本科实验报告

课程名称：计算机体系结构

设计名称：Pipelined CPU

姓　　名：曾帅王异鸣

学　　号：3190105729 3190102780

学　　院：计算机学院

专　　业：计算机科学与技术

班　　级：图灵 1901

指导老师：姜晓红

2021 年 10 月 16 日

目录

1 实验目的

本次实验要求我们实现支持中断与异常的 RV32IZicsr 流水线 CPU

1. 理解 CPU 中断异常的原理并掌握它的执行过程
2. 掌握设计 RV32I 流水线支持中断与异常的 CPU 的方法
3. 掌握验证 CPU 正确性的方式

2 实验内容

实验的基本要求是实现 RISC-V 架构下，指令集为 RV32I 并支持中断与异常的流水线 CPU。
本实验已给出主要框架，需要完成的实验内容如下：

1. 补充修改数据通路
2. 设计中断异常控制模块
3. 利用测试程序验证 CPU 并观察指令的执行情况

3 实验原理

3.1 CSR 指令

实现 RSIC-V 的中断与异常首先就需要能够对 CSR 进行读写操作，M 特权模式提供了如下六种指令进行操作。

31 25	24 20	19 15	14 12	11 7	6 0	
csr	rs1	001	rd	1110011		I csrrw
csr	rs1	010	rd	1110011		I csrrs
csr	rs1	011	rd	1110011		I csrrc
csr	zimm	101	rd	1110011		I csrrwi
csr	zimm	110	rd	1110011		I cssrrsi
csr	zimm	111	rd	1110011		I csrrci

图 3.1: CSR 指令

介绍完指令格式之后，CSR 每个指令的作用如下

操作名	使用方式	操作方式
csrrw	csrrw rd, csr, rs1	t = CSRs[csr]; CSRs[csr] = x[rs1]; x[rd] = t
csrrs	csrrs rd, csr, rs1	t = CSRs[csr]; CSRs[csr] = t | x[rs1]; x[rd] = t
csrrc	csrrc rd, csr, rs1	t = CSRs[csr]; CSRs[csr] = t & x[rs1]; x[rd] = t
csrrwi	csrrwi rd, csr, zimm[4:0]	x[rd] = CSRs[csr]; CSRs[csr] = zimm
csrrsi	csrrsi rd, csr, zimm[4:0]	t = CSRs[csr]; CSRs[csr] = t | zimm; x[rd] = t
csrrci	csrrci rd, csr, zimm[4:0]	t = CSRs[csr]; CSRs[csr] = t & zimm; x[rd] = t

3.2 CSR 寄存器

RISC-V 定义了一些控制和状态寄存器 (CSR), 用于配置或记录一些运行的状态，而 CSR 寄存器是处理器内核内部的寄存器，使用专有的 12 位地址编码空间，此次实验使用到了下述寄存器

Number	Privilege	Name	Description
Machine Information Registers			
0xF11	MRO	mvendorid	Vendor ID.
0xF12	MRO	marchid	Architecture ID.
0xF13	MRO	mimpid	Implementation ID.
0xF14	MRO	mhartid	Hardware thread ID.
Machine Trap Setup			
0x300	MRW	mstatus	Machine status register.
0x301	MRW	misa	ISA and extensions
0x302	MRW	medeleg	Machine exception delegation register.
0x303	MRW	mideleg	Machine interrupt delegation register.
0x304	MRW	mie	Machine interrupt-enable register.
0x305	MRW	mtvec	Machine trap-handler base address.
0x306	MRW	mcounteren	Machine counter enable.
Machine Trap Handling			
0x340	MRW	mscratch	Scratch register for machine trap handlers.
0x341	MRW	mepc	Machine exception program counter.
0x342	MRW	mcause	Machine trap cause.
0x343	MRW	mtval	Machine bad address or instruction.
0x344	MRW	mip	Machine interrupt pending.

图 3.2: CSR 寄存器

下面对各个控制与状态寄存器做具体的描述：

mtvec Riscv 处理器 trap 后跳入的 PC 地址由一个叫做机器模式异常入口基地址寄存器 mtvec 的 csr 寄存器指定。mtvec 是一个可读可写的寄存器，软件可以编程设定它的值。

xlen-1											2	1	0
Interrupt	BASE[xlen-1:2](WARL)												
xlen-2												2	

图 3.3: mtvec

1. 假设 mode 的值为 0，则所有的异常响应时处理器均跳转到 base 值指示的 pc 地址

2. 假设 mode 的值为 1，则狭义的异常发生时候，处理器均跳转到 base 值指示的 pc 地址。狭义的中断发生时候，处理器跳转到 base+4*cause 值指示的 pc 地址。cause 的值表示中断对应的异常编号 (exception code)。

mcause Riscv 架构规定，进入异常时候，机器模式异常原因寄存器 mcause 被同时更新，以反映当前的异常种类，软件可以通过读此寄存器查询造成异常的具体原因。

xlen-1	xlen-2												0
Interrupt	异常编码(WLRL)												
1	xlen-1												

图 3.4: mcause

异常编号域定义了中断和异常类型，如下表所示：

Interrupt / Exception mcause[XLEN-1]	Exception Code mcause[XLEN-2:0]	Description
1	1	Supervisor software interrupt
1	3	Machine software interrupt
1	5	Supervisor timer interrupt
1	7	Machine timer interrupt
1	9	Supervisor external interrupt
1	11	Machine external interrupt
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store address misaligned
0	7	Store access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	15	Store page fault

