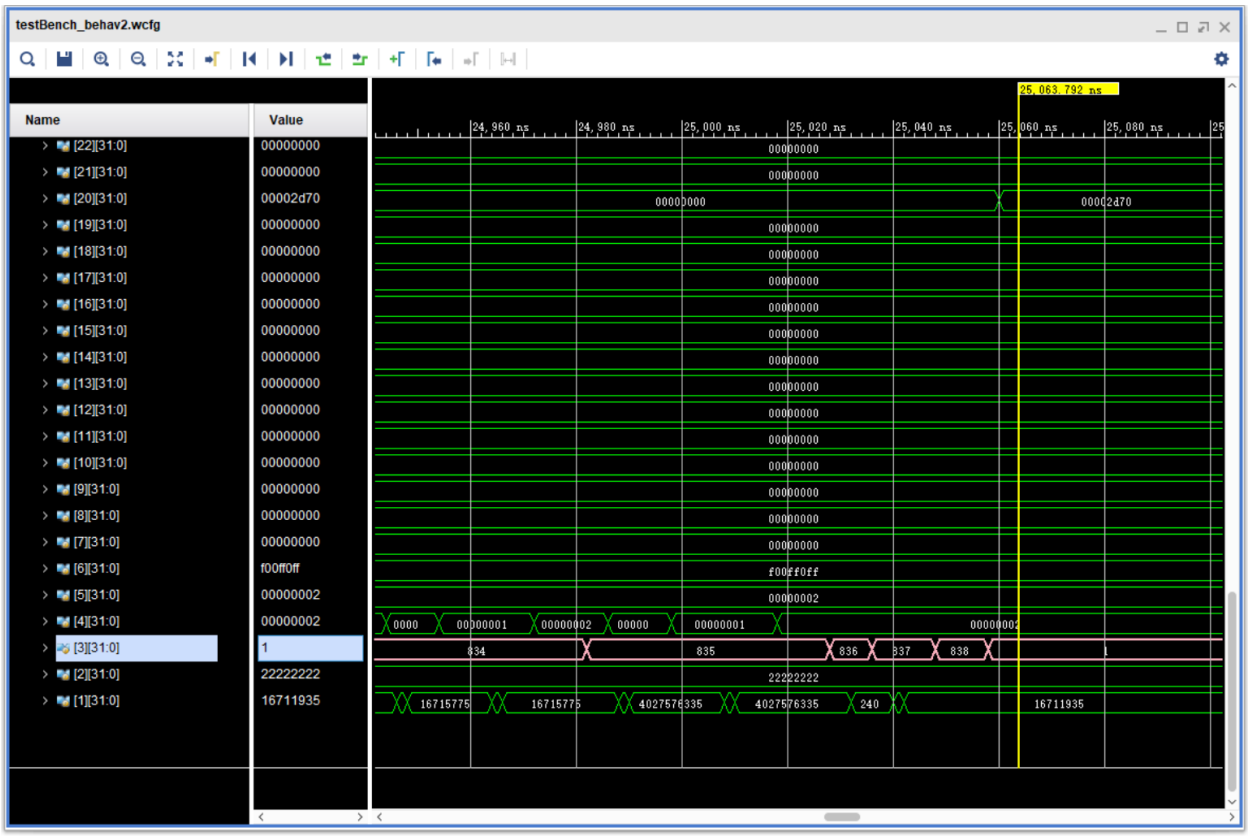
2testAll.inst

仿真结果，　523 号测试后 3 号寄存器的值变为 1，　且不再改变。

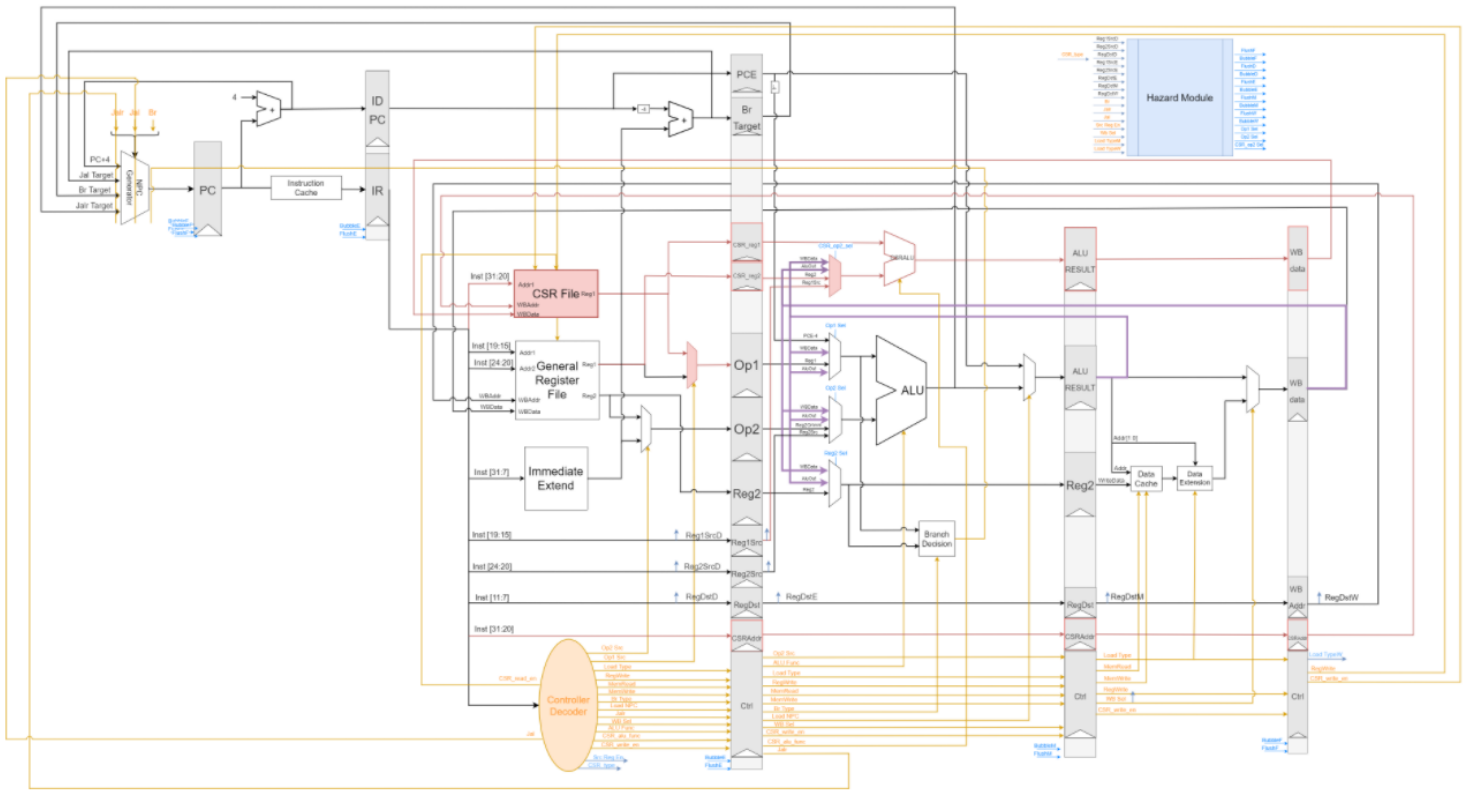


3testAll.inst

仿真结果，　838 号测试后 3 号寄存器的值变为 1，　且不再改变。



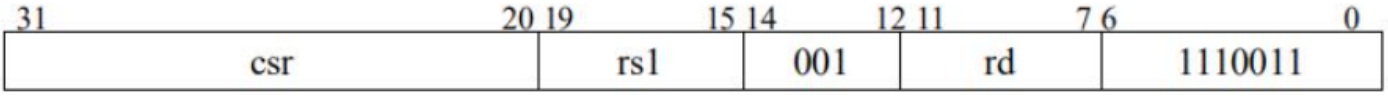
阶段三：



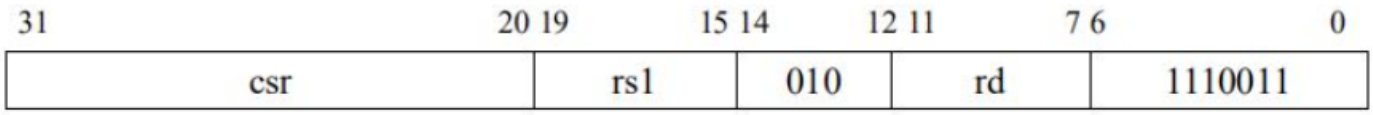
主要增加了 CSR 寄存器，CSR_ALU 以及各个段间寄存器的修改。

首先介绍各个 CSR 指令，及其作用：

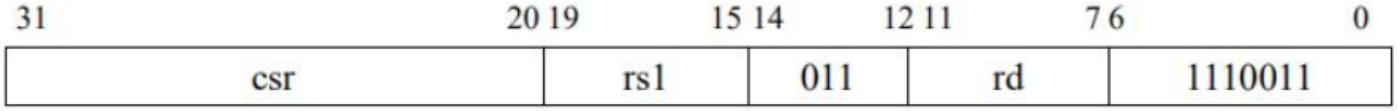
