实验2 - 流水线异常和中断设计

1 实验步骤

本次要写的代码基本在 `ExceptionUnit.v` 中，里面实现了 CSR 指令处理和异常处理。

1.1 CSR 指令处理

根据已经给出的框架代码可以看到，CSR 的读写均是在 MEM 阶段进行的。具体代码如下：

```
    // 处理 CSR 指令
    always @(*) begin
        if(rst) begin
            csr_raddr = 0;
            csr_waddr = 0;
            csr_wdata = 0;
            csr_w = 0;
            csr_wsc = 0;
        end
        else begin
            if(csr_rw_in) begin
                csr_raddr = csr_rw_addr_in;
            end
            else csr_raddr = 0;

            if(csr_rw_in && csr_wsc_mode_in != 2'b00) begin
                csr_waddr = csr_rw_addr_in;
                if(csr_w_imm_mux) begin
                    csr_wdata = csr_w_data_imm;         // 截断，然后 0 扩展
                end
                else begin
                    csr_wdata = csr_w_data_reg;
                end
                csr_w = 1;
                csr_wsc = csr_wsc_mode_in;
            end
            else begin
                csr_waddr = 0;
                csr_wdata = 0;
                csr_w = 0;
                csr_wsc = 0;
            end
        end
    end
```

CSR 指令的处理比较简单，根据输入的 `csr_rw_in` 和 `csr_wsc_mode_in` 来决定是否进行 CSR 操作，以及是读还是写。如果要进行读操作，那么就将 `csr_raddr` 设置为输入的 `csr_rw_addr_in`，否则就将其置零。如果要进行写操作，那么就将 `csr_waddr` 设置为输入的 `csr_rw_addr_in`，将 `csr_wdata` 设置为输入的 `csr_w_data_imm` 或 `csr_w_data_reg`（根据 `csr_w_imm_mux` 区分是否使用立即数），将 `csr_w` 设置为 1，将 `csr_wsc` 设置为输入的 `csr_wsc_mode_in`，否则就将它们都置零。

1.2 异常检测及处理

根据已经给出的框架代码可以看到，异常检测及处理均是在 WB 阶段（除了 `mret` 在 MEM 阶段）进行的。
异常处理的思路是

- 如果检测到当前在执行 `ecall` 或者出现访问地址越界、非法指令格式或者出现中断信号的情况，说明我们需要进行异常（中断）处理。
  - 同时修改 `mepc`，`mcause`，`mtval`，`mstatus` 特权寄存器的值，并将下一条执行的 `PC` 设为 `mtvec`，即异常处理程序的入口。
  - 因为我们是在 WB 阶段才会进行检测，所以我们需要 flush 掉之前阶段的所有指令，即将 `FD_flush`，`DE_flush`，`EM_flush`，`MW_flush` 都置为 1。
- 如果检测到当前在执行 `mret` 指令，说明我们需要进行异常（中断）返回。
  - 同时修改 `mepc`，`mstatus` 特权寄存器的值，并将下一条执行的 `PC` 设为 `mepc`。
  - 同理，需要将 `FD_flush`，`DE_flush`，`EM_flush`，`MW_flush` 都置为 1 进行 flush。

具体代码如下：

- 修改 `mepc`（`ExceptionUnit.v` 中）：
  值得注意是的，如果是中断，那么我们的 `mepc` 应该设为下一条执行的 PC，否则应该设为当前要执行的 PC。

```
    always @(*) begin
        if (rst) mepc = 0;
        else begin
            if (interrupt) mepc = epc_next;
            else if (illegal_inst || l_access_fault || s_access_fault ||
ecall_m) mepc = epc_cur;
            else mepc = mepc;
        end
    end
```

- 修改 `mcause`，`mtval`，`mstatus`（`CSRRegs.v` 中）：
  因为异常处理模块可能用到 `mtvec`，`mepc` 的值，因此我们直接将其作为 CSR 模块的输出接出来。此外当异常发生或者 `mret` 指令执行时，我们需要修改 `mcause`，`mtval`，`mstatus` 的值，这里我们直接向 CSR 寄存器堆中修改。
  此外，我们添加了一个寄存器用来记录当前程序的特权级，即 `now_privilege`。

```
...     // 在文件中添加了下面这些端口，以便我们同步读写这些寄存器
```

```
input Exception,
input Interrupter,
input is_mret,
input [31:0] mepc,
input [31:0] mcause,
input [31:0] mtval,
output [31:0] mtvec,
output [31:0] mepc_out
...
reg [1:0] now_privilege;
assign mtvec = CSR[5];
assign mepc_out = CSR[9];
always@(posedge clk) begin
    if(rst) begin
        ...
    end
    else if(csr_w) begin
        ...
    end
    else begin
        if (Exception || is_mret) begin
            CSR[9] <= mepc;
            CSR[10] <= mcause;
            CSR[11] <= mtval;
        end
        if (Exception) begin
            CSR[0][3] <= 0;
            CSR[0][7] <= CSR[0][3];
            CSR[0][12:11] <= now_privilege;
            now_privilege <= CSR[0][12:11];
        end
        else if (is_mret) begin
            CSR[0][3] <= CSR[0][7];
            CSR[0][7] <= 1;
            now_privilege <= CSR[0][12:11];
            CSR[0][12:11] <= 3;
        end
    end
end
```

