1. 实验目的和要求

Purpose

- Understand the principle of piplines that support multicycle operations.
- Master the design methods of piplines that support multicycle operations.
- Master verification methods of pipelined CPU supporting multicycle operations.

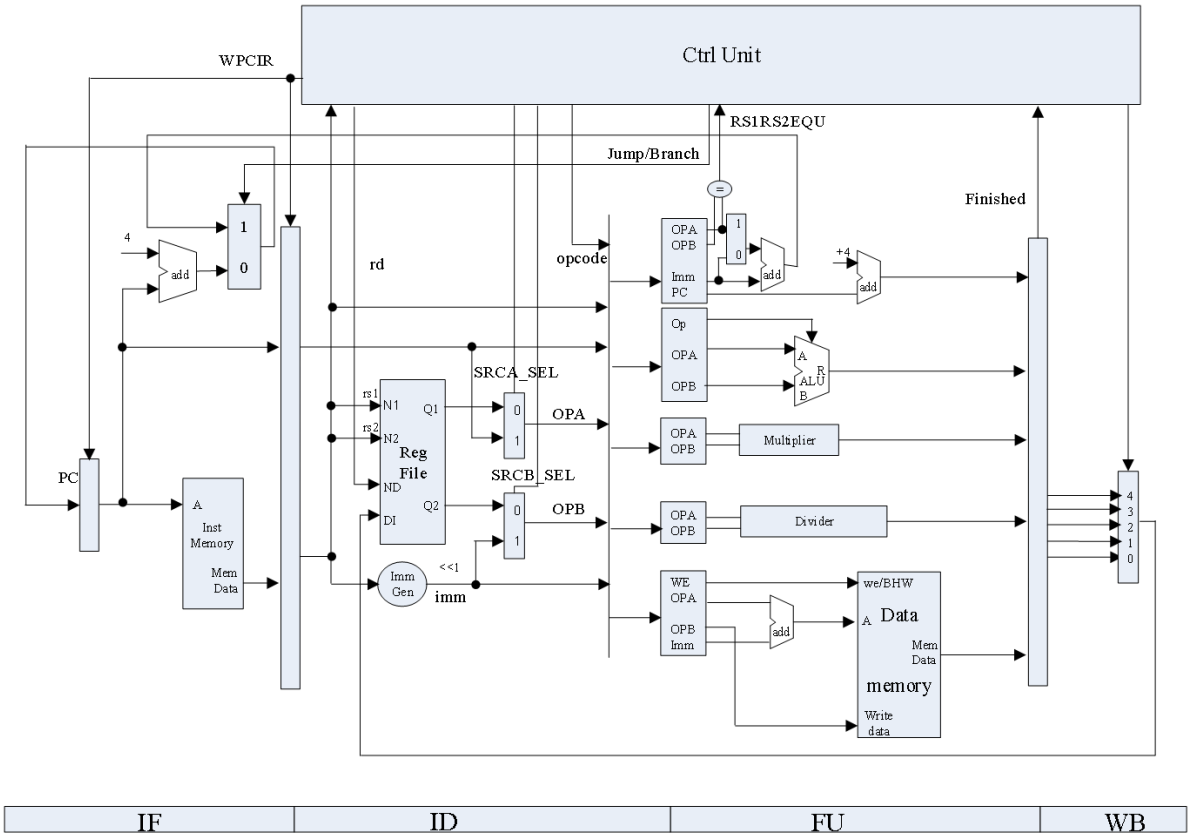
Task

- Redesign the pipelines with IF/ID/FU/WB stages and FU stage supporting multicycle operations.
- Redesign of CPU Controller.
- Verify the Pipelined CPU with program and observe the execution of program.

2. 实验内容和原理

Pipeline

The whole system is divided into 4 stages: Instruciotn Fetch, Instruction Decode, Function Unit and Writeback. The modules of execution and memory stages in the previous pipeline are reformed into separate function units.



