浙江大学

本科实验报告

课程名称：计算机体系结构

姓　　名：

学　　院：计算机科学与技术学院

系：计算机科学与技术系

专　　业：计算机科学与技术

学　　号：

指导教师：卜凯

2023 年　　1 月　　1 日

浙江大学实验报告

课程名称：计算机体系结构　　实验类型：综合

实验项目名称：Lab5：Dynamic Scheduled Pipelines using Scoreboarding

学生姓名：　　专业：计算机科学与技术　　学号：

同组学生姓名：　　指导老师：卜凯

实验地点：曹西 301　　实验日期：2023 年 1 月 1 日

1 Tasks and requirements

1.1 Tasks

The main tasks of Lab-5 are:

1. Redesign the pipelines with IF/ID/FU/WB stages and FU stage supporting **multicycle** operations.
2. Redesign of cache controller.

1.2 Requirements

This experiment would be based on SWORD development board, with xc7k325tffg676-2L FPGA.

We are given a Verilog project complementing a CPU core and other components for debugging on board; this CPU should be able to deal with integer and float point operations. Most of the parts have been finished. There are still some units we need to complete, they are:

FU_ALU.v. This module is used to deal with integral arithmetic operations.

FU_div.v. This module is used to deal with float-point divide operations.

FU_mul.v. This module is used to deal with float-point multiply operations.

FU_mem.v. This module is used to deal with memory operations.

FU_jump.v. This module is used to deal with branch operations.

RV32core.v. This is the float-point CPU top module.

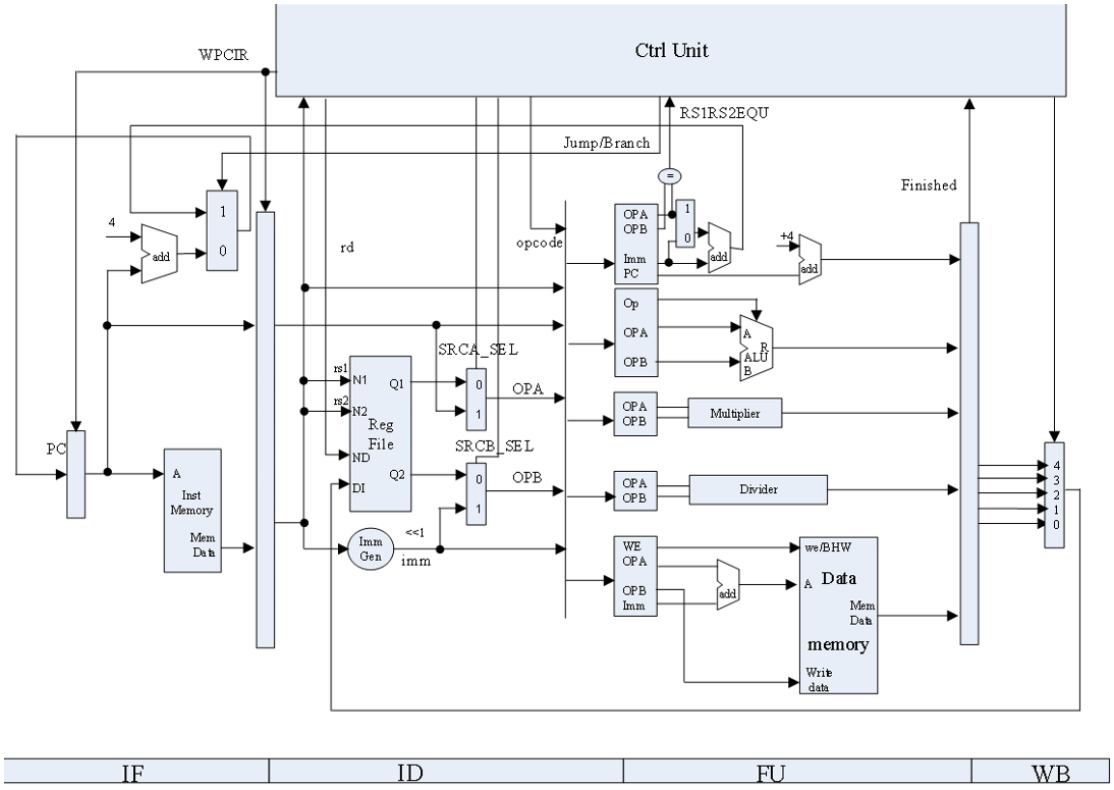
When this work is finished, we shall verify the simulation results of a preset program.