此外，我们在 `ExceptionUnit.v` 中也需要进行相应的修改：

- 根据异常来源设置 `mcause`：外部中断为 0x8000_000B，非法指令为 2，load 访问地址越界为 5，store 访问地址越界为 7，`ecall` 为 11。

- 根据异常类型设置 `mtval`：非法指令就把当前指令的值写入，load 和 store 就把越界访问的地址写入，其他情况就把当前指令 PC 值写入。
    - 为了能让 `mtval` 输出越界访问的地址和当前指令的值，我们需要在流水线集成中将对应的值传入异常检测模块。

```
// ExceptionUnit.v
...
input [31:0] addr,
input [31:0] inst,
...
// RV32core.v
ExceptionUnit exp_unit(...
.inst(inst_WB),
.addr(ALUout_WB),...);
```

- 设置 `reg_PC_redirect` 和 `reg_redirect_mux`：如果是中断或者异常，我们需要把 `reg_PC_redirect` 设为 `mtvec`，并将 `reg_redirect_mux` 设为 1。如果是 `mret` 指令我们需要把 `reg_PC_redirect` 设为 `mepc`，并将 `reg_redirect_mux` 设为 1。其他时候这两个信号应为 0.

```
always @(*) begin
    if (Exception) begin
        reg_PC_redirect = mtvec;
        reg_redirect_mux = 1;
        // 非法指令，val 设为 inst。访问异常，val 设为 addr。其他，val 设为 epc_cur
        if (illegal_inst) mtval = inst;
        else if (l_access_fault || s_access_fault) mtval = addr;
        else mtval = epc_cur;
        // 设置 mcause
        if (interrupt) mcause = 32'h8000_000B;
        else if (illegal_inst) mcause = 2;
        else if (l_access_fault) mcause = 5;
        else if (s_access_fault) mcause = 7;
        else if (ecall_m) mcause = 11;
    end
    else if (mret) begin
        reg_PC_redirect = mepc_out;
        reg_redirect_mux = 1;
        mcause = 0;
        mtval = 0;
    end
    else begin
```

```
        reg_redirect_mux = 0;
        mcause = 0;
        mtval = 0;
        reg_PC_redirect = 0;
    end
end
```

- 处理 flush 的问题（`ExceptionUnit.v` 中）：
  值得注意的是，并非产生了外部中断信号就可以开启中断，还需要保证 `mstatus[MIE]` 位置为 1，即允许中断才可以。

```
assign Exception = illegal_inst || l_access_fault || s_access_fault || ecall_m || Interrupter;
assign Interrupter = interrupt && mstatus[3];

reg FD_flush, DE_flush, EM_flush, MW_flush, reg_RegWrite_cancel;
assign reg_FD_flush = FD_flush;
assign reg_DE_flush = DE_flush;
assign reg_EM_flush = EM_flush;
assign reg_MW_flush = MW_flush;
assign RegWrite_cancel = reg_RegWrite_cancel;

always @(*) begin
    if (rst) begin
        FD_flush = 0;
        DE_flush = 0;
        EM_flush = 0;
        MW_flush = 0;
        reg_RegWrite_cancel = 0;
    end
    if (Interrupter) begin
        FD_flush = 0;
        DE_flush = 0;
        EM_flush = 0;
        MW_flush = 0;
        reg_RegWrite_cancel = 0;
    end
    else if ((!Interrupter && Exception) || mret) begin
        FD_flush = 1;
        DE_flush = 1;
        EM_flush = 1;
        MW_flush = 1;
        reg_RegWrite_cancel = 1;
    end
```

