浙江大学

本科实验报告

课程名称：	计算机组成与设计
姓　　名：	熊子宇
学　　号：	3200105278
学　　院：	竺可桢学院
专　　业：	混合班
指导教师：	姜晓红/叶大源
报告日期：	2022 年 5 月 4 日

实验四 Single Cycle CPU

一、实验目的

1. 设计 IMem 和 DMem
2. 实现 ALU-RR, ALU-RI 指令并仿真验证
3. 实现 LW, SW 指令，LED 读写并仿真验证
4. 实现 branch 指令并仿真验证
5. 实现基本 SCPU 的其他指令(jal, jalr, lui, auipc)并仿真验证
6. 基本 SCPU 指令上板验证
7. 设计 CSR 部件，实现 CSRRX 系列指令并仿真验证
8. 实现 ecall 和 mret 指令并仿真验证
9. 实现非法指令、PC, lw, sw 地址非对齐异常并仿真验证
10. 实现 timer 和 external 中断并仿真验证
11. 对中断和异常部分指令上板验证

二、实验方法与步骤

1 设计 IMem 和 DMem

助教在 Framework 中给出了 IMem 和 DMem，但是我没有注意到，因此走了一些弯路。我仿照 Lab3 中的 data_providor.v，利用 verilog 的语法糖写出了 IMem 和 DMem。

```
(* ram_style = "block" *) reg [31:0] imem[0:4095];
initial $readmemh("imem_data.mem", imem);
assign data = imem[addr >> 2];
```

这里需要注意的点有两个：

问题一：IMem 和 DMem 的单位访问长度。

IMem 和 DMem 定义为 32 位的寄存器组，实质上是 word-addressed，与理论课中的 byte-addressed 的要求不符合。在 IMem 中，送入了 pc（也即 imem_addr）后，需要将 pc 右移两位，才能得到 word 的地址；在 Dmem 中，送入 dmem_addr 后也是如此。这种设计有利有弊。利在于方便实现 lw, sw 指令，弊在于不易于实现 lb, lh 等指令。因为实验没有要求实现 lb 指令，因此还是选择采用 32 位的寄存器。

问题二：寄存器部件，读口是组合逻辑（使用 assign 语句）的，而写口是时序逻辑（使用 always @(posedge clk)语句）的。

在写 DMem.v 时，一开始代码如下：

```
always @(posedge clk) begin
    if (wen) dmem[addr] <= i_data;
    else if (ren) o_data <= dmem[addr];//不能放在里面，否则 reg 读不到值
end
```

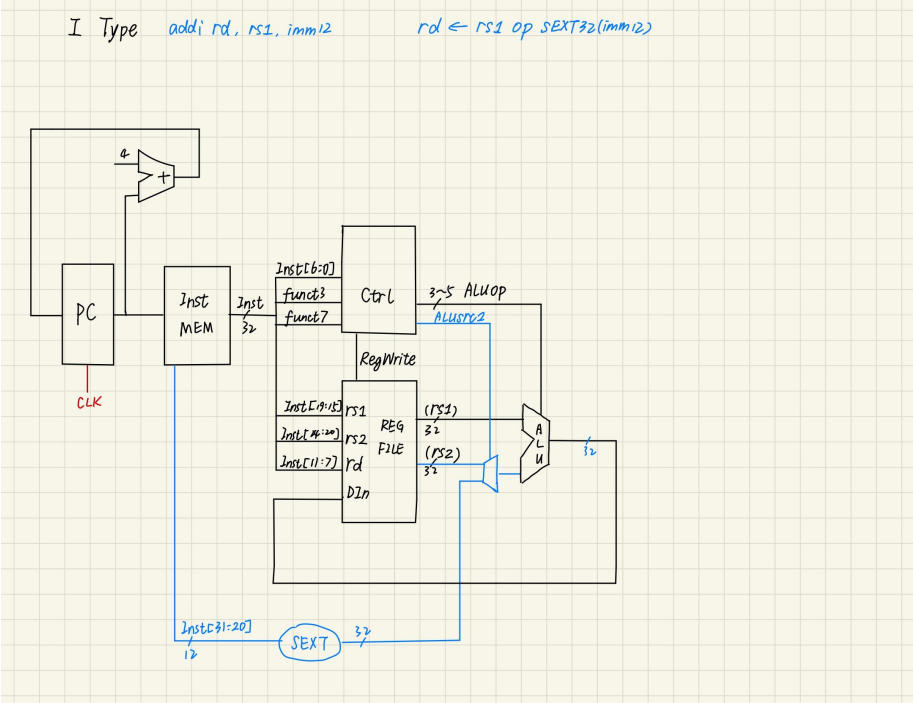
后来注意到在实例化 DMem 和 RegFile 时，都使用了.clk(~clk_cpu)。这样 o_data 只会在时钟负边沿更新。但 RegFile 的写口也是在时钟负边沿写入 data_in。这样一来，data_in 取的是 clk = 1 时的值，而不是从 DMem 最新读到的值。因此 lw 操作怎么调试都是失败的。

正确的写法应该如下。也即 DMem 中的 clk 作用仅是在时钟负边沿时将 i_data 写入 DMem。

```
always @(posedge clk) begin
    if (wen) dmem[addr] <= i_data;
end
assign o_data = ren ? dmem[addr] : 32'hZZZZZZZZ;
```

2 实现 ALU-RR, ALU-RI 指令并仿真验证

2.1 数据通路



如图是 ALU-RR 和 ALU-RI 指令的数据通路。可以看到图中的单元中，IMem, RegFile, ALU 已经设计完毕，差 PC，控制信号单元 Controller 和立即数生成器 ImmGen（为了其他指令方便，在此处先设计好）。

2.2 功能实现

在 Core.v 中，PC 设计如下：

```
assign imem_addr = pc;
reg [31:0] pc;
always @(posedge clk or posedge rst) begin
    if (rst) begin
```

```
            pc <= 0;
        end
        else begin
            pc <= pc + 4;
        end
    end
```

新建 Controller.v，控制信号如下：

```
assign is_aluRR = (opcode == `OPCODE_ALURR) ? 1 : 0;
assign is_aluRI = (opcode == `OPCODE_ALURI) ? 1 : 0;

always @* begin
        if (is_aluRR || is_aluRI) begin
            case (funct3)
                `OP_ADD: alu_ctrl = inst[30] ? `ALU_SUB : `ALU_ADD;
                `OP_SLL: alu_ctrl = `ALU_SLL;
                `OP_SLT: alu_ctrl = `ALU_SLT;
                `OP_SLTU: alu_ctrl = `ALU_SLTU;
                `OP_XOR: alu_ctrl = `ALU_XOR;
                `OP_SRL: alu_ctrl = inst[30] ? `ALU_SRA : `ALU_SRL;
                `OP_OR: alu_ctrl = `ALU_OR;
                `OP_AND: alu_ctrl = `ALU_AND;
            endcase
        end
        else alu_ctrl = `ALU_ADD;
        //convenient for other offset operations

assign is_imm = is_aluRI; //即为数据通路中的 alu_src2

if (is_aluRI) begin
        if (alu_ctrl == `ALU_SLTU) immType = `IMM12_I_U;
        else immType = `IMM12_I_S;
end

assign reg_wen = is_aluRR | is_aluRI;
```

新建 ImmGen.v，立即数生成器如下：（为了方便起见将所有立即数类型先定义好）

```
always @* begin
    case (immType)
        `IMM12_I_S: imm = {{20{inst[31]}}, inst[31:20]};
        `IMM12_I_U: imm = {20'h00000, inst[31:20]};
        `IMM12_S: imm = {{20{inst[31]}}, inst[31:25], inst[11:7]};
        `IMM12_SB:  imm  =  {{20{inst[31]}},  inst[7],  inst[30:25],
inst[11:8], 1'b0};
        `IMM20_UJ:  imm  =  {{12{inst[31]}},  inst[19:12],  inst[20],
inst[30:21], 1'b0};
        `IMM20_U: imm = {inst[31:12], 12'b0};
        endcase
    end
```

2.3 仿真验证

汇编代码以及预期结果如下：

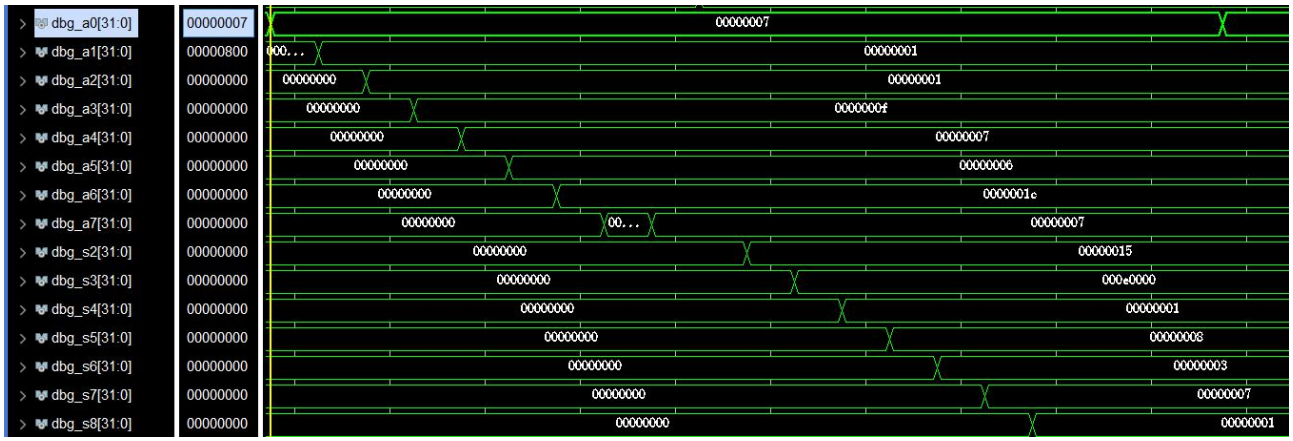
```
# test I type
    addi    a0, zero, 7     # a0 = 0x0000_0007
    slti    a1, a0, 9       # a1 = 0x0000_0001
    sltiu   a2, a0, 8       # a2 = 0x0000_0001
    xori    a3, a0, 8       # a3 = 0x0000_000F
```

```
    ori     a4, a0, 6       # a4 = 0x0000_0007
    andi    a5, a0, 6       # a5 = 0x0000_0006
    slli    a6, a0, 2       # a6 = 0x0000_001C
    srli    a7, a0, 2       # a7 = 0x0000_0001
    srai    a7, a0, 0       # a7 = 0x0000_0007
# test R type
    add     s1, a0, a0      # s1 = 0x0000_000E
    sub     s2, a6, a0      # s2 = 0x0000_0015
    sll     s3, a6, a3      # s3 = 0x000E_0000
    slt     s4, a0, s3      # s4 = 0x0000_0001
    xor     s5, a0, a3      # s5 = 0x0000_0008
    srl     s6, a0, a2      # s6 = 0x0000_0003
    or      s7, a0, a4      # s7 = 0x0000_0007
    and     s8, a0, a2      # s8 = 0x0000_0001
```

