Lab 4: Multi-cycle CPU Lab

EE312 Computer Architecture

Professor: John Kim

TA: Minhoo Kang (EMAIL: minhoo.kang@kaist.ac.kr)

Due date: May 3rd, 11:59pm

1. Overview

Lab 4 is intended to give you 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 instruction do not need the MEM stage and a branch instruction does not need the MEM and WB stage. To adapt 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.
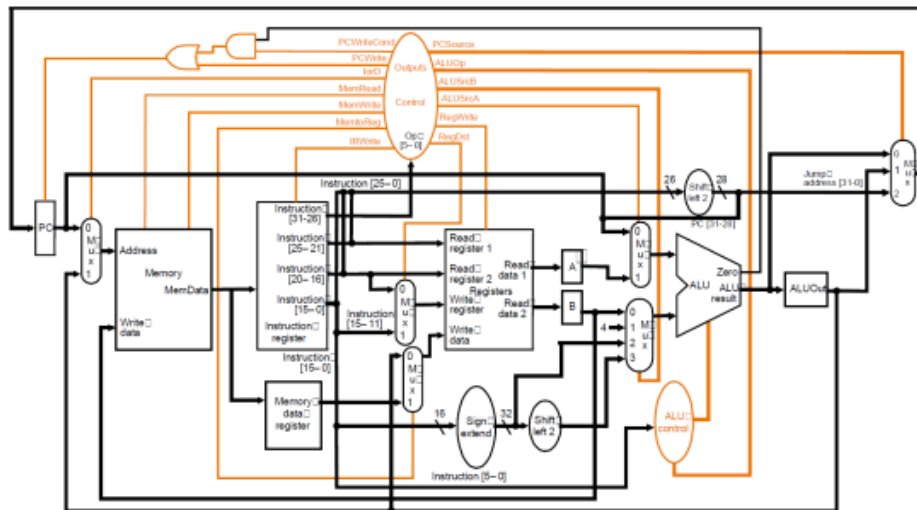
Figure 1. Data & Control Path

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.
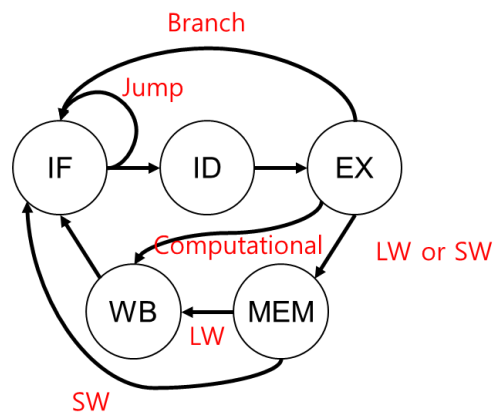
Figure 3. FSM for Multi-cycle CPU

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 (as discussed during Lecture 6, 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 add to a variable called *counter* or *micro_counter* to keep your stages and generate control signals.
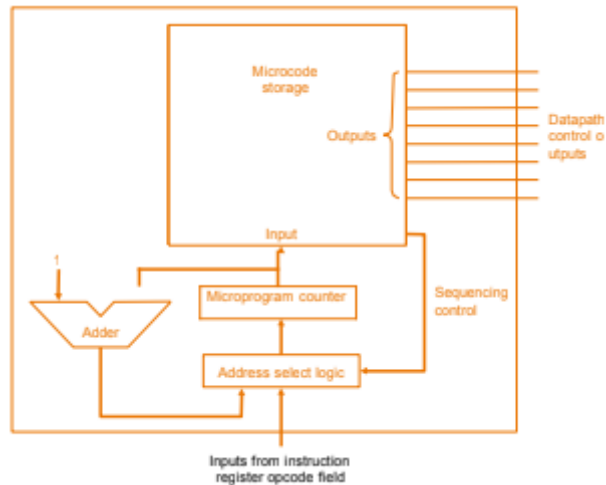
Figure 5. Microarchitecture for the Control Unit

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/foorloop.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. (ex. RISCV_TOP.v and RISCV_CTRL.v)

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)

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

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 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 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 follows 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.

b. TB_RISCV_inst only contains integer computational instructions

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.