图 10.3：RISC-V 异常和中断的原因。中断时 mcause 的最高有效位置 1，同步异常时置 0，且低有效位标识了中断或异常的具体原因。只有在实现了监管者模式时才能处理监管者模式中断和页面错误异常

图 3.5: cause

mepc Riscv 架构定义异常的返回地址由机器模式异常 PC 寄存器 mepc 保存。在进入异常时候，硬件将自动更新 mepc 寄存器的值为当前遇到异常的指令 PC 值 (即当前程序的停止执行点)。该寄存器的值将作为异常的返回地址，在异常结束后，能够使用它保存的 pc 值返回之前停止执行的程序点。注意：mepc 虽然被自动更新，但它是可读可写的，软件可以直接读写该寄存器的值。

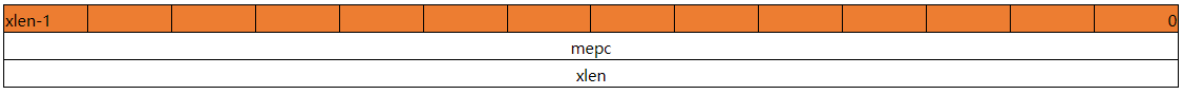

图 3.6: mepc

出现中断时候，中断返回地址 mepc 的值被更新为下一条尚未执行的指令。出现异常时候，中断返回地址 mepc 的值被更新为当前发生异常的指令 pc。注意：如果异常是有 ecall 和 ebreak 产生，由于 mepc 的值被更新为 ecall 或者 ebreak 指令自己的 PC。因此，在异常返回时候，如果直接使用 mepc 保存的 pc 值作为返回地址，则会再次进入异常，形成死循环。正确的做法是在异常处理程序中软件改变 mepc 指向下一条指令。

mtval Riscv 规定，在进入异常时候，硬件将自动更新机器模式异常值寄存器 mtval，以反映引起当前异常的存储器访问地址或者指令编码。

mtval	xlen-1												0
	mtval												
	xlen												

图 3.7: mtval

如果是由访问存储器造成的异常，比如硬件断点，取指令，存储器读写造成的异常，则将存储器访问的地址更新到 mtval。如果是由非法指令造成的异常，则将该指令的指令编码更新到 mtval 寄存器中。

mstatus mstatus 是机器模式下的状态寄存器。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SD	WPRI								TSR	TW	TVM	MXR	SUM	MPRV	XS[1]	XS[0]	FS[1]	FS[0]	MPP[1]	MPP[0]	WPRI		SPP	MPIE	WPRI	SPIE	UPIE	MIE	WPRI	SIE	UIE

图 3.8: mstatus

MIE 域表示全局中断使能。当该 MIE 域值为 1 时，表示所有中断的全局开关打开，当 MIE 域的值为 0 时候，表示全局关闭所有中断。MPIE 用于保存进入异常之前 MIE 域的值。MPP 用于保存进入异常之前特权模式的值。处理器进入异常时候：MPIE 域的值被更新为 MIE 的值。MIE 的值被更新为 0（意味着进入异常，中断被屏蔽)。MPP 的值被更新为异常发生前的模式（如果只实现机器模式，则 MPP 的值永远为 11)。

mie/mip Riscv 架构上定义的异常是不可屏蔽的，但狭义上的中断是可以屏蔽的，通过设置 mie 寄存器来屏蔽中断。mip 寄存器用于查询中断的等待状态，软件可以通过读 mip 寄存器达到查询中断状态的结果。

mip	xlen-1	11	10	9	8	7	6	5	4	3	2	1	0
	WPRI	MEIP	WPRI	SEIP	UEIP	MTIP	WPRI	STIP	UTIP	MSIP	WPRI	SSIP	USIP
	xlen-12	1	1	1	1	1	1	1	1	1	1	1	1
mie	xlen-1	11	10	9	8	7	6	5	4	3	2	1	0
	WPRI	MEIE	WPRI	SEIE	UEIE	MTIE	WPRI	STIE	UTIE	MSIE	WPRI	SSIE	USIE
	xlen-12	1	1	1	1	1	1	1	1	1	1	1	1

图 3.9: mip_mie

机器模式下外部中断的屏蔽由 csr 寄存器 mie 中 MEIE 域控制，等待标志 (pending) 则反映在 csr 寄存器 mip 的 MEIP 域。mip 和 mie 寄存器的高 20 位可以用于扩展其它的自定义中断类型。

3.3 处理过程

发生异常/中断时，硬件自动经历如下的状态转换：

1. 异常指令的 PC 被保存在 mepc 中，PC 被设置为 mtvec。mepc 指向导致异常的指令；对于中断，它指向中断处理后应该恢复执行的位置。
2. 根据异常来源设置 mcause，并将 mtval 设置为出错的地址或者其它适用于特定异常的信息字。
3. 把控制状态寄存器 mstatus 中的 MIE 位置零以禁用中断，并把先前的 MIE 值保留到 MPIE 中。
4. 发生异常之前的权限模式保留在 mstatus 的 MPP 域中，再把权限模式更改为 M。

从中断与异常处理返回时需要用到 mret 指令，处理过程如下：

1. PC 被设置为 mepc
2. 恢复 mstatus 的 MIE 位
3. 恢复保存在 mstatus 的 MPP 域中的权限模式

3.4 精确 Exception

在本次实验中，我们实现的是精确异常。也即异常所有之前的指令必须回收，异常之后的指令之后的指令一律不得回收，即不可以更新体系结构状态。为实现这一要求，我们做了如下设计：