使用 venus 插件，生成 PC、机器码以及汇编代码如下：（后续其他指令仿真验证部分将省略此内容）

```
0x00000034      0x00700513      addi x10 x0 7
0x00000038      0x00952593      slti x11 x10 9
0x0000003C      0x00853613      sltiu x12 x10 8
0x00000040      0x00854693      xori x13 x10 8
0x00000044      0x00656713      ori x14 x10 6
0x00000048      0x00657793      andi x15 x10 6
0x0000004C      0x00251813      slli x16 x10 2
0x00000050      0x00255893      srli x17 x10 2
0x00000054      0x40055893      srai x17 x10 0
0x00000058      0x00A504B3      add x9 x10 x10
0x0000005C      0x40A80933      sub x18 x16 x10
0x00000060      0x00D819B3      sll x19 x16 x13
0x00000064      0x01352A33      slt x20 x10 x19
0x00000068      0x00D54AB3      xor x21 x10 x13
0x0000006C      0x00C55B33      srl x22 x10 x12
0x00000070      0x00E56BB3      or x23 x10 x14
0x00000074      0x00C57C33      and x24 x10 x12
```

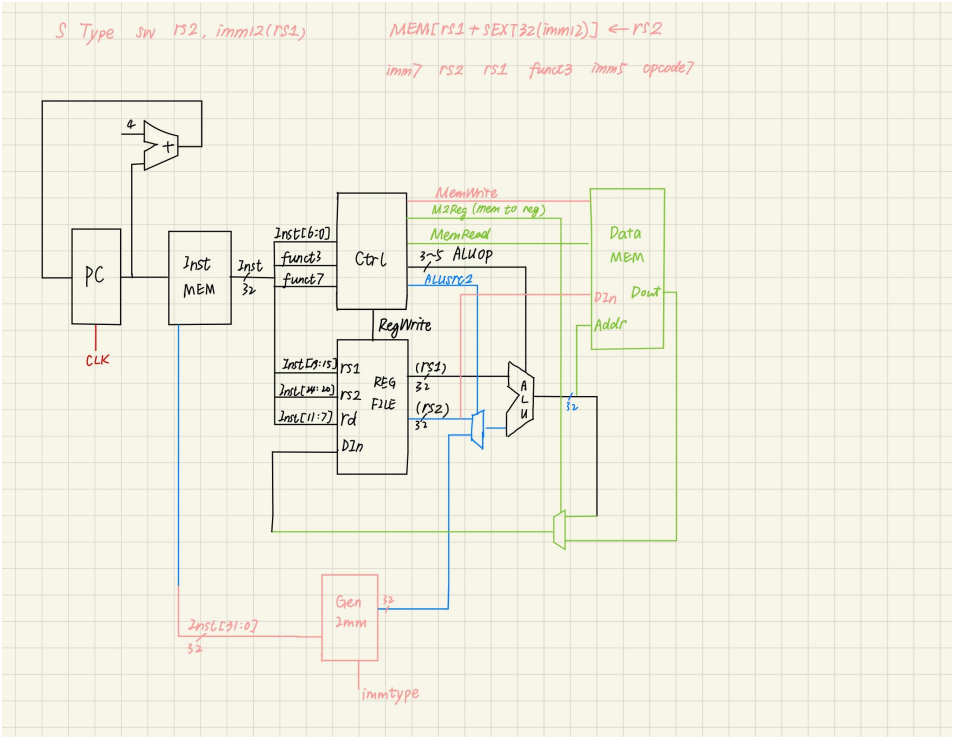
对应部分的仿真波形如下：



可见仿真完全正确。

3 实现 LW, SW 指令，LED 控制并仿真验证

3.1 数据通路



如图是添加完 Load 和 Store 指令的数据通路。可见增加了 DMem 和新的控制信号，如 mem_wen, mem_ren 和 m2reg。其余不变。

3.2 功能实现

Controller.v，控制信号如下：

```
assign is_load = (opcode == `OPCODE_LOAD) ? 1 : 0;
assign is_store = (opcode == `OPCODE_STORE) ? 1 : 0;

...
else if (is_load | is_jalr) immType = `IMM12_I_S;
       else if (is_store) immType = `IMM12_S;

assign reg_wen = is_aluRR | is_aluRI | is_load ;
assign is_imm = is_aluRI | is_load | is_store;
assign mem_ren = is_load;
assign mem_wen = is_store;
assign m2reg = is_load;
```

在 Lab1 中我们还实现了 LEDCtrl，即当地址为 0xFE000000 时，向 LED 读写。改写 LED 模块如下：在 Top.v 中，将 LEDCtrl 模块接入~clk_cpu。LEDCtrl.v 中仿照 DMem 改写。

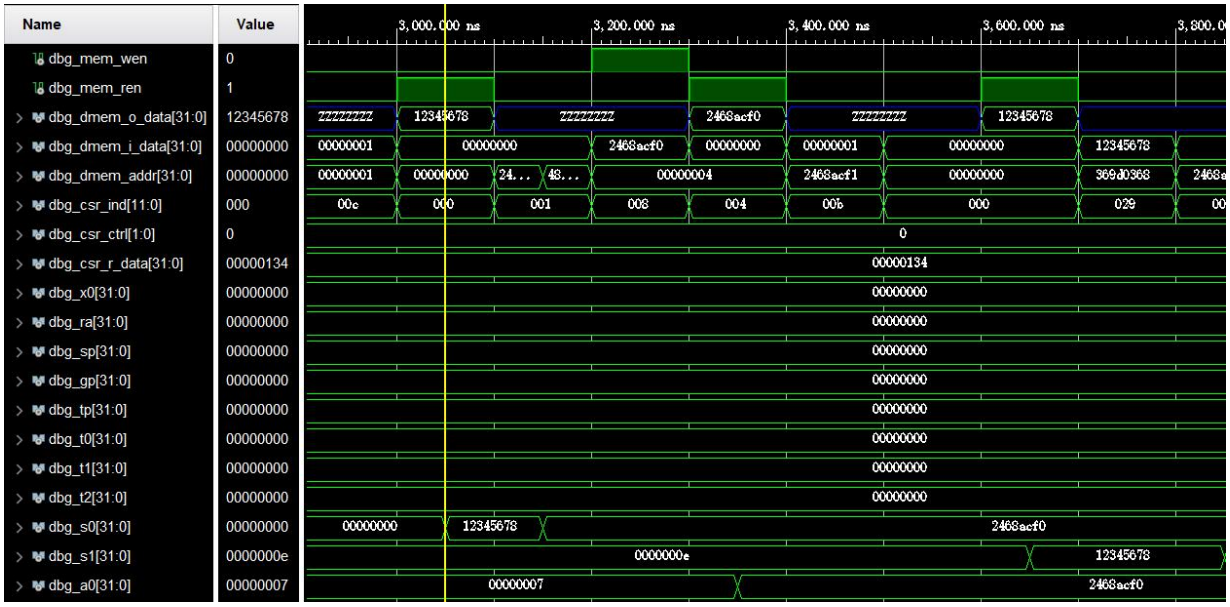
```
reg [31:0] data;
always @(posedge clk or posedge rst) begin
   if (rst) data <= 32'h000000FF;
   else if (wen) data <= i_data; // lack of else
end
assign o_data=data;
assign o_led_ctrl=data[7:0];
```

3.3 仿真验证

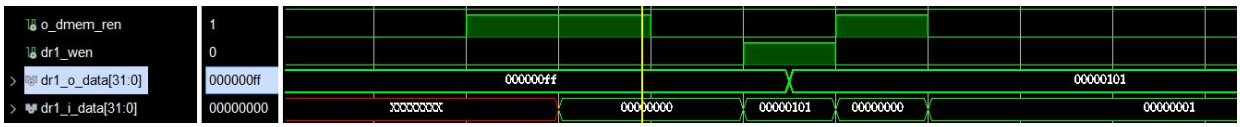
汇编代码以及预期结果如下：

```
# test LW and SW
    lw      s0, 0(zero)     # s0 = 0x1234_5678
    slli    s0, s0, 1       # s0 = 0x2468_acf0
    sw      s0, 4(zero)     # (* no GPRs modified *)
    lw      a0, 4(zero)     # a0 = 0x2468_acf0

# test LED
    lw      t0, 8(x0)           # t0 = FE00000
    lw      t1, 0(t0)           # t1 = LED_data
    addi    t1, t1, 2           # t1 = t1 + 2
    sw      t1, 0(t0)           # t1 -> LED
    lw      t2, 0(t0)
```

对应部分的仿真波形如下：



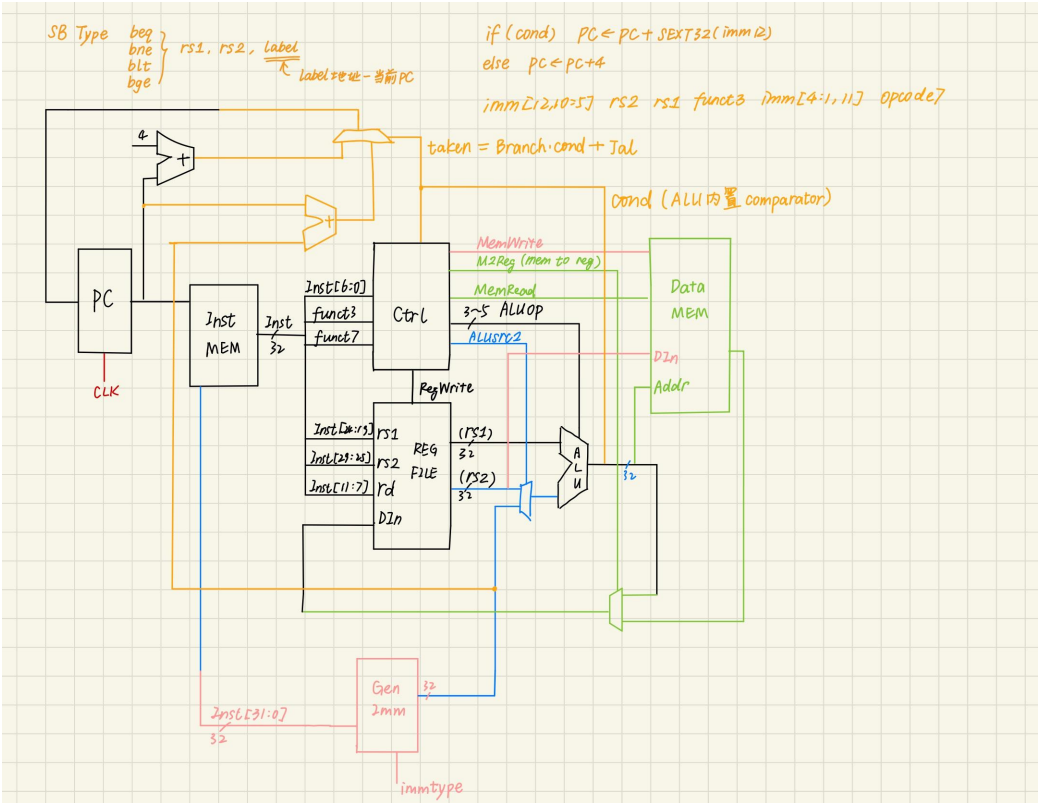
可见 lw, sw 指令正确。



可见 led 读写正确。

4 实现 branch 指令并仿真验证

4.1 数据通路



如图是添加完 branch 指令的数据通路。在图中 ALU 和 Comparator 二合一，而在实验中 Comparator 和 ALU 是分开的。主要新增了 PC 的跳转控制信号和 PC+offset 计算模块。

