Lab 5: Pipelined CPU Lab

EE312 Computer Architecture

Professor: Minsoo Rhu

TA (Lab 5): Youngeun Kwon, Head TA : Youngeun Kwon

EMAIL: kaist.ee312.ta@gmail.com

1. Overview

Lab 5 is intended to give you a hands-on experience in designing a modern microprocessor, whose operations are conducted in a pipelined fashion. Based on the concepts and skills you have acquired through the lab assignments, you are now ready to implement a pipelined CPU. Through this lab assignment, you will be gaining an in-depth understanding on the fundamentals of designing a CPU microarchitecture.

2. Backgrounds

Why Do We Use Pipelining?

In a pipelined CPU, each instruction utilizes difference parts of CPU. If the pipeline stages are not stalled, the pipelined CPU completes one instruction per every clock cycle, thereby enhancing the throughput. Note that latency experienced by the instruction remains unchanged.

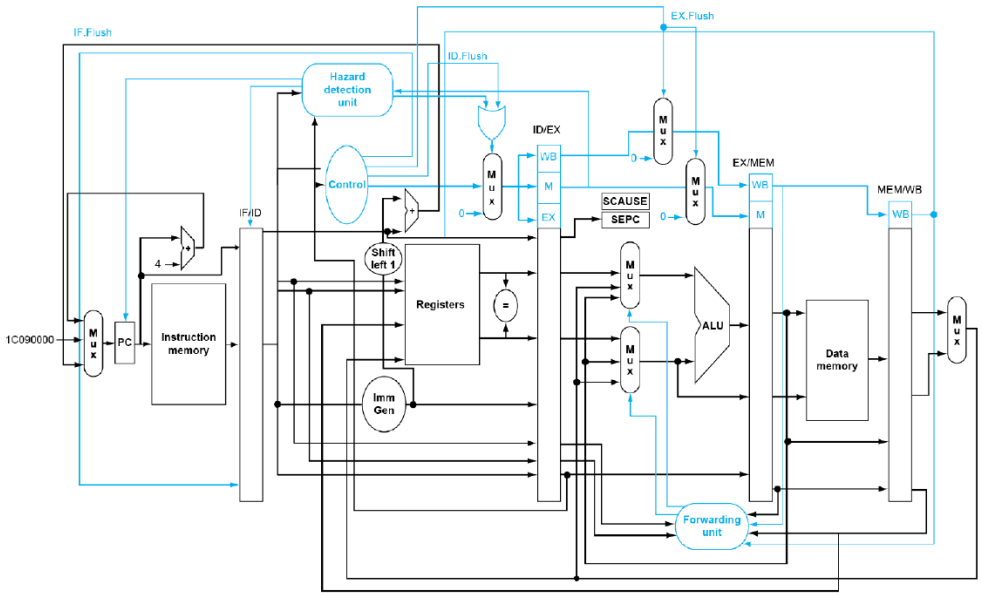

Figure 1. Pipelined CPU

Pipeline Implementation

You won't need to significantly modify the data path used in the multi-cycle CPU; however, you will be needing a bunch of additional registers called the pipeline registers, in order to forward the control signals and data that are needed in each pipeline stage. As always, when and where to generate the control information depends on the implementation, but one option is to generate the control information in the ID stage and forwarded to each pipeline stages. After certain control information is used and won't be used anymore, you don't need to forward that control information to the next pipeline stage.

The Hazards

In a pipelined CPU, problems called hazards arise, which wouldn't have arisen if it were a multi-cycle CPU. The definition of "hazard" is the situation where the next instruction cannot be executed in the following clock cycle. The hazards are of three kinds: data hazards, control hazards, and structural hazards.

Fundamentally, the reason why a hazard occurs is because the instruction cannot see the change in the architectural states that has been changed (or will be changed) by the previous instruction unless it waits for the previous instruction to change the architectural stages. However, we don't want every instruction to wait until the previous instruction changes the architectural stages, so we need a clever way to resolve the hazards.

Data Hazard

In Figure 2, the first *addi* instruction changes *$ra*; the following instructions use *$ra* as an operand. As a result, the following instructions have to wait until the first *addi* instruction changes the architectural states.

Instead of stalling the pipeline stages until the first *addi* changes the architectural states, bypassing the results to the following instructions can reduce clock cycles when the first *addi* reaches the WB stage, which is called "forwarding". More specifically, when the second instruction reaches the EX stage, we can bypass the result of the first instruction that is in the MEM stage. To enable forwarding, we need a unit called the "forwarding unit" that detects this situation and determines when and where to bypass the results. See section 5 for the grading details.

