1. 实验目的和要求

Purpose

- Understand the principle of piplines that support multicycle operations.
- Understand the principle of Dynamic Scheduling With a Scoreboard.
- Master the design methods of piplines that support multicycle operations.
- Master the design methods of Dynamically Scheduled Pipelines using Scoreboarding.
- Master verification methods of Dynamically Scheduled Pipelines using Scoreboarding.

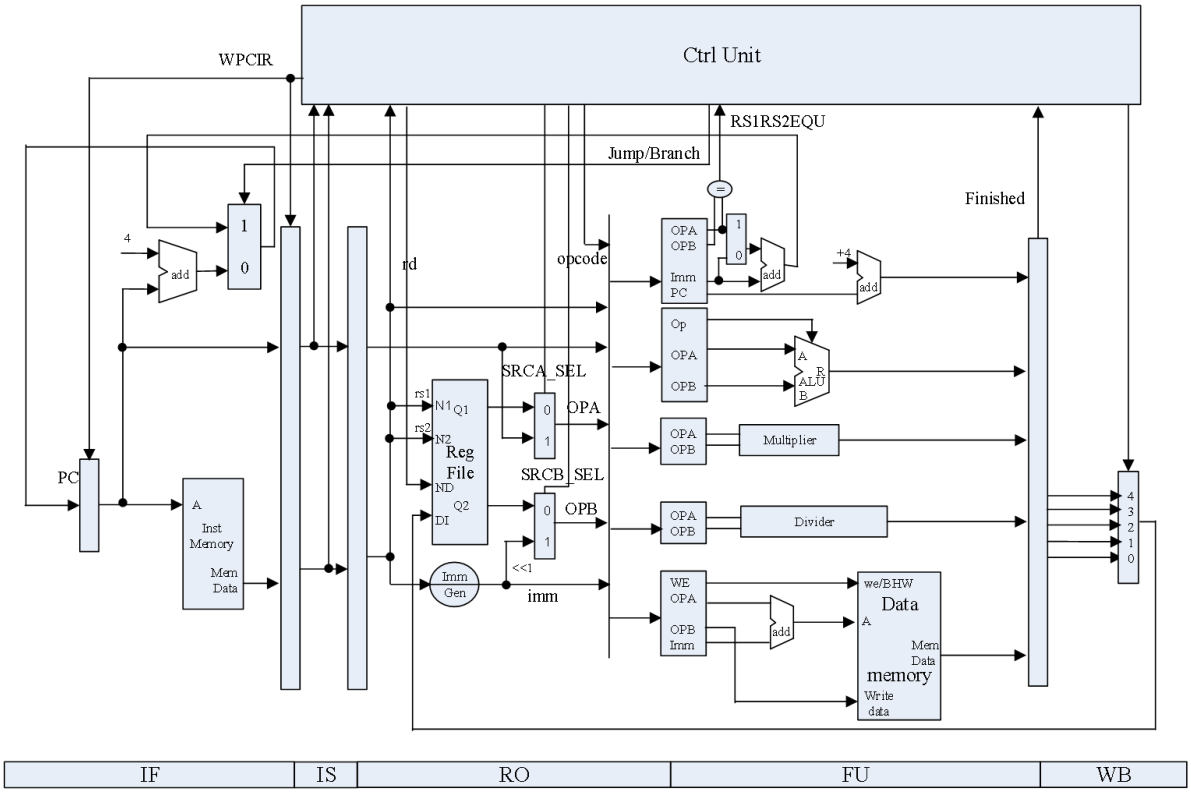
Task

- Redesign the pipelines with IF/IS/RO/FU/WB stages and supporting multicycle operations.
- Design of a scoreboard and integrate it to CPU.
- 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 5 stages: Instruciotn Fetch, Instruction Issue, Read Operands, Function Unit and Writeback. Comparing to the previous pipeline in lab 5, this one allows more parallelism, that is, out-of-order execution. Scoreboarding technique is adopted here. Background knowledge is introduced in class, we can refer to the sildes for help.