4.2 功能实现

PC 修改如下：

```
reg [31:0] pc;
wire [31:0] pc_branch;
assign do_branch = is_branch & cmp_res;
assign pc_branch = do_branch ? pc+imm : pc+4;

always @(posedge clk or posedge rst) begin
     if (rst) begin
        pc <= 0;
     end
     else begin
        pc <= pc_branch;
     end
end
```

Controller.v，控制信号如下：

```
assign is_branch = (opcode == `OPCODE_BRANCH) ? 1 : 0;

...
else if (is_branch) immType = `IMM12_SB;

case (funct3)
          `BRANCH_EQ: cmp_ctrl = `CMP_EQ;
          `BRANCH_NE: cmp_ctrl = `CMP_NE;
```

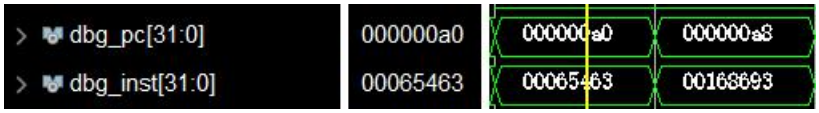
```
        `BRANCH_LT: cmp_ctrl = `CMP_LT;
        `BRANCH_GE: cmp_ctrl = `CMP_GE;
        `BRANCH_LTU: cmp_ctrl = `CMP_LTU;
        `BRANCH_GEU: cmp_ctrl = `CMP_GEU;
    endcase
```

4.3 仿真验证

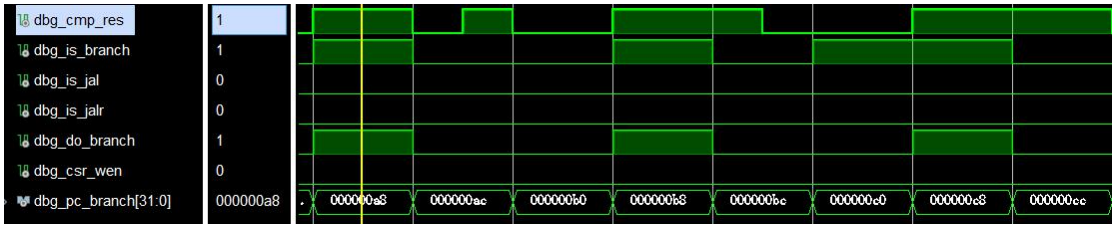
以如下三条指令为例。第一列为 PC 值，预期结果为从 0xA0 跳转至 0xA8。

```
0x000000A0    0x00065463    bge x12 x0 8
0x000000A4    0x02000263    beq x0 x0 36
0x000000A8    0x00168693    addi x13 x13 1
```

对应部分的 PC 和 inst 变化如下：



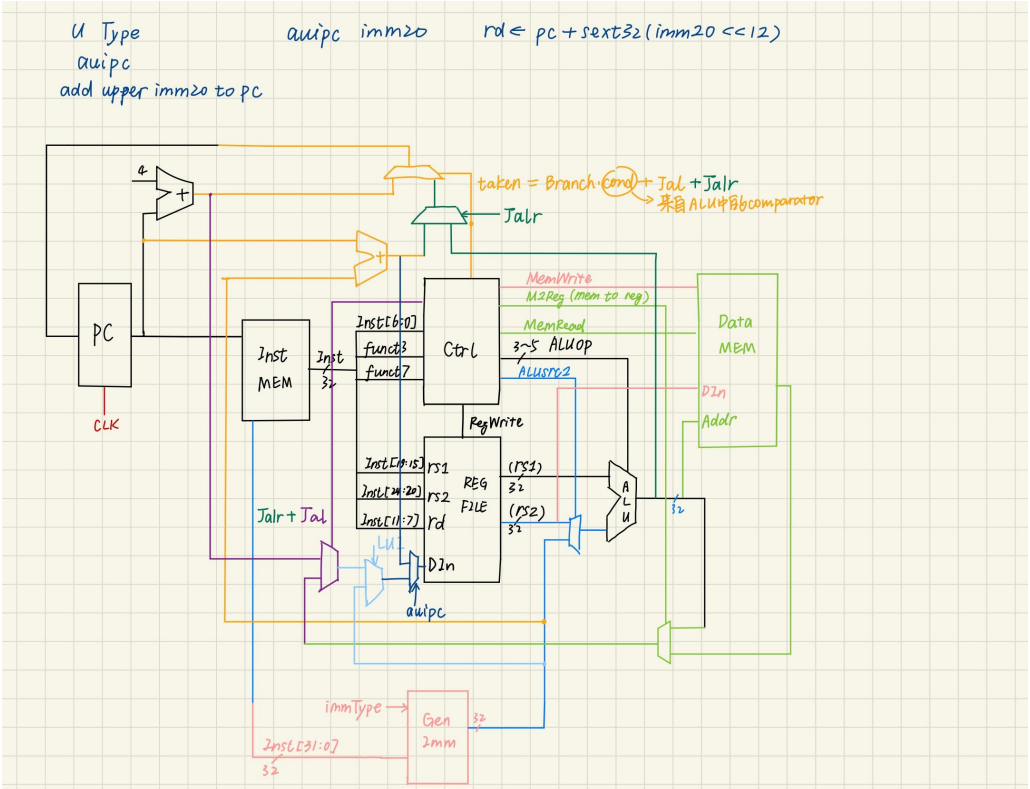
此部分以及其他 branch 测试指令的 cmp_res, is_branch, do_branch, pc_branch 信号如下：



可见 branch 指令正确。

5 实现基本 CPU 的其他指令(jal, jalr, lui, auipc)并仿真验证

5.1 数据通路



如图是添加完 jal, jalr, lui, auipc 指令后的数据通路。至此基本 SCPU 的数据通路如上图所示。主要是对 RegFile 写口的输入和 PC 跳转信号来源和控制信号做出进一步的选择。比较丑陋的方法是用 4 个 2 选 1 的多路选择器选择，当然也可以整合成 4 选 1 的多路选择器。

5.2 功能实现

PC 修改如下：

```
reg [31:0] pc;
wire [31:0] pc_branch;
assign do_branch = is_branch & cmp_res | is_jal | is_jalr;
wire [31:0] pc_jalOrJalr;
assign pc_jalOrJalr = (is_jalr) ? alu_res : pc + imm;
wire [31:0] pc_jOrAdd4;
assign pc_branch = do_branch ? pc_jalOrJalr : pc + 4;
```

Controller.v，控制信号如下：

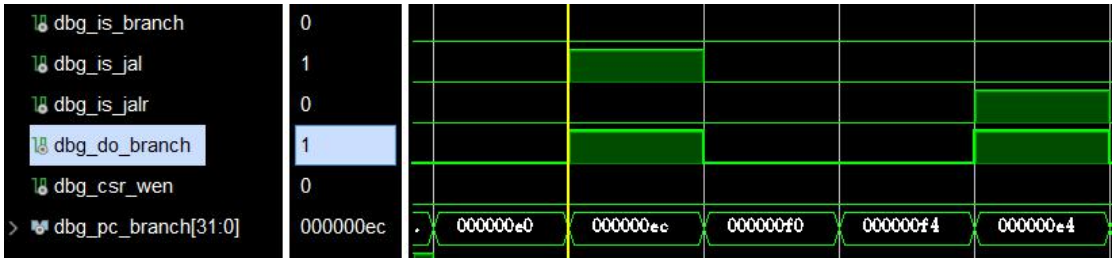
```
assign is_jal = (opcode == `OPCODE_JAL) ? 1 : 0;
assign is_jalr = (opcode == `OPCODE_JALR) ? 1 : 0;
assign is_lui = (opcode == `OPCODE_LUI) ? 1 : 0;
assign is_auipc = (opcode == `OPCODE_AUIPC) ? 1 : 0;

...
else if (is_jal) immType = `IMM20_UJ;
else if (is_lui | is_auipc) immType = `IMM20_U;

assign reg_wen = is_aluRR | is_aluRI | is_load | is_jal | is_lui |
is_jalr | is_auipc;
assign is_imm = is_aluRI | is_load | is_store | is_jalr;
```

5.3 仿真验证

测试汇编代码如下：

```
# test JAL, LUI, JALR and AUIPC
    jal     x1, test_jal
    add     x1, x0, x0
    beq     x1, x0, EXIT
test_jal:
    lui     t3, 0x12345
    auipc   t4, 0x32001
    jalr    x0, x1, 0
EXIT:
```