2 Contents and principles

All of the following principles are based on RISC-V pipelined float-point CPU.

2.1 Float-point Architecture

To run float-point operations on our CPU, we need to modify the architecture in order to support operations finished in different numbers of cycles, as following:



In our experiment, in order to simplify the design, we need to wait until all instructions finished executing, then to **issue** another instruction.

2.2 Float-point Function Units

As was mentioned above, float-point operations may cost different numbers of cycles in FU stage. We know that integer operations and branch operations can be done in 1 cycle, and memory operations in 2 cycles. However, float-point operations will be **multi-cycle**. This will influence our design of the control unit in next

experiments.

3 Steps and data records

3.1 Completed Verilog source files

For clarity, we will omit some pre-set code segments, and only analysis the code blocks that we need to fill in.

3.1.1 RV32core.v

In fact, the code filled in this file is quite simple and we need not to explain much. The principles we need can be found in part 2.1. Following is the implement:

```
//Issue
ImmGen imm_gen(
    .ImmSel(ImmSel_ctrl),.inst_field(inst_ID),.Imm_out(Imm_out_ID));
MUX2T1_32 mux_imm_ALU_ID_A(
    .I0(rs1_data_ID),.I1(PC_ID),.s(ALUSrcA_ctrl),.o(ALUA_ID));
MUX2T1_32 mux_imm_ALU_ID_B(
    .I0(rs2_data_ID),.I1(Imm_out_ID),.s(ALUSrcB_ctrl),.o(ALUB_ID));
//WB
MUX8T1_32 mux_DtR(.s(DatatoReg_ctrl),
    .I1(ALUout_WB),.I2(mem_data_WB),.I3(mulres_WB),
    .I4(divres_WB),.I5(PC_wb_WB),.o(wt_data_WB));
```

3.1.2 FU_ALU.v

The only thing we need to fill in is the starting state of ALU. As ALU operations costs only 1 cycle in FU stage, we write as follows.

```
always@(posedge clk) begin
    if(EN & ~state) begin // state == 0
        A <= ALUA;
        B <= ALUB;
        Control <= ALUControl;
        state <= 1;
```

```
    end
    else state <= 0;
end
```

3.1.3 FU_mem.v

Similar to above, one thing we need to fill in is the starting state and state transferring of memory operations. As memory operations will cost 2 cycles in FU stage, we write as follows.

```
always@(posedge clk) begin
    if(EN & ~|state) begin
        mem_w_reg <= mem_w;
        bhw_reg <=   bhw;
        rs1_data_reg <= rs1_data;
        rs2_data_reg <= rs2_data;
        imm_reg <= imm;
        state <= 2'b10;
    end
    else begin
        state <= state >> 1;
    end
end
```

At last, we need to calculate the address we access also at this unit.

```
wire[31:0] addr;
add_32 add(.a(rs1_data_reg),.b(imm_reg),.c(addr));
```

3.1.4 FU_jump.v

Same as above, one thing we need to fill in is the starting state and state transferring of branch operations. As branch operations will cost 1 cycle in FU stage, we write as follows.

```
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
```

Then we use 3 smaller units to give out the needed addresses and the result of the comparator unit.

```
cmp_32 cmp(.a(rs1_data_reg),.b(rs2_data_reg),.ctrl(cmp_ctrl_reg),.c(cmp_res));
add_32 a(.a(imm_reg),.b(JALR_reg ? rs1_data_reg : PC_reg),.c(PC_jump));
add_32 b(.a(pc_reg),.b(4),.c(PC_wb));
```