In this experiment, we should implement the control unit in the pipeline, which controls the functionality of the pipeline.

3. 实验过程和数据记录

There is only one module to be filled out.

CtrlUnit.v

The first signal `normal_stall` indicates whether there is a structural hazard or WAW. If the function unit to be used is busy now, there is a structural hazard, which is judged by the first four clauses. The last clause judges if the destination register will be written by the previous instructions currently in other function units.

```
// normal stall: structural hazard or WAW
assign normal_stall = (FUS[`FU_ALU][`BUSY] & use_ALU) |
    (FUS[`FU_MEM][`BUSY] & use_MEM) |
    (FUS[`FU_MUL][`BUSY] & use_MUL) |
    (FUS[`FU_DIV][`BUSY] & use_DIV) |
    (FUS[`FU_JUMP][`BUSY] & use_JUMP) |
    (|RRS[dst] & dst != 0);           //fill sth. here
```

We judge a WAR. The analysis of all funciton units is similar, so we only take ALU as an example. If ALU is going to write x0, there is no WAR for sure. Otherwise, if the destination register of ALU is the same as the source register of previous instructions, which reside in other function unit, and the value of such register hasn't been read by other funciton units, there is a WAR.

Here, the transition of `RDY` bit is noteworthy. In the beginning, set to 0 if the source register is not ready to be read. Then, if the source register is ready to be read but hasn't been read, `RDY` is set to 1. Finally, if the ready value is read, `RDY` is set to 0 again. To prevent WAR, we need to ensure that an FU has read the value, so we need to judge whether `RDY` is 0 here. Three states share only one bit here will not cause any problem since `ALU_WAR` will be used together with `DONE` bit in the following instructions.

```
// ensure WAR:
// If an FU hasn't read a register value (RO), don't write to it.
wire ALU_WAR = (
    ((FUS[`FU_MEM][`SRC1_H:`SRC1_L]  != FUS[`FU_ALU][`DST_H:`DST_L]) |
!FUS[`FU_MEM][`RDY1] )  &
    ((FUS[`FU_MEM][`SRC2_H:`SRC2_L]  != FUS[`FU_ALU][`DST_H:`DST_L]) |
!FUS[`FU_MEM][`RDY2] )  &
    ((FUS[`FU_MUL][`SRC1_H:`SRC1_L]  != FUS[`FU_ALU][`DST_H:`DST_L]) |
!FUS[`FU_MUL][`RDY1] )  &
    ((FUS[`FU_MUL][`SRC2_H:`SRC2_L]  != FUS[`FU_ALU][`DST_H:`DST_L]) |
!FUS[`FU_MUL][`RDY2] )  &
    ((FUS[`FU_DIV][`SRC1_H:`SRC1_L]  != FUS[`FU_ALU][`DST_H:`DST_L]) |
!FUS[`FU_DIV][`RDY1] )  &
    ((FUS[`FU_DIV][`SRC2_H:`SRC2_L]  != FUS[`FU_ALU][`DST_H:`DST_L]) |
!FUS[`FU_DIV][`RDY2] )  &
    ((FUS[`FU_JUMP][`SRC1_H:`SRC1_L] != FUS[`FU_ALU][`DST_H:`DST_L]) |
!FUS[`FU_JUMP][`RDY1])  &
    ((FUS[`FU_JUMP][`SRC2_H:`SRC2_L] != FUS[`FU_ALU][`DST_H:`DST_L]) |
!FUS[`FU_JUMP][`RDY2])
) | |FUS[`FU_ALU][`DST_H:`DST_L];
```

The following block maintains all kinds of information for the pipeline. If stage RO is enabled, we should record the information of the current instruction in RRS, FUS and IMM.

In stage RO, if both source registers are ready, values will be read in another block, and we should set `RDY` bits to 0 here, which indicate that the value has been read.

In stage FU, each funciton unit will be working if it is assigned an instruction with all operands. Once the execution of a function unit is done, a signal will be sent back to control unit. Here, if such signal is received, we should set `DONE` bit to 1, which indicates that the execution of one function unit is done. However, if such signal is 0, the `DONE` bit should remain unchanged.