对应部分的 is_jal, is_jalr, do_branch, pc_branch 信号如下：



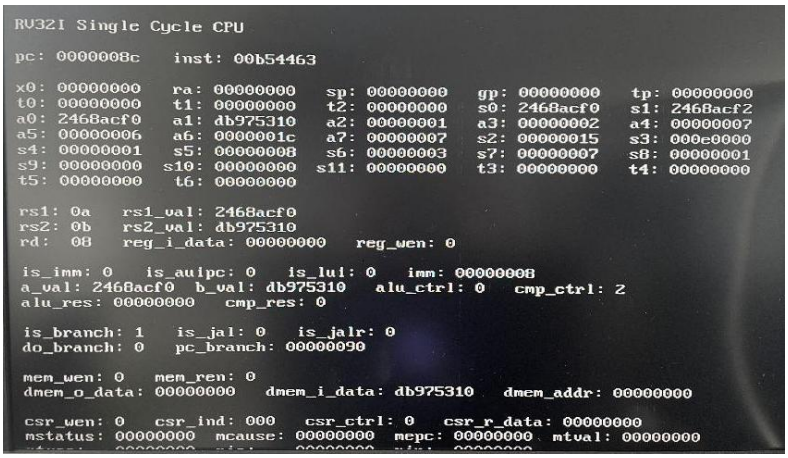
lui, auipc 测试如下：

可见四条指令均正确。

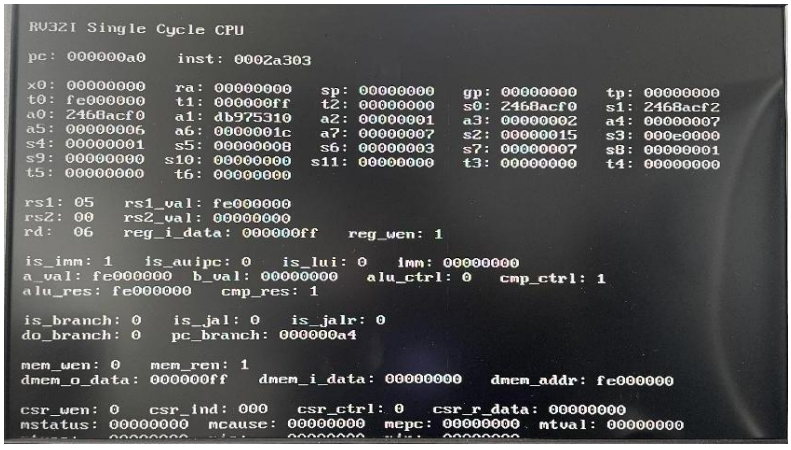
6 基本 SCPU 指令上板验证

由于指令条数较多，无法一一展示。此处仅以测试 LED 读写的指令为例。

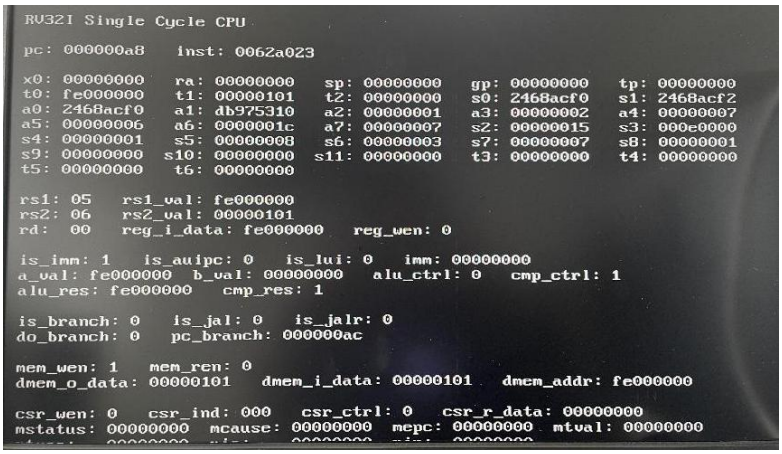
```
# test LED
    lw      t0, 8(x0)           # t0 = FE00000
    lw      t1, 0(t0)           # t1 = LED_data
    addi    t1, t1, 2           # t1 = t1 + 2
    sw      t1, 0(t0)           # t1 -> LED
    lw      t2, 0(t0)
```



t0 = t1 = 00000000

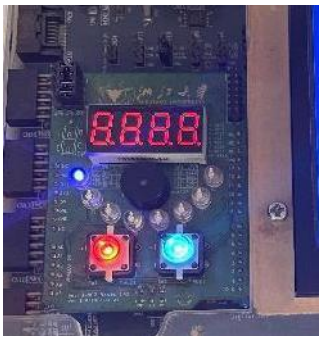


经过两条 lw 指令后，t0 = fe000000, t1 = 000000ff, mem_ren = 1



经过 addi 指令后 t1 = 00000101。sw 指令时 mem_wen = 1

sw 指令前，led[7:0] = ff



sw 指令后，led[7:0] = 01

由图可知上板现象正确。

7 设计 CSR 部件，实现 CSRRX 系列指令并仿真验证

7.1 设计 CSR

CSRs 理论上有 2^12 = 4096 个，但是本实验只需要实现其中 7 个。为了节省资源，我只定义了所需要的 7 个寄存器。

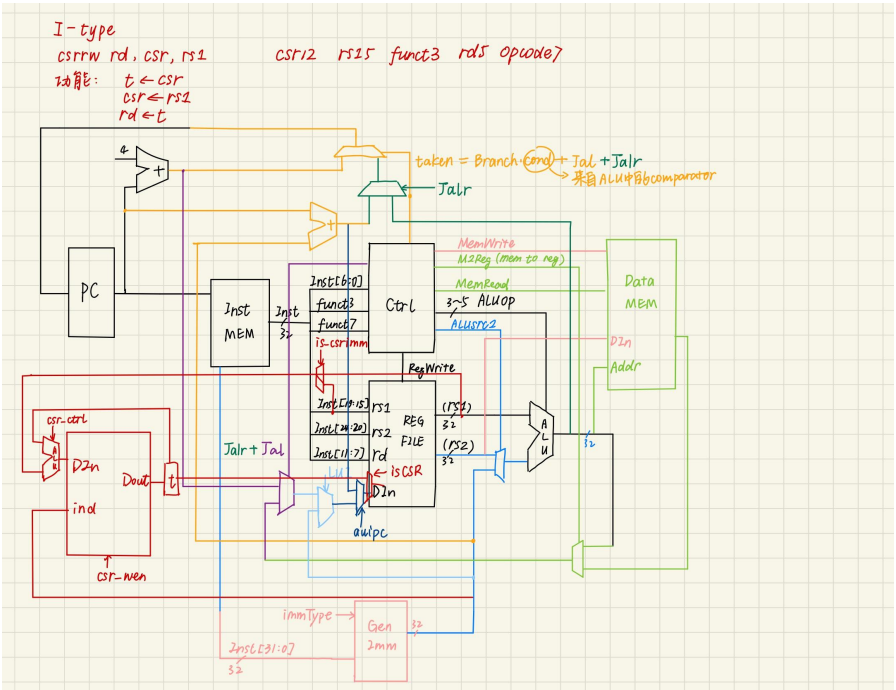
如果只需要实现 CSRRX 系列指令，只需要一个写口，一个读口。写口需要送入写入数据、写使能信号和寄存器编号。但是如果要实现异常和中断，则需要在同一个时钟周期内同时改写 mip, mepc, mstatus, mcause, mtval 等多个寄存器，说明写口和写使能只有一个是远远不够的，每一个寄存器都要有对应的写入数据。

定义 CSR 模块如下：

```
module CSR(
    `VGA_DBG_Csr_Outputs //7 个寄存器的 debug 信号
    input clk, rst,
    input wen,            //供 CSRR 指令使用的写使能
    input [11:0] ind,    //供 CSRR 指令使用的写入寄存器编号
    input [31:0] i_data,//供 CSRR 指令使用的写入数据
    input [31:0] EPCval,//处理异常使用的写入 mepc 数据，下同
    input [31:0] CauseVal,
    input [31:0] tval,
    input is_exp,       //发生异常信号
    input is_mret,      //mret 指令信号
    output reg [31:0] o_data, //供 CSRR 指令使用的读口数据
    output reg INT,     //生成的中断使能
    output reg [31:0] pc_exp //生成的 pc 跳转地址
    );
    reg [31:0] mstatus_o;
    reg [31:0] mcause_o;
    reg [31:0] mepc_o;
    reg [31:0] mtval_o;
    reg [31:0] mtvec_o;
    reg [31:0] mie_o;
    reg [31:0] mip_o;

    `VGA_DBG_Csr_Assignments
    always @* begin
       case (ind)
           `CSR_MSTATUS: o_data = mstatus_o;
           `CSR_MIE: o_data = mie_o;
           `CSR_MIP: o_data = mip_o;
```

```
            `CSR_MEPC: o_data = mepc_o;
            `CSR_MCAUSE: o_data = mcause_o;
            `CSR_MTVAL: o_data = mtval_o;
            `CSR_MTVEC: o_data = mtvec_o;
        endcase
        if (INT) pc_exp = mtvec_o;
        else if (is_mret) pc_exp = mepc_o;
    end
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            mstatus_o <= 0;
            mcause_o <= 0;
            mepc_o <= 0;
            mtval_o <= 0;
            mtvec_o <= 0;
            mie_o <= 0;
            mip_o <= 0;
            INT <= 0;
        end
        else if (wen) begin
            INT    <=    mstatus_o[3]    &    mie_o[mcause_o[3:0]]    &
mip_o[mcause_o[3:0]]; //这一句是为了增加异常和中断做准备
            case (ind)
                `CSR_MSTATUS: mstatus_o <= i_data;
                `CSR_MIE: mie_o <= i_data;
                `CSR_MIP: mip_o <= i_data;
                `CSR_MEPC: mepc_o <= i_data;
                `CSR_MCAUSE: mcause_o <= i_data;
                `CSR_MTVAL: mtval_o <= i_data;
                `CSR_MTVEC: mtvec_o <= i_data;
            endcase
        end
```

7.2 数据通路



如图是添加完 csrrw, csrrs, csrrc, csrrwi, csrrsi, csrrci 指令后的数据通路。可见增加了 CSR

寄存器和csr_ctrl, csr_ind, csr_wen, is_csrimm, is_csr等控制信号。图中的t寄存器不是必要的，可以在实践中省略。

至此所有基本指令已经添加完毕。基本的控制信号真值表如下所示：

7.3 功能实现

Core.v，代码如下：

```
assign csr_val2 = (csr_src2) ? {27'b0, inst[19:15]} : rs1_val;
    always @* begin
        case(csr_ctrl)
            `CSR_CTRL_W: csr_i_data = csr_val2;
            `CSR_CTRL_S: csr_i_data = csr_r_data | csr_val2;
            `CSR_CTRL_C: csr_i_data = csr_r_data & ~csr_val2;
        endcase
        csr_ind = inst[31:20];
end
```

Controller.v，控制信号如下：

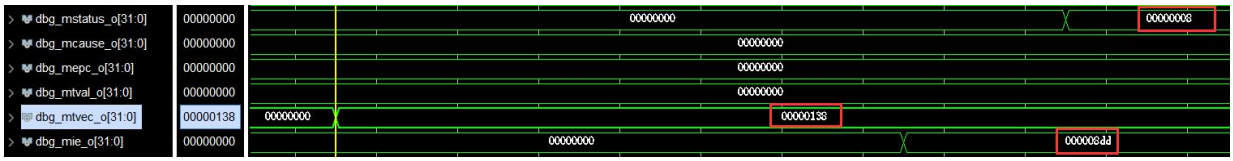
```
    assign is_sys = (opcode == `OPCODE_SYSTEM) ? 1 : 0;
    assign is_csr = (is_sys && (funct3 == `SYSTEM_CSRRC || funct3 ==
`SYSTEM_CSRRCI
        || funct3 == `SYSTEM_CSRRS || funct3 == `SYSTEM_CSRRSI ||
        funct3 == `SYSTEM_CSRRW || funct3 == `SYSTEM_CSRRWI)) ? 1 : 0;
    assign csr_wen = is_csr;

    ...
    if (is_csr) begin
                case (funct3)
                `SYSTEM_CSRRW: csr_ctrl = `CSR_CTRL_W;
                `SYSTEM_CSRRWI: csr_ctrl = `CSR_CTRL_W;
                `SYSTEM_CSRRS:  csr_ctrl = `CSR_CTRL_S;
                `SYSTEM_CSRRSI:  csr_ctrl = `CSR_CTRL_S;
                `SYSTEM_CSRRC: csr_ctrl = `CSR_CTRL_C;
                `SYSTEM_CSRRCI: csr_ctrl = `CSR_CTRL_C;
                endcase
                if (funct3 == `SYSTEM_CSRRWI || funct3 == `SYSTEM_CSRRSI
|| funct3 == `SYSTEM_CSRRCI)
                    csr_src2 = 1;
                else csr_src2 = 0;
    end
```