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.

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.

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_Student1Name_Student1ID_Student2Name_Student2 ID.pdf**.

- You should compress the lab report, and source codes, then name the compressed zip file as **Lab4_Student1Name_Student1ID_Student2Name_Student2ID.zip**, and submit the zip file on the KLMS.

Code Review of Testbench Files

1. *TB_RISCV_inst* is a testbench file for testing the I-type and R-type integer computational instructions of RISC-V.

2. *TB_RISCV_forloop* and *TB_RISCV_sort* are testbench files that test a basic for-loop and a sort algorithm, respectively.

3. *TB_RISCV_inst* checks the correctness of the output value for every instruction execution; it executes one instruction at a time and checks the correctness of that instruction, then move on to the next instruction. The TAs recommend you to test with *TB_RISCV_inst* first since it tests the most primitive integer computational instructions.

4. In the testbench files,

   a. *riscv_clkrst1* generates the clock and reset signals,
   b. *riscv_top1* creates the data path of the processor core,
      a. It does NOT include major sequential logics (e.g., instruction/data memory, RF)
   c. *i_mem1* creates the instruction memory,
   d. *d_*mem1 creates the data memory,
   e. *reg_file1* creates the register files.

Clock Generator

All modules must be able to be initialized by the *RSTn* signal, generated by *riscv_clkrst1*. In addition, the core, memory, and the register file should be synchronous to the *CLK* signal.

In *riscv_top1*, you only need to implement the core module. The core module receives the *CLK* signal to be able to be synchronous to the clock signal.

The Core Module

In the core module, *I_MEM_CSN, I_MEM_DI, I_MEM_ADDR, D_MEM_CSN, D_MEM_DI, D_MEM_DOUT, D_MEM_ADDR, D_MEM_WEN,* and *D_MEM_BE* signals are generated to access the (instruction/data) memory and receives data from the memory when necessary.

a. To operate the memory correctly, *I_MEM_CSN* and *D_MEM_CSN* must be assigned 0 when *RSTn* is 1; otherwise, they are assigned 1.

b. To access a specific address in the instruction memory, the core module should specify the address using *I_MEM_ADDR*.

c. To access a specific address in the data memory, the core module should specify the address using *D_MEM_ADDR*.

d. *I_MEM_DI* and *D_MEM_DI* are the data values the core receives from the instruction and data memory, respectively (i.e., instruction/data -> core); these values contain the data inside the memory specified by the *I_MEM_ADDR* and *D_MEM_ADDR*.

e. *D_MEM_BE* is a byte-enable signal, which is used to properly designate the data access granularity when reading or writing data from/to data memory.

   a. When executing the *SW, LW* instructions in RISC-V, *D_MEM_BE* must be properly set to match the correct semantics of the instruction. For example, *SW: b1111, LW: b1111*.

f. In testbench files, the memory modules and the core module are already instantiated and properly connected/interfaced using wires, so you can read/write from the memory if you specify the correct values to the input/output ports of the core and the two memories.

g. *RF_WE* stands for Register File Write Enable. To enable a write operation to the register file, *RF_WE* must be set to 1.

h. *RF_RA1, RF_RA2*, and *RF_WA* specify the address of the source register #1, source register #2, and the destination register.

i. *RF_WD* specifies the data that is going to be written into the register file.

j. *RF_RD1* and *RF_RD2* designates the data that is going to be read out from the two source registers, *RF_RA1* and *RF_RA2*.

k. In testbench files, the register file and the core module are connected via wires, the core module can read/write from/to the register file if the core module specifies the register number and operation type.

l. *HALT* is used to halt the program when a specific condition is met. The terminate condition is (*RF_RD1 == 0x0000000c*) when the received instruction is *0x00008067*. You must set *HALT* wire to 1 when the terminal condition is met.

m. *NUM_INST* is the number of instructions executed in the core module. *NUM_INST* must correctly contain the number of instructions executed, since it is used in testbench files. The grading mechanism is explained later.

