# Lab 4: Multi-cycle CPU Lab

EE312 Computer Architecture

Professor: John Kim

TA (Lab 4): Seongmin Song, Bongjoon Hyun

EMAIL: kaist.ee312.ta@gmail.com

Assigned Date: Oct 13, 2022

Due Date: Nov 1, 2022

## 1. Overview

*Lab* 4 is intended to give you a hands-on experience in designing a modern microprocessor, whose operations are conducted within multiple clock cycles. Based on the concepts and skills you have acquired through *Lab 1, 2* and *3*, you are now ready to implement a multi-cycle CPU (Fig. 1). Through this lab assignment, you will have gained an in-depth understanding on the fundamentals of designing a CPU microarchitecture.

## 2. Backgrounds

### Why Do We Need a Multi-cycle CPU?

Single-cycle CPU completes all instructions within one clock cycle. Since instruction latency is not equal for all instructions, the clock signal must be the longest latency to guarantee functionality. Fig. 2 represents the instruction latency for each instruction. As you can see in Figure 2, the R-type and I-type instructions do not need the MEM stage and a branch instruction does not need the MEM and WB stage. To adapt to the varying latency of instructions, a multi-cycle CPU takes multiple cycles rather than a single cycle; different instructions may require different number of clock cycles.

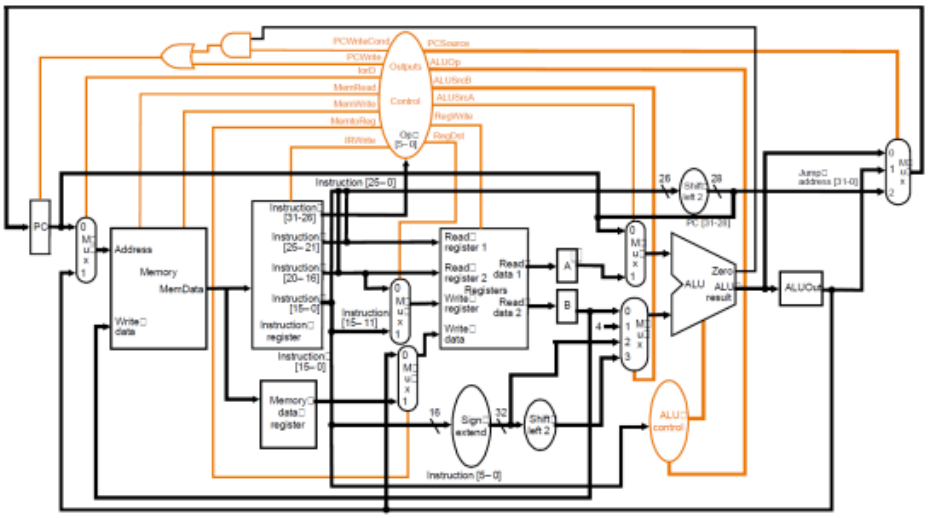


**Figure 1. Data & Control Path**

| steps | IF | ID | EX | MEM | WB | Delay |
|---|---|---|---|---|---|---|
| resources | mem | RF | ALU | mem | RF | |
| R-type | 200 | 50 | 100 | | 50 | 400 |
| I-type | 200 | 50 | 100 | | 50 | 400 |
| LW | 200 | 50 | 100 | 200 | 50 | 600 |
| SW | 200 | 50 | 100 | 200 | | 550 |
| Branch | 200 | 50 | 100 | | | 350 |
| Jump | 200 | | | | | 200 |

**Figure 2. Instruction latency Table**

**Multi-cycle CPU Implementation**

In our memory model, the register file and ALU are designed to perform within one clock cycle. Therefore, unlike what you have learned in the class, you are required to design a multi-cycle CPU that operates five stages like Figure 3. Each stage takes one clock cycle. The actual Finite State Machine (FSM) for RISC-V can be slightly different from Figure 3 because of the JAL and JALR instruction; other details are equal except the two instructions.

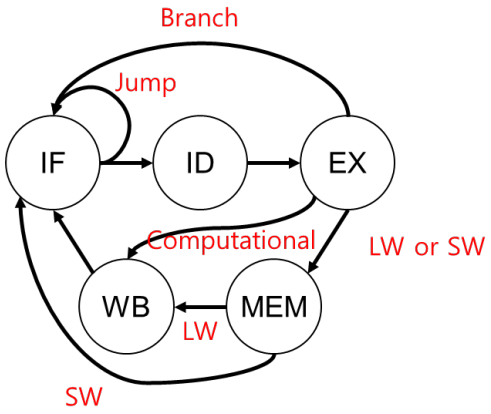


**Figure 3. FSM for Multi-cycle CPU**

The TAs recommend you to make the transition table when you finish designing the FSM. A simple example of the transition table is shown in Figure 4. Moreover, the transition table for RISC-V can be slightly different from Figure 4. The transition table would help you implement the multi-cycle CPU.

| state label | contrl flow | conditional targets | | | | |
|---|---|---|---|---|---|---|
| | | R/I-type | LW | SW | Br | Jump |
| $IF_1$ | next | - | - | - | - | - |
| $IF_2$ | next | - | - | - | - | - |
| $IF_3$ | next | - | - | - | - | - |
| $IF_4$ | go to | ID | ID | ID | ID | $IF_1$ |
| ID | next | - | - | - | - | |
| $EX_1$ | next | - | - | - | - | |
| $EX_2$ | go to | WB | $MEM_1$ | $MEM_1$ | $IF_1$ | |
| $MEM_1$ | next | - | - | - | | |
| $MEM_2$ | next | - | - | - | | |
| $MEM_3$ | next | - | - | - | | |
| $MEM_4$ | go to | - | WB | $IF_1$ | | |
| WB | go to | $IF_1$ | $IF_1$ | | | |

**Figure 4. Transition Table for Multi-cycle CPU**

**Control Unit Implementation**

You have two options on how you could go about implementing your control unit. One option is to simply follow the FSM-based control unit design (as explored during Lab 2). The other option is to employ the "microprogram + microcoding" based controller (see Figure 5 below). You can choose either ways but if you choose to implement the control unit based on the first option, make sure you draw the complete state diagram in your report.

