



1. 实验目的和要求

Purpose

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

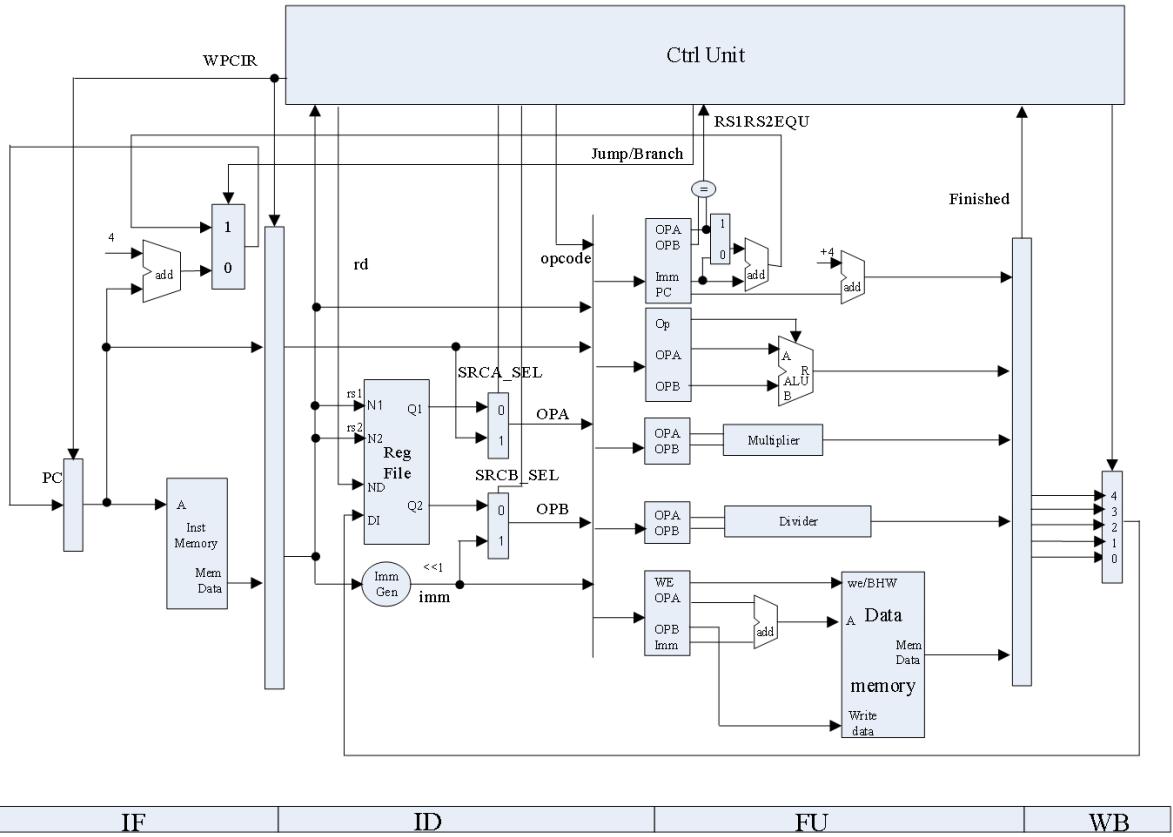
Task

- 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: Instruction 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 instruction finishes its FU and WB stages can the next instruction enter 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 condition 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_ctrl1_reg <= cmp_ctrl1;
```

```

    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_ctrl1_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

```

