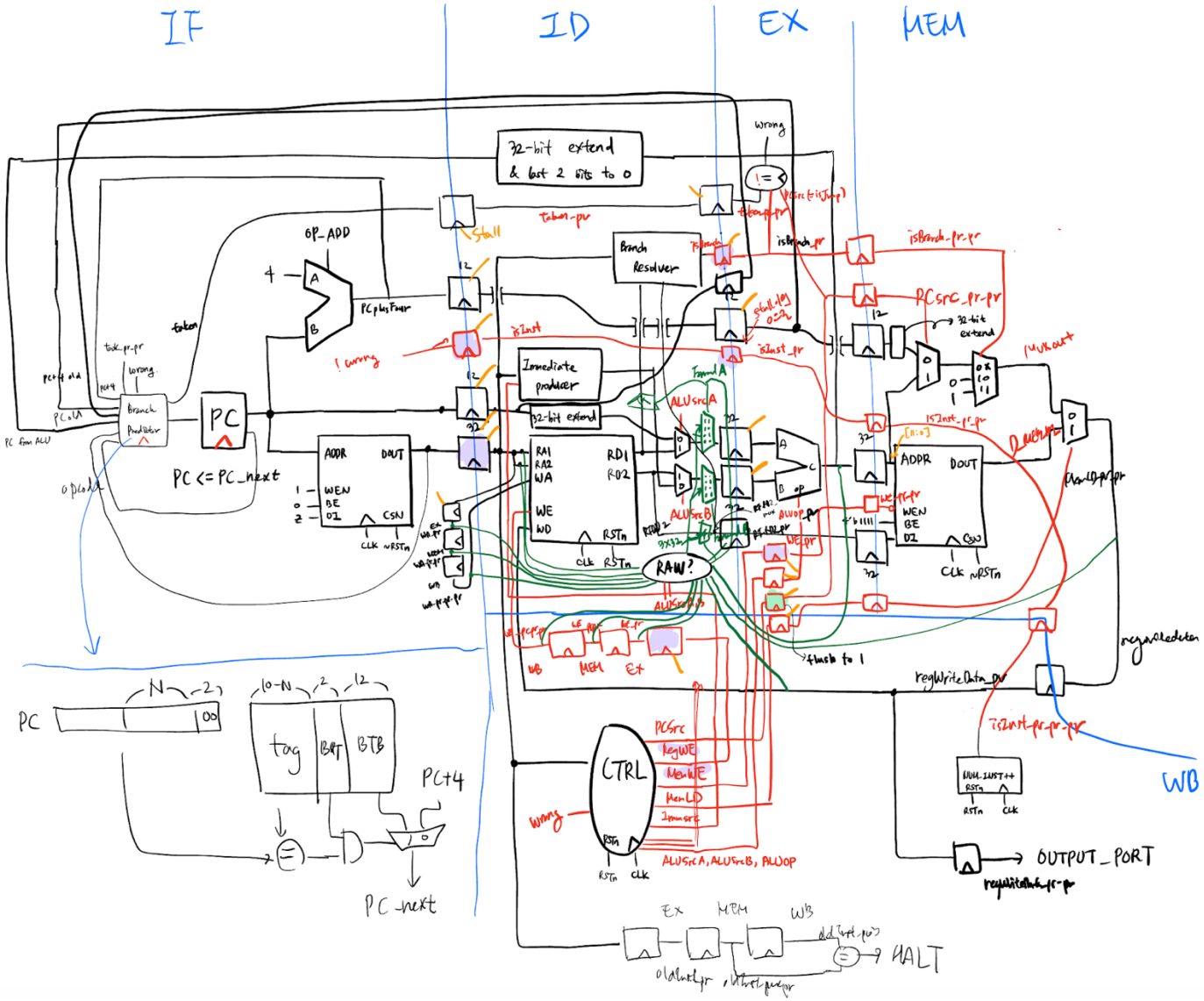
EE312 Lab #5 Report

20180610 정의준

1. Introduction

In Lab 5, we implement a pipelined CPU. We add forwarding module, branch predictor, and many pipeline registers to the multi-cycle CPU from Lab 4. Also, since a lot of data and control hazards may occur in a pipelined CPU, we should rigorously design the logical gates, control signals, and the error flags. Such error controlling includes flushing the wrong instructions and stalling the architecture to prevent RAW(Read-after-write) hazards.