Hazards

- Data hazards & Structural hazards
  - only after an instruciton finishes its FU and WB stages can the next instruction enters FU stage
- Control hazards
  - predict-not-taken
  - if a branch is taken, instructions in IF and ID should be flushed

3. 实验过程和数据记录

FU_ALU.v

The implementation of an ALU has been given so we just need to assign values to the operands and control signal. The ALU needs just one cycle to finish everything.

```
always@(posedge clk) begin
    if(EN & ~state) begin // state == 0
        //to fill sth.in
        A <= ALUA;
        B <= ALUB;
        Control <= ALUControl;
        state <= 1;
    end
    else state <= 0;
end
```

FU_mul.v

Simply assign values to operands and we need to set `state`. We should notice that the multiplier needs 7 cycles to obtain the result and singal `finish = state[0] == 1'b1`. When operands come, set `state` to 7'b1000000 and shift right 1 bit for every cycle spent calculating. In such manner, `finish` is set to 1 when calculation ends.

```
always @(posedge clk) begin
    if(EN & ~|state)begin   // enable == 1 && state == 0
        A_reg <= A;
        B_reg <= B;
        state <= 7'b1000000;
    end
    else
        state <= state >> 1;

end
```

FU_div.v

SImilarly, we assign values to operands. Since the divider module requires valid bits, we set valid bits as well. When calculation ends, we reset the valid bits.

```
always@(posedge clk) begin
    if(EN & ~state) begin // state == 0
        A_reg <= A;
        B_reg <= B;
        A_valid <= 1;
        B_valid <= 1;
        state <= 1;
    end
    else if(res_valid) begin
        A_valid <= 0;
        B_valid <= 0;
        state <= 0;
    end
end
```

FU_jump.v

At first, we assign values to each register. Then, since this module needs to judge the branch condtion and calculate target address, we need sub-modules to calculate target address, return address and judge branch condition. Specifically, for target address, if it is a `JALR`, we should use data of `rs1` to calculate target address. Otherwise, value of PC is used.

```
always@(posedge clk) begin
    if(EN & ~state) begin
        JALR_reg <= JALR;
        cmp_ctrl_reg <= cmp_ctrl;
```

```
        rs1_data_reg <= rs1_data;
        rs2_data_reg <= rs2_data;
        imm_reg <= imm;
        PC_reg <= PC;
        state <= 1;
    end
    else begin
        state <= 0;
    end
end

add_32 adder_pc_jump(JALR_reg ? rs1_data_reg : PC_reg, imm_reg, PC_jump);
add_32 adder_pc_wb(PC_reg, 'd4, PC_wb);
cmp_32 cmp(rs1_data_reg, rs2_data_reg, cmp_ctrl_reg, cmp_res);
```

FU_mem.v

Load/Store requires 2 cycles. Hence besides assigning values, counting cycles is also necessary. A shifting method is used again here. A sub-module is used here for calculating address.

```
always@(posedge clk) begin
    if(EN && state=='b00) begin
        mem_w_reg <= mem_w;
        bhw_reg <= bhw;
        rs1_data_reg <= rs1_data;
        rs2_data_reg <= rs2_data;
        imm_reg <= imm;
        state <= 'b10;
    end
    else begin
        state <= state >> 1;
    end
end

wire [31:0]addr;
add_32 adder(rs1_data_reg, imm_reg, addr);
```

RV32core.v

According to the datapath given, we can easily fill out the blank in this file. The first module is to generate immediates. The second and third ones are for selecting sources of ALU operands. The last module selects which data to write back to registers.