CSRRW(control and status register read and write，读后立即写控制状态寄存器)指令格式为CSRRW rd，csr，rs1。其机器码如图2所示，CSRRW的funct3是001。该指令是把CSR寄存器中的值读出并赋值到rd寄存器中，再把rs1寄存器中的值写入CSR寄存器。



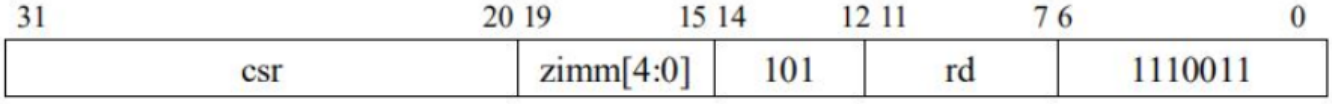
CSRRS(control and status register read and set，读后置位控制状态寄存器)指令格式为CSRRS rd，csr，rs1。其机器码如图3所示，CSRRS的funct3是010。该指令是把CSR寄存器中的值读出并赋值到rd寄存器中，且将CSR寄存器中的值和寄存器rs1中的值按位或(bitwise OR)的结果写入CSR寄存器。



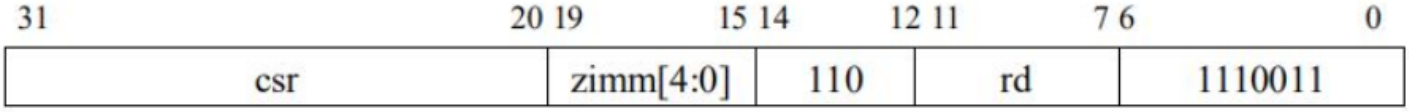