2. Design



Compared to Lab 4, many modules were included to implement the pipelined CPU.

First, branch predictor at the far left replaces all the logics that determined the next PC value. I made a **BTB with 2-bit saturation counter** to predict the PC values. The BTB can contain up to 2^8=256 entries. Also, many D-flipflops were added in between every stage. The colored D-FF's are the ones that should be flushed when the branch is mis-predicted. Also, the ones with orange wire connected means that the register will not pass its value to the output when 'stall' signal is enabled.

Then, **forwarding module** is added at ID stage. This module determines which values will be used for the ALU operands, branch resolver, and the memory file. It is very important to not forward values that are flushed. I assumed that the register memory file may require a long period of time to read the register values, and made the forwarding module to fetch the values from three stages: EX, MEM, and WB.

Third, **branch resolver is added at EX stage**, which means that the **mis-predict penalty** will be of **2 cycles**. This module produces a 'wrong' signal when the PC after a branch or jump instruction is incorrectly predicted. Finally, a single bit flag 'isInst' is added to count the number of valid instructions; that is, this flag will help CPU to not count the flushed or stalled instructions that flow through the pipeline.

The control module has become simpler because we no more need the 5-state mealy machine to distinguish stages and the corresponding states. The control module will produce suitable control signals in every cycle - except when the CPU is stalled. Also, the 'wrong' flag can replace flush control signals to zero at the upcoming cycle. For instance, the MemWE & RegWE(write-enable) controls should be turned off when the instructions are flushed.

3. Implementation

About naming the pipeline registers and wires, I added _pw to at the end of the registers and wires, which means 'pipeline wire.' If the value is connected to multiple pipeline registers, I named them as _pw_pw and _pw3, if there are 2 or 3 pipeline registers.

We do not necessarily have to flush every data and control signal in the pipeline. Instead, we can only flush some signals that can change the architectural states, for instance the WE control signals and the isBranch signal. If the WE signals are all turned off, the architectural state of register and memory will not change no matter what values are flowing through the pipeline. I highlighted the registers that should be flushed in purple and green.

Also, stalling the pipeline should only occur on the IF->ID and ID->EX registers. I connected orange wires to the D-FF's as the figure to make the register to not pass the input value to the output.

For initializing BHT's 2-bit counter, I let its value to be 2'b10. This is because most branches are made to be 'taken' from the start of the program. Changing this number to 2'b00 or 2'b11 changed few numbers of total cycles, but the differences were a quite negligible.

The followings are some brief explanations for each modules used in my implementation.

1. BCOND.v

This module produces a 2-bit binary. The first bit indicates whether the instruction is a branch instruction or not. The second bit means whether the branch is taken or not.

2. RESOLVER.v

This module produces the 'wrong' signal on every negedge. Wrong signal becomes 1 when the instruction is jump or mis-predicted branch. If 'wrong' signal is activated, some corresponding registers will be flushed and/or stalled. The reason why I used negedge statement to update 'wrong' is to prevent signal hazards, because some registers were not properly flushed when the branch prediction is wrong.

3. BP.v

This module is a branch predictor. I made two branch predictors in the .v file to compare the number of cycles. The second module in the .v file is the branch predictor with BTB and the 2-bit saturation counter.

First, the overall scheme of the BTB is the same as BTB of the lecture note. A new entry is added to the BTB when the input instruction is jump or branch. Since the target address of the instruction is not ready in IF stage, the predictor first predicts PC+4 and waits for the resolver to see whether it was right or wrong. If it was wrong, the BTB is updated to the right target address which is calculated from the ALU.

The 'taken' signal is activated when the branch predictor predicts the branch to be taken. This signal will flow through the pipeline and be used as an input of the resolver. Also, BHT in the branch predictor will update its value when 'wrong' signal from the resolver is activated.

4. CTRL.v

I made a 2-state Mealy machine to make the control module, to distinguish the normal state, and the 'wrong' state. 'wrong' state is activated when the 'wrong' signal from the resolver is enabled. When the 'wrong' state is activated, write-enable signals for the Reg and Mem is disabled. This is to make sure that the flushed instructions do not change the architectural states. The other aspects of the control module is very much similar to the one from the single cycle lab.