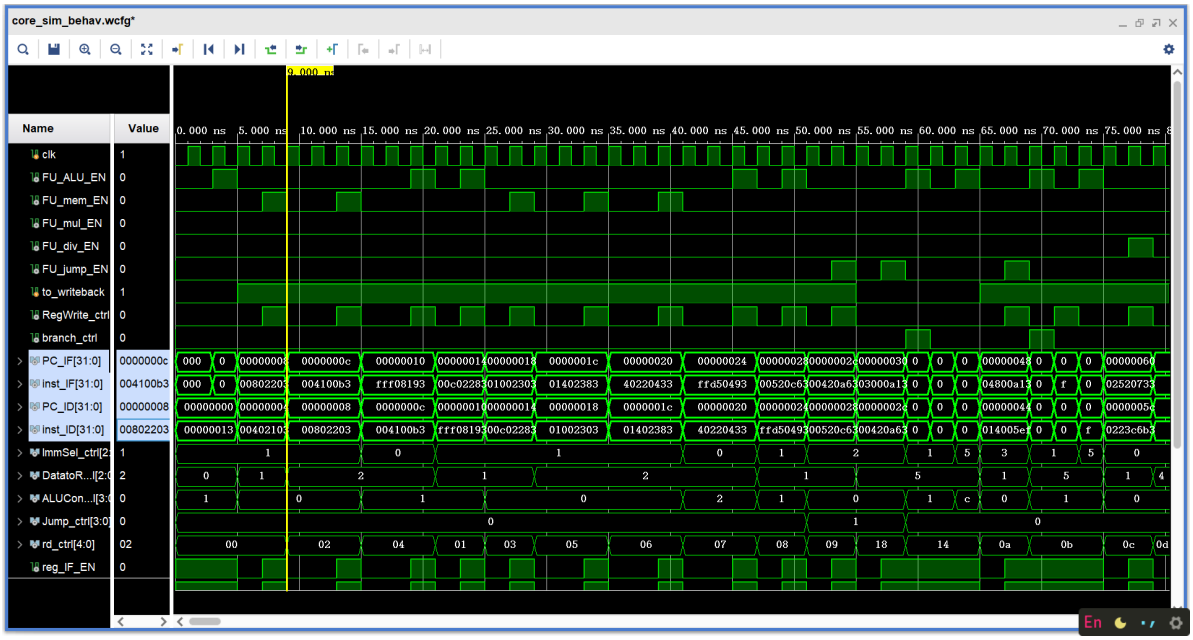
```
ImmGen imm_gen(ImmSel_ctrl, inst_ID, Imm_out_ID);
MUX2T1_32 mux_imm_ALU_ID_A(rs1_data_ID, PC_ID, ALUSrcA_ctrl, ALUA_ID);
MUX2T1_32 mux_imm_ALU_ID_B(rs2_data_ID, Imm_out_ID, ALUSrcB_ctrl, ALUB_ID);
MUX8T1_32 mux_DtR(
    .s(DatatoReg_ctrl),
    .I0('d0),               // use nothing
    .I1(ALUout_WB),         // use ALU
    .I2(mem_data_WB),       // use MEM
    .I3(mulres_WB),         // use MUL
```

```
    .I4(divres_WB),          // use DIV
    .I5(PC_wb_WB),           // use JUMP
    .I6('d0),                // use nothing
    .I7('d0),                // use nothing
    .o(wt_data_WB)
);
```

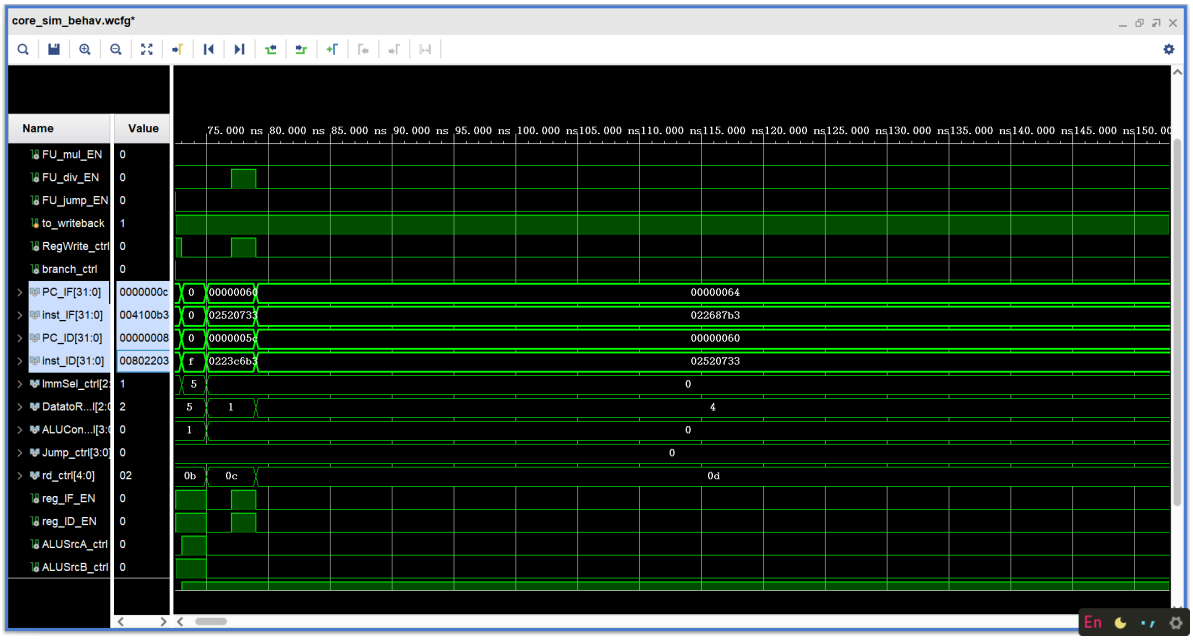
4. 实验结果分析

Simulation

As we can see, a load instruction occupies 2 cycles and then there is an enable signal set.