CSRRC(control and status register read and clear，读后清除控制状态寄存器)指令格式为CSRRC rd，csr，rs1。其机器码如图4所示，CSRRC的funct3是011。该指令是把CSR寄存器中的值读出并赋值到rd寄存器中，且将CSR寄存器中的值和寄存器rs1中的值按位与(bitwise AND)的结果写入CSR寄存器。



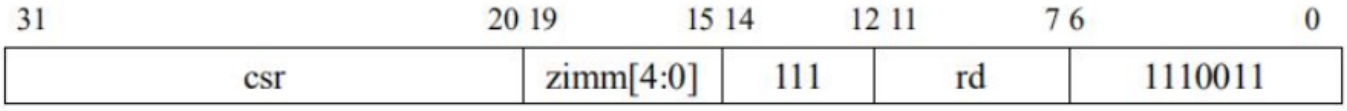
CSRRWI(control and status register read and write immediate，立即数读后写控制状态寄存器)指令格式为CSRRWI rd，csr，zimm[4:0]，这里的zimm[4:0]表示高位由0(zero)扩展的立即数。其机器码如图5所示，CSRRWI的funct3是101。该指令是把CSR寄存器中的值读出并赋值到rd寄存器中，再把五位的零扩展的立即数zimm写入CSR寄存器。