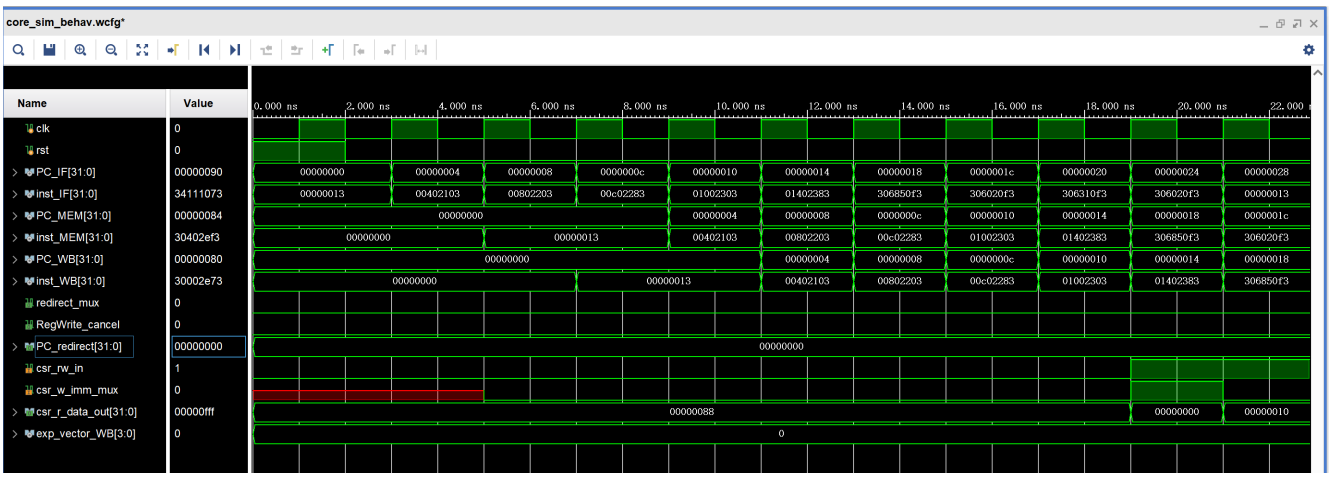
```
    else begin
        FD_flush = 0;
        DE_flush = 0;
        EM_flush = 0;
        MW_flush = 0;
        reg_RegWrite_cancel = 0;
    end
end
```

2 实验评估

2.1 仿真

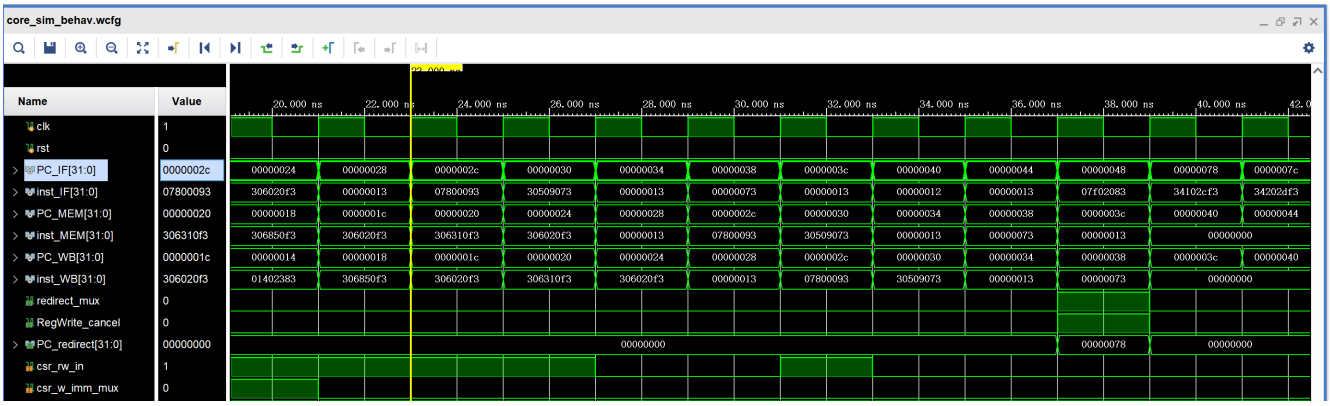
这里我们使用实验框架给出的代码进行仿真。

```
addi x0, x0, 0          # PC=0x0
lw x2, 4(x0)
lw x4, 8(x0)
lw x5, 12(x0)
lw x6, 16(x0)
lw x7, 20(x0)
csrrwi x1, 0x306, 16
csrr x1, 0x306
csrrw x1, 0x306, x6
csrr x1, 0x306
addi x0, x0, 0          # PC=0x28
```



可以看到，19ns 时 PC_MEM=18，此时 `csrrwi x1, 0x306, 16` 处于 MEM 阶段，准备将 `mstatus` 的值设为 16，即禁止中断。此时 `csr_rw_in` 和 `csr_w_imm_mux` 均为 1。下个周期时（21ns） `csrr x1, 0x306` 处于 MEM 阶段，我们将 `mstatus` 的值读出，发现 `csr_r_data` 值为 16，说明我们写入成功。

```
csrr x1, 0x306          # PC=0x24
addi x0, x0, 0
addi x1, x0, 120
csrw 0x305, x1
addi x0, x0, 0
ecall
addi x0, x0, 0
addi x0, x0, 0 # change to illegal
addi x0, x0, 0
lw   x1, 127(x0)        # PC=0x48
```