In stage WB, the last stage of an instruction, the information recorded in FUS and RRS should be cleared. Also, `RDY` bits of the following instructions should be set to 1 for the pipeline to continue working.

```
// maintain the table
always @ (posedge clk or posedge rst) begin
    if (rst) begin
        // reset the scoreboard
        for (i = 0; i < 32; i = i + 1) begin
            RRS[i] <= 3'b0;
        end

        for (i = 1; i <= 5; i = i + 1) begin
            FUS[i] <= 32'b0;
            IMM[i] <= 32'b0;
        end
    end
```

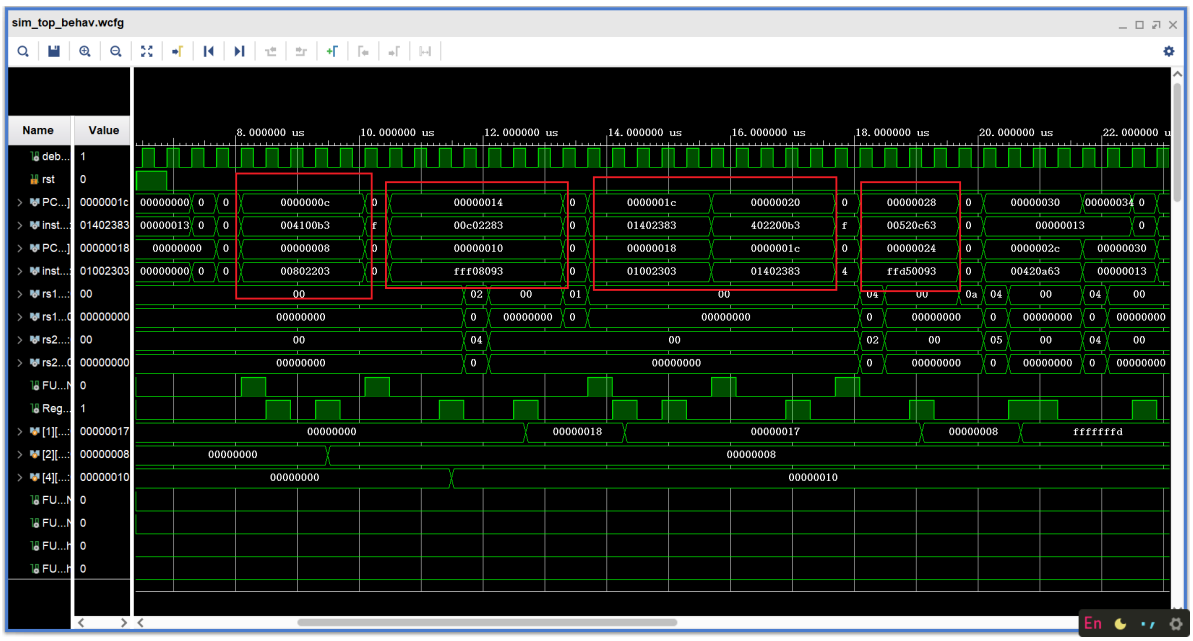
```
    else begin
        // IS issue
        if (RO_en) begin
            // not busy, no WAW, write info to FUS and RRS
            if (|dst) RRS[dst] <= use_FU;
            FUS[use_FU][`BUSY] <= 1'b1;
            FUS[use_FU][`OP_H:`OP_L] <= op;                          //fill sth.
here.
            FUS[use_FU][`DST_H:`DST_L] <= dst;
            FUS[use_FU][`SRC1_H:`SRC1_L] <= src1;
            FUS[use_FU][`SRC2_H:`SRC2_L] <= src2;
            FUS[use_FU][`FU1_H:`FU1_L] <= fu1;
            FUS[use_FU][`FU2_H:`FU2_L] <= fu2;
            FUS[use_FU][`RDY1] <= rdy1;
            FUS[use_FU][`RDY2] <= rdy2;

            IMM[use_FU] <= imm;
            PCR[use_FU] <= PC;
        end

        // RO read operands
        if (FUS[`FU_JUMP][`RDY1] & FUS[`FU_JUMP][`RDY2]) begin
            // JUMP
            FUS[`FU_JUMP][`RDY1] <= 1'b0;
            FUS[`FU_JUMP][`RDY2] <= 1'b0;
        end
        else if (FUS[`FU_ALU][`RDY1] & FUS[`FU_ALU][`RDY2]) begin
//fill sth. here.
            // ALU
            FUS[`FU_ALU][`RDY1] <= 1'b0;
            FUS[`FU_ALU][`RDY2] <= 1'b0;
        end
        else if (FUS[`FU_MEM][`RDY1] & FUS[`FU_MEM][`RDY2]) begin
//fill sth. here.
            // MEM
            FUS[`FU_MEM][`RDY1] <= 1'b0;
            FUS[`FU_MEM][`RDY2] <= 1'b0;
        end