```

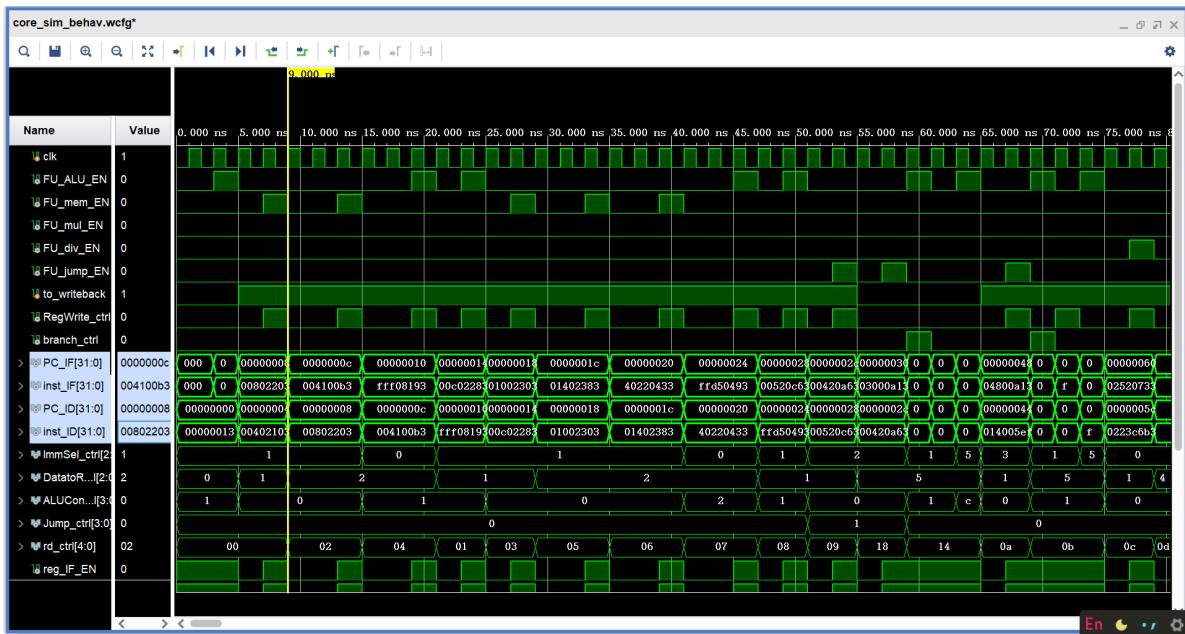
.I4(divres_WB),           // use DIV
.I5(PC_wb_WB),           // use JUMP
.I6('d0),                 // use nothing
.I7('d0),                 // use nothing
.o(wt_data_WB)
);

```

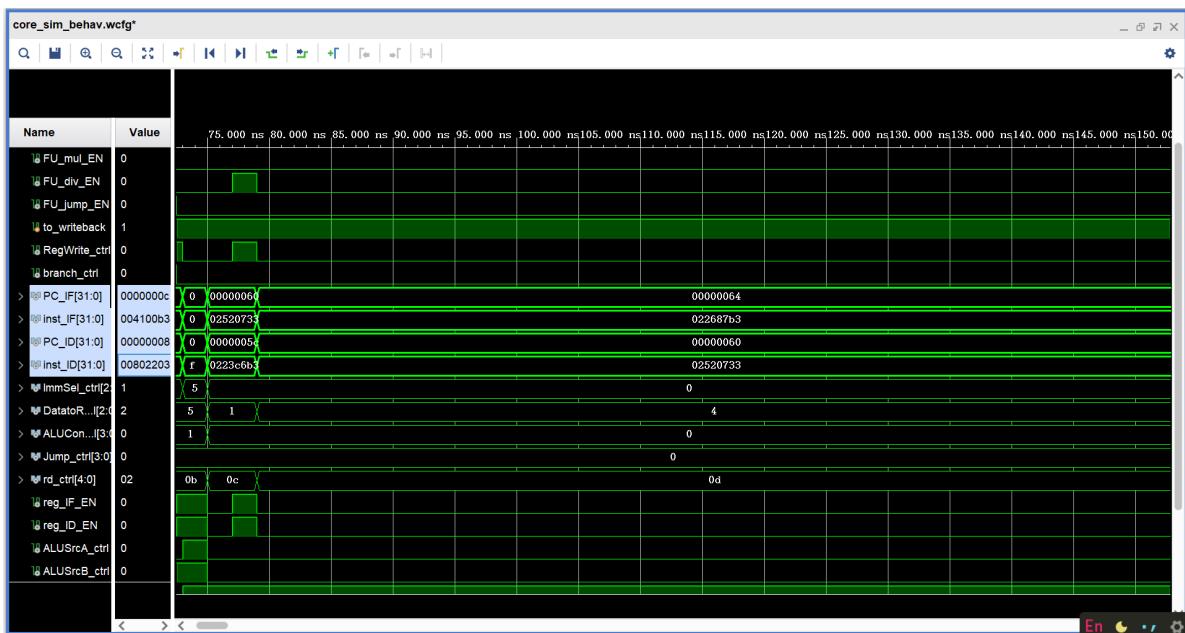
4. 实验结果分析

Simulation