If you're motivated enough, you can choose to implement microprogram-based control unit. Figure 5 represents microarchitecture for the control unit. The important thing is the "microprogram counter", which represents the state ID. You will generate control signals by using **instruction and the microprogram counter**. You might need a variable called *counter* or *micro_counter* to keep track of your stages and generate control signals.

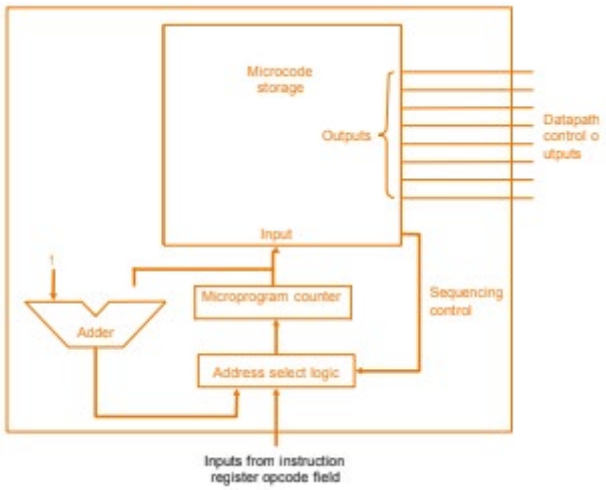


**Figure 5. Microarchitecture for the Control Unit**

### 3. Files

In *Lab 4*, you are given files that are in three folders:

1. *template* folder includes templates of the multi-cycle CPU.
2. *testbench* folder includes files you can test with.
3. *testcase* folder includes instruction streams in either assembly code or binary format.

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 multi-cycle 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. Multi Cycle CPU Lab**

In *Lab 4*, you are required to implement a multi-cycle CPU.

The template assumes the system with following rules:

1. The instruction memory and data memory are 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. Since we have not covered the concept of "Virtual Memory" in this course 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. Each stage takes one clock cycle.
    A. e.g.) jump instruction takes one cycle.
3. You only need to implement the below instructions (**some instructions are removed**):
    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
4. The number of cycles to execute each instruction are as follows.
    A. I-type, R-type, **custom type instruction**, JAL, JALR: 4 cycles
    B. Branch instruction: 3 cycles
    C. Load instruction: 5 cycles
    D. Store instruction: 4 cycles

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 multi-cycle CPU. 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 instruction sequences "0x00c00093 //(addi x1, x0, 0xc) and 0x00008067 //(jalr x0, x1, 0)", you should halt the program. HALT output wire should be set to 1.

**Simulation**

To test your multi-cycle 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. **You will get zero points if the CPU microarchitecture is not implemented in a multi-cycle fashion** – in principle, all the testcases will show as "PASS" even if your implementation is based on a single-cycle microarchitecture.

Assuming your implementation is indeed based on a multi-cycle microarchitecture design, you will get 55% of your maximum possible score for this lab by passing all the three test cases (15% from *inst*, 20% from *forloop*

and 20% from *sort*). The remaining 40% of the score is determined by "**how well**" your multi-cycle CPU microarchitecture is designed. Nonetheless, even if you pass all the test and get the correct number of cycles, if your design contains flaws, **we may deduct significant amount of your points**. The rest of the 5% will be evaluated based on the report you've submitted. Further details follow below:

1. *inst* testbench comprises 25% of the total scores assigned for this lab; you will earn 15% if the test passes correctly, and the remaining 10% will be evaluated based on whether the total number of clock cycles elapsed is correct.
    a. For the integer computational instructions, each instruction should take 4 cycles to complete one instruction.
2. *forloop* comprises 35% of the total scores assigned for this lab; you will earn 20% if the test passes correctly, and the remaining 15% will be evaluated based on whether the total number of clock cycles elapsed is correct.
    a. *Reference cycles for the forloop test case is 310.*
3. *sort* testbench comprises 35% of the total scores assigned for this lab; you will earn 20% if the test passes correctly, and the remaining 15% will be evaluated based on whether the total number of clock cycles elapsed is correct.
    a. *Reference cycles for the sort testbench is 41478.*

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. 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 your time writing a lab report. Please keep the report **short**. **Three to four pages** are enough for the report unless you have more to show. You don't need to 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.
3. Implementation
    a. *Implementation* includes a detail 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 *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 **Lab4_YourName1_StudentID1_YourName2_StudentID2.pdf**.
- You should compress the honor pledge(s), lab report, simulation results, and source code, then name the compressed zip file as **Lab4_YourName1_StudentID1_YourName2_StudentID2.zip**, and submit the zip file on 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. **Note that if you do not submit honor pledge(s), your score will get deducted.**
- Make sure that all characters in your submission zip files and their content files' names be **in English**.
  (e.g., Lab1_GilDongHong_20220000.zip (O),
  Lab1_홍길동_20220000.zip (X),
  Lab1_GilDongHong_20220000_JohnDoe_20220001.zip (O),
  Lab1_홍길동_20220000_존도_20220001.zip (X)
  )