Instruction Memory

The instruction memory is initialized by loading a hex file. You must specify your location of where the hex file is located in *ROMDATA*. The path must not contain any Korean alphabets (한글).

AWIDTH and *SIZE* filed of the instruction memory are initialized to 10 and 1024, respectively. As a result,

2^{10} index-able entries are created into the RAM; each entry is 4-bytes wide and accordingly contains a single 32-bit wide instruction.

The instruction memory is synchronous to the *CLK* and receives the *I_MEM_CSN* input.

Since the instruction memory is 1) read-only and need not have to support a write operation and 2) the is designed to support the byte-granularity addressing mode, *BE, WEN,* and *DI* are fixed to 0, 1, and z, respectively.

DOUT is where the instruction data is read out of the instruction memory, the address of which is specified using *I_MEM_ADDR*. Note that the upper 10 bits of *I_MEM_ADDR (I_MEM_ADDR*[11:2]) are used to access the RAM entries.

Data Memory

AWIDTH, SIZE field of the data memory are initialized to 12 and 4096, respectively. As a result, the data memory contains 2^{12} index-able entries, each of which contains 4-bytes of data.

The data memory is also synchronous to the *CLK* signal, and receives the *D_MEM_CSN* input.

The data memory should be able to support byte-granularity read/write operations depending on the instruction type. The core module can coordinate the granularity by *D_MEM_BE*. For the write operation, *D_MEM_WEN* is set to 1 and the values fed into *D_MEM_DI* are written to the RAM.

DOUT is where the data comes out of the data memory whose address is specified in *D_MEM_ADDR*. Note that the address fed into *D_MEM_ADDR* isn't sliced unlike in the instruction memory.

D_MEM_BE enables to access data memory in byte granularity, and *D_MEM_WEN* is received from the core so that it can be used to enable write operation and the value inside the *D_MEM_DI* can be written to ram. *DOUT* is the data output from memory, and *ADDR* is connected to *D_MEM_ADDR*. At this time, *ADDR* is generated with *D_MEM_ADDR*. Again, you should be aware that the address is not sliced unlike instruction address.

Data memory and instruction memory are already implemented to be able to export or write the value synchronous to the clock signal. Data can be exported or written after one cycle when interfaces (*WEN, BE* …) are set correctly.

Register File

DWIDTH, MDEPTH, and *AWIDTH* field of the register file are initialized to 32, 32, and 5, respectively, which specifies the width of the register data, the number of general-purpose registers, and the address width. In our case, the register file has 32 general-purpose registers, each of which can store 32-bits of data.

The register file is initialized by the *RSTn* signal, and synchronous to the *CLK* signal. The values fed into *RF_RA1* and *RF_RA2* are the index of the source register #1 and #2, whose data will be read out through *RF_RD1* and *RF_RD2*. The write signal (*WE*) must be fed into *Write* in order to conduct a write operation; the data fed into *WD* are written to the register file.

Each register is initialized in the *REG_FILE module*, once it receives the *RSTn* signal. You don't have to worry about which values to initialize the register file as it's already implemented inside the *REF_FILE.v* file.