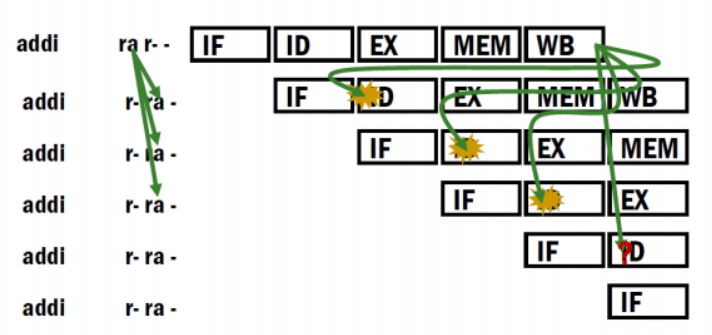

Figure 2. Data Hazard

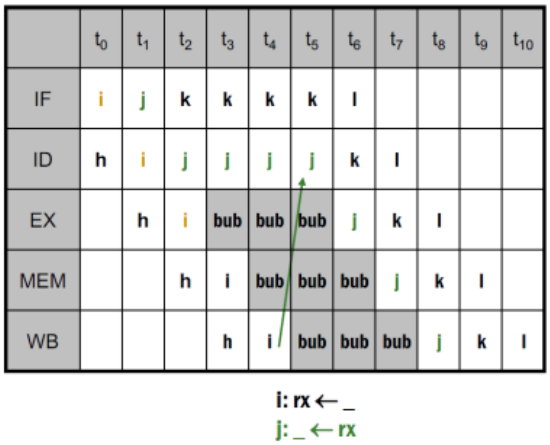

Figure 3. Data Hazard

Control Hazard

Another hazard is called the "control hazard". All the instructions in the RISV-V ISA need the PC value in the IF stage. However, generating the next PC value is different depending on the type of instruction. Therefore, the following instruction needs to wait until the next PC is generated. In Figure 4, the J type instruction produces the next PC value in the ID stage, so the following instruction needs to wait one cycle for the next PC value to be ready. Actually, we need to stall the pipeline stages at least one cycle after every instruction regardless of its type. Is it good?

	R/I-Type	LW	SW	Br	J	Jr
IF	use	use	use	use	use	use
ID	produce	produce	produce		produce	produce
EX				produce		
MEM						
WB						

Figure 4. Use-and-produce of PC

Definitely not. Therefore, we have to predict the next PC value and if it is mis-predicted, meaning the predicted next PC value turns out to be wrong, we flush the pipeline stages and fetch correct instructions.

The most naïve way to predict the PC value is predicting always PC+4, because most of the time, the right next instruction is executed unless the current instruction is a branch or jump instruction.

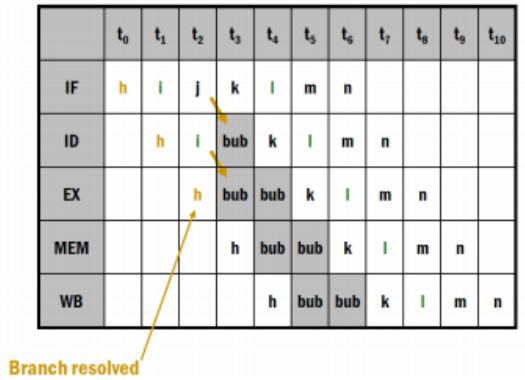

Figure 5. Control Hazard

Not surprisingly, there are many cleverer prediction methods than this, for example, BTB, BHT, and GShare. To implement such prediction method, we need to make a unit called the "branch prediction" that predicts the next PC value. See section 5 for the grading details.

3. Files

In *Lab 5*, you are given files that are in three folders:

1. *template* folder includes templates of the pipelined CPU.
2. *testbench* folder includes files you can test with.
3. *testcase* folder includes instruction streams in either assembly code or binary format that can give a hint for.

In the *template* folder, you are given four files:

1. *Mem_Model.v* defines the memory model.
2. *REG_FILE.v* defines the register file.
3. *RISCV_CLKRST.v* generates the clock signal.
4. *RISCV_TOP.v* includes templates you start with.

You **SHOULD NOT** modify *Mem_Model.v, REG_FILE.v, and RISCV_CLKRST.v*. You only need to implement modules that consist the data path in *RISCV_TOP.v*. You may create additional files that include modules that consist the control path in the *templete* folder.