3.1.5 FU_mul.v

The only thing we need to fill in is the starting state and state transferring of float-point multiply operations. As float-point multiply operations will cost 8 cycles in FU stage, we write as follows.

```
always@(posedge clk) begin
    if(EN & ~|state) begin
        A_reg <= A;
        B_reg <= B;
        state <= 7'b1000000;
    end
    else begin
        state <= {1'b0, state[6:1]};
```

```
    end
end
```

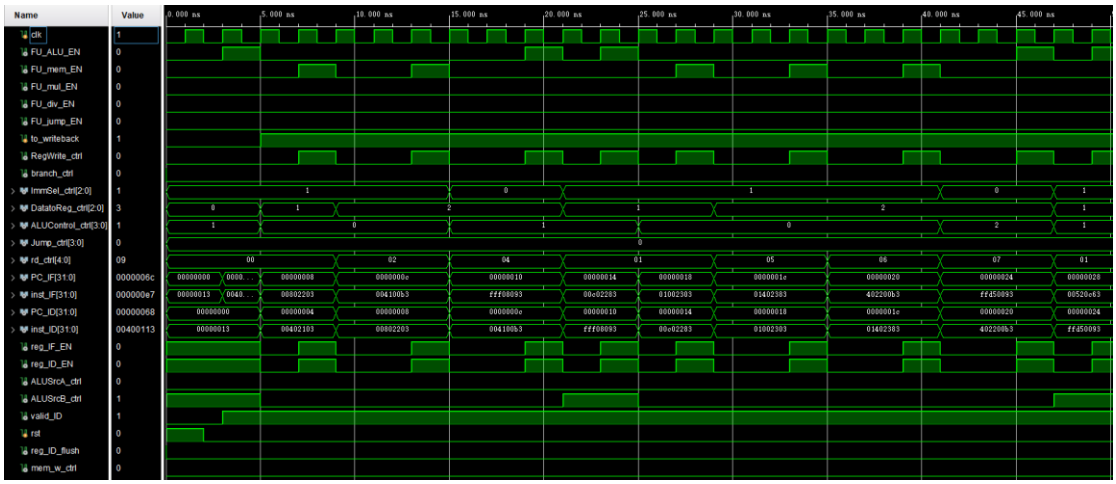
3.1.6 FU_div.v

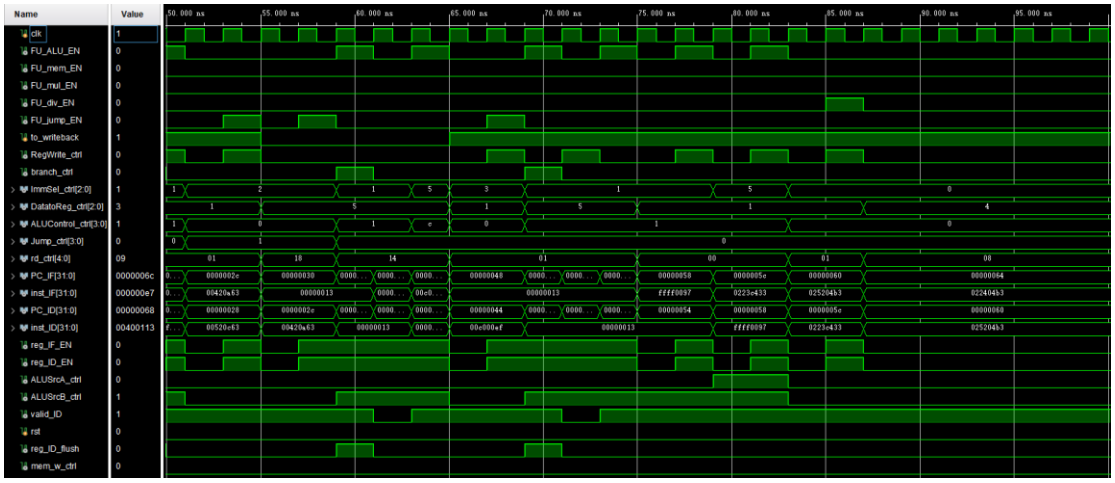
Same as above, the only thing we need to fill in is the starting state and state transferring of float-point divide operations. As float-point divide operations is using a divider and will stay in FU stage until the valid signal is set, we write as follows.