1. 所以异常的跳转均需要在 WB 阶段进行，也就是说前几个阶段侦测到的异常不会立刻进入异常处理，而是要流入 WB 后再同一处理
2. 如果多阶段产生了异常，处理最早产生的异常
3. 使用异常向量来完成这一操作：如果异常被检测到异常信号将会被加入异常向量并且取消所有修改系统状态的写信号

根据上述原理，我组设计的中断异常模块如下：

```
1 `timescale 1ns / 1ps
2
3 module ExceptionUnit(
4     input clk, rst,
```

```
5       input csr_rw_in,
6       input[1:0] csr_wsc_mode_in,
7       input csr_w_imm_mux,
8       input[11:0] csr_rw_addr_in,
9       input[31:0] csr_w_data_reg,
10      input[4:0] csr_w_data_imm,
11      input[31:0] fault_addr,
12      input isZero,
13      output[31:0] csr_r_data_out,
14
15      input interrupt,
16      input illegal_inst,
17      input l_access_fault,
18      input s_access_fault,
19      input ecall_m,
20
21      input mret,
22
23      input[31:0] epc_cur,
24      input[31:0] epc_next,
25      output[31:0] PC_redirect,
26      output redirect_mux,
27
28      output reg_FD_flush, reg_DE_flush, reg_EM_flush, reg_MW_flush,
29      output RegWrite_cancel
30  );
31
32      localparam mstatus = 4'd0;
33      localparam mie     = 4'd4;
34      localparam mtvec   = 4'd5;
35      localparam mepc    = 4'd9;
36      localparam mcause  = 4'd10;
37      localparam mtval   = 4'd11;
38      localparam mip     = 4'd12;
39      localparam csrrw   = 3'b001;
40      localparam csrrs   = 3'b010;
41      localparam csrrc   = 3'b011;
42      localparam csrrwi  = 3'b101;
43      localparam csrrsi  = 3'b110;
44      localparam csrrci  = 3'b111;
45
46      wire Inter_Exp = illegal_inst | l_access_fault | s_access_fault |
```

```
        ecall_m;
47      /*将CSR模块提出*/
48      reg[31:0] CSR [0:15];
49      reg[1:0] mp = 2'b11;
50      wire [11:0] raddr, waddr;
51      wire [31:0] wdata;
52      wire [1:0] csr_wsc_mode;
53      wire csr_w;
54
55      assign raddr = csr_rw_addr_in;
56      assign waddr = csr_rw_addr_in;
57      assign csr_w = csr_rw_in;
58      assign wdata = csr_w_imm_mux ? {27'b0, csr_w_data_imm} :
            csr_w_data_reg;
59      assign csr_wsc_mode = csr_wsc_mode_in;
60
61      // Address mapping. The address is 12 bits, but only 4 bits are used
            in this module.
62      wire raddr_valid = raddr[11:7] == 5'h6 && raddr[5:3] == 3'h0;
63      wire[3:0] raddr_map = (raddr[6] << 3) + raddr[2:0];
64      wire waddr_valid = waddr[11:7] == 5'h6 && waddr[5:3] == 3'h0;
65      wire[3:0] waddr_map = (waddr[6] << 3) + waddr[2:0];
66
67      always@(posedge clk or posedge rst) begin
68          if(rst) begin
69              CSR[0] <= 32'h88;
70              CSR[1] <= 0;
71              CSR[2] <= 0;
72              CSR[3] <= 0;
73              CSR[4] <= 32'hfff;
74              CSR[5] <= 0;
75              CSR[6] <= 0;
76              CSR[7] <= 0;
77              CSR[8] <= 0;
78              CSR[9] <= 0;
79              CSR[10] <= 0;
80              CSR[11] <= 0;
81              CSR[12] <= 0;
82              CSR[13] <= 0;
83              CSR[14] <= 0;
84              CSR[15] <= 0;
85
```

```
86          end
87          else if(csr_w) begin
88              case(csr_wsc_mode)
89                  2'b01: CSR[waddr_map] = wdata;
90                  2'b10: CSR[waddr_map] = CSR[waddr_map] | wdata;
91                  2'b11: CSR[waddr_map] = CSR[waddr_map] & ~wdata;
92                  default: CSR[waddr_map] = wdata;
93              endcase
94
95          end
96          else if(mret) begin
97              CSR[mstatus][3] <= CSR[mstatus][7];
98              mp <= CSR[mstatus][12:11];
99          end
100         else if(interrupt & CSR[mstatus][3] & CSR[mie][11] & ~isZero)
                begin
101             CSR[mepc]       <= epc_next;
102             CSR[mcause]     <= {1'b1, 31'd11};
103             //CSR[mtval]在中断时无需修改
104             CSR[mstatus][7] <= CSR[mstatus][3];
105             CSR[mstatus][3] <= 1'b0;
106             //权限模式变化 mstatus.mpp<=mp;  mp<=11
107             CSR[mstatus][12:11] <= mp;
108             mp <= 2'b11;
109         end
110         else if(Inter_Exp & CSR[mstatus][3]) begin
111             CSR[mepc]       <= epc_cur;
112             if(illegal_inst) begin
113                 CSR[mcause] <= {1'b0, 31'd2};
114             end
115             else if(ecall_m) begin
116                 CSR[mcause] <= {1'b0, 31'd11};
117             end
118             else if(l_access_fault) begin
119                 CSR[mcause] <= {1'b0, 31'd4};
120             end
121             else if(s_access_fault) begin
122                 CSR[mcause] <= {1'b0, 31'd6};
123             end
124
125             if(l_access_fault | s_access_fault) begin
126                 CSR[mtval]  <= fault_addr;
```

```
127             end
128             else if(illegal_inst) begin
129                 CSR[mtval] <= epc_cur;
130             end
131             else begin
132                 CSR[mtval]  <= 0;
133             end
134             CSR[mstatus][7] <= CSR[mstatus][3];
135             CSR[mstatus][3] <= 1'b0;
136             //权限模式变化 mstatus.mpp<-mp;  mp<-11
137             CSR[mstatus][12:11] <= mp;
138             mp <= 2'b11;
139         end
140     end
141
142     assign redirect_mux = mret | (((interrupt & ~isZero) | Inter_Exp) &
            CSR[mstatus][3]);
143     assign reg_FD_flush = mret | (((interrupt & ~isZero) | Inter_Exp) &
            CSR[mstatus][3]);
144     assign reg_DE_flush = mret | (((interrupt & ~isZero) | Inter_Exp) &
            CSR[mstatus][3]);
145     assign reg_EM_flush = mret | (((interrupt & ~isZero) | Inter_Exp) &
            CSR[mstatus][3]);
146     assign reg_MW_flush = mret | (((interrupt & ~isZero) | Inter_Exp) &
            CSR[mstatus][3]);
147     assign RegWrite_cancel = mret | (((interrupt & ~isZero) | Inter_Exp) &
             CSR[mstatus][3]);
148     assign PC_redirect  = rst                                   ? 32'h0 :
149                           mret                                    ? CSR[mepc] :
150                           ((interrupt & ~isZero) & CSR[mstatus][3]) ?
                                CSR[mtvec] :
151                           (Inter_Exp & CSR[mstatus][3]) ? CSR[mtvec] :
152                           CSR[raddr_map];
153     assign csr_r_data_out = CSR[raddr_map];
154 endmodule
```

