

Lab 3: Single-cycle CPU Lab

EE312 Computer Architecture

TA: 정기훈 / Kihoon Jung (EMAIL: jkih0021@gmail.com)

TA: 조상훈 / Sanghun Cho (EMAIL: chosanghoon1118@gmail.com)

Assigned date: 2019 / 09 / 26

Due date: 2019 / 10 / 17, 13:00 (before class)

1. Overview

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

2. Backgrounds

Control Path & Data Path

A CPU is composed of two types of units, which are the control units and data units. A control path consists of multiple control units, which “controls” the data path of the processor (e.g., ALUs, memory) how to respond to the instruction that is fetched to the processor. Likewise, a data path is a collection of functional units such as arithmetic logic units (ALUs) or multipliers, that perform data processing operations, registers, and buses. You can simply think of the data path as it literally means; the path where the data flow inside of the processor.

Single-cycle CPU

A single-cycle CPU processes an instruction per every single clock cycle. At every single clock cycle, the single-cycle CPU must perform Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory operation (MEM), and Write Back (WB) stages of a single instruction, so the architectural states are updated as the instruction “instructs” the CPU. The IF stage is to fetch the instruction that the Program Counter (PC) points to in the instruction memory. The ID stage is to decode the fetched instruction, figure out which operations are required to carry out, which is defined by the Instruction Set Architecture (ISA). The EX stage is where primitive operations (e.g., add, multiply, bit-wise operations) as well as memory address calculations are performed. The MEM stage is to perform memory operations (e.g., load, store) on specified address by the instruction. The WB stage is to update the architectural states of the processor; the values of registers or memory are updated as a result of executing the instruction. The exact control and data path are shown in Fig. 1, which is the MIPS version of a single cycle CPU design. Remember that the goal of Lab 3 is to design your own "RISC-V" version of such single cycle CPU microarchitecture.

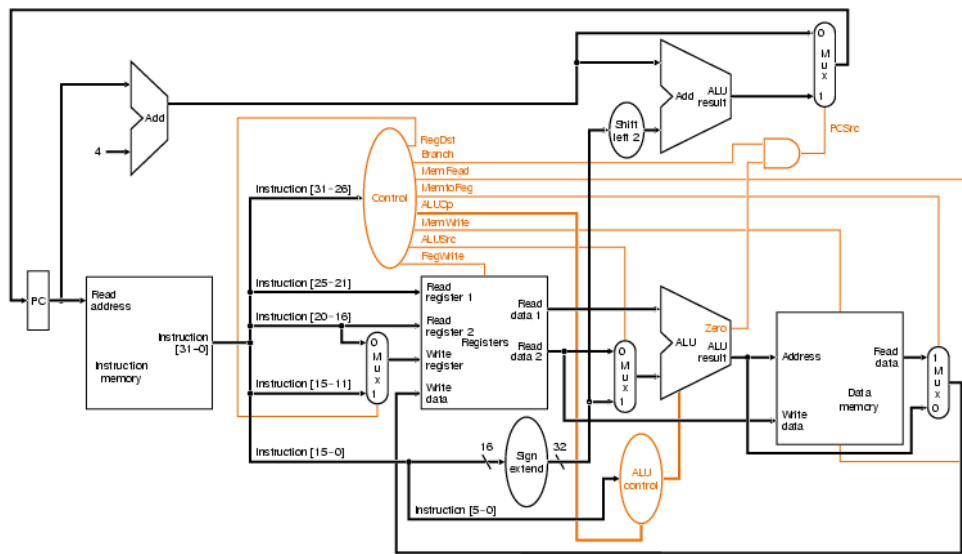


Figure 1. Control & Data Path of the Single-cycle CPU (MIPS ver)

Instruction Set Architecture (ISA)

Instruction Set Architecture (ISA) is an abstract model of a computer. An ISA defines everything that a machine language programmer needs to know to program the computer. “x86”, “ARM”, “MIPS”, and “RISC-V” are common ISAs that you have probably heard of many times. As you have already been noticed in the lectures, you are going to implement a CPU that follows the RISC-V ISA throughout the lab assignments.

RISC-V is an open Instruction Set Architecture (ISA) based on established reduced instruction set computing principles. The TAs already gave you a RISC-V tutorial, so we hoped you have gained a useful amount of knowledge to complete *Lab 3*. However, if you have anything unclear about the RISC-V ISA, please review the RISC-V tutorial and RISC-V manual. In *Lab 3* and the rest of the lab assignments, we focus on the basic 32-bit integer set, which is RV32I. A full documentation of RV32I is given to you, please refer to the documentation to make sure your implementation of the single-cycle CPU follows the RV32I correctly. You can refer to the following manual to get an idea on the semantics of some of the RV32I instructions (remember though that your goal is to implement the RV32I instruction set, not the tinyRV version below which only accounts for a subset of RV32I).