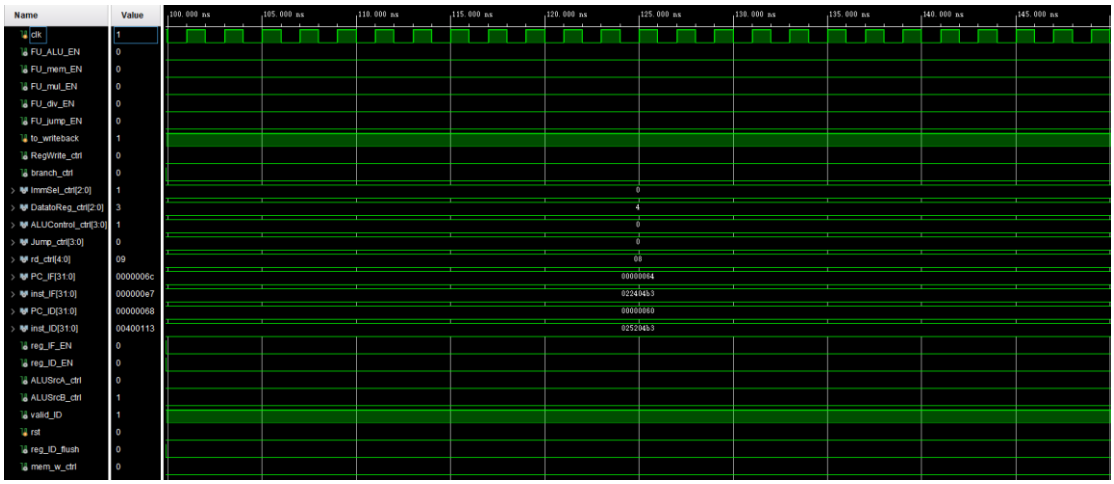
```
always@(posedge clk) begin
    if(EN & ~state) begin
        A_reg <= A;
        A_valid <= 1;
        B_reg <= B;
        B_valid <= 1;
        state <= 1;
    end
    else if(res_valid) begin
        A_valid <= 0;
        B_valid <= 0;
        state <= 0;
    end
end
```