        else if (FUS[`FU_MUL][`RDY1] & FUS[`FU_MUL][`RDY2]) begin
//fill sth. here.
            // MUL
            FUS[`FU_MUL][`RDY1] <= 1'b0;
            FUS[`FU_MUL][`RDY2] <= 1'b0;
        end
        else if (FUS[`FU_DIV][`RDY1] & FUS[`FU_DIV][`RDY2]) begin
//fill sth. here.
            // DIV
            FUS[`FU_DIV][`RDY1] <= 1'b0;
            FUS[`FU_DIV][`RDY2] <= 1'b0;
        end

        // EX execute
        FUS[`FU_ALU][`FU_DONE] <= ALU_done ? 1'b1 :FUS[`FU_ALU][`FU_DONE];
//fill sth. here
        FUS[`FU_MEM][`FU_DONE] <= MEM_done ? 1'b1 :FUS[`FU_MEM][`FU_DONE];
        FUS[`FU_MUL][`FU_DONE] <= MUL_done ? 1'b1 :FUS[`FU_MUL][`FU_DONE];
        FUS[`FU_DIV][`FU_DONE] <= DIV_done ? 1'b1 :FUS[`FU_DIV][`FU_DONE];
```

```
        FUS[`FU_JUMP][`FU_DONE] <= JUMP_done ? 1'b1 :FUS[`FU_JUMP][`FU_DONE];

        // WB write back
        if (FUS[`FU_JUMP][`FU_DONE] & JUMP_WAR) begin
            // clear everything
            FUS[`FU_JUMP] <= 32'b0;
            RRS[FUS[`FU_JUMP][`DST_H:`DST_L]] <= 3'b0;

            // ensure RAW
            if (FUS[`FU_ALU][`FU1_H:`FU1_L] == `FU_JUMP) FUS[`FU_ALU][`RDY1] <=
1'b1;
            if (FUS[`FU_MEM][`FU1_H:`FU1_L] == `FU_JUMP) FUS[`FU_MEM][`RDY1] <=
1'b1;
            if (FUS[`FU_MUL][`FU1_H:`FU1_L] == `FU_JUMP) FUS[`FU_MUL][`RDY1] <=
1'b1;
            if (FUS[`FU_DIV][`FU1_H:`FU1_L] == `FU_JUMP) FUS[`FU_DIV][`RDY1] <=
1'b1;

            if (FUS[`FU_ALU][`FU2_H:`FU2_L] == `FU_JUMP) FUS[`FU_ALU][`RDY2] <=
1'b1;
            if (FUS[`FU_MEM][`FU2_H:`FU2_L] == `FU_JUMP) FUS[`FU_MEM][`RDY2] <=
1'b1;
            if (FUS[`FU_MUL][`FU2_H:`FU2_L] == `FU_JUMP) FUS[`FU_MUL][`RDY2] <=
1'b1;
            if (FUS[`FU_DIV][`FU2_H:`FU2_L] == `FU_JUMP) FUS[`FU_DIV][`RDY2] <=
1'b1;
        end
        // ALU
        ...
        // MEM
        ...
        // MUL
        ...
        // DIV
        ...
    end
end
```