The Grading Mechanism

Not all testbench files checks the correctness of the output value for every instruction execution. Testbench files checks *NUM_INST*, and if the number of executed instructions matches to the predefined number of instructions, then compares *OUTPUT_PORT* with the test case value, grading the score.
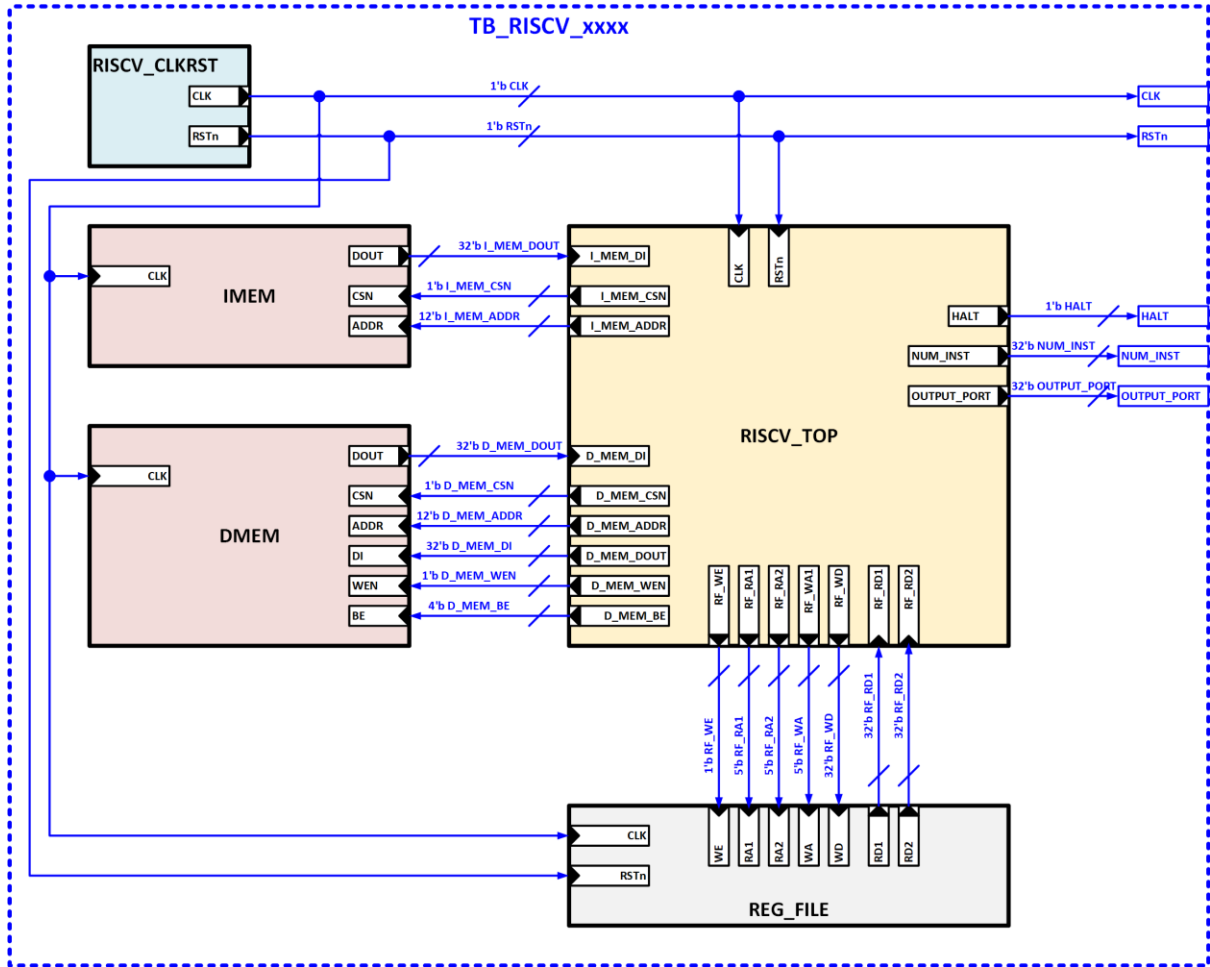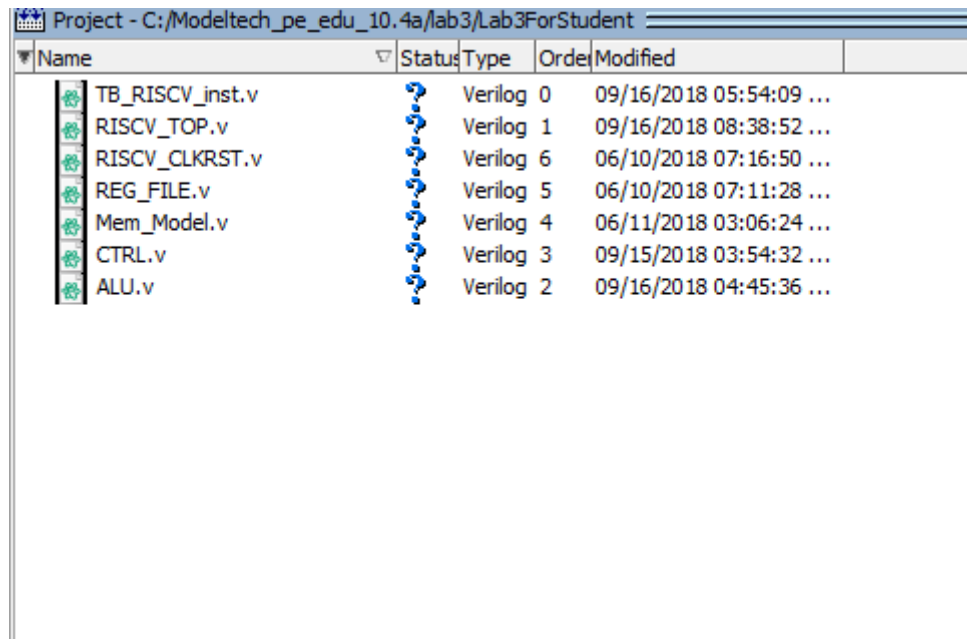
Figure 6. *TB_RISCV* **Module**