<https://www.csl.cornell.edu/courses/ece4750/handouts/ece4750-tinyrv-isa.txt>

3. Files

In *Lab 3*, you are given files that are in three folders:

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

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* defines the clock signal.
4. *RISCV_TOP.v* includes templates you can 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 *template* 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 single-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. Single Cycle CPU Lab

In *Lab 3*, you are required to implement a single-cycle CPU, which is the simplest form of CPU.

Your implementation **MUST** comply with the 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 RISC_V_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.
9. You need to implement only below instructions
 - a. LUI, AUIPC
 - b. JAL
 - c. JALR
 - d. BEQ, BNE, BLT, BGE, BLTU, BGEU
 - e. LB, LH, LW, LBU, LHU,
 - f. SB, SH, SW
 - g. ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, SRAI
 - h. ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND

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 single-cycle CPU. If you implement correctly, you will see a “Success” message in the console log when you run the testbench file, as shown in Fig. 2. 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.

```
VSIM 2> run -all
# Finish:      73 cycle
# Success.
# ** Note: $finish      : C:/Users/sh520/Desktop/lab/rv32i-verilog/3. single-cycle/TB_RISCV.v(166)
# Time: 845 ns  Iteration: 1  Instance: /TB_RISCV
```

Figure 2. Success Message

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), 0x00008067 //(jalr x0, x1, 0)”, you should halt the program. HALT output wire should be set to 1.

Simulation

To test your single-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 and one additional *testbench* file that is hidden to you. Your score is determined how many tests you pass.

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 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.
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 vending_machine_TB.v
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 **Lab3_YourName_StudentID.pdf**.
One report per team
You do not have to use the same teams as previous labs
- You should compress the lab report, simulation results, and source code, then name the compressed zip file as **Lab3_YourName_StudentID.zip**, and submit the zip file on the KLMS.
- You should not change the interface of the vending machine defined in *vending_machine.v*.

Reference

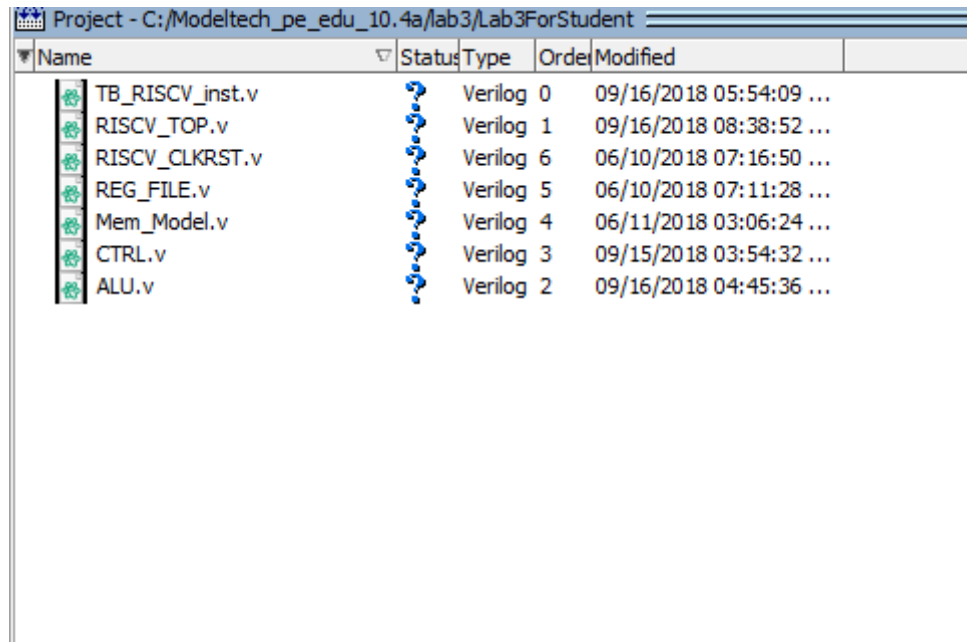
1. <https://www.csl.cornell.edu/courses/ece4750/handouts/ece4750-tinyrv-isa.txt>

How To Start Guide

1. Load all files in the template folder, your CPU implementation and a testbench that you want to test.

The CTRL.v and ALU.v files are one of TA's implementation of single cycle CPU.