CSRRSI(control and status register read and set immediate，立即数读后设置控制状态寄存器)指令格式为CSRRSI rd，csr，zimm[4:0]。其机器码如图6所示，CSRRSI的funct3是110。该指令是把CSR寄存器中的值读出并赋值到rd寄存器中，且将CSR寄存器中的值和五位的零扩展的立即数zimm按位或(bitwise OR)的结果写入CSR寄存器(CSR寄存器的第五位及更高位不变)。



CSRRCI(control and status register read and clear immediate，立即数读后清除控制状态寄存器)指令格式为CSRRCI rd，csr，zimm[4:0]。其机器码如图7所示，CSRRCI的funct3是111。该指令是把CSR寄存器中的值读出并赋值到rd寄存器中，且将CSR寄存器中的值和寄存器rs1中的值按位与(bitwise AND)的结果写入CSR寄存器(CSR寄存器的第五位及更高位不变)。



CSR 寄存器模块：

```
module CSR(
    input wire clk,
    input wire rst,
    input wire read_en,
    input wire write_en,
    input wire [4:0] addr, wb_addr,
    input wire [31:0] wb_data,
    output wire [31:0] csr_out
    );
    reg [31:0] RegFile[31:0];
    integer i;
    initial
    begin
        for(i = 0; i < 32; i = i + 1)
            RegFile[i][31:0] <= 32'b0;
    end
    always@(negedge clk or posedge rst)
    begin
        if (rst)
            for (i = 0; i < 32; i = i + 1)
                RegFile[i][31:0] <= 32'b0;
        else if(write_en==1'b1)
            RegFile[wb_addr] <= wb_data;
    end
    assign csr_out =(read_en==1'b1)?RegFile[addr]:32'b0;
endmodule
```

具体设计类似通用寄存器，不再赘述。

CSR_ALU:（在参数模块增加`define CSR 4'd11）

```
module CSR_ALU(
    input wire [31:0] op1,
    input wire [31:0] op2,
    input wire [1:0] csrALU_ctrl,
    output reg [31:0] ALU_out
    );
    always @ (*)
    begin
    case(csrALU_ctrl)
        2'b01:ALU_out = op2;
        2'b10:ALU_out = op1 | op2;
        2'b11:ALU_out = op1 & (~ op2);
        default:
        ALU_out = 32'h0;
    endcase
    end
endmodule
```

RV32Core 增加相应的信号、及修改对应的信号：

```
//CSR
wire csr_read_ID,csr_write_ID;
wire csr_read_enID;
wire csr_write_enID,csr_write_enEX,csr_write_enMEM,csr_write_enWB;// csr write or not
wire [4:0] csr_dest_ID,csr_dest_EX,csr_dest_MEM,csr_dest_WB;//the addrass which be writen back
wire [31:0] csr_Alu_out,csr_Alu_outMEM;
wire [31:0] csr_WB;//write back to csr
wire [31:0] csr_data_ID,csr_op1_EX,csr_op2_EX_IN;//read from csr; csralu op1,op2
wire [31:0] csr_op2_EX,csr_op2_EX_IN_F;
wire csr_op_ID,csr_op_EX;// need harzardUnit?
wire [1:0] csr_AluFun_ID,csr_AluFun_EX;
wire csr_imm_ID,csr_imm_EX;//1 -imm 0-rs1
wire [1:0]csr_Forward;//harzard
wire [1:0] csr_op_Forward;
wire [31:0] csr_op_src1;
wire [4:0] csr_op_EX_1;
assign csr_write_enID=(Rs1D==5'b0)?0:1;//0号寄存器,optional
assign csr_read_enID=((Funct3D==3'b001&&RdD==5'b0)||(Funct3D==3'b101&&RdD==5'b0))?0:1;
assign csr_op2_EX_IN=(csr_imm_EX)?{27'h0,Rs1E}:csr_op2_EX_IN_F;
assign csr_op2_EX_IN_F=(csr_Forward[1])?(AluOutM):(csr_Forward[0]?RegWriteData:csr_op2_EX);
//wire values assignments
assign {Funct7D, Rs2D, Rs1D, Funct3D, RdD, OpCodeD} = Instr;
assign JalNPC=ImmD+PCD;
assign ForwardData1 = Forward1E[1]?(AluOutM):( Forward1E[0]?RegWriteData:RegOut1E );
assign csr_op_src1=csr_op_Forward[1]?(csr_Alu_outMEM):(csr_op_Forward[0]?csr_WB:csr_op1_EX);
assign Operand1 = csr_op_EX?(csr_op_src1):(AluSrc1E?PCE:ForwardData1);
assign ForwardData2 = Forward2E[1]?(AluOutM):( Forward2E[0]?RegWriteData:RegOut2E );
assign Operand2 = AluSrc2E[1]?(ImmE):( AluSrc2E[0]?Rs2E:ForwardData2 );
assign ResultM = LoadNpcM ? (PCM+4) : AluOutM;
assign RegWriteData = `MemToRegW?ResultW:DM_RD_Ext;
```