5. DFF.v

This D-flipflop is updated little bit compared to the non-pipelined multicycle CPU. 'wrong' and 'stall' signals are added to the D-FF. When 'stall' is enabled, the register does not pass the value to the output. Also, when 'flush' is enabled, all registers except one flush the value to 0. The register for PCsrc signal is reset to 1 when 'flush' is enabled, and it is depicted in 'green' in the above figure.

6. FORWARD.v

```
if(RegWE_pw && RF_RA1 == RF_WA1_pw && ALUsrcA && isInst_pw) fwdA = 2'b01;
else if(RegWE_pw_pw && RF_RA1 == RF_WA1_pw_pw && ALUsrcA && isInst_pw_pw) fwdA = 2'b10;
else if(RegWE_pw3 && RF_RA1 == RF_WA1_pw3 && ALUsrcA && isInst_pw3) fwdA = 2'b11;
else fwdA = 2'b00;
```

The 32-bit operands for the ALU and the BCOND is connected to the 4-to-1 muxes, because the operands can be the forwarded values. The controller of the mux is produced by this forwarding module. Considering

the conditions as forwarding module in the lecture note, I used the above code to implement the forwarding module. fwdA in the above code is for controlling the ALU's first operand. If it is 00, the operand comes from the register file. If it is 01, 10, or 11, the value is forwarded from the EX, MEM, and WB stage respectively. We have to make sure to forward the 'latest' value, which will be in the 'closest' state from the ID stage. Also, we have to make sure that the flushed value is not forwarded. This is considered by adding isInst_pw_pw to the if statement as the above.

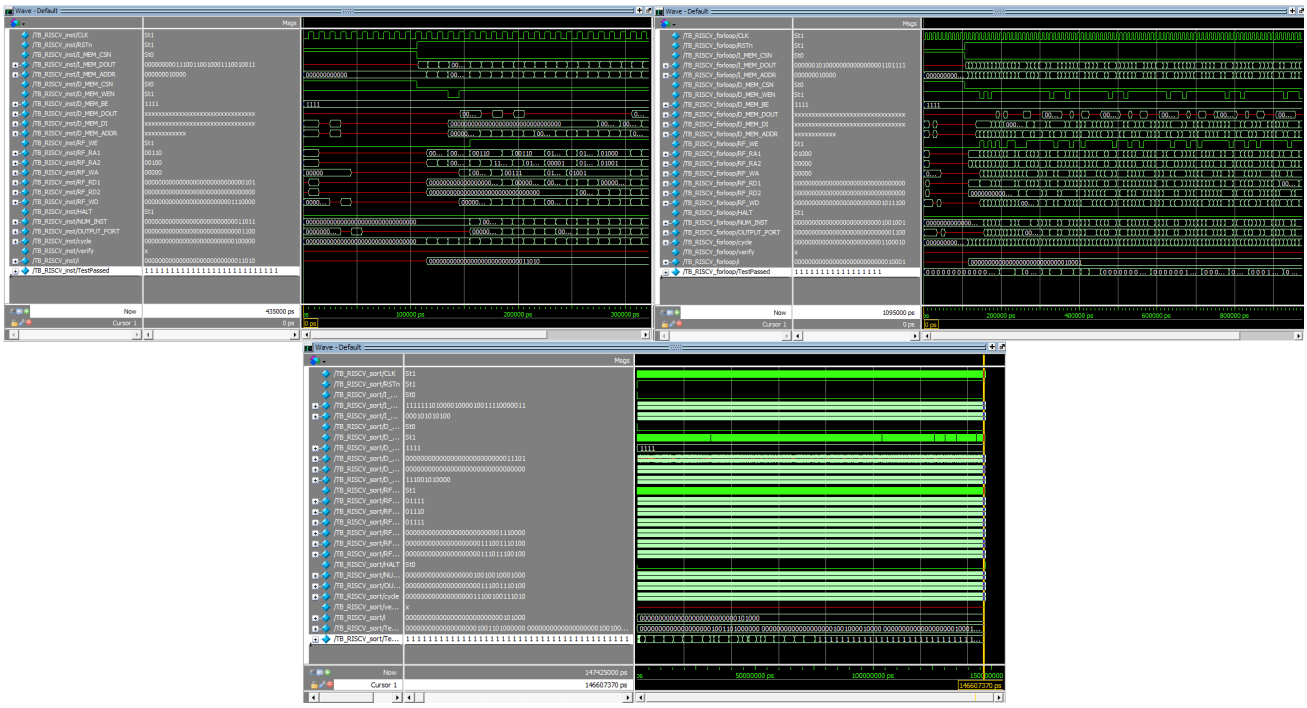
7. IM.v

This module produces the immediate for I-type instruction. It is not changed from the Lab 4.