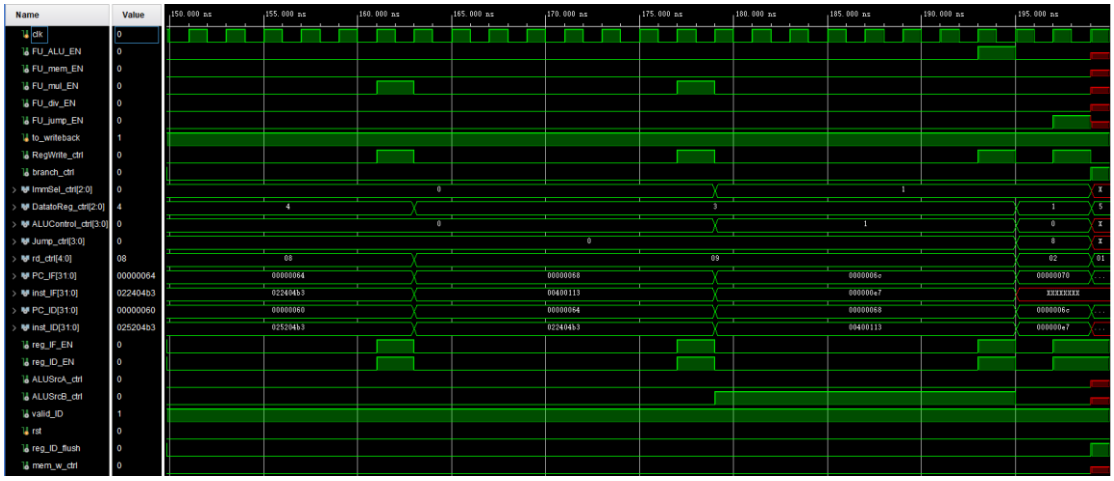
3.2 Implementation results

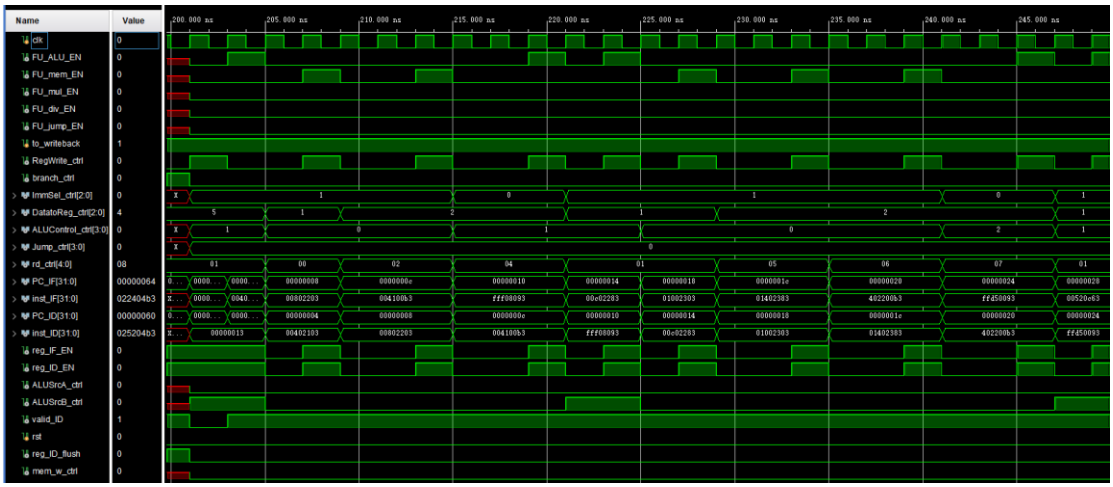
Here we record all our behavioral simulation results. Most will not be used for analysis.






Name
Value
100.000 ns
105.000 ns
110.000 ns
115.000 ns
120.000 ns
125.000 ns
130.000 ns
135.000 ns
140.000 ns
145.000 ns
clk
1
FU_ALU_EN
0
FU_mem_EN
0
FU_mul_EN
0
FU_div_EN
0
FU_jump_EN
0
to_writeback
1
RegWrite_ctrl
0
branch_ctrl
0
ImmSel_ctrl[2:0]
1
DatatoReg_ctrl[2:0]
3
ALUControl_ctrl[3:0]
1
Jump_ctrl[3:0]
0
rd_ctrl[4:0]
09
PC_IF[31:0]
0000006c
inst_IF[31:0]
000000e7
PC_ID[31:0]
00000068
inst_ID[31:0]
00400113
reg_IF_EN
0
reg_ID_EN
0
ALUSrcA_ctrl
0
ALUSrcB_ctrl
1
valid_ID
1
rst
0
reg_ID_flush
0
mem_w_ctrl
0