特别注意对通用寄存器中 0 号寄存器的写一律是无效的。

一般 ALU 的修改：

```
`CSR: AluOut = Operand1;
```

ExSegReg 增加以下相应信号：

```
//CSR
input wire op1_csr_ID,
output reg op1_csr_EX,
input wire [31:0] csr_Reg_1_ID,
input wire [31:0] csr_Reg_2_ID,
output reg [31:0] csr_Reg_1_EX,
output reg [31:0] csr_Reg_2_EX,
input wire [4:0] csr_dest_ID,
output reg [4:0]csr_dest_EX,
input wire csr_write_enID,
output reg csr_write_enEX,
input wire [1:0] csr_AluFun_ID,
output reg [1:0] csr_AluFun_EX,
input wire csr_imm_ID,
output reg csr_imm_EX,
input wire [4:0] csr_op_ID_1,
output reg [4:0] csr_op_EX_1
```

同理，MemSegReg 也增加相应信号：

```
//csr
input wire [31:0] csr_Alu_outEX,
output reg [31:0]   csr_Alu_outMEM,
input wire [4:0] csr_dest_EX,
output reg [4:0]  csr_dest_MEM,
input wire  csr_write_enEX,
output reg csr_write_enMEM
```

WBSegReg 增加：

```
//csr
input wire [31:0]   csr_Alu_outMEM,
output  reg [31:0] csr_WB,
input wire [4:0] csr_dest_MEM,
output reg [4:0]  csr_dest_WB,
input wire  csr_write_enMEM,
output reg csr_write_enWB
```

控制单元：

```
//CSR
output wire csr_read_ID,
output wire csr_write_ID,
output wire csr_op_ID, //1- csr to reg  0 not csr to reg
output reg [1:0] csr_AluFun,
output wire csr_imm_ID
);
assign csr_write_ID=(Op[6:0]==7'b1110011);
assign csr_read_ID=(Op[6:0]==7'b1110011);
assign csr_op_ID=(Op[6:0]==7'b1110011);
assign csr_imm_ID=(Op[6:0]==7'b1110011)&&(Fn3==3'b101||Fn3==3'b110||Fn3==3'b111);

always@(*)//csr_alu
begin
   if(Op[6:0]==7'b1110011)
   begin
           if(Fn3==3'b001||Fn3==3'b101)
           csr_AluFun=2'b01;//csrrw
           else if(Fn3==3'b010||Fn3==3'b110)
           csr_AluFun=2'b10;//csrrs
           else if(Fn3==3'b011||Fn3==3'b111)
           csr_AluFun=2'b11;//csrrc
           else
           csr_AluFun=2'b0;
   end
   else
   csr_AluFun=2'b0;
end
```

```
7'b1110011://CSR
   case(Fn3)
       3'b001: `ControlOut = {{2'b0_0},{1'b0,`LW},{4'b0000},{1'b0},{2'b00},{`NOBRANCH},{`CSR},{3'b0_00},{`RTYPE}};/*CSRRW*/
       3'b010: `ControlOut = {{2'b0_0},{1'b0,`LW},{4'b0000},{1'b0},{2'b00},{`NOBRANCH},{`CSR},{3'b0_00},{`RTYPE}};/*CSRRS*/
       3'b011: `ControlOut = {{2'b0_0},{1'b0,`LW},{4'b0000},{1'b0},{2'b00},{`NOBRANCH},{`CSR},{3'b0_00},{`RTYPE}};/*CSRRC*/
       3'b101: `ControlOut = {{2'b0_0},{1'b0,`LW},{4'b0000},{1'b0},{2'b00},{`NOBRANCH},{`CSR},{3'b0_00},{`RTYPE}};/*CSRRWI*/
       3'b110: `ControlOut = {{2'b0_0},{1'b0,`LW},{4'b0000},{1'b0},{2'b00},{`NOBRANCH},{`CSR},{3'b0_00},{`RTYPE}};/*CSRRSI*/
       3'b111: `ControlOut = {{2'b0_0},{1'b0,`LW},{4'b0000},{1'b0},{2'b00},{`NOBRANCH},{`CSR},{3'b0_00},{`RTYPE}};/*CSRRCI*/
   endcase
```