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



Instruction 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

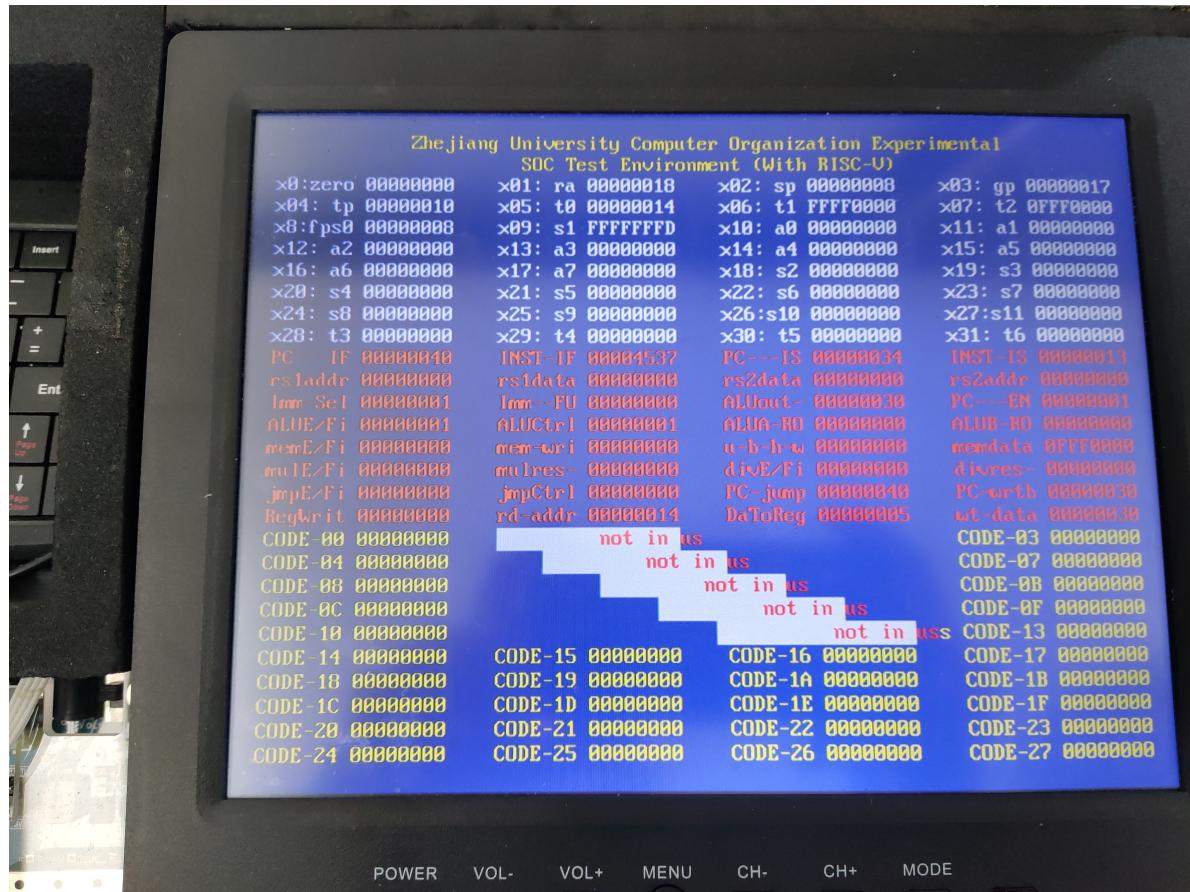
For the first load instruction, we can see the pipeline is stalled. The attributes of it are shown correctly, like the destination register is x2.