In the *testbench* folder, you are given three *testbench* files:

1. TB_RISCV_forloop.v
2. TB_RISCV_inst.v
3. TB_RISCV_sort.v

In the *testcase* folder, you are given five files:

1. *asm/forloop.asm*
2. *asm/sort.asm*
3. *hex/forloop.hex*
4. *hex/inst.hex*
5. *hex/sort.hex*

You can test your CPU with *testbench* files in the *testbench* folder. Before you test with *testbench* files, you need to specify the instruction stream stored in the *testcase* folder. For example, if you want to test with *TB_RISCV_forloop.v*, you must change the file path to *testcase/hex/forloop.hex*. You can find a human-readable assembly code in *testcase/asm/forloop.asm*, which can be helpful for debugging.

The TB file reads hex from the hex file and puts instructions in the instruction memory. Then, it executes the instruction from the first instruction in the memory according to the instruction flow. While executing the program, if the number of executed instructions becomes the pre-defined number, your output port is compared to the expected value.

4. Pipelined CPU Lab

In *Lab 5*, you are required to implement a pipelined CPU.

The template assumes the system with following rules:

1. The instruction memory and data memory is physically separated as two independent modules (check the testbench files).
2. The overall memory size is 4KB for instructions and 16KB for data.
3. The memory follows byte addressing, which supports accessing individual bytes of data rather than only larger units called words (for instructions, this is naturally handled whereas for data, this is enabled using the D_MEM_BE port, check the testbench files and the RISCV_TOP.v file).
4. **Little-endian**
5. The control path and data path should be separated.
6. As we have not covered the concept of "Virtual Memory" in this course just yet, you can assume that the lower N-bits of the instruction and data memory addresses are used as-is to access instruction memory and data memory.
    a. For accessing instructions, use (Effective_Address & 0xFFF) as the translation function
    b. For accessing data, use (Effective_Address & 0x3FFF) as the translation function
7. The initial value of the Program Counter (PC) is 0x000.

8. The initial value of the stack pointer is 0xF00.

Your implementation **MUST** comply with the following rules:

1. RISC-V ISA (RV32I) + custom instructions
   A. Refer to RISC_V_101.pdf for the detailed descriptions about the custom instructions.
2. You're required to implement a 5 stage pipeline
3. You need to resolve data hazards and control hazards
   Refer the grading policy
4. You need to implement only below instructions
   A. JAL
   B. JALR
   C. BEQ, BNE, BLT, BGE, BLTU, BGEU
   D. LW
   E. SW
   F. ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, SRAI
   G. ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND
   H. MULT, MODULO, IS_EVEN

The template of memory and register file is already given to you. You are only required to implement the control and data path of the pipeline. If you implement correctly, you will see a "Success" message in the console log when you run the testbench file. The TAs recommend you to carefully design which modules are needed, how to connect them, and which control signals are necessary to transfer to the data units, before you start to write the code.

Terminal condition

Since we don't implement instructions which are used to transfer control to the operating system, we set a flag instruction to quit the program. If you get **the two instructions** "0x00c00093 //(addi x1, x0, 0xc)" and "0x00008067 //(jalr x0, x1, 0)" **in a row**, you should halt the program. HALT output wire should be set to 1.

Simulation

To test your CPU, you need to implement two additional output, which are *NUM_INST* and *OUTPUT_PORT*.

1. NUM_INST: the number of executed instructions

2. OUTPUT_PORT: the output result,

    a. If the instruction has a destination register (*rd*), then the value that is supposed to be written in the destination register should also be written to *OUTPUT_PORT*.

    b. If the instruction is a branch instruction, 1 is written to *OUTPUT_PORT* if taken; otherwise, 0 is written.

    c. If the instruction is a store instruction, the target address of the store instruction is written to *OUTPUT_PORT*.

5. Grading

The TAs will grade your lab assignments with three *testbench* files in the *testbench folder* that are already given to you. If you passed all the testcases but your pipeline stalls every data hazard and control hazard, you will get 75% of the total score. Nonetheless, even if you pass all the test, if your design contains flaws, **we may deduct significant amount of your points**. If you implement the forwarding unit, you will get additional 10%. If you implement the "always-not-taken" branch predictor, you will get another additional 10%. Therefore, the implementation takes up 95% of the total score, and the report takes up the remaining 5%. You can get a full score (100%), even if you implement the "always-not-taken" branch prediction. However, you will get extra credits if you implement the "always-taken" branch prediction with the BTB or "2-bit saturation counter" with the BTB. If you implement the "always-taken" branch predictor with the BTB, you will get 15% of the total score, instead of 10% that you would get if you implemented the "always-not-taken" branch predictor. If everything goes perfectly, you will get total 105%. On the other hand, if you implement the "2-bit saturation counter with the BTB, you will get 20% of the total score. Again, if everything goes perfectly, you will get total 110%. **If you do not design your CPU in a five-stage pipeline CPU fashion, you will get zero points. The lab report takes up 5% of your Lab 5 score.** Number of cycles is going to be compared to check your branch predictor and data forwarding unit. Also, there will be additional deduction if you do not submit your **honor code**.