3.5 数据通路

完成了个部件的设计，数据通路就是将各个部分连接起来，保证各功能正常实现的关键部分，下面是总体的原理图

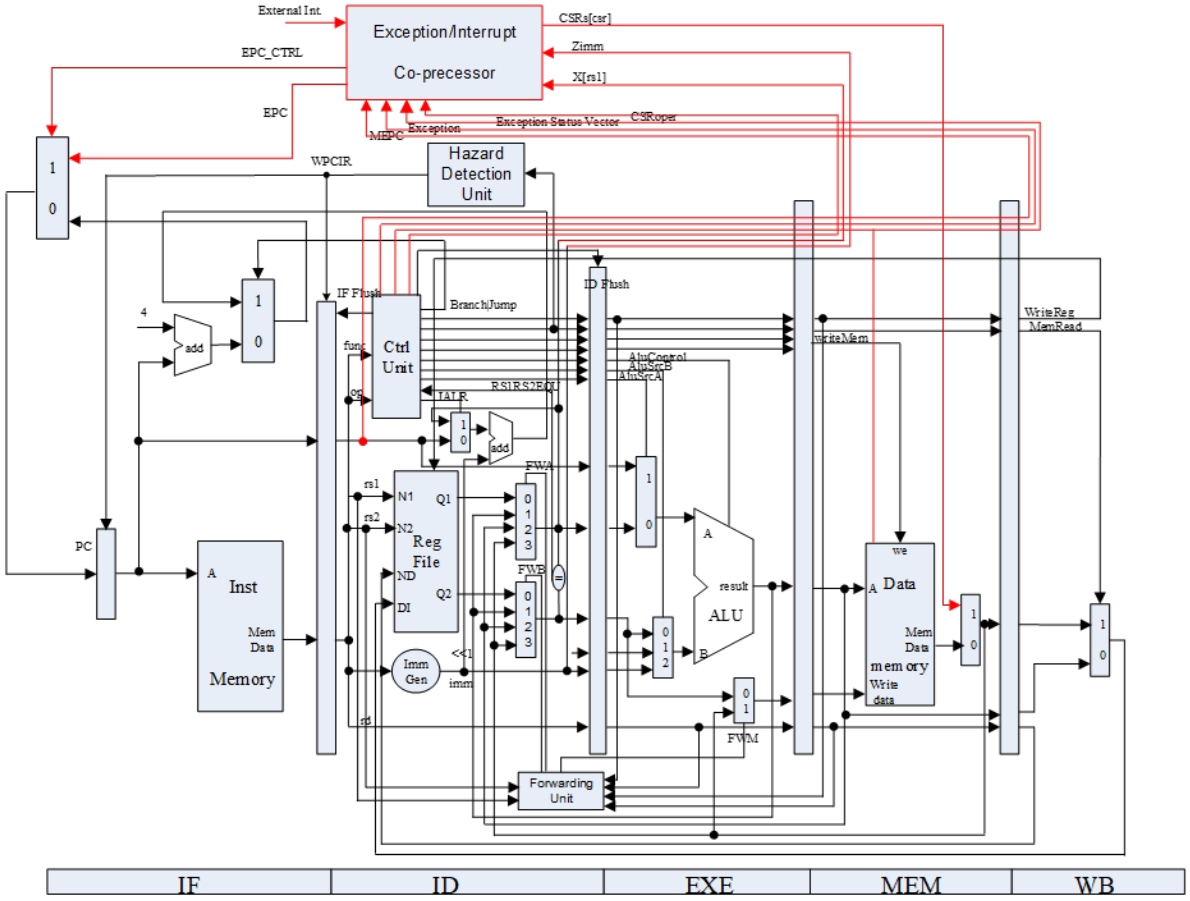

图 3.10: DataPath

数据通路的补充就是根据原理图连线即可，具体的源代码如下

```
1  `timescale 1ns / 1ps
2
3
4  module  RV32core(
5          input debug_en,  // debug enable
6          input debug_step,  // debug step clock
7          input [6:0] debug_addr,  // debug address
8          output[31:0] debug_data,  // debug data
9          input clk,  // main clock
10         input rst,  // synchronous reset
11         input interrupter  // interrupt source, for future use
12     );
13
14     wire debug_clk;
15
16     debug_clk clock(.clk(clk),.debug_en(debug_en),.debug_step(debug_step)
```

```
        ,.debug_clk(debug_clk));
17
18      wire Branch_ctrl, JALR, RegWrite_ctrl, mem_w_ctrl, mem_r_ctrl,
19          ALUSrc_A_ctrl, ALUSrc_B_ctrl, DatatoReg_ctrl, rs1use_ctrl,
                rs2use_ctrl,
20          MRET, csr_rw_ctrl, csr_w_imm_mux_ctrl;
21      wire[1:0] hazard_optype_ctrl, exp_vector_ctrl;
22      wire[2:0] ImmSel_ctrl, cmp_ctrl;
23      wire[3:0] ALUControl_ctrl;
24
25      wire forward_ctrl_ls;
26      wire[1:0] forward_ctrl_A, forward_ctrl_B;
27
28      wire PC_EN_IF;
29      wire [31:0] PC_IF, next_PC_IF, PC_4_IF, inst_IF, final_PC_IF;
30
31      wire reg_FD_stall, reg_FD_flush, isFlushed_ID, cmp_res_ID;
32      wire [31:0] jump_PC_ID, PC_ID, inst_ID, Debug_regs, rs1_data_reg,
            rs2_data_reg,
33          Imm_out_ID, rs1_data_ID, rs2_data_ID, addA_ID;
34
35      wire reg_DE_flush, isFlushed_EXE, RegWrite_EXE, mem_w_EXE, mem_r_EXE,
36          ALUSrc_A_EXE, ALUSrc_B_EXE, ALUzero_EXE, ALUoverflow_EXE,
                DatatoReg_EXE,
37          csr_rw_EXE, csr_w_imm_mux_EXE, mret_EXE;
38      wire[1:0] exp_vector_EXE;
39      wire[2:0] u_b_h_w_EXE;
40      wire[3:0] ALUControl_EXE;
41      wire[4:0] rs1_EXE, rs2_EXE, rd_EXE;
42      wire[31:0] ALUout_EXE, PC_EXE, inst_EXE, rs1_data_EXE, rs2_data_EXE,
            Imm_EXE,
43          ALUA_EXE, ALUB_EXE, Dataout_EXE;
44
45      wire redirect_mux_exp, reg_FD_flush_exp, reg_DE_flush_exp,
46          reg_EM_flush_exp, reg_MW_flush_exp, RegWrite_cancel_exp;
47      wire[31:0] PC_redirect_exp;
48
49      wire isFlushed_MEM, RegWrite_MEM, DatatoReg_MEM, mem_w_MEM, mem_r_MEM,
50          csr_rw_MEM, csr_w_imm_mux_MEM, mret_MEM, l_access_fault_MEM,
                s_access_fault_MEM;
51      wire[1:0] exp_vector_MEM;
52      wire[2:0] u_b_h_w_MEM;
```

```
53      wire[4:0] rs1_MEM, rd_MEM;
54      wire[31:0] ALUout_MEM, PC_MEM, inst_MEM, Dataout_MEM, Datain_MEM,
55          rs1_data_MEM, CSRout_MEM, RAMout_MEM;
56
57
58      wire isFlushed_WB, RegWrite_WB, DatatoReg_WB;
59      wire[3:0] exp_vector_WB;
60      wire[4:0] rd_WB;
61      wire [31:0] wt_data_WB, PC_WB, inst_WB, ALUout_WB, Datain_WB;
62
63
64      // IF
65      REG32 REG_PC(.clk(debug_clk),.rst(rst),.CE(PC_EN_IF),.D(final_PC_IF),.
            Q(PC_IF));
66
67      add_32 add_IF(.a(PC_IF),.b(32'd4),.c(PC_4_IF));
68
69      MUX2T1_32 mux_IF(.I0(PC_4_IF),.I1(jump_PC_ID),.s(Branch_ctrl),.o(
            next_PC_IF));
70
71      MUX2T1_32 redirectPC(.I0(next_PC_IF),.I1(PC_redirect_exp),.s(
            redirect_mux_exp),.o(final_PC_IF));
72
73      ROM_D inst_rom(.a(PC_IF[8:2]),.spo(inst_IF));
74
75
76      // ID
77      REG_IF_ID reg_IF_ID(.clk(debug_clk),.rst(rst),.EN(1'b1),.Data_stall(
            reg_FD_stall),
78          .flush(reg_FD_flush | reg_FD_flush_exp),.PCOUT(PC_IF),.IR(inst_IF)