CTRL.v is for control signal and RISC_V_TOP.v & ALU.v are for datapath.



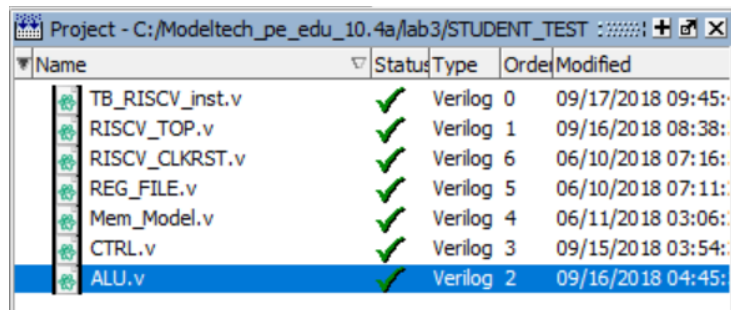
Project - C:/Modeltech_pe_edu_10.4a/lab3/Lab3ForStudent

Name	Status	Type	Order	Modified
TB_RISCV_inst.v	?	Verilog	0	09/16/2018 05:54:09 ...
RISCV_TOP.v	?	Verilog	1	09/16/2018 08:38:52 ...
RISCV_CLKRST.v	?	Verilog	6	06/10/2018 07:16:50 ...
REG_FILE.v	?	Verilog	5	06/10/2018 07:11:28 ...
Mem_Model.v	?	Verilog	4	06/11/2018 03:06:24 ...
CTRL.v	?	Verilog	3	09/15/2018 03:54:32 ...
ALU.v	?	Verilog	2	09/16/2018 04:45:36 ...

2. You should change file location in testbench files.

```
.NUM_INST (NUM_INST),  
.OUTPUT_PORT (OUTPUT_PORT)  
);  
  
//I-Memory  
SP_SRAM #(  
    .ROMDATA ("C:\\Users\\jjeong\\Lab3\\release\\testcase\\inst.hex"), //Initialize I-Memory  
    .AWIDTH (10),  
    .SIZE (1024)  
) i_mem1 (
```

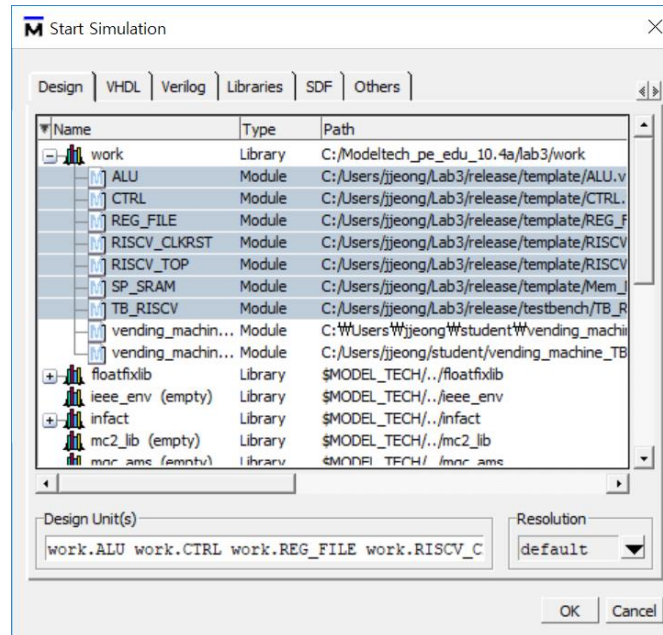
3. Compile all the files.



Project - C:/Modeltech_pe_edu_10.4a/lab3/STUDENT_TEST

Name	Status	Type	Order	Modified
TB_RISCV_inst.v	✓	Verilog	0	09/17/2018 09:45:...
RISCV_TOP.v	✓	Verilog	1	09/16/2018 08:38:...
RISCV_CLKRST.v	✓	Verilog	6	06/10/2018 07:16:...
REG_FILE.v	✓	Verilog	5	06/10/2018 07:11:...
Mem_Model.v	✓	Verilog	4	06/11/2018 03:06:...
CTRL.v	✓	Verilog	3	09/15/2018 03:54:...
ALU.v	✓	Verilog	2	09/16/2018 04:45:...

4. Choose all files relevant to single cycle CPU when you start simulation.



5. Run the simulation.

6. You get the simulation result.

```
# Finish:          24 cycle
# Success.
# ** Note: $finish   : C:/Users/jjeong/Lab3/release/testbench/TB_RISC_V_inst.v(175)
#   Time: 355 ns   Iteration: 1   Instance: /TB_RISC_V
```