7.4 仿真验证

汇编代码以及预期结果如下：

```
test_csrr:
    addi    a0, zero, 0x138
    csrrw   zero, mtvec, a0 # set mtvec = 异常处理程序的首地址
    addi    a0, zero, 8
    slli    a1, a0, 4
    or      a0, a0, a1
    slli    a1, a1, 4
    or      a0, a0, a1
    addi    a0, a0, 0x55
    csrrw   zero, mie, a0   # set mie[11,7,6,4,3,2] = 1
    addi    s0, zero, 255
    csrrsi  s0, mstatus, 8  # set mstatus.mie = 1, s0 重置为 0
```

对应部分的仿真波形如下：



可见 mtvec, mie, mstatus 都成功写入，csrr 指令正确。

8 实现 ecall 和 mret 指令并仿真验证

8.1 ecall 和 mret 指令功能

ecall 是我实现的第一种异常。当执行到 ecall 指令时，首先检验是否能执行 ecall。判断条件为 mstatus.MIE & mip[11] & mie[11] = 1。其他异常类似。

在执行 ecall 指令的这个时钟周期内，要在 CSR 中完成如下操作：

- mepc <- PC
- PC <- mtvec
- mstatus[7] <- mstatus[3]
- mstatus[3] <- 0

这里涉及到多个寄存器的读写判断，非常复杂，花费了我大量时间 debug。我在实验心得部分的问题二做了更详细的讨论。

异常处理程序的最后一条指令是 mret。当执行 mret 时，将完成如下操作：

- PC <- mepc
- mstatus[3] <- mstatus[7]
- mstatus[7] <- 1
- mip[i] <- 0

相较而言 mret 指令不那么复杂。

8.2 功能实现

Controller.v，控制信号如下：

```
assign is_ecall = (is_sys && inst[31:7] == 0) ? 1 : 0;
assign is_mret = (inst[31:0] == 32'h30200073) ? 1 : 0;
```

Core.v 中，增加代码如下：

```
assign pc_branch = (INT | is_mret) ? pc_exp : pc_jOrAdd4;
assign is_exp = is_ecall;
assign EPCval = pc;
if (is_ecall) CauseVal = `CSR_CAUSE_ECALL_FROM_M;
```

CSR.v 中，增加如下代码：

```
always @* begin
       ...
       if (INT) pc_exp = mtvec_o;
       else if (is_mret) pc_exp = mepc_o;
end

...
else begin
          if (is_exp) mip_o[CauseVal[3:0]] = 1;
          else if (is_mret) mip_o[mcause_o[3:0]] = 0;
          if     (mstatus_o[3]     &     mie_o[CauseVal[3:0]]     &
mip_o[CauseVal[3:0]] | is_mret) begin
              if (is_mret) begin
                 mstatus_o[3] <= mstatus_o[7];
                 mstatus_o[7] <= 1;
              end
              else begin
                 mepc_o <= EPCval;
                 mcause_o = CauseVal;
                 mtval_o <= tval;
                 mstatus_o[7] <= mstatus_o[3];
                 mstatus_o[3] <= 0;
              end
          end
          INT    =    mstatus_o[3]    &    mie_o[mcause_o[3:0]]    &
mip_o[mcause_o[3:0]];
       end
```

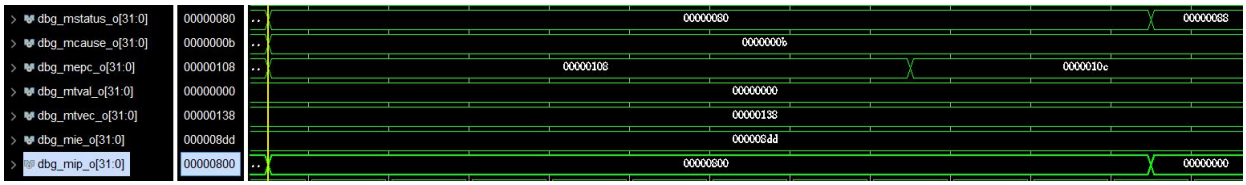
8.3 仿真验证

汇编代码以及预期结果如下：

```
# test_ecall
0x00000108    0x00000073    ecall

# 所有异常/中断程序的起始地址
trap_m:
    csrrs   a0, mcause, zero    # a0 = mcause
    slt     a1, a0, zero # 解析是异常还是中断，a1 = 1 为中断，a1 = 0 为异常
    bne     a1, x0, handle_int
    # exception
    # mepc = mepc + 4
    csrrs   t0, mepc, zero
    addi    t0, t0, 4
    csrrw   zero, mepc, t0
    addi    t0, zero, 11
```

```
    beq     a0, t0, handle_ecall
    addi    t0, zero, 2
    beq     a0, t0, handle_illop
    addi    t0, zero, 4
    beq     a0, t0, handle_lw_misalign
    addi    t0, zero, 6
    beq     a0, t0, handle_sw_misalign
handle_int:
handle_ecall:
out_int:
    mret
```

对应部分的仿真波形如下：



ecall 指令的 PC 值为 0x108。可见发生 ecall 时，mip[11] = 0，mepc = 0x108，mcause = 0x0000000b，为 ecall 类型。mstatus[7] = mstatus[3]，mstatus[3] = 0。与设计功能完全一致。

在异常程序执行过程中，由于发生的是异常而不是中断，mepc 由软件实现+4。

当发生 mret 指令时，mip 相应位置清 0，mstatus[3] = 1。返回至原先的 PC+4。

PC 变化如下：

可见 ecall 和 mret 指令正确。

9 实现非法指令、PC, lw, sw 地址非对齐异常并仿真验证

9.1 四种异常简述

非法指令异常检测非常容易。只要解析 inst[6:0]的 opcode7，发现不是现有指令的任何一种，则发生非法指令异常。则在中断使能为 1 的情况下，准备跳转至中断处理程序。

PC 地址非对齐异常检测也很容易。只要当前 PC[1:0] != 2'b00，则发生 PC 非对齐异常。则在中断使能为 1 的情况下，准备跳转至中断处理程序。

lw, sw 地址非对齐异常也比较容易。只要当前是 lw 指令或 sw 指令，而且检测 dmem_addr[1:0] != 2'b00，则发生此两种异常。需要注意的是，要小小修改一下 dmem_ren 和 dmem_wen，当发生 lw 地址非对齐异常时，要关闭 dmem_ren 使能；当发生 sw 地址非对齐异常时，要关闭 dmem_wen 使能。不要错误地读取和写入 DMem。

这四种异常大体框架和 ecall 指令非常类似。

9.2 功能实现

Controller.v，控制信号如下：

```
assign illegal_op = ~(is_aluRR | is_aluRI | is_load |
    is_store | is_branch | is_jal | is_jalr | is_lui |
    is_auipc | is_sys);
```

Core.v 中，增加或修改代码如下：

```
assign is_exp = is_ecall | illegal_op | is_pc_misalign | is_lw_misalign
| is_sw_misalign;
```

```
assign is_sw_misalign = (mem_wen && dmem_addr[1:0] != 2'b00) ? 1 : 0;
assign is_lw_misalign = (mem_ren && dmem_addr[1:0] != 2'b00) ? 1 : 0;
assign is_pc_misalign = (pc[1:0] != 2'b00) ? 1 : 0;

...
else if (illegal_op) CauseVal = `CSR_CAUSE_ILLEGAL_INST;
else if (is_pc_misalign) CauseVal = `CSR_CAUSE_INST_ADDR_MISALIGN;
else if (is_lw_misalign) CauseVal = `CSR_CAUSE_LOAD_ADDR_MISALIGN;
else if (is_sw_misalign) CauseVal = `CSR_CAUSE_STORE_ADDR_MISALIGN;

if (illegal_op) tval = inst;
       else if (is_pc_misalign) tval = pc;
       else if (is_lw_misalign | is_sw_misalign) tval = dmem_addr;
       else tval = 0;

assign dmem_wen = mem_wen & ~is_sw_misalign;
assign dmem_ren = mem_ren & ~is_lw_misalign;
```

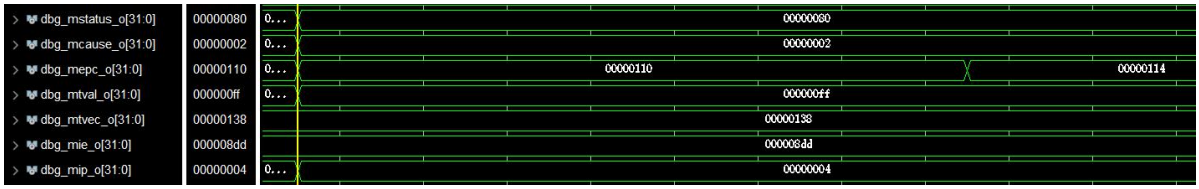
9.3 仿真验证

汇编代码以及预期结果如下：

```
   test_illegal_inst:
      addi    a0, zero, 0 # 修改成 0x000000ff,成为非法指令
      addi    a0, zero, 0 # 缓冲
   test_pc_misalign:
      beq     zero, zero, test_lw_misalign    # 修改成 0x00000163, pc 不
与 4 对齐
   test_lw_misalign:
      addi    t0, zero, 3
      lw      t1, 0(t0)
   test_sw_misalign:
      addi    t0, zero, 3
      sw      a0, 0(t0)

   # 异常处理程序中处理 pc 非对齐部分
   handle_pc_misalign:
      csrrs   t5, mtval, zero
      andi    t3, t5, 3
      addi    t4, zero, 1
      bne     t3, t4, check2
      addi    t5, t5, 3
      jal     x0, correctPC
   check2:
      addi    t4, t4, 1
      bne     t3, t4, check3
      addi    t5, t5, 2
      jal     x0, correctPC
   check3:
      addi    t5, t5, 1
   correctPC:
      csrrw   zero, mepc, t5
      jal     x0, out_int
```