                ,
79
80          .IR_ID(inst_ID),.PCurrent_ID(PC_ID),.isFlushed(isFlushed_ID));
81
82      CtrlUnit ctrl(.inst(inst_ID),.cmp_res(cmp_res_ID),.Branch(Branch_ctrl)
            ,.ALUSrc_A(ALUSrc_A_ctrl),
83          .ALUSrc_B(ALUSrc_B_ctrl),.DatatoReg(DatatoReg_ctrl),.RegWrite(
                RegWrite_ctrl),
84          .mem_w(mem_w_ctrl),.mem_r(mem_r_ctrl),.rs1use(rs1use_ctrl),.rs2use
                (rs2use_ctrl),
85          .hazard_optype(hazard_optype_ctrl),.ImmSel(ImmSel_ctrl),.cmp_ctrl(
                cmp_ctrl),
```

```
86          .ALUControl(ALUControl_ctrl),.JALR(JALR),.MRET(MRET),.csr_rw(
                csr_rw_ctrl),
87          .csr_w_imm_mux(csr_w_imm_mux_ctrl),.exp_vector(exp_vector_ctrl));
88
89      Regs register(.clk(debug_clk),.rst(rst),.L_S(RegWrite_WB & ~
            RegWrite_cancel_exp),
90          .R_addr_A(inst_ID[19:15]),.R_addr_B(inst_ID[24:20]),
91          .rdata_A(rs1_data_reg),.rdata_B(rs2_data_reg),
92          .Wt_addr(rd_WB),.Wt_data(wt_data_WB),
93          .Debug_addr(debug_addr[4:0]),.Debug_regs(Debug_regs));
94
95      ImmGen imm_gen(.ImmSel(ImmSel_ctrl),.inst_field(inst_ID),.Imm_out(
            Imm_out_ID));
96
97      MUX4T1_32 mux_forward_A(.I0(rs1_data_reg),.I1(ALUout_EXE),.I2(
            ALUout_MEM),.I3(Datain_MEM),
98          .s(forward_ctrl_A),.o(rs1_data_ID));
99
100     MUX4T1_32 mux_forward_B(.I0(rs2_data_reg),.I1(ALUout_EXE),.I2(
            ALUout_MEM),.I3(Datain_MEM),
101         .s(forward_ctrl_B),.o(rs2_data_ID));
102
103     MUX2T1_32 mux_branch_ID(.I0(PC_ID),.I1(rs1_data_ID),.s(JALR),.o(
            addA_ID));
104
105     add_32 add_branch_ID(.a(addA_ID),.b(Imm_out_ID),.c(jump_PC_ID));
106
107     cmp_32 cmp_ID(.a(rs1_data_ID),.b(rs2_data_ID),.ctrl(cmp_ctrl),.c(
            cmp_res_ID));
108
109     HazardDetectionUnit hazard_unit(.clk(debug_clk),.Branch_ID(Branch_ctrl
            ),.rs1use_ID(rs1use_ctrl),
110         .rs2use_ID(rs2use_ctrl),.hazard_optype_ID(hazard_optype_ctrl),.
                rd_EXE(rd_EXE),
111         .rd_MEM(rd_MEM),.rs1_ID(inst_ID[19:15]),.rs2_ID(inst_ID[24:20]),.
                rs2_EXE(rs2_EXE),
112         .PC_EN_IF(PC_EN_IF),.reg_FD_stall(reg_FD_stall),.reg_FD_flush(
                reg_FD_flush),
113         .reg_DE_flush(reg_DE_flush),.forward_ctrl_ls(forward_ctrl_ls),.
                forward_ctrl_A(forward_ctrl_A),
114         .forward_ctrl_B(forward_ctrl_B));
115
```

```
116
117     // EX
118     REG_ID_EX reg_ID_EX(.clk(debug_clk),.rst(rst),.EN(1'b1),
119         .flush(reg_DE_flush | reg_DE_flush_exp | isFlushed_ID),
120         .IR_ID(inst_ID),.PCurrent_ID(PC_ID),.rs1_addr(inst_ID[19:15]),.
                rs2_addr(inst_ID[24:20]),
121         .rs1_data(rs1_data_ID),.rs2_data(rs2_data_ID),.Imm32(Imm_out_ID),.
                rd_addr(inst_ID[11:7]),
122         .ALUSrc_A(ALUSrc_A_ctrl),.ALUSrc_B(ALUSrc_B_ctrl),.ALUC(
                ALUControl_ctrl),.DatatoReg(DatatoReg_ctrl),
123         .RegWrite(RegWrite_ctrl),.WR(mem_w_ctrl),.u_b_h_w(inst_ID[14:12])
                ,.mem_r(mem_r_ctrl),
124         .csr_rw(csr_rw_ctrl),.csr_w_imm_mux(csr_w_imm_mux_ctrl),.mret(MRET
                ),
125         .exp_vector(exp_vector_ctrl),
126
127         .PCurrent_EX(PC_EXE),.IR_EX(inst_EXE),.rs1_EX(rs1_EXE),.rs2_EX(
                rs2_EXE),
128         .A_EX(rs1_data_EXE),.B_EX(rs2_data_EXE),.Imm32_EX(Imm_EXE),.rd_EX(
                rd_EXE),
129         .ALUSrc_A_EX(ALUSrc_A_EXE),.ALUSrc_B_EX(ALUSrc_B_EXE),.ALUC_EX(