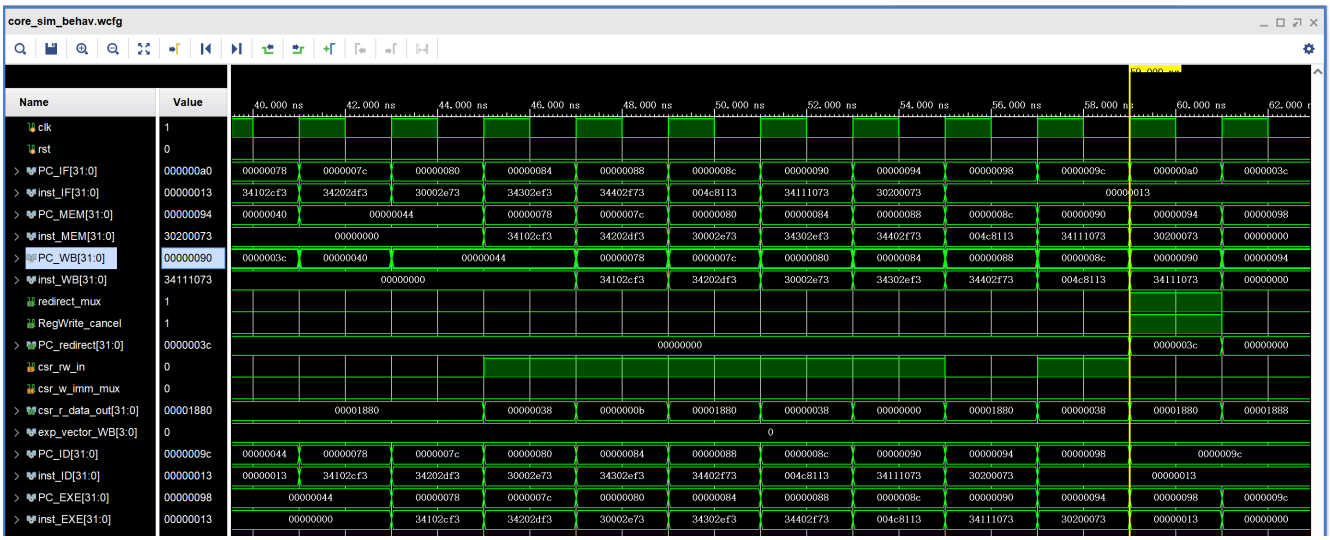


可以看到，23ns 时 PC_MEM=0x20，此时 `csrrw x1, 0x306, x6` 处于 MEM 阶段，准备将寄存器 `x6` 的值放入 `mstatus` 寄存器。此时 `csr_rw_in` 为 1。结合之前代码并查看 RAM 内存可知 `x6` 的值为 0xFFFF0000，因此下个周期（25ns）`csrr x1, 0x306` 处于 MEM 阶段，我们将 `mstatus` 的值读出，发现 `csr_r_data` 值为 0xFFFF0000，说明我们写入成功。

接着我们用相同的方式将 `x1` 的值写入 `mtvec`（31ns），此时 `x1` 的值为 0x78，即我们异常处理程序的首地址。随后 37ns 时，`ecall` 处于 WB 阶段（PC_WB=0x38），此时我们要跳到异常处理程序，因此可以看到 `redirect_mux` 为 1，而对应的 `PC_redirect` 的值也为 0x78，即我们要跳去的地址。此外我们还看到 `RegWrite_cancel` 为 1，说明此时我们会进行 flush 操作。

```
csrr x25, 0x341  # mepc      # PC=0x78
csrr x27, 0x342  # mcause
csrr x28, 0x300  # mstatus
csrr x29, 0x304  # mie
csrr x30, 0x344  # mip
addi x2, x25, 4
csrw 0x341, x2
mret
addi x0, x0, x0
addi x0, x0, x0
addi x0, x0, x0             # PC=0xA0