*** **Please note that** you need to **explicitly** describe whether or not you implement a forwarding unit and a branch predictor, which policy you adopted and give some numbers or figures to prove that you've implemented those modules in your report. **If you do not mention these things in your report, you may lose some of your points**. Writing down comments describing where you implement a forwarding unit and a branch predictor in your source code would be very helpful to TAs recognizing them.

Also, there are some guidelines that can affect your grades.

- **Do not** use "blocking operator" in the sequential logic part (e.g., always @(posedge clk)). Use non-blocking operator for sequential logic and blocking operator for combinational logic. If you don't abide by this rule, there will be a deduction.
- **Do not** use delay operator (e.g., #50, #100). If you use "delay operator" in your code (except the given testbench code), you will automatically get zero points for that lab assignment.
- **Do not** use "wait()" function for your design. We will not give any points for a design that is using "wait()" function.

- **DO NOT CHEAT**

6. Some Statistics and TA's Recommendations

1. Students remove all of their source files 1.5 times to re-design their pipelined CPU
2. It takes 10 (extremely rare) ~ 60 hours to complete this assignment. (Median is 40 ~ 50 hours) So we highly recommend you to start this assignment as early as you can.
3. We recommend you draw your design diagram as detailed as you can before you start to code in Verilog.
4. While drawing your design diagram, open an Excel spreadsheet and try to write down all the control signals you need for each instruction.
5. We recommend you start Verilog coding after you have your final version of the diagram and Excel sheet of your control signals, which are verified through your thought experiment.

7. Lab Report Guidance

You are required to submit a lab report for every lab assignment. You can write your report either in Korean or English. We don't want you to waste too much time writing a lab report. **Three to five pages** are enough for the report unless you have more to show. You don't need to have too much concern about the report.

Your lab report **MUST** include the following sections:

1. Introduction
    a. *Introduction* includes what you think you are required to accomplish from the lab assignment and a brief description of your design and implementation.
2. Design (Try to assign most of your lab report pages explaining your design)
    a. *Design* includes a high-level description of your design of the Verilog modules (e.g., the relationship between the modules).
    b. Figures are very helpful for the TAs to understand your Verilog code.
    c. The TAs recommend you to include figures because drawing the figures helps you how to *design* your modules.
    d. **Explicitly** describe whether or not you implement a forwarding unit and a branch predictor, which policy you adopted.
3. Implementation
    a. *Implementation* includes a detailed description of your implementation of what you design.
    b. Just writing the overall structure and meaningful information is enough; you do not need to explain minor issues that you solve in detail.

c. **Do not copy and paste your source code.**

4. Evaluation
    a. *Evaluation* includes how you evaluate your design and implementation and the simulation results.
    b. *Evaluation* **must** include how many tests you pass in the *testbench folder*.
5. Discussion
    a. *Discussion* includes any problems that you experience when you follow through the lab assignment or any feedbacks for the TAs.
    b. Your feedbacks are very helpful for the TAs to further improve *EE312* course*!*
6. Conclusion
    a. *Conclusion* includes any concluding remarks of your work or what you accomplish through the lab assignment.

7. Requirements

You **MUST** comply with the following rules:

- You should implement the lab assignment in **Verilog**.
- You should only implement the **TODO** parts of the given template.
- You should name your lab report as **Lab5_YourName1_StudentID1_YourName2_StudentID2.pdf**.
- You should name your honor pledges **HonorPledge_YourName1_StudentID1.pdf** and **HonorPledge_YourName2_StudentID2.pdf**
- You should compress the honor pledge(s), lab report, and source codes, then name the compressed zip file as **Lab5_YourName1_StudentID1_YourName2_StudentID2.zip**, and submit the zip file on the KLMS. *All* names of your submitted files, including your zipped file, **should not contain any spaces in them**. If you need to put a space in the name, substitute it with an underscore ("_") instead. You **will be penalized** if you fail to comply with this rule.

8. FAQ

- The size of BTB is your design choice.
- Branch miss prediction penalty could be 1 or 2 cycles. Choosing one from there is your design choice. (This is related to choosing which pipeline stage to resolve a branch)
- We decided not to provide the expected execution cycle of the test benches.