Computer organization lab06 report

2019-12172 이재원

1 Introduction

Understand why pipelined cpu is faster than multicycle cpu, and calculate the performance of pipelined cpu compared to multicycle cpu. Also, implement data forwarding and branch predictor and see how much performance improved.

2 Design

2.1 Control signals

When the instructions are decoded at ID stage, opcodes and function_codes are known, where control signals are determined. There are 2 ways to design control signals.

First, all control signals that are used at ID, EX, MEM, WB stage are determined at ID stage, and the signals are moved through pipeline registers.

Second, the instruction moves through the pipeline register and the instruction is decoded at every stage and determines control signals.

In this implementation, the first method is used. Therefore, module *control_unit.v* gets the decoded instruction at ID stage and prints out control signals at ID stage. As it works at ID stage, it is a combinational circuit.

2.2 Pipeline registers

A pipelined cpu needs to latch data and control signals, implement flush and stalls. Therefore, there needs to be a register in the datapath or in the control signals. The pipeline register is implemented in *pipleline_register.v,* where both control signals and data are both latched. To avoid confusion, control signals and data is differentiated with comments in *pipleline_register.v*.

2.3 num_inst

num_inst should be increased when the instruction gets out of the pipeline. This is because instruction can be flushed or interrupted while at the pipeline. Although there are no case with exception and interruptions, num_inst will be designed to increase at the end of the stage.

2.4 RF

Usually there are 2 read ports, and 1 write port for RF. However, by adding another read port at RF will make in possible to read RF at ID stage, and another stage. The instruction WWD reads RF which can lead to RAW hazards. But by reading RF at WB, there is no any kind of data hazards. And as the num_inst is increased at the end of the pipeline, the output port of WWD should print out at WB stage, which is efficient to read RF at WB stage and print it out right away.

Also, RF is written at negedge clock to reduce RAW hazard.

2.5 Branch predictor

In branch predictor module BTB, Tag Table will be designed. Also for there is BHT for 2-bit saturation counter.

2.6 Data forwarding

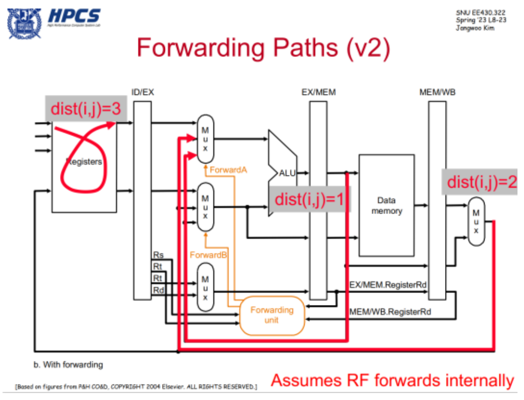


Figure 1. Data forwarding from lecture slide Lec08_pipelines_CPU_data_hazard

Will design exactly same as Figure 1. However, as JAR, JRL instruction is used at ID stage, and figure 1 is not data forwarded at ID stage. Therefore, there still needs to be stall at ID stage for JAR, JRL instructions.

2.7 Stall

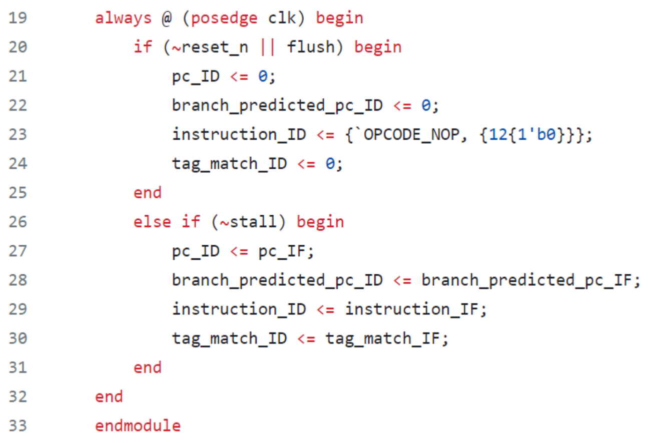
In this implementation you only need to stall at ID stage. Therefore, by stalling IF/ID register and changing instructions at EX to nop, it is same as stalling for 1 cycle.

2.8 Flush

Flush occurs when branch prediction was incorrect. The cpu knows jump instructions are predicted wrong at ID stage, and I type branch instructions at EX stage. So, for mispredicted jump instructions printing out nop at IF/ID register makes it flush. And for I type branch instructions, printing out nop at IF/ID, ID/EX register will do the same.

3 Implementation

3.1 pipeline_register.v



```verilog
always @ (posedge clk) begin
    if (~reset_n || flush) begin
        pc_ID <= 0;
        branch_predicted_pc_ID <= 0;
        instruction_ID <= {`OPCODE_NOP, {12{1'b0}}};
        tag_match_ID <= 0;
    end
    else if (~stall) begin
        pc_ID <= pc_IF;
        branch_predicted_pc_ID <= branch_predicted_pc_IF;
        instruction_ID <= instruction_IF;
        tag_match_ID <= tag_match_IF;
    end
end
endmodule
```

Figure 2. part of pipeline_register.v

Like figure 2 all pipeline registers have the same format. When reset or flush it prints out nop, and when not stall, in prints out the input. When stall, it prints out the same output at the previous cycle.

3.2 hazard_control_unit.v

It determines stall and flush signals.