```

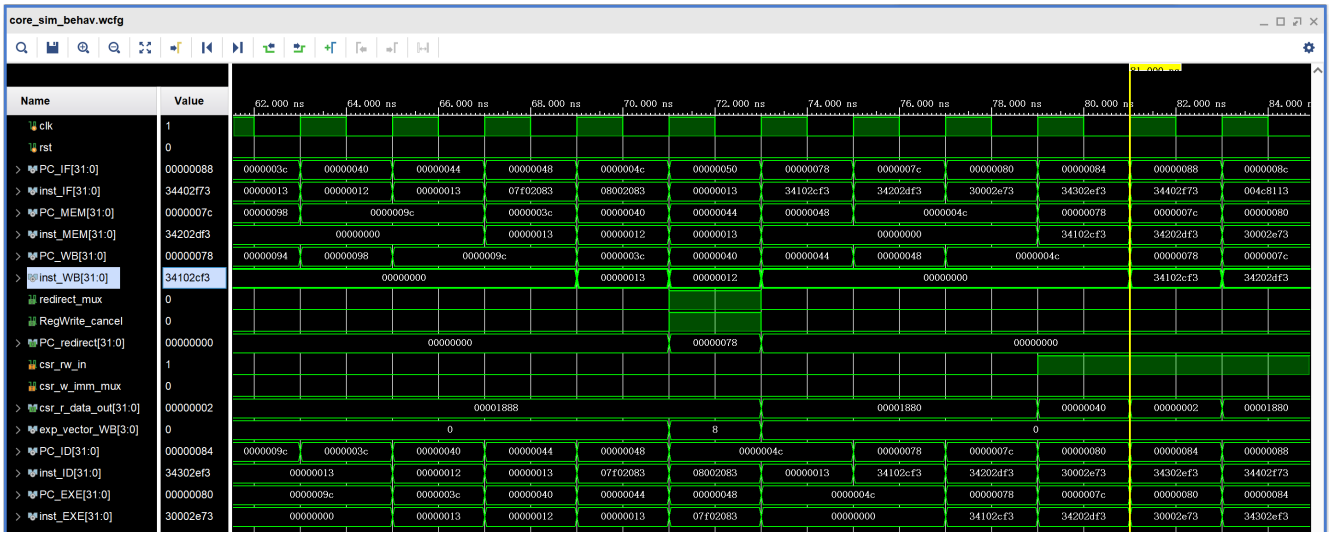


上面的分析中，我们知道程序是从 PC=0x38 跳到 PC=0x78 的，但在 jump 的同时我们已经取指到了 0x44，即此时我们有。

从 45ns 开始到 55ns，依次是读取 `mepc`，`mcause`，`mstatus`，`mtval`，`mip` 的 CSR 的指令进入到了 MEM 阶段。通过 `csr_r_data_out` 的值我们可以看到对应寄存器的值，比如 `mepc` 存放是 0x38，即 `ecall` 指令的 PC，又比如 `mcause` 为 0xb 与 `ecall` 对应的信息相符。`mstatus` 为 0x1880，其中 MPP 为 11（因为我们这里只有 M 模式）。`mtval` 存放的是 0x38（即 `ecall` 的下一条指令地址）

随后 57ns 时，`csrw 0x341, x2` 处于 MEM 阶段，我们将 `x2` 的值存回 `mepc`。（`mepc←x2=mepc+4=0x38+4=0x3c`）59ns 时，`mret` 处于 MEM 阶段，可以看到此时 `redirect_mux` 和 `RegWrite_cancel` 被设为 1，而 `PC_redirect` 的值为 0x3c，而下一条取指的结果就是 0x3c。

```
addi x0, x0, 0 # change to illegal      # PC=0x40
addi x0, x0, 0
lw   x1, 127(x0)
lw   x1, 128(x0)
addi x0, x0, 0                          # PC=0X50
```



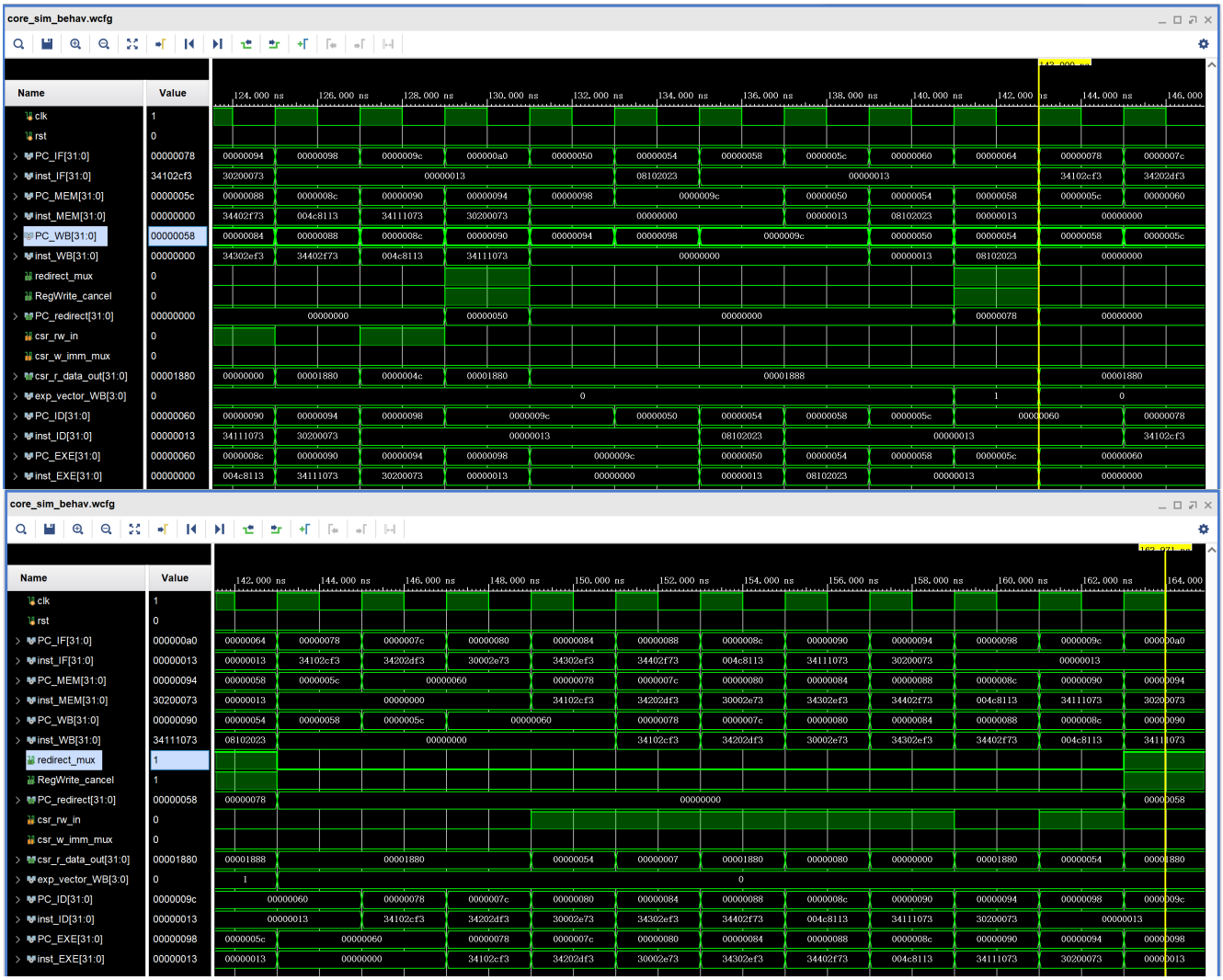
这里 0x40 被改为了一条非法的指令（可以看到 inst 内容为 0x00000012），因此在 71ns 时，我们检测到了这个异常并准备跳去处理程序，因此可以看到 `redirect_mux`，`RegWrite_cancel` 和 `PC_redirect` 和之前 `ecall` 时相同，设为了对应的值。随后我们进入处理程序，依次是读取 `mepc`，`mcause`，`mstatus`，`mtval`。可以看到此时的 `mepc` 为 0x40，`mcause` 为 2，`mstatus` 为 0x1880，`mtval` 为 0x12 即非法指令的值，均符合预期。

```
1  lw   x1, 127(x0)                    # PC=0x48
2  lw   x1, 128(x0) # l access fault
3  addi x0, x0, 0
4  sw   x1, 128(x0) # s access fault
5  addi x0, x0, 0
6  addi x0, x0, 0                      # PC=0x5C
```