How to Start

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

   CTRL.v and *ALU.v* files are one of TA's implementation.

   CTRL.v is for the control signal and *RISCV_TOP.v* & *ALU.v* are for the data path.
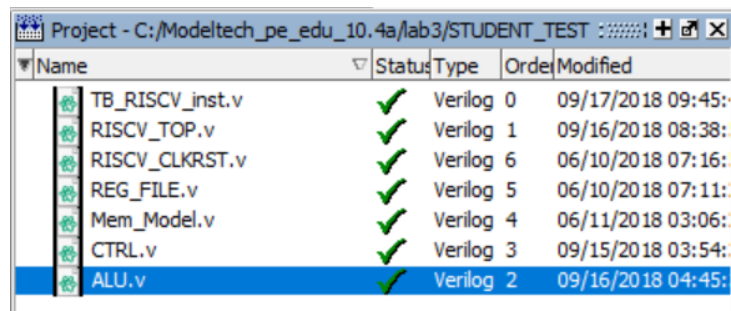

2. Change the file location in testbench files.

```
        .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.

4. Choose all the files which are relevant to the multi cycle CPU when you start simulation.
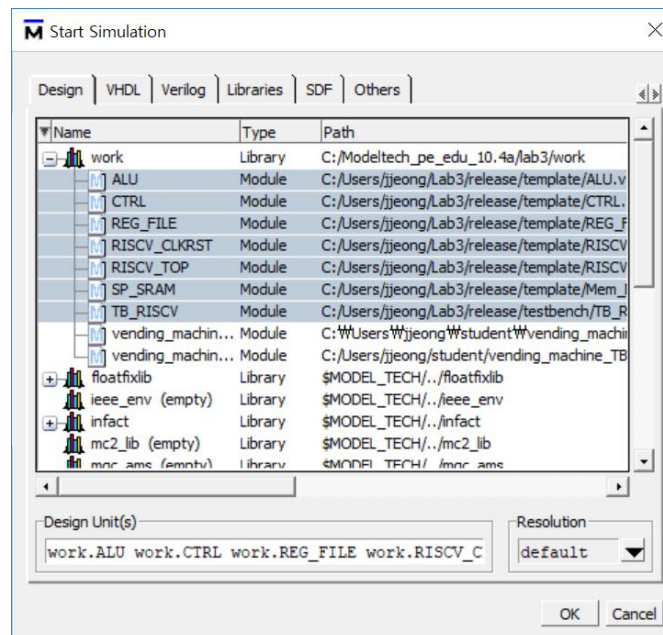
5. Run the simulation.

6. Get the simulation result.