```
Zhejiang University Computer Organization Experimental  
SOC Test Environment (With RISC-U)  
  
x0:zero 00000000  x01: ra 00000000  x02: sp 00000008  x03: gp 00000000  
x04: tp 00000000  x05: t0 00000000  x06: t1 00000000  x07: t2 00000000  
x8:fps0 00000008  x09: s1 00000000  x10: a0 00000000  x11: a1 00000000  
x12: a2 00000000  x13: a3 00000000  x14: a4 00000000  x15: a5 00000000  
x16: a6 00000000  x17: a7 00000000  x18: s2 00000000  x19: s3 00000000  
x20: s4 00000000  x21: s5 00000000  x22: s6 00000000  x23: s7 00000000  
x24: s8 00000000  x25: s9 00000000  x26:s10 00000000  x27:s11 00000000  
x28: t3 00000000  x29: t4 00000000  x30: t5 00000000  x31: t6 00000000  
PC---IF 0000000C  INST-IF 004100B3  PC---IS 00000008  INST-IS 00002203  
rs1addr 00000000  rs1data 00000000  rs2data 00000000  rs2addr 00000000  
Imm-Sel 00000001  Imm--FU 00000008  ALUout- 00000000  PC---EN 00000001  
ALUE/Fi 00000000  ALUCtrl 00000000  ALUA-RO 00000000  ALUB-RO 00000000  
memE/Fi 00010000  mem-wri 00000000  u-b-h-w 00000002  memdata 00000000  
mulE/Fi 00000000  mulres- 00000000  divE/Fi 00000000  divres- 00000000  
jmpE/Fi 00000000  jmpCtrl 00000000  PC-jump FFFFF000  PC-wrtb 00000004  
RegWrit 00000001  rd-addr 00000002  DaToReg 00000002  wt-data 00000000  
CODE-00 00000000  not in us  CODE-03 00000000  
CODE-04 00000000  not in us  CODE-07 00000000  
CODE-08 00000000  not in us  CODE-0B 00000000  
CODE-0C 00000000  not in us  CODE-0F 00000000  
CODE-10 00000000  not in us  CODE-13 00000000  
CODE-14 00000000  CODE-15 00000000  CODE-16 00000000  CODE-17 00000000  
CODE-18 00000000  CODE-19 00000000  CODE-1A 00000000  CODE-1B 00000000  
CODE-1C 00000000  CODE-1D 00000000  CODE-1E 00000000  CODE-1F 00000000  
CODE-20 00000000  CODE-21 00000000  CODE-22 00000000  CODE-23 00000000  
CODE-24 00000000  CODE-25 00000000  CODE-26 00000000  CODE-27 00000000
```

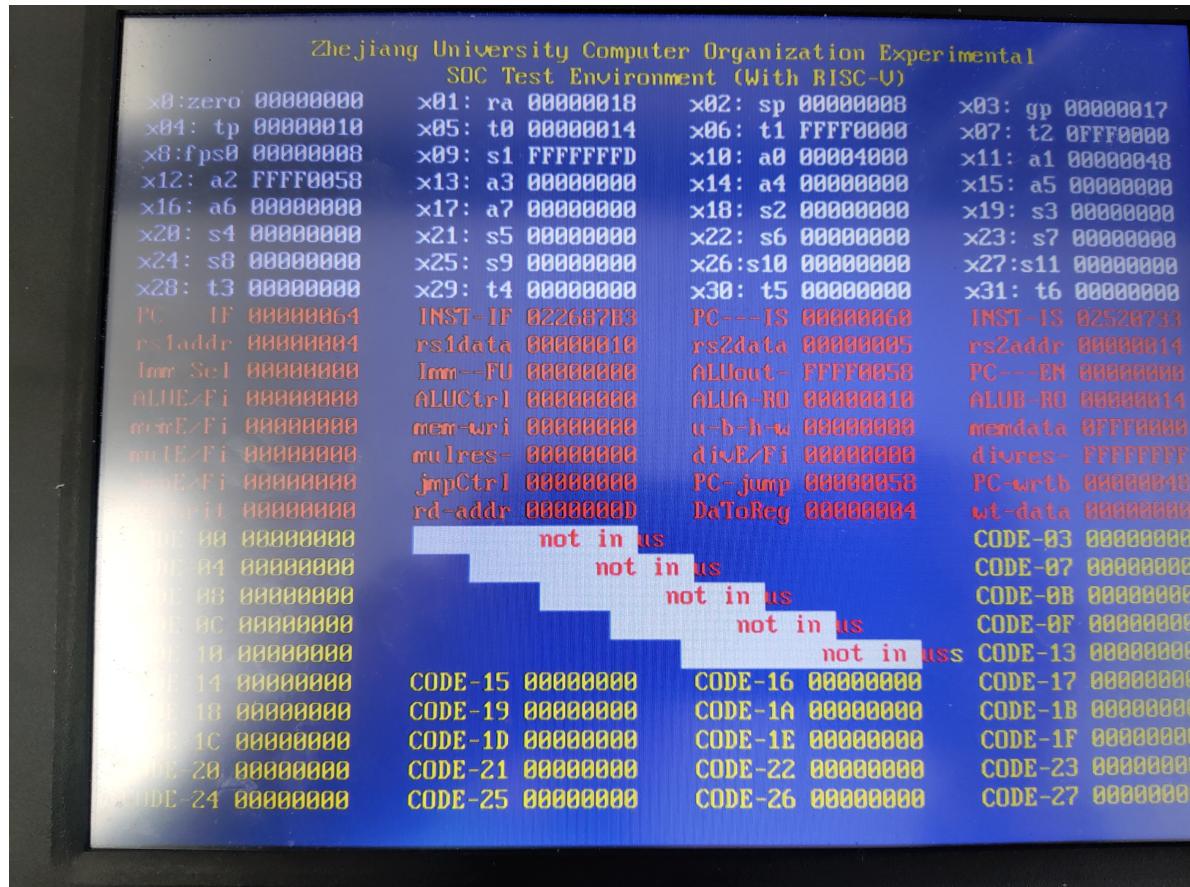
Then we can see, for the first branch instruction, the branch condition does not hold. The target address is 0x40 as shown, but the branch is not taken.