处理程序里面的过程同前，这里不再重复，最后我们调用 `mret` 回到 `mepc+4=0x44` 的地方继续执行。在 107ns 时，`lw x1, 128(x0)` 处于 WB 阶段，而这条指令访问的地址超过了范围，因此我们需要再次进入处理程序处理这个异常。进入处理程序后我们观察 `mepc`，`mcause`，`mstatus`，`mtval` 的值。可以看到 `mepc` 为 0x4c，`mcause` 为 0x5，`mstatus` 为 0x1880，`mtval` 为 0x80（即 128）均符合预期。129ns 时我们通过 `mret`

准备跳回到 `mepc+4` 的位置，即 0x50。

```
sw   x1, 128(x0) # s access fault       # PC=0x54
addi x0, x0, 0
addi x0, x0, 0
addi x0, x0, 0
addi x0, x0, 0                          # PC=0x64
```



这里我们跳回到正常程序后，可以看到 141ns 时 PC_WB=0x54，即 `sw x1, 128(x0)`，此时访问了非法的地址范围，故要进入异常处理程序。随后我们观察 `mepc`，`mcause`，`mstatus`，`mtval` 的值，可以看到 `mepc` 为 0x54，`mcause` 为 0x7，`mstatus` 为 0x1880，`mtval` 为 0x80 均符合预期。163ns 时我们通过 `mret` 准备跳回到 `mepc+4` 的位置，即 0x58。

后续过程与异常无关，这里不再赘述。

2.2 上板结果

0x38 处有一条 ecall 指令，于是在 `WB_PC=0x44` 时后跳入中断处理程序。

```
WB_PC  =0x00000044
WB_INST=0x00000000
MEMADDR=0x00000001
MEMDATA=0x00000001
x01=0x00000078
x02=0x00000008
x03=0x00000000
x04=0x00000010
x05=0x00000014
x06=0xFFFF0000
x07=0x0FFF0000
x08=0x00000000
x09=0x00000000
x10=0x00000000
x11=0x00000000
x12=0x00000000
x13=0x00000000
x14=0x00000000
x15=0x00000000
x16=0x00000000
x17=0x00000000
x18=0x00000000
x19=0x00000000
x20=0x00000000
x21=0x00000000
x22=0x00000000
x23=0x00000000
x24=0x00000000
x25=0x00000038
x26=0x00000000
x27=0x00000000
x28=0x00000000
x29=0x00000000
x30=0x00000000
x31=0x00000000
WB_PC  =0x00000078
WB_INST=0x34102CF3
MEMADDR=0x00000001
MEMDATA=0x00000001
```

`WB_PC=0x88` 时可以看到，`mepc` 的值在 `x25` 中，为 0x38；`mcause` 的值在 `x27` 中，为 0xb；`mstatus` 的值在 `x28` 中，为 0x1880；`mtval` 的值在 `x29` 中，为 0x38；`mip` 的值在 `x30` 中，为 0x0。

```
x25=0x00000038
x26=0x00000000
x27=0x0000000B
x28=0x00001880
x29=0x00000038
x30=0x00000000
x31=0x00000000
WB_PC  =0x00000088
WB_INST=0x34402F73
MEMADDR=0x00000001
MEMDATA=0x00000001
```