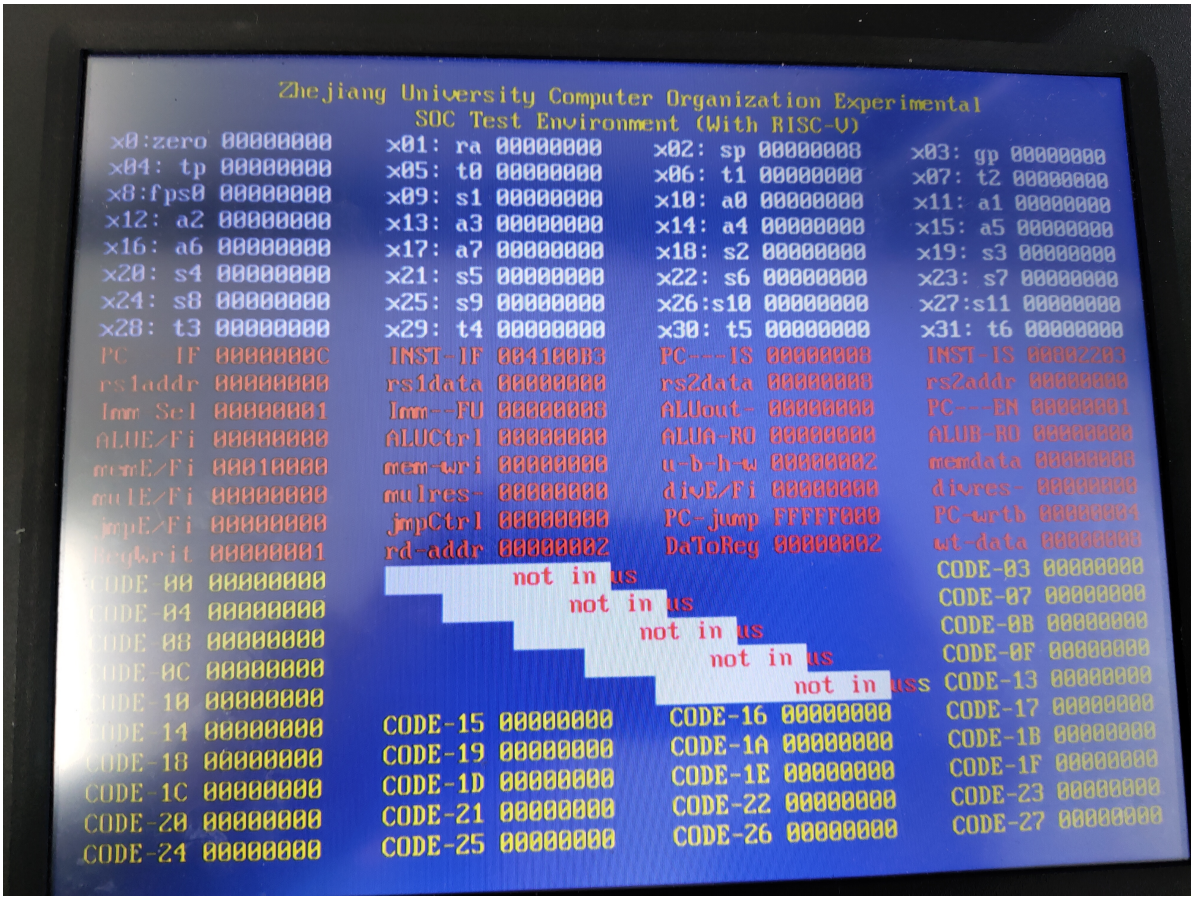
Instrucion at 0x5C is an div instruction, which occupies multiple cycles. We can see when it enters stage FU, irrelevant enable signals are set, and the pipeline is stalled waiting for the result.