```
Zhejiang University Computer Organization Experimental  
SOC Test Environment (With RISC-U)  
  
x0:zero 00000000  x01: ra 00000018  x02: sp 00000008  x03: gp 00000017  
x04: tp 00000010  x05: t0 00000014  x06: t1 FFFF0000  x07: t2 0FFF0000  
x8:fps0 00000008  x09: s1 FFFFFFFD  x10: a0 00000000  x11: a1 00000000  
x12: a2 00000000  x13: a3 00000000  x14: a4 00000000  x15: a5 00000000  
x16: a6 00000000  x17: a7 00000000  x18: s2 00000000  x19: s3 00000000  
x20: s4 00000000  x21: s5 00000000  x22: s6 00000000  x23: s7 00000000  
x24: s8 00000000  x25: s9 00000000  x26:s10 00000000  x27:s11 00000000  
x28: t3 00000000  x29: t4 00000000  x30: t5 00000000  x31: t6 00000000  
PC---IF 00000030  INST-IF 030000A13  PC---IS 0000002C  INST-IS 004200A3  
rs1addr 00000004  rs1data 00000010  rs2data 00000004  rs2addr 00000010  
Imm-Sel 00000002  Imm--FU 00000014  ALUout- FFFFFFFD  PC---EN 00000001  
ALUE/Fi 00000000  ALUCtrl 00000000  ALUA-RO 00000018  ALUB-RO 00000018  
memE/Fi 00000000  mem-wri 00000000  u-b-h-w 00000000  memdata 0FFF0000  
mulE/Fi 00000000  mulres- 00000000  divE/Fi 00000000  divres- 00000000  
jmpE/Fi 00010000  jmpCtrl 00000001  PC-jump 00000040  PC-wrtb 00000002  
RegWrit 00000000  rd-addr 00000018  DaToReg 00000005  wt-data 00000002  
CODE-00 00000000  not in us  CODE-03 00000000  
CODE-04 00000000  not in us  CODE-07 00000000  
CODE-08 00000000  not in us  CODE-0B 00000000  
CODE-0C 00000000  not in us  CODE-0F 00000000  
CODE-10 00000000  not in us  CODE-13 00000000  
CODE-14 00000000  CODE-15 00000000  CODE-16 00000000  CODE-17 00000000  
CODE-18 00000000  CODE-19 00000000  CODE-1A 00000000  CODE-1B 00000000  
CODE-1C 00000000  CODE-1D 00000000  CODE-1E 00000000  CODE-1F 00000000  
CODE-20 00000000  CODE-21 00000000  CODE-22 00000000  CODE-23 00000000  
CODE-24 00000000  CODE-25 00000000  CODE-26 00000000  CODE-27 00000000
```

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.