处理程序结束后，可以看到 `x2` 为 0x3c，最后我们也回到了 `mepc+4` 的位置，即 0x3c。

```
x01=0x00000078
x02=0x0000003C
x03=0x00000000
x04=0x00000010
x05=0x00000014
x06=0xFFFF0000
x07=0x0FFF0000
x08=0x00000000
x09=0x00000000
x10=0x00000000
x11=0x00000000
x12=0x00000000
x13=0x00000000
x14=0x00000000
x15=0x00000000
x16=0x00000000
x17=0x00000000
x18=0x00000000
x19=0x00000000
x20=0x00000000
x21=0x00000000
x22=0x00000000
x23=0x00000000
x24=0x00000000
x25=0x00000038
x26=0x00000000
x27=0x0000000B
x28=0x00001880
x29=0x00000038
x30=0x00000000
x31=0x00000000
WB_PC  =0x0000003C
WB_INST=0x00000013
MEMADDR=0x00000001
MEMDATA=0x00000001
```

0x40 处有一条非法指令，于是我们在 `WB_PC=4c` 时跳入中断处理程序。

`WB_PC=0x88` 时可以看到，`mepc` 的值在 `x25` 中，为 0x40；`mcause` 的值在 `x27` 中，为 0x2；`mstatus` 的值在 `x28` 中，为 0x1880；`mtval` 的值在 `x29` 中，为 0x12；`mip` 的值在 `x30` 中，为 0x0。

```
x25=0x00000040
x26=0x00000000
x27=0x00000002
x28=0x00001880
x29=0x00000012
x30=0x00000000
x31=0x00000000
WB_PC  =0x00000088
WB_INST=0x34402F73
MEMADDR=0x00000001
MEMDATA=0x00000001
```

处理程序结束后，我们也回到了 `mepc+4` 的位置（存在 `x2` 中），即 0x44。

```
x01=0x00000078
x02=0x00000044
x03=0x00000000
x04=0x00000010
x05=0x00000014
x06=0xFFFF0000
x07=0x0FFF0000
x08=0x00000000
x09=0x00000000
x10=0x00000000
x11=0x00000000
x12=0x00000000
x13=0x00000000
x14=0x00000000
x15=0x00000000
x16=0x00000000
x17=0x00000000
x18=0x00000000
x19=0x00000000
x20=0x00000000
x21=0x00000000
x22=0x00000000
x23=0x00000000
x24=0x00000000
x25=0x00000040
x26=0x00000000
x27=0x00000002
x28=0x00001880
x29=0x00000012
x30=0x00000000
x31=0x00000000
WB_PC  =0x00000044
WB_INST=0x00000013
MEMADDR=0x00000001
MEMDATA=0x00000000
```

0x4c 处有一条访问越界的 load 指令，于是我们在 `WB_PC=0x58` 时跳入中断处理程序。

`WB_PC=0x88` 时可以看到，`mepc` 的值在 `x25` 中，为 0x4c；`mcause` 的值在 `x27` 中，为 0x5；`mstatus` 的值在 `x28` 中，为 0x1880；`mtval` 的值在 `x29` 中，为 0x80；`mip` 的值在 `x30` 中，为 0x0。

```
x25=0x0000004C
x26=0x00000000
x27=0x00000005
x28=0x00001880
x29=0x00000080
x30=0x00000000
x31=0x00000000
WB_PC  =0x00000088
WB_INST=0x34402F73
MEMADDR=0x00000001
MEMDATA=0x00000001
```

处理程序结束后，我们也回到了 `mepc+4` 的位置（存在 `x2` 中），即 0x50。

```
x01=0x0080BF00
x02=0x00000050
x03=0x00000000
x04=0x00000010
x05=0x00000014
x06=0xFFFF0000
x07=0x0FFF0000
x08=0x00000000
x09=0x00000000
x10=0x00000000
x11=0x00000000
x12=0x00000000
x13=0x00000000
x14=0x00000000
x15=0x00000000
x16=0x00000000
x17=0x00000000
x18=0x00000000
x19=0x00000000
x20=0x00000000
x21=0x00000000
x22=0x00000000
x23=0x00000000
x24=0x00000000
x25=0x0000004C
x26=0x00000000
x27=0x00000005
x28=0x00001880
x29=0x00000080
x30=0x00000000
x31=0x00000000
WB_PC  =0x00000050
WB_INST=0x00000013
MEMADDR=0x00000000
MEMDATA=0x00000000
```

0x54 处有一条访问越界的 store 指令，现象类似，这里不再重复。

0x74 是一条 jump 指令，跳回到程序开头，可以看到 `WB_PC=0x74` 下一条的 `WB_PC=0x0` 。

```
WB_PC  =0x00000074
WB_INST=0x00000000
MEMADDR=0x00000001
MEMDATA=0x00000001
x01=0x0080BF00
x02=0x00000058
x03=0x00000000
x04=0x00000010
x05=0x00000014
x06=0xFFFF0000
x07=0x0FFF0000
x08=0x00000000
x09=0x00000000
x10=0x00000000
x11=0x00000000
x12=0x00000000
x13=0x00000000
x14=0x00000000
x15=0x00000000
x16=0x00000000
x17=0x00000000
x18=0x00000000
x19=0x00000000
x20=0x00000000
x21=0x00000000
x22=0x00000000
x23=0x00000000
x24=0x00000000
x25=0x00000054
x26=0x00000000
x27=0x00000007
x28=0x00001880
x29=0x00000080
x30=0x00000000
x31=0x00000000
WB_PC  =0x00000000
```