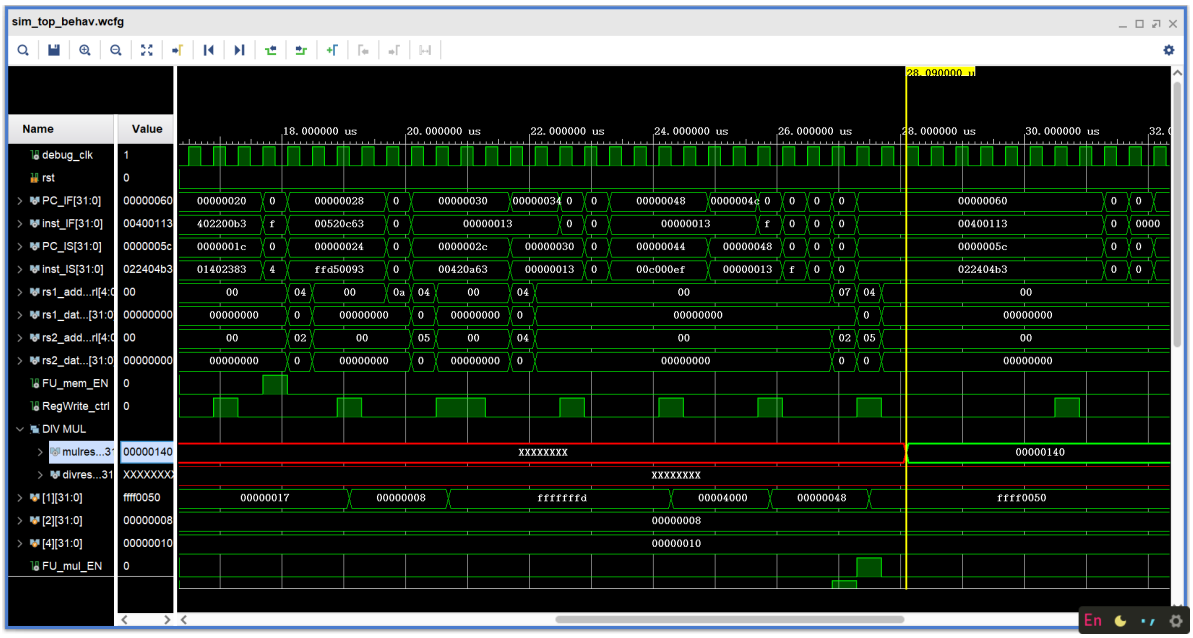
4. 实验结果分析

Simulation

As the simulation shows, whenever there is a hazard or WAW, the pipeline stops issuing. Check the value of each signal, we can find the values are correct.



We can see although the mul instruction comes following the div instruction, the result of multiplication comes out first. This is because there is no dependency and they can be both issued without stalling. Multiplication requires less cycles so the result of multiplication comes out first.



The other part of the result is the same as that given in the guidance, so is omitted here.

Program Device

When the first load instruction is issued, we can see that the pipeline stalls because of a structural hazard. From the value of signals, we can infer that FU memory is generating immediate.

Loading continues. From the value of signals, we can infer that FU memory is reading from memory.

Loading continues. From the value of signals, we can infer that FU memory is writing value to destination register. We can see the value read out at `memdata` is 0x08, which is correct.

Then, at the beginning of the mul instruction, we can see the result is cleared.

Multiplication continues for several cycles and we can see the result at `mulres` is 0x140, which is consistent with the simulation result.

5. 讨论与心得

This experiment is a relatively difficult one. The organization of the code is not that clear as other experiments. Also, compared to the scoreboarding introduced in clss, this pipeline introduce a `DONE` bit, which works slightly differently from `BUSY` bit.

As for me, the three states of `RDY` bit causes me some trouble. When I was coding for WAR, I didn't figure out that and I used

```
FUS[FU_MEM][RDY1]
```

rather than

```
!FUS[FU_MEM][RDY1]
```

This makes the pipeline works abnormally. After figuring out the transition of `RDY`, as explained above, I finally corrected this error.