                ALUControl_EXE),
130         .DatatoReg_EX(DatatoReg_EXE),.RegWrite_EX(RegWrite_EXE),.WR_EX(
                mem_w_EXE),
131         .u_b_h_w_EX(u_b_h_w_EXE),.mem_r_EX(mem_r_EXE),.isFlushed(
                isFlushed_EXE),
132         .csr_rw_EX(csr_rw_EXE),.csr_w_imm_mux_EX(csr_w_imm_mux_EXE),.
                mret_EX(mret_EXE),
133         .exp_vector_EX(exp_vector_EXE));
134
135     MUX2T1_32 mux_A_EXE(.I0(rs1_data_EXE),.I1(PC_EXE),.s(ALUSrc_A_EXE),.o(
            ALUA_EXE));
136
137     MUX2T1_32 mux_B_EXE(.I0(rs2_data_EXE),.I1(Imm_EXE),.s(ALUSrc_B_EXE),.o
            (ALUB_EXE));
138
139     ALU alu(.A(ALUA_EXE),.B(ALUB_EXE),.Control(ALUControl_EXE),
140         .res(ALUout_EXE),.zero(ALUzero_EXE),.overflow(ALUoverflow_EXE));
141
142     MUX2T1_32 mux_forward_EXE(.I0(rs2_data_EXE),.I1(Datain_MEM),.s(
            forward_ctrl_ls),.o(Dataout_EXE));
143
```

```
144
145     // MEM
146     REG_EX_MEM reg_EXE_MEM(.clk(debug_clk),.rst(rst),.EN(1'b1),.flush(
            reg_EM_flush_exp | isFlushed_EXE),
147         .IR_EX(inst_EXE),.PCurrent_EX(PC_EXE),.ALUO_EX(ALUout_EXE),.B_EX(
                Dataout_EXE),
148         .rd_EX(rd_EXE),.rs1_EX(rs1_EXE),.rs1_data_EX(rs1_data_EXE),.
                DatatoReg_EX(DatatoReg_EXE),
149         .RegWrite_EX(RegWrite_EXE),.WR_EX(mem_w_EXE),.u_b_h_w_EX(
                u_b_h_w_EXE),.mem_r_EX(mem_r_EXE),
150         .csr_rw_EX(csr_rw_EXE),.csr_w_imm_mux_EX(csr_w_imm_mux_EXE),.
                mret_EX(mret_EXE),
151         .exp_vector_EX(exp_vector_EXE),
152
153         .PCurrent_MEM(PC_MEM),.IR_MEM(inst_MEM),.ALUO_MEM(ALUout_MEM),.
                Datao_MEM(Dataout_MEM),
154         .rd_MEM(rd_MEM),.rs1_MEM(rs1_MEM),.rs1_data_MEM(rs1_data_MEM),.
                DatatoReg_MEM(DatatoReg_MEM),
155         .RegWrite_MEM(RegWrite_MEM),.WR_MEM(mem_w_MEM),.u_b_h_w_MEM(
                u_b_h_w_MEM),.mem_r_MEM(mem_r_MEM),
156         .isFlushed(isFlushed_MEM),.csr_rw_MEM(csr_rw_MEM),.
                csr_w_imm_mux_MEM(csr_w_imm_mux_MEM),
157         .mret_MEM(mret_MEM),.exp_vector_MEM(exp_vector_MEM));
158
159     RAM_B data_ram(.addra(ALUout_MEM),.clka(debug_clk),.dina(Dataout_MEM),
160         .wea(mem_w_MEM),.rea(mem_r_MEM),.douta(RAMout_MEM),.mem_u_b_h_w(
                u_b_h_w_MEM),
161         .l_access_fault(l_access_fault_MEM),.s_access_fault(
                s_access_fault_MEM));
162
163     ExceptionUnit exp_unit(.clk(debug_clk),.rst(rst),.csr_rw_in(csr_rw_MEM
            ),.csr_wsc_mode_in(inst_MEM[13:12]),
164         .csr_w_imm_mux(csr_w_imm_mux_MEM),.csr_rw_addr_in(inst_MEM[31:20])
                ,
165         .csr_w_data_reg(rs1_data_MEM),.csr_w_data_imm(rs1_MEM),
166         .csr_r_data_out(CSRout_MEM),
167
168         .interrupt(interrupter),
169         .illegal_inst(~isFlushed_WB & exp_vector_WB[3]),
170         .ecall_m(~isFlushed_WB & exp_vector_WB[2]),
171         .l_access_fault(~isFlushed_WB & exp_vector_WB[1]),
172         .s_access_fault(~isFlushed_WB & exp_vector_WB[0]),
```

```
173             .mret(mret_MEM),
174             /*自行添加*/
175             .fault_addr(ALUout_WB),
176             .isZero(inst_WB ? 0 : 1),
177             /*自行添加*/
178             .epc_cur(PC_WB),
179             .epc_next(~isFlushed_MEM ? PC_MEM : ~isFlushed_EXE ? PC_EXE :
180             ~isFlushed_ID ? PC_ID : PC_IF),
181             .PC_redirect(PC_redirect_exp),.redirect_mux(redirect_mux_exp),