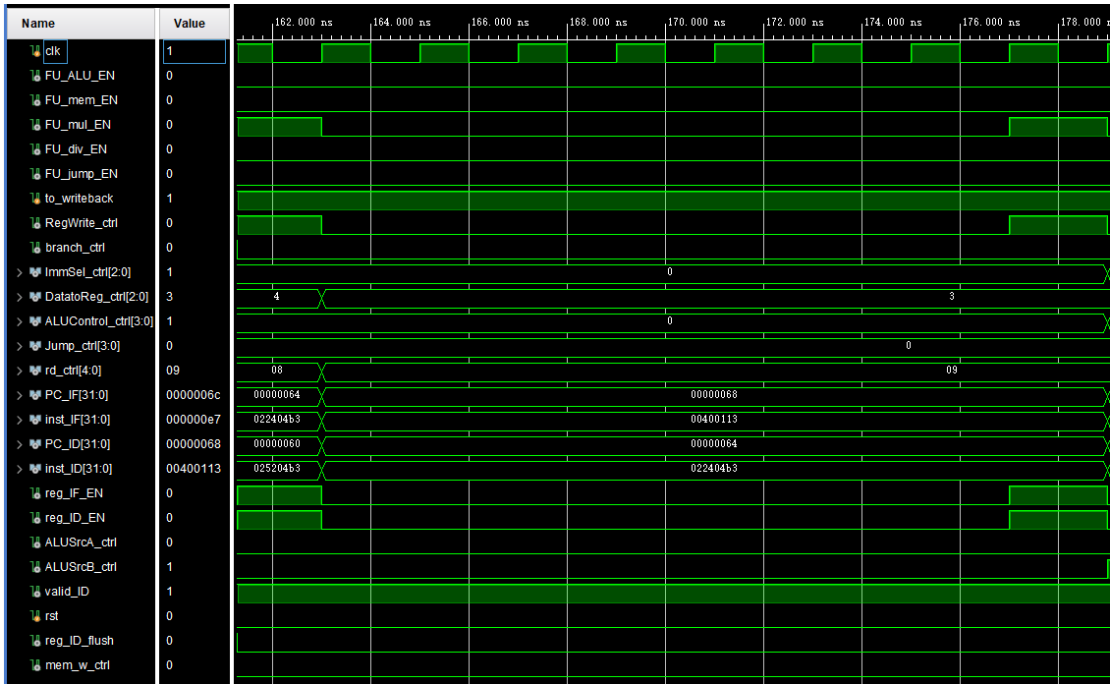

Name
Value
150.000 ns
155.000 ns
160.000 ns
165.000 ns
170.000 ns
175.000 ns
180.000 ns
185.000 ns
190.000 ns
195.000 ns
clk
0
FU_ALU_EN
0
FU_mem_EN
0
FU_mul_EN
0
FU_div_EN
0
FU_jump_EN
0
to_writeback
1
RegWrite_ctrl
0
branch_ctrl
0
ImmSel_ctrl[2:0]
0
DatatoReg_ctrl[2:0]
4
ALUControl_ctrl[3:0]
0
Jump_ctrl[3:0]
0
rd_ctrl[4:0]
08
PC_IF[31:0]
00000064
inst_IF[31:0]
022404b3
PC_ID[31:0]
00000060
inst_ID[31:0]
025204b3
reg_IF_EN
0
reg_ID_EN
0
ALUSrcA_ctrl
0
ALUSrcB_ctrl
0
valid_ID
1
rst
0
reg_ID_flush
0
mem_w_ctrl
0


Name
Value
200.000 ns
205.000 ns
210.000 ns
215.000 ns
220.000 ns
225.000 ns
230.000 ns
235.000 ns
240.000 ns
245.000 ns
clk
0
FU_ALU_EN
0
FU_mem_EN
0
FU_mul_EN
0
FU_div_EN
0
FU_jump_EN
0
to_writeback
1
RegWrite_ctrl
0
branch_ctrl
0
ImmSel_ctrl[2:0]
0
DatatoReg_ctrl[2:0]
4
ALUControl_ctrl[3:0]
0
Jump_ctrl[3:0]
0
rd_ctrl[4:0]
08
PC_IF[31:0]
00000064
inst_IF[31:0]
022404b3
PC_ID[31:0]
00000060
inst_ID[31:0]
025204b3
reg_IF_EN
0
reg_ID_EN
0
ALUSrcA_ctrl
0
ALUSrcB_ctrl
0
valid_ID
1
rst
0
reg_ID_flush
0
mem_w_ctrl
0

4 Analysis of the results

Here we only discuss some typical results.

4.1 Multi-cycle Operations



Take in 163-179ns as an example.

In 163-179ns, the CPU core uses 8 cycles to finish a float-point multiply operation. Noticeably, in 177-179ns, the signal **FU_mul_EN** is set, indicating that the float-point multiply operation is dealt. Then the CPU will be able to deal with the next instruction after 179ns.

We could see the process runs smoothly.

5 Discussion and Conclusion

5.1 Problems

We got no problem worth to be mentioned in lab-5.

5.2 Achievements and conclusion

In this experiment, I implemented a float-point CPU supporting multicycle operations, and learned the knowledge about float-point functions in architecture design. Overall, the experiment is successful.