The other result is the same as the one given in guidance and so is omitted here.

Program Device

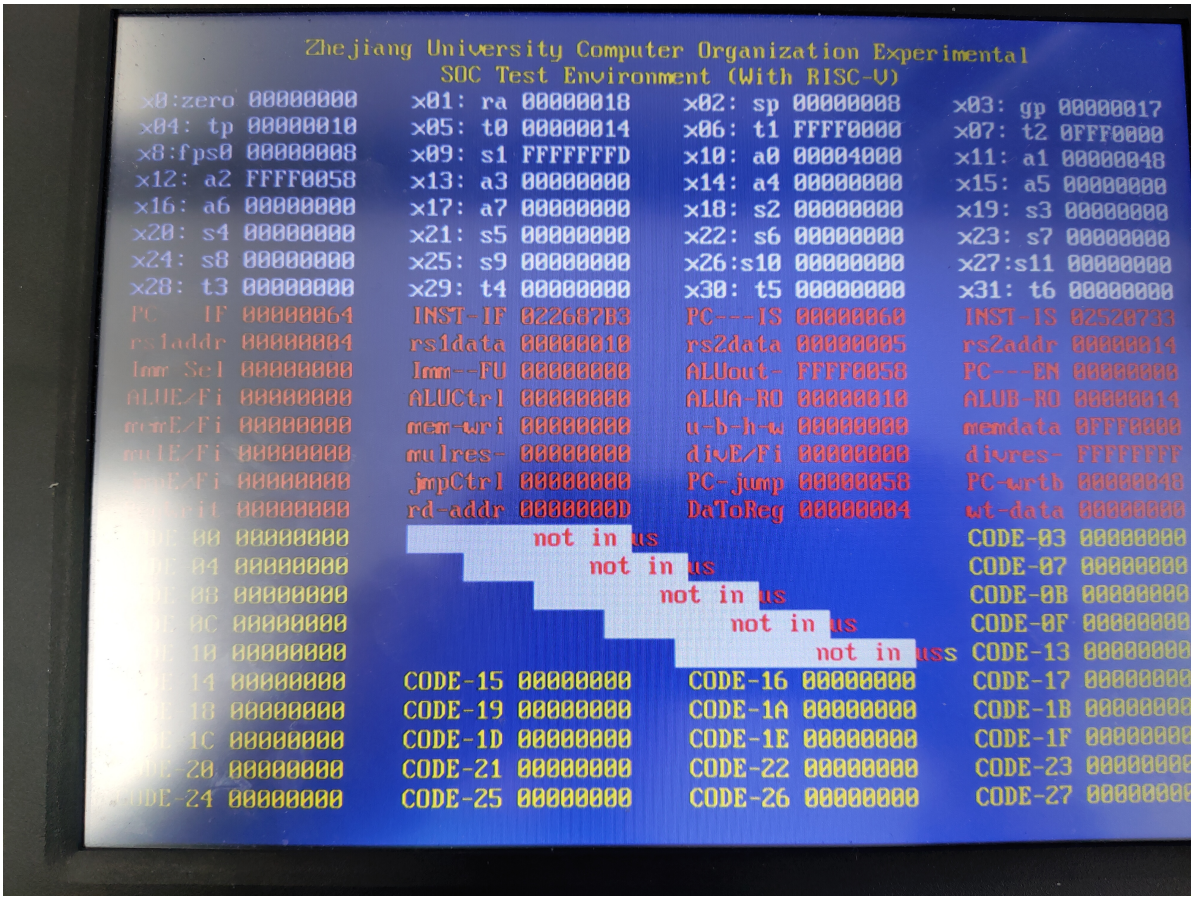
For the first load instruction, we can see the pipeline is stalled. The attributes of it are shown correctly, like the destination register is x2.



Then we can see, for the first branch instruction, the branch condition does not hold. The traget address is 0x40 as shown, but the branch is not taken.

Then, the next branch is taken. We can see PC changes to 0x40 after that.

Finally, for the div instruction, the pipeline is stalled for multiple cycles. In general, the result is the same as the result given by simulation, which can be proved to be correct.



5. 讨论与心得

The experiment is not difficult in general. The only point puzzling us is that we forgot to judge the source of target address calculation, which caused the `FU_jump` module functions incorrectly. Except for that, this experiment can be finished easily with full comprehension of the pipeline given.