8. MUX.v

This module has the implementations for 4-to-1 mux and 2-to-1 mux. I made this file to simplify the TOP.v file's code.

4. Evaluation



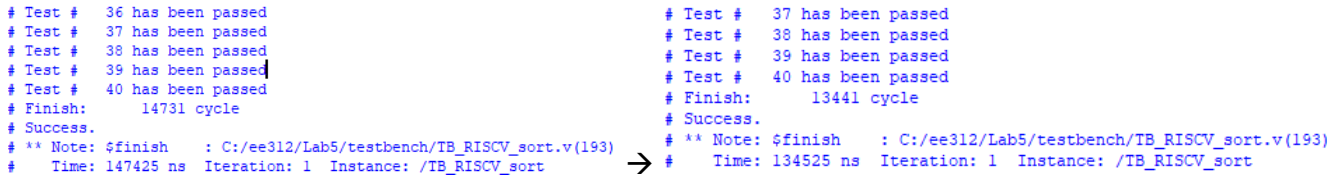
```
# Test #    23 has been passed
# Test #    24 has been passed
# Test #    25 has been passed
# Test #    26 has been passed
# Finish:          32 cycle
# Success.
# ** Note: $finish    : C:/ee312/Lab5/TB_RISCV_inst.v(179)
#    Time: 435 ns  Iteration: 1  Instance: /TB_RISCV_inst
```

Always non-taken branch → Branch predictor with 2-bit saturation counter and BTB

```
# Test #    14 has been passed
# Test #    15 has been passed
# Test #    16 has been passed
# Test #    17 has been passed
# Finish:         104 cycle
# Success.
# ** Note: $finish    : C:/ee312/Lab5/testbench/TB_RISCV_forloop.v(167)
#    Time: 1155 ns  Iteration: 1  Instance: /TB_RISCV_forloop
```

→

```
# Test #    14 has been passed
# Test #    15 has been passed
# Test #    16 has been passed
# Test #    17 has been passed
# Finish:          98 cycle
# Success.
# ** Note: $finish    : C:/ee312/Lab5/testbench/TB_RISCV_forloop.v(167)
#    Time: 1095 ns  Iteration: 1  Instance: /TB_RISCV_forloop
```



I have passed **3 out of 3** given testbenches. With **always-non-taken** branch predictor, I was able to get **32, 104, and 14731 cycles** for each testbench. With the **2-bit saturation BTB**, I was able to improve the CPU to use **98, 13441 cycles** on the last two testbenches. The BTB let the testbench to run with -5.8%p and -8.8%p smaller number of cycles on the for-loop TB and the sort TB.

5. Discussion

I tried to separate the logic circuit into many modules, to simplify the code in the TOP.v file. I am very happy that the "always" statement in the TOP.v file is very short and simple.

The debugging part was very confusing. It has taken a lot of time to find that my BTB was not working properly at the first time. I was barely able to fix the BTB by looking at the waveforms when the branch prediction was wrong.

There are more rooms for improving(=reducing) the number of cycles. First, we can change the overall design to reduce the mis-prediction penalty. To do so, we should move the branch resolver to ID stage. Also, we can design a better branch predictor. I saw that one branch in 'sort' testbench is bouncing between 'taken' and 'non-taken'. So, we were not able to predict this branch correctly, where the BHT were almost always wrong. By using some advanced branch-predicting techniques that we have talked about in the lectures, we'll be able to predict such branches more accurately.

6. Conclusion

In this lab, I was able to experience how a rigorous logic design should be done. It has taken about 5 hours to think and draw the logic circuit of the CPU. Also, the circuit design has been revised many times when I noticed a flaw in my design. However, it was very fun to think and imagine about multiple instructions flowing through my CPU at the same time. Thinking about the CPU's design to make it work with all the on-flight instructions was very challenging but interesting.