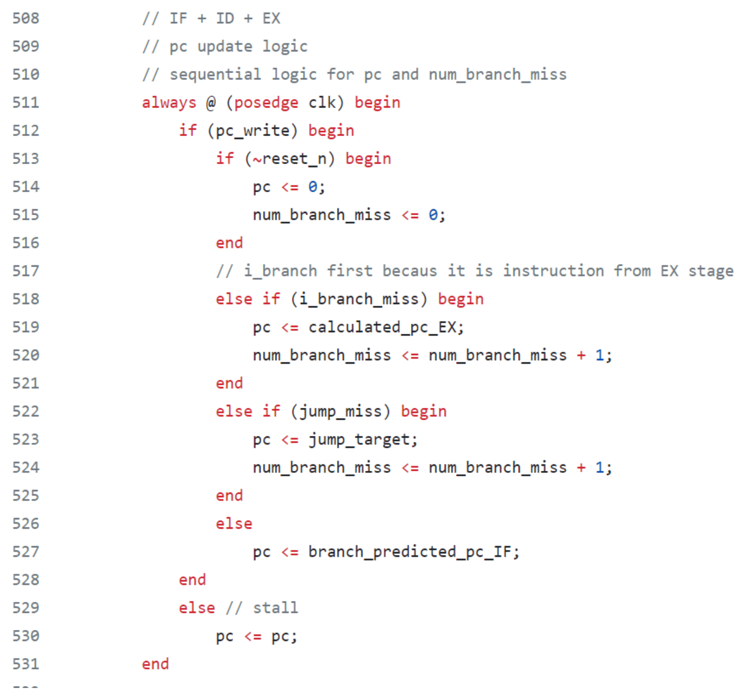
For stall at ID stage, same instruction should be at ID stage, and nop at EX stage. Therefore, it prints out stall to IF/ID register and flush at ID/EX register like explained in the design section.

For mispredicted jump type instructions, it prints out flush at IF/ID register.

For mispredicted I type instructions, it prints out flush at IF/ID register and ID/EX register as explained at the design section.

Just like *control_unit.v* it is combinational circuit that works on ID stage.

3.3 pc update logic

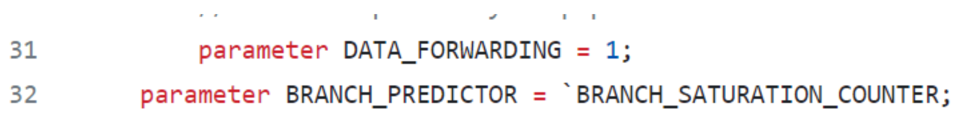


```verilog
// IF + ID + EX
// pc update logic
// sequential logic for pc and num_branch_miss
always @ (posedge clk) begin
    if (pc_write) begin
        if (~reset_n) begin
            pc <= 0;
            num_branch_miss <= 0;
        end
        // i_branch first becaus it is instruction from EX stage
        else if (i_branch_miss) begin
            pc <= calculated_pc_EX;
            num_branch_miss <= num_branch_miss + 1;
        end
        else if (jump_miss) begin
            pc <= jump_target;
            num_branch_miss <= num_branch_miss + 1;
        end
        else
            pc <= branch_predicted_pc_IF;
    end
    else // stall
        pc <= pc;
end
```

Figure 3. pc update logic from datapath.v

By default, update pc with the pc predicted by the branch predictor. If the cpu detects prediction was wrong at ID stage, update pc with jump target. If the cpu detects prediction was wrong at EX stage, update pc with branch target.

3.4 Parameter



```verilog
    parameter DATA_FORWARDING = 1;
parameter BRANCH_PREDICTOR = `BRANCH_SATURATION_COUNTER;
```

Figure 4. parameter at cpu.v

By using parameter, you can implement 4 cases. If DATA_FORWRDING is 0 and BRANCH_PREDICTOR is `BRANCH_ALWAYS_TAKEN, it is the baseline cpu. By changing the parameters implementation for extra credit is done. The submitted code parameter is as figure 4.

3.5 Performance comparison

By looking at num_inst port, the number of total instructions is known, which is 3036 instructions.

	TOTAL INSTRUCTIONS	TOTAL CYCLE	IPC
MULTICYCLE	3,036	11,702	0.259443
PIPELINE BASELINE		5,685	0.534037
PIPELINE DATA FORWARDING & BRANCH ALWAYS TAKEN		3,789	0.801267
PIPELINE SATURATION COUNTER		5,687	0.533849
PIPELINE DATA FORWARDING & SATURATION COUNTER		3,814	0.796015

Table 1. performance comparison with IPC

3.6 Accuracy of branch predictor

The accuracy of branch predictor can be easily calculated by looking at the value of register *num_branch* and *num_branch_miss*.

	TOTAL BRANCH	BRANCH MISPREDICT	ACCURACY
ALWAYS TAKEN	528	102	80.7%
SATURATION COUNTER		126	76.1%

4 Discussion

The results show that pipelined cpu performance is improved than multicycle implementation. Pipelined baseline is 105.8% faster than multicycle.

Also, with data forwarding it is 50% faster than pipeline baseline.

However, as the accuracy of saturation counter is lower than predicting branch always taken, the IPC of using 2-bit saturation counter decreased. Several reasons can be cause of lower performance.

First, the branch predictor was too simple. In this implementation 2-bit saturation counter was used, where there are only 4 states. By increasing the states, branch predictor accuracy will increase.

Second, not using global prediction. In this implementation only used local prediction.

Third, not using return address stack. The return address is stored at register files, and it is current pc + 1. This could never be predicted correctly without return address stack because it always changes.

Most of all the instructions was not long enough for saturation counter to work correctly.

5 Conclusion

Understood why pipelined cpu has better performance to multicycle cpu and successfully designed branch predictor and data forwarding. (all pass!)