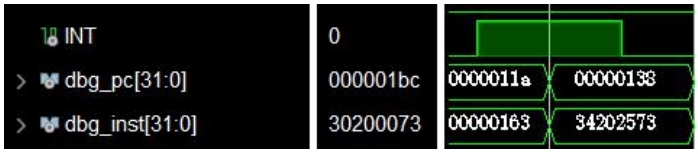
对应部分的仿真波形如下：

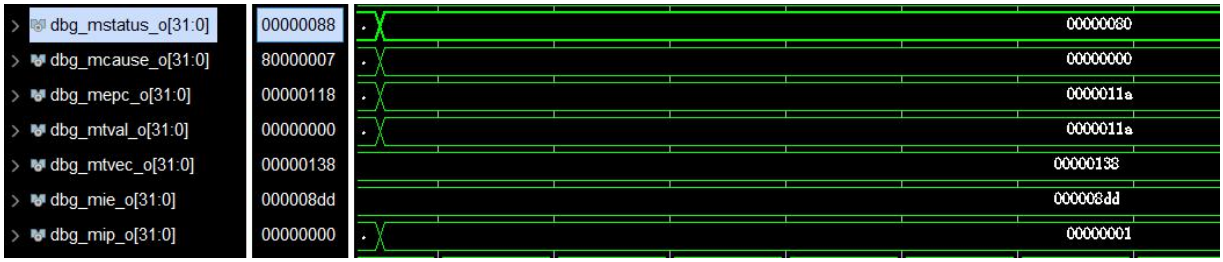
非法指令：



mcause = 2，表示为非法指令异常。除了更改 mepc, mip, mstatus 以外，还要用 mtval 记录非法指令。

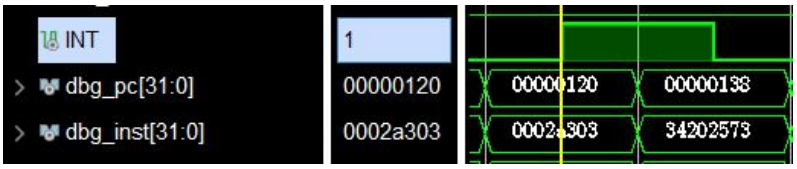
PC 地址非对齐：

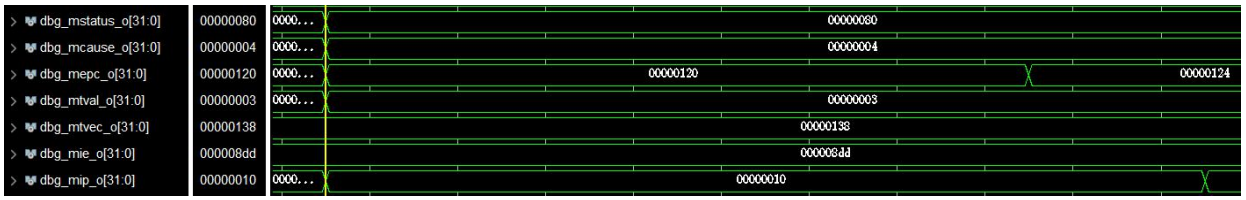




mcause = 0，表示为 PC 地址非对齐异常。除了更改 mepc, mip, mstatus 以外，还要用 mtval 记录非法 PC。

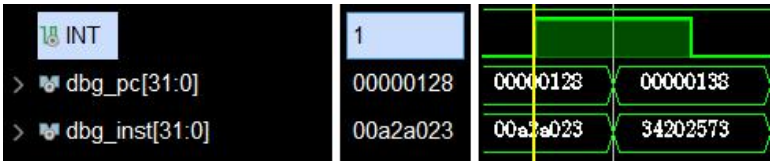
lw 地址非对齐：

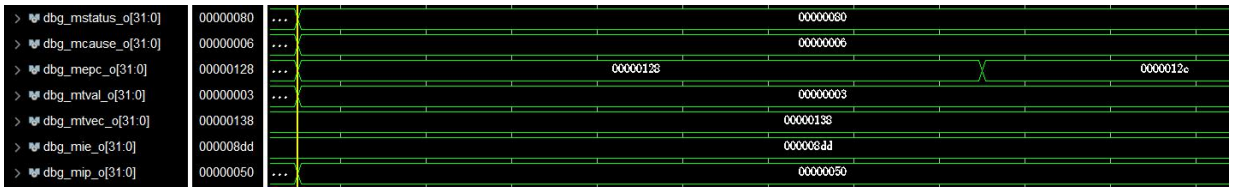




mcause = 4，表示为 lw 地址非对齐异常。除了更改 mepc, mip, mstatus 以外，还要用 mtval 记录非法的 dmem_addr。

sw 地址非对齐：





mcause = 6，表示为 lw 地址非对齐异常。除了更改 mepc, mip, mstatus 以外，还要用 mtval 记录非法的 dmem_addr。

可见四种异常都可以正确处理。

10 实现 timer 和 external 中断并仿真验证

10.1 两种中断简述

外部中断处理比较简单。只要在 Core.v 的 input 信号中接入 key_x[1]，当按下该列键盘时，如果中断使能允许，则会开始处理外部中断。

计时器中断则相对来说复杂。在 PPT 中助教写道从 Core 外部连入 clk_div[31]作为信号。理论上可行，但上板时 VGA 又无法显示，说明 clk_div[31]信号又是不与 clk_cpu 同步的。我采取的解决办法是在 Core.v 内部使用计数器分频 clk_cpu，得到一个新的时钟，作为 timer 中断的输入。详细见下方代码。

处理一层中断很简单，和异常无殊。但是嵌套中断，或者正在处理异常时发生中断（从逻辑上来说应该没有嵌套异常和处理中断时发生异常这两种情况，因为至少保证异常/中断处理程序是正确的），则非常麻烦。关于嵌套中断，我将在实验心得部分进一步讨论。

10.2 功能实现

Core 内部的分频时钟（虽然逻辑上来说，时钟中断不应该产生自 core 内部，但是为了能够正常上板子，我选择用这样的方式模拟。）

```
reg [4:0] cnt;
    always @ (posedge clk or posedge rst) begin
        if (rst) begin
            tim_int <= 0;
            cnt <= 0;
        end
        else if (cnt < 31) cnt <= cnt + 1;
        else begin
          cnt <= 0;
          tim_int <= ~tim_int;
     end
end
```

Core.v 中，增加或修改代码如下：

```
 assign is_exp = is_ecall | illegal_op | is_pc_misalign | is_lw_misalign
| is_sw_misalign | tim_int | ext_int;

else if (tim_int) CauseVal = `CSR_CAUSE_M_TIMER_INT;
else if (ext_int) CauseVal = `CSR_CAUSE_M_EXTERNAL_INT;
```

Top.v 中，接入外部中断信号：

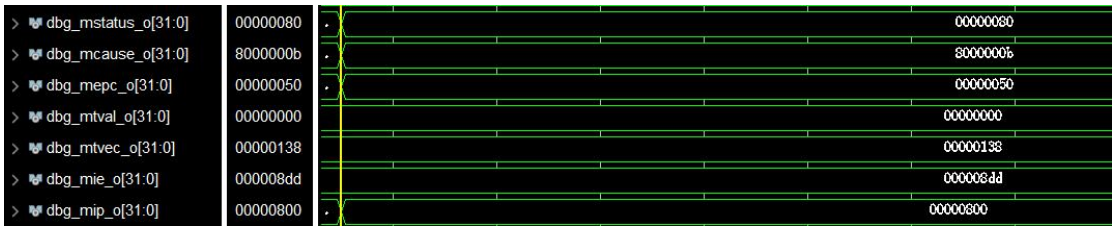
```
Core core(
        `VGA_DBG_Core_Arguments
        ...
        .ext_int(key_x[1])
    );
```

10.3 仿真验证

在 core_tb.v 中加入外部中断信号：

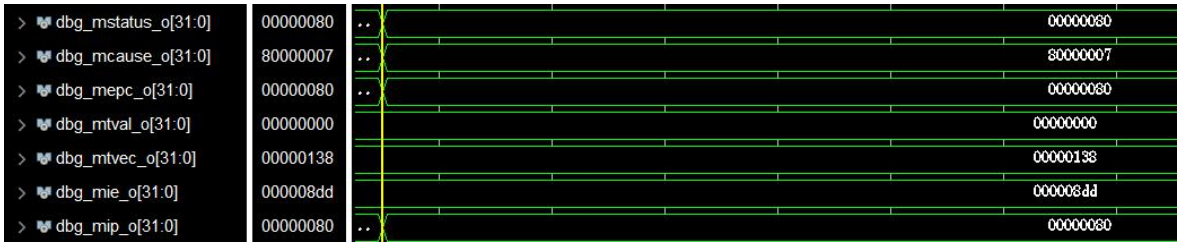
```
initial begin
        rst = 1;
        clk_cpu = 1;
        ext_int = 0;
        #10;
        rst = 0;
        #2000
        ext_int = 1;
        #100
        ext_int = 0;
    end
```