冒险单元：

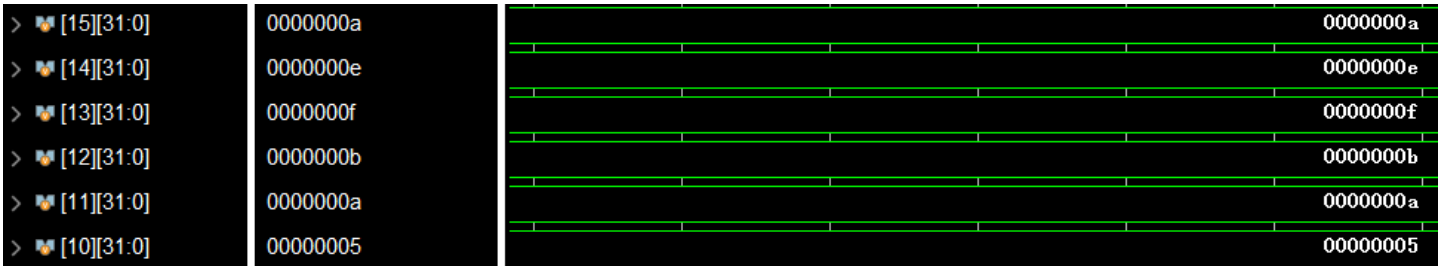
```
//csr
output reg [1:0] csr_Forward,
input wire [4:0] csr_op_11, csr_dest_MEM, csr_dest_WB,
input wire csr_write_MEM,csr_write_WB,
output reg [1:0]csr_op_Forward
```

```
//CSR
always@(*) begin
if(csr_op_11==csr_dest_MEM&&csr_write_MEM)
    csr_op_Forward<=2'b10;
else if(csr_op_11==csr_dest_WB&&csr_write_WB)
    csr_op_Forward<=2'b01;
else
    csr_op_Forward<=2'b00;
end
//csr
always@(*) begin
if(RdM!=5'b0&&RdM==Rs1E&&RegWriteM!=3'b0)
    csr_Forward<=2'b10;
else if(RdW!=5'b0&&RdW==Rs1E&&RegWriteW!=3'b0)
    csr_Forward<=2'b01;
else
    csr_Forward<=2'b00;
end
```

实验结果：

```
1   csrrwi  a0, cycle, 5
2   li      t0, 10
3   csrrw     a0, cycle, t0         #a0 = 5(00101)
4   csrrsi  a1, cycle, 0b0001     #a1 = 10(01010)
5   li      t0, 0b0100
6   csrrs   a2, cycle, t0         #a2 = 11(01011)
7   csrrci  a3, cycle, 0b0001     #a3 = 15(01111)
8   csrrc   a4, cycle, t0         #a4 = 14(01110)
9   csrrs     a5, cycle, zero       #a5 = 10(01010)
10  finish:
11      j finish
```

a0 到 a5 寄存器对应 10 号到 15 号通用寄存器，他们最终的值应该对应上图：



实验成功。

实验总结：

在有了助教提供的框架的前提下，实现所有的数据通路其实不困难。

因为 COD 中已经实现过流水线，所以第一阶段和第二阶段比较顺利。

困难主要集中在第三阶段 CSR 指令上，因为不熟悉 CSR，前期花了大量时间学习 CSR 指令以及构思数据通路。

改进意见：

希望在实验前对 CSR 进行讲解，并提供相应的测试代码。