随后在 `WB_PC=0x4` 时我们打开 SW[12]，即中断信号，可以看到在 `WB_PC=0x14` 之后我们跳入了中断处理程序。

`WB_PC=0x88` 时可以看到， `mepc` 的值在 `x25` 中，为 0x8； `mcause` 的值在 `x27` 中，为 0xb； `mstatus` 的值在 `x28` 中，为 0x1880； `mtval` 的值在 `x29` 中，为 0x4； `mip` 的值在 `x30` 中，为 0x0。

```
x25=0x00000008
x26=0x00000000
x27=0x8000000B
x28=0x00001880
x29=0x00000004
x30=0x00000000
x31=0x00000000
WB_PC  =0x00000088
WB_INST=0x34402F73
MEMADDR=0x00000001
MEMDATA=0x00000001
```

处理程序结束后，我们也回到了 `mepc+4` 的位置（存在 `x2` 中），即 0xc。

```
x01=0x0080BF00
x02=0x0000000C
x03=0x00000000
x04=0x00000010
x05=0x00000014
x06=0xFFFF0000
x07=0x0FFF0000
x08=0x00000000
x09=0x00000000
x10=0x00000000
x11=0x00000000
x12=0x00000000
x13=0x00000000
x14=0x00000000
x15=0x00000000
x16=0x00000000
x17=0x00000000
x18=0x00000000
x19=0x00000000
x20=0x00000000
x21=0x00000000
x22=0x00000000
x23=0x00000000
x24=0x00000000
x25=0x00000008
x26=0x00000000
x27=0x8000000B
x28=0x00001880
x29=0x00000004
x30=0x00000000
x31=0x00000000
WB_PC  =0x0000000C
WB_INST=0x00C02283
MEMADDR=0x00000001
MEMDATA=0x00000001
```

此时我们没有关闭开关，即中断信号仍然被使能，因此 `WB_PC=0xc` 后下一条指令我们又回到了中断处理程序。

```
WB_PC  =0x0000009C
WB_INST=0x00000000
MEMADDR=0x00000000
MEMDATA=0x00000000
x01=0x0080BF00
x02=0x0000000C
x03=0x00000000
x04=0x00000010
x05=0x00000014
x06=0xFFFF0000
x07=0x0FFF0000
x08=0x00000000
x09=0x00000000
x10=0x00000000
x11=0x00000000
x12=0x00000000
x13=0x00000000
x14=0x00000000
x15=0x00000000
x16=0x00000000
x17=0x00000000
x18=0x00000000
x19=0x00000000
x20=0x00000000
x21=0x00000000
x22=0x00000000
x23=0x00000000
x24=0x00000000
x25=0x00000008
x26=0x00000000
x27=0x8000000B
x28=0x00001880
x29=0x00000004
x30=0x00000000
x31=0x00000000
WB_PC  =0x0000000C
WB_INST=0x00C02283
MEMADDR=0x00000001
MEMDATA=0x00000001
```

后续过程略。

3 思考题

1. 精确异常和非精确异常的区别是什么？

   精确异常是指异常发生时，PC 指向的指令不会被执行，而非精确异常是指异常发生时，PC 指向的指令会被执行。

2. 阅读测试代码，第一次导致trap的指令是哪条？trap之后的指令做了什么？如果实现了U mode，并以U mode从头开始执行测试指令，会出现什么新的异常？

   第一次导致 trap 的指令是 `ecall`（`PC=0x38`），trap 之后指令先依次读取 `mepc, mcause, mstatus, mtval, mip` 的值到通用寄存器，随后将 `x25`（读的 `mepc`）的值加 4 并存回到 `mepc` 中，最后 `mret` 返回。

如果实现了 U mode，在 `PC=0x18`（`csrrwi`）时会出现越级访问，即访问了 M mode 的 CSR 寄存器，因此会出现访问异常。

3. 为什么异常要传到最后一段即WB段后，才送入异常处理模块？可不可以一旦在某一段流水线发现了异常就送入异常处理模块，如果可以请说明异常处理模块应该如何处理异常；如果不可以，请说明理由。

   我认为可以，但是需要有其他操作。因为在异常发生的前面指令，可能会修改 CSR 的相关值（如 `mtvec`），进而影响到异常的处理。因此需要等待前面的指令执行结束后才能送入异常。在 WB 阶段送异常处理模块可以做到这一点；如果发现了异常就送入异常处理模块，我们需要若干拍 stall，以确保在这之前的指令都已经执行完毕。