外部中断的仿真波形如下：



mcause = 0x8000000b，表示为外部中断。

时钟中断的仿真波形如下：

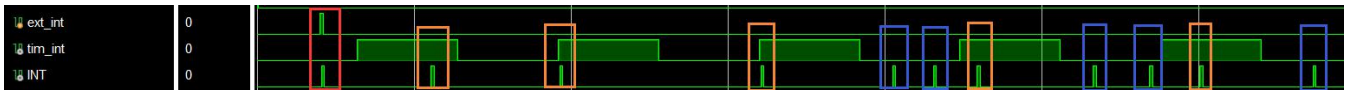


mcause = 0x80000007，表示为时钟中断。

可见两种中断都可以正确处理。

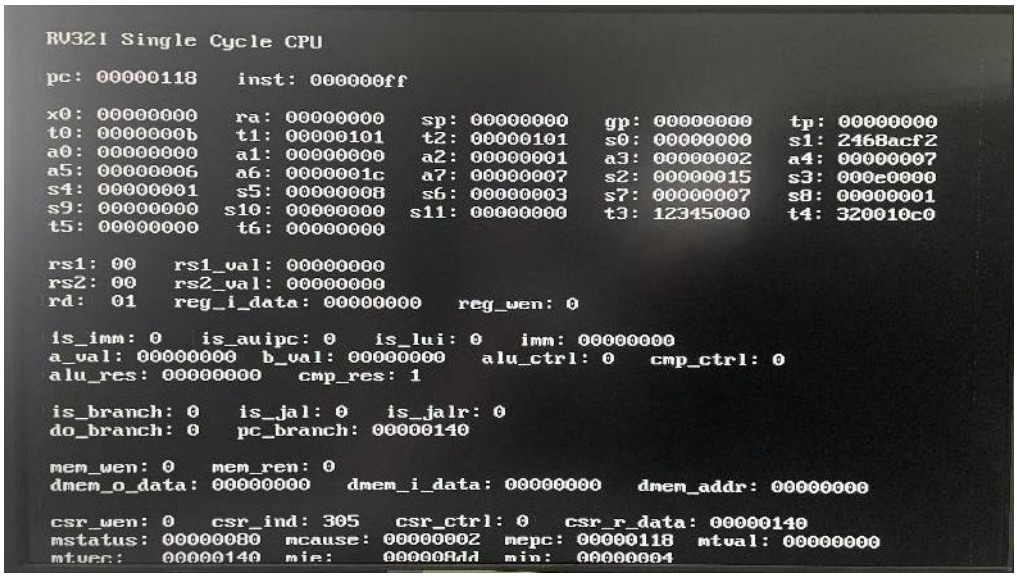
11 对中断和异常部分指令上板验证

仿真整个测试程序，可见如下结果：

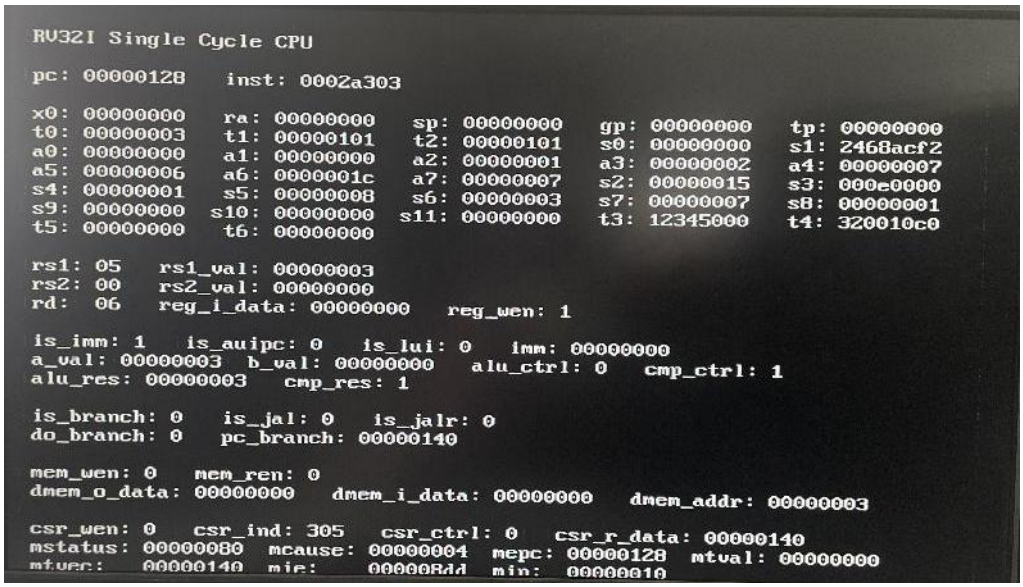


上图中，红色方框表示模拟的外部中断，橙色方框表示时钟中断，蓝色方框依次是 ecall, illegal instruction, PC misalign, load address misalign, store address misalign。

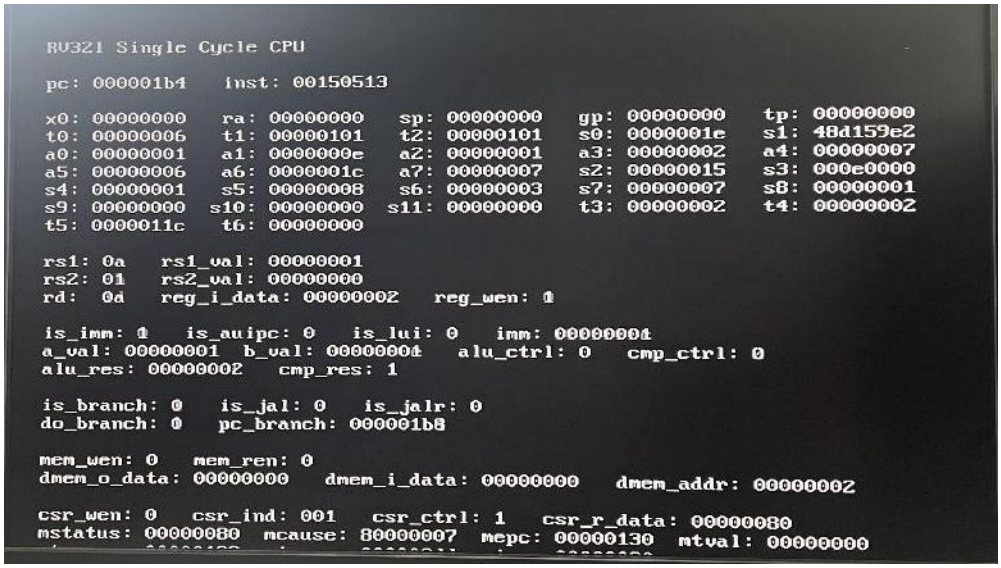
取部分异常和中断情况，上板验证情况如下（有的显示器的分辨率和助教给的 vga 似乎不匹配，因此无法显示最后一行的 mtvel, mie 和 mip，不过问题不大）



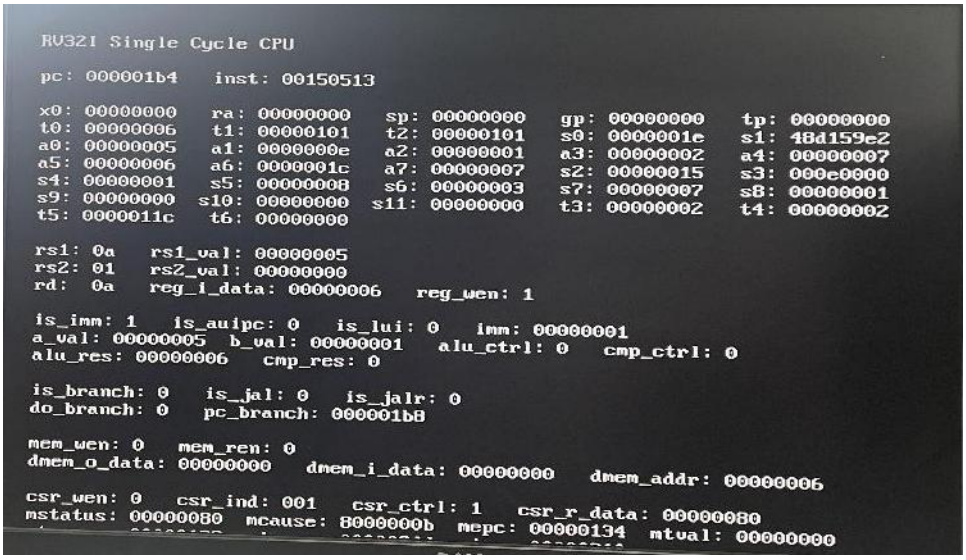
inst = 0x000000ff,非法指令异常，见下方 mcause = 2，mepc=0x118



dmem_addr = 3,lw 地址非对齐异常，见下方 mcause = 4，mepc=0x128



时钟中断处理程序执行中，见下方 mcause = 0x80000007，mepc=0x130



按下 key_x[1]，外部中断处理程序执行中，见下方 mcause = 0x8000000b，mepc=0x134

由图可知上板现象正确。

三、实验心得

这一部分主要叙述我做 Lab4 时遇到的 Bug 或困难。

问题一：IMem 和 DMem 的单位访问长度

除第 1 节中提及的问题外，我还走了一些弯路。

我最初的设计，不是在 IMem 中将 imem_addr >> 2，而是在 core 中将 pc >> 2。也即仿真时是 PC = PC + 1。这与 RISC-V ISA 设计初衷不符。除去这个问题外，在这里我还遇到非常奇异的语法 bug。如果使用

```
assign pc_branch = do_branch ? pc + $signed(imm) >>> 2 : pc + 1;
```

来对生成的立即数进行算术右移，在仿真时发现无论如何都只会逻辑右移，从而导致计算出的 PC 错误。如果折中一下，先计算出立即数/4，再加给 PC，这样仿真就能成功。

```
assign pc_offset = ($signed(imm)) >>> 32'd2;
assign pc_branch = do_branch ? pc + pc_offset : pc + 1;
```

可能 Verilog 不能在单条语句中嵌入太多复杂逻辑。

问题二：寄存器部件问题汇总。

第一，整个数据通路中的时序逻辑部件及其功能如下：

- PC，时钟正边沿时改变值

- RegFile，时钟负边沿时可能写入值(reg_wen = 1)，而读口是组合逻辑的

- DMem，时钟负边沿时可能写入值（dmem_wen = 1），而读口是组合逻辑的

- CSR，时钟负边沿时可能写入值(csr_wen = 1)，而读口是组合逻辑的

第二，关于 assign 和 always 语句。

以下是我在实践中遭遇过众多坑以后的总结：

1）组合逻辑应使用 assign 或 always@*语句，且在 always 语句中使用阻塞型赋值（=）

2）时序逻辑应使用 always@(posedge clk)语句，并最好使用非阻塞型赋值（<=）

3）不能在多个 always 语句块中对同一个 reg 变量进行赋值。这种方式在仿真时不会报错，甚至运行良好。但是在综合时会报错[DRC MDRV-1] Multiple Driver Nets。其硬件逻辑上的原因是，verilog 无法判断 always 语句会何时触发，可能会造成赋值的冒险和竞争情况。

4）在同一个 always 语句块中，对同一个 reg 变量根据不同条件进行赋值的语句，应该放置

在不可并行的 if...else if 语句块中。如下第二条语句就是错误的。因为 mip 将在不同情况下赋予不同的值，造成冒险竞争。

```
always@(posedge clk) begin
    if (rst) mip <= 0;
    //if (wen) mip <= i_data; //error!
    else if (wen) mip <= i_data;
end
```

4）不要在 always 语句内使用延时和延迟赋值语句。仿真完全正常，上板子会出大问题。

如果使用#10 语句，会造成寄存器的时钟和 vga 时钟不一致，vga 无法正常显示。

如果使用#50 语句，时钟和 vga 同步，vga 可以正常显示，但是板子会自动忽略延时赋值，导致设计的功能失效。

第三，关于阻塞型赋值(=)和非阻塞型赋值(<=)。

在设计 CSR 和异常功能之前，我从来没有注意过两种赋值的区别，因此又掉进了一个大坑，又花了很多力气修改逻辑。如下是阻塞型赋值。当时钟正边沿到来时，b 的值会赋给 a，然后 a 的值赋给 c，结果是 c = a = b。之所以称之为阻塞型赋值，是因为要先执行 a = b 才会执行 c = a，即前一条语句阻塞了后一条语句赋值。

```
always@(posedge clk) begin
    a = b;
    c = a;
end
```

非阻塞型赋值如下。在时钟正边沿到来之前，先计算好 b 和 a 的值。当时钟正边沿到来时，将原先的 b 和 a 分别同时赋给 a 和 c。这两条语句是同时进行的，因此称之为非阻塞型赋值。

```
always@(posedge clk) begin
    a <= b;
    c <= a;
end
```