182             .reg_FD_flush(reg_FD_flush_exp),.reg_DE_flush(reg_DE_flush_exp),
183             .reg_EM_flush(reg_EM_flush_exp),.reg_MW_flush(reg_MW_flush_exp),
184             .RegWrite_cancel(RegWrite_cancel_exp));
185
186         MUX2T1_32 mux_csrout(.I0(RAMout_MEM),.I1(CSRout_MEM),.s(csr_rw_MEM),.o
                (Datain_MEM));
187
188
189         // WB
190         REG_MEM_WB reg_MEM_WB(.clk(debug_clk),.rst(rst),.EN(1'b1),.flush(
                reg_MW_flush_exp | isFlushed_MEM),
191             .IR_MEM(inst_MEM),.PCurrent_MEM(PC_MEM),.ALUO_MEM(ALUout_MEM),.
                    Datai(Datain_MEM),
192             .rd_MEM(rd_MEM),.DatatoReg_MEM(DatatoReg_MEM),.RegWrite_MEM(
                    RegWrite_MEM),
193             .exp_vector_MEM({exp_vector_MEM, l_access_fault_MEM,
                    s_access_fault_MEM}),
194
195             .PCurrent_WB(PC_WB),.IR_WB(inst_WB),.ALUO_WB(ALUout_WB),.MDR_WB(
                    Datain_WB),
196             .rd_WB(rd_WB),.DatatoReg_WB(DatatoReg_WB),.RegWrite_WB(RegWrite_WB
                    ),
197             .isFlushed(isFlushed_WB),.exp_vector_WB(exp_vector_WB));
198
199         MUX2T1_32 mux_WB(.I0(ALUout_WB),.I1(Datain_WB),.s(DatatoReg_WB),.o(
                wt_data_WB));
200
201
202         wire [31:0] Test_signal;
203         assign debug_data = debug_addr[5] ? Test_signal : Debug_regs;
204
205         CPUTEST     U1_3(.PC_IF(PC_IF),
206                          .PC_ID(PC_ID),
```

```
207                 .PC_EXE(PC_EXE),
208                 .PC_MEM(PC_MEM),
209                 .PC_WB(PC_WB),
210                 .PC_next_IF(next_PC_IF),
211                 .PCJump(jump_PC_ID),
212                 .inst_IF(inst_IF),
213                 .inst_ID(inst_ID),
214                 .inst_EXE(inst_EXE),
215                 .inst_MEM(inst_MEM),
216                 .inst_WB(inst_WB),
217                 .PCEN(PC_EN_IF),
218                 .Branch(Branch_ctrl),
219                 .PCSource(Branch_ctrl),
220                 .RS1DATA(rs1_data_reg),
221                 .RS2DATA(rs2_data_reg),
222                 .Imm32(Imm_out_ID),
223                 .ImmSel(ImmSel_ctrl),
224                 .ALUC(ALUControl_ctrl),
225                 .ALUSrc_A(ALUSrc_A_ctrl),
226                 .ALUSrc_B(ALUSrc_B_ctrl),
227                 .A(ALUA_EXE),
228                 .B(ALUB_EXE),
229                 .ALU_out(ALUout_MEM),
230                 .Datai(Datain_MEM),
231                 .Datao(Dataout_MEM),
232                 .Addr(Addr),
233                 .WR(MWR),
234                 .MIO(mem_r_MEM),
235                 .WDATA(wt_data_WB),
236                 .DatatoReg(DatatoReg_WB),
237                 .RegWrite(RegWrite_WB),
238                 .data_hazard(reg_FD_stall),
239                 .control_hazard(Branch_ctrl),
240                 .exp_sig({csr_rw_MEM,2'b0,csr_w_imm_mux_MEM,
241                     isFlushed_ID, isFlushed_EXE, isFlushed_MEM,
                            isFlushed_WB,
242                     interrupter,3'b0,exp_vector_WB,
243                     4'b0,
244                     3'b0,redirect_mux_exp,
245                     reg_FD_flush_exp,reg_DE_flush_exp,reg_EM_flush_exp
                            ,reg_MW_flush_exp,
246                     3'b0,RegWrite_cancel_exp}),
```

```
247
248                         .Debug_addr(debug_addr[4:0]),
249                         .Test_signal(Test_signal)
250                         );
251
252 endmodule
```

4 实验步骤与调试

4.1 仿真

根据已经写好的代码，进行仿真模拟

4.2 综合

选择左侧面板的 Run Synthesis 或者点击上方的绿色小三角，选择 Synthesis

4.3 实现

选择左侧面板的 Run Implementation 或者点击上方的绿色小三角，选择 Implementation。值得注意的是执行 implementation 之前应该确保引脚约束存在且正确，同时之前已经综合过最新的代码。

4.4 验证设计

选择左侧面板的 Open Elaborated Design，输出的结果如下，根据原理图来判断，基本没有问题

4.5 生成二进制文件

选择左侧面板的 Generate Bitstream 或者点击上上的绿色二进制标志。同时生成 Bitstream 前要确保: 之前已经综合、实现过最新的代码。如没有，直接运行会默认从综合、实现开始。此过程还要注意生成的 bit 文件默认存放在.runs 下相应的 implementation 文件夹中

4.6 烧写上板

点击左侧的 Open Hardware Manager → 点击 Open Target → Auto Connect → 点击 Program Device → 选择 bistream 路径，烧写。验证结果见实验结果部分。

5 实验结果与分析

6 讨论与心得

本实验是在上学期的最后一个实验的基础上进行了功能的添加与改进。由于在本次实验中，我们已经有了框架代码，因此我们需要阅读框架代码并了解其主要意思，并在此框架的基础上完成我们最终的代码，这要求我们有一定的代码阅读与理解能力。另外，我们还需要尽快捡起上学期已经学过的知识，打好之后学习和实验的基础。