第二和第三点都是在设计异常时遇到的问题。在异常发生时，首先要将 mip 相应位置 1，然后根据 mstatus.MIE & mip[i] & mie[i]来判断是否可以执行异常处理程序，如果需要处理异常则进行其他的 CSRs 的改写操作，最后还要将 mstatus.MIE 置 0。这些寄存器之间存在着对我来说比较复杂的时序逻辑依赖关系，让我比较糊涂。

第一个解决方法是将 mip 放在 always@(negedge clk)里面，首先改写，其他 CSRs 放在 always@(posedge clk)里面再改写，这样可以处理其他 CSRs 改写对 mip 的依赖关系。但是 CSR 模块同时处理 csrr 指令，always@(posedge clk)模块里也有对 mip 的赋值语句。正如第

二点的 3)所示，会有 multiple driven 错误，失败。

第二个解决方法是使用延时赋值语句。在同一个 always 块中，先对 mip 赋值，然后延时#10，再改写其他 CSRs，对于 mstatus.MIE 置 0 则使用 mstatus[3] <= #50 0，手动使其下一个时钟正边沿置 0。结果是#10 会导致 vga timing 不同步问题，不显示；延时赋值在板子上不起作用。

第三个也是最终的解决办法是阻塞型赋值和非阻塞型赋值混合使用。

- 先对 mip 进行阻塞赋值，再对其他 CSRs 非阻塞赋值。

- 设计 INT 信号，来判断是否需要处理中断/异常。

```
    if (is_exp) mip_o[CauseVal[3:0]] = 1;
    else if (is_mret) mip_o[mcause_o[3:0]] = 0;
    if (mstatus_o[3] & mie_o[CauseVal[3:0]] & mip_o[CauseVal[3:0]] |
is_mret) begin
        if (is_mret) begin
            mstatus_o[3] <= mstatus_o[7];
            mstatus_o[7] <= 1;
        end
        else begin
            mepc_o <= EPCval;
            mcause_o = CauseVal;
            mtval_o <= tval;
            mstatus_o[7] <= mstatus_o[3];
            mstatus_o[3] <= 0;
        end
    end
    INT = mstatus_o[3] & mie_o[mcause_o[3:0]] & mip_o[mcause_o[3:0]];
```

如上代码所示，首先使用阻塞型赋值将 mip 首先改写，接下来依据 mstatus_o[3] & mie_o[CauseVal[3:0]] & mip_o[CauseVal[3:0]]的情况再决定是否改写其他 CSRs。这里 mstatus, mepc, mtval 使用非阻塞型赋值，mcause 使用阻塞型赋值，是为了能够正确处理 INT 信号。

还有一个小 bug 是 CSR 模块除了处理异常，还要在当 csr_wen = 1 时处理 csrr 语句。在这两个 if else 语句当中，INT 都应该被赋值，否则执行 csrr 语句就没法修改 INT。会出现这种 bug：INT = 1 后跳转至中断处理程序，中断处理程序中第一句即为 csrrs 解析 mcause，CSR 进入 csrr 处理模块，没有修改新的 INT，导致 INT 仍然为 1，接着执行了嵌套中断，造成错误。

问题三：挂起中断的处理

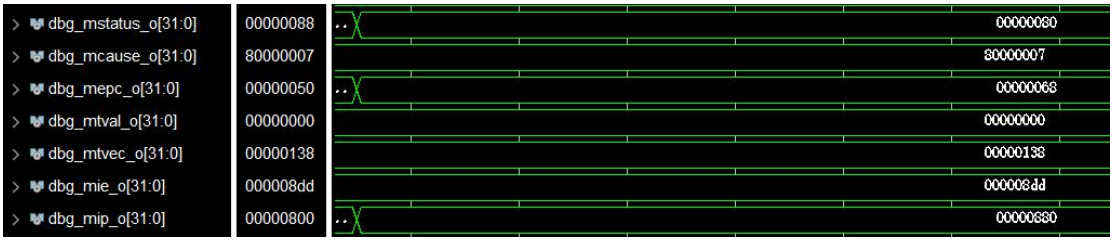
设想如下情形：在 PC = 0x110 处是 ecall 指令。遇到 ecall 后 mip[11] = 1，进入异常处理程序。此时发生时钟中断请求，mip[7] = 1，但是由于在异常程序内部 mstatus.MIE = 0，所以该中断请求被挂起。在执行完 ecall 命令之前，时钟中断信号已经消失，又出现了另一个

中断信号（如外部中断），也被挂起。当执行完 ecall 命令以后，返回 PC = 0x114。此时 mip[11] & mie[11] & mstatus.MIE = 1，应当继续处理曾经存在过的时钟中断请求。

而在我设计的逻辑里，我不是依据 mip[11] & mie[11] & mstatus.MIE = 1 来判断是否进行中断，而是用 mstatus_o[3] & mie_o[CauseVal[3:0]] & mip_o[CauseVal[3:0]]来判断的。该 CauseVal 是一个组合逻辑电路，即时钟中断信号存在时，CauseVal 为 0x80000007。如果时钟中断信号消失后，又出现了其他中断信号，CauseVal 就会被刷新。那么在上述情形中，返回 PC=0x114 时，在被挂起的时钟中断和外部中断信号中，只有外部中断信号会被处理，而时钟中断信号会被遗忘。mip[7]会一直置 1，直到下一次出现时钟中断。

我意识到我上述的逻辑有一些问题，没办法正确处理在异常处理程序中出现多次其他中断的情况。但是似乎 mip[i] & mie[i] & mstatus.MIE = 1 的中断使能逻辑也有一些问题。第一，ecall 会置 mip[11] = 1,外部中断也会置 mip[11] = 1，假如在 ecall 处理时发生外部中断，结束 ecall 会使得 mip[11] = 0，那么外部中断也被清除了，相当于没有发生。第二，类似地，假如在处理某种异常时发生外部中断使得 mip[11] = 1，结束异常处理时发现 mip[11] & mie[11] & mstatus.MIE = 1，但此时不知道是发生过 ecall 还是外部中断。归根结底是 mip 中相同编码有歧义，无法辨别发生过异常还是中断。

我的逻辑还存在一个小 bug。在发生嵌套中断（将后一个中断挂起）时，我前一个中断处理完成时 mip 不会置 0。如下图 mip = 0x880。不知道是什么原因（从代码上看理应清 0），可能还是和我没有特别理解透彻寄存器的赋值方式有关系。



问题四：各种 Bug 集锦

1）同名变量过多，忘记连其中某处线就会导致通路断连。

比如 dmem_wen 信号，在 Core.v 中有 dmem_wen 和 mem_wen 两个信号，在 Top.v 中还有 o_dmem_wen 和 i_dmem_wen，只要有一处忘记连线就不能正常传输数据。而且这种错误有些隐蔽，要无谓地 debug 很久。我在尝试 lw 指令时一直不成功，后来发现忘记在 Core.v 中将 dmem_wen = mem_wen 了。

2）新增 LED 模块时忘记把 LEDCtrl 实例加入 core_tb.v 中，导致仿真时始终无法显示 LED

信号。

3）Vivado 的语法报错提示有的不是很靠谱。模块内 begin 和 end 不配对居然会在完全不相关的地方报错。如下图。

```
Alu alu(
    .a_val(a_val),
    .b_val(b_val),
    .ctrl(alu_ctrl),
    .result(alu_res)
);
```

4）助教在给出的测试代码中使用了 la 伪指令，在用 venus 生成机器码时会自动转成 auipc。但是由于测试程序太短，offset < 0x1000，所以会转成 auipc rd, 0，导致没有成功 load address。解决办法是手动用 addi rd, x0, offset。

其他实验心得

第一，感谢助教提供的 VGA 和 define.vh，让仿真验证和上板验证都变得直观且方便。为了理清楚众多宏定义之间的关系，我写了一个 aboutDefine.md 来提示自己。

```
放在top.v
`define VGA_DBG_RegFile_Declaration \
    wire [31:0] dbg_x0; \
    wire [31:0] dbg_ra; \
    wire [31:0] dbg_sp; \
    wire [31:0] dbg_gp; \


放在Module定义中
`define VGA_DBG_RegFile_Outputs \
    output wire [31:0] dbg_x0, \
    output wire [31:0] dbg_ra, \

放在module内部
`define VGA_DBG_RegFile_Assignments \
    assign dbg_x0 = regs[0]; \
    assign dbg_ra = regs[1]; \

放在调用regfile模块的形参表中
`define VGA_DBG_RegFile_Arguments \
    .dbg_x0(dbg_x0), \
    .dbg_ra(dbg_ra), \
    .dbg_sp(dbg_sp), \
```

第二，感谢 vscode 和 Venus 插件，让我不必再花费时间将汇编代码转成机器码。生成的 assembly 文件将 PC、机器码和汇编指令相对应，给 debug 提供了极大的方便。

```
0x00000010      0x30551073      csrrw x0 773 x10
0x00000014      0x00800513      addi x10 x0 8
0x00000018      0x00451593      slli x11 x10 4
0x0000001C      0x00B56533      or x10 x10 x11
0x00000020      0x00459593      slli x11 x11 4
0x00000024      0x00B56533      or x10 x10 x11
0x00000028      0x05550513      addi x10 x10 85
0x0000002C      0x30451073      csrrw x0 772 x10
0x00000030      0x0FF00413      addi x8 x0 255
0x00000034      0x30046473      csrrsi x8 768 8
```

我甚至还写了一份 C++代码，使用重定向来将第二列的机器码提取出来。

```cpp
int main() {
    string s;
    while (getline(cin, s)) {
        istringstream iss(s);
        string inst;
        iss >> inst >> inst;
        inst.erase(inst.begin(), inst.begin()+2);
        cout << inst << endl;
    }
    return 0;
}
```

```
g++ trim.cpp
gc in.txt | ./a.exe | Out-File out.txt
```

第三，Verilog 语言入门很简单，但是如果没有深入了解其语法特性，最后坑害的都是自己。正如问题二所说，有些指令仿真有效，综合时就会报各种各样的错误。有些指令不会报错，上板时就莫名其妙地不起作用。

第四，很多指令逻辑上我完全理解，但是实验时就会出现各种意外差错，尤其是一些隐秘的低级的错误很难发现，会浪费大量时间在找这些错误上。

从开始做实验到写好这份实验报告大约花了 36 个小时，基本 SCPU 指令大约花费 11 个小时，CSR 到中断异常花费 20 多小时，写报告花费 4-5 个小时。5 月 1 日这一天仿真了 120 多次，累计仿真次数可能多达 300 次。由此可见，写出一个看起来非常简单的单周期 CPU 其实是非常不容易的。接下来的流水线 CPU 实验，我还要面临更大的挑战。