Lab 5: Pipeline CPU Lab

1. Overview

Lab 5 is intended to give you hands-on experience in designing a modern microprocessor, whose operations are conducted in pipelined fashion. Based on the concepts and skills you have acquired through the lab assignments, you are now ready to implement a pipeline 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 pipeline CPU, each instruction utilizes difference part of CPU. If the pipeline stages are not stalled, the pipeline CPU completes one instruction per every clock cycle, thereby enhancing the throughput. Note that latency experienced by the instruction remains unchanged.
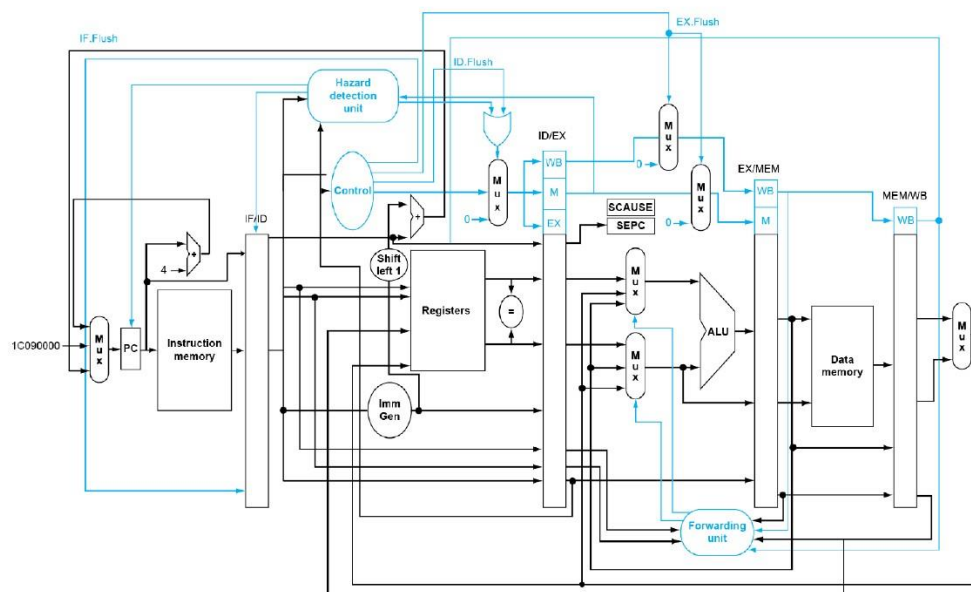

Figure 1. Pipeline CPU

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

Fundamentally, the reason why the hazards arise is an instruction can't see the change in the architectural states that is 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.
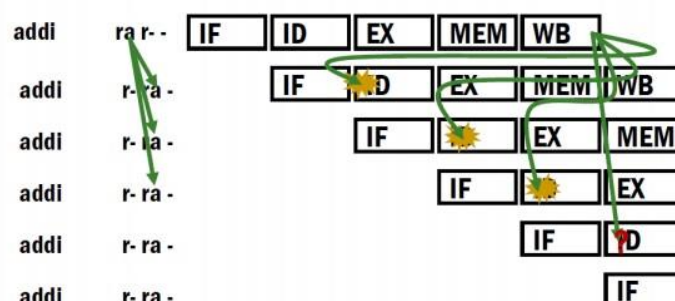
Figure 2. Data Hazard

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
IF	i	j	k	k	k	k	l				
ID	h	i	j	j	j	j	k	l			
EX		h	i	bub	bub	bub	j	k	l		
MEM			h	i	bub	bub	bub	j	k	l	
WB				h	i	bub	bub	bub	j	k	l

i: rx ← _
j: _ ← rx

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, which means the predicted next PC value turns out to be wrong, then 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.
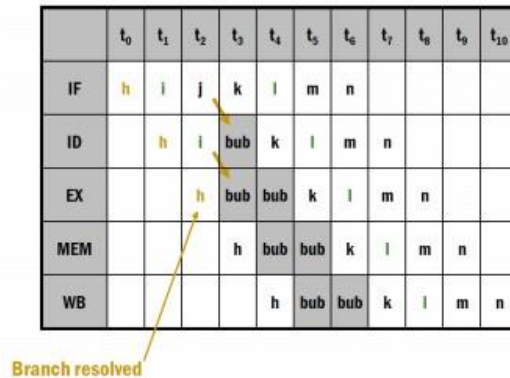
3

Figure 5. Control Hazard

Not surprisingly, 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 pipeline 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/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. Pipeline CPU Lab

In *Lab 5*, you are required to implement a pipeline 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 (RV 32I)

2. You're required to implement 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

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 (e.g. if your pipelined CPU implementation is not 5 stage), **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.

*** 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.

6. Some Statistic

1. Students re-wrote their code 1.5 times to re-design their pipeline CPU

2. It takes 10 ~ 60 hours to complete this assignment

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 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 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 compress the 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.

Code Review of the 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 clock and reset signals,
   b. *riscv_top1* creates the processor core datapath,
      a. *Does NOT include major sequential logics (e.g., instruction/data memory, RF)*
   c. *i_mem1* creates the instruction memory,
   d. *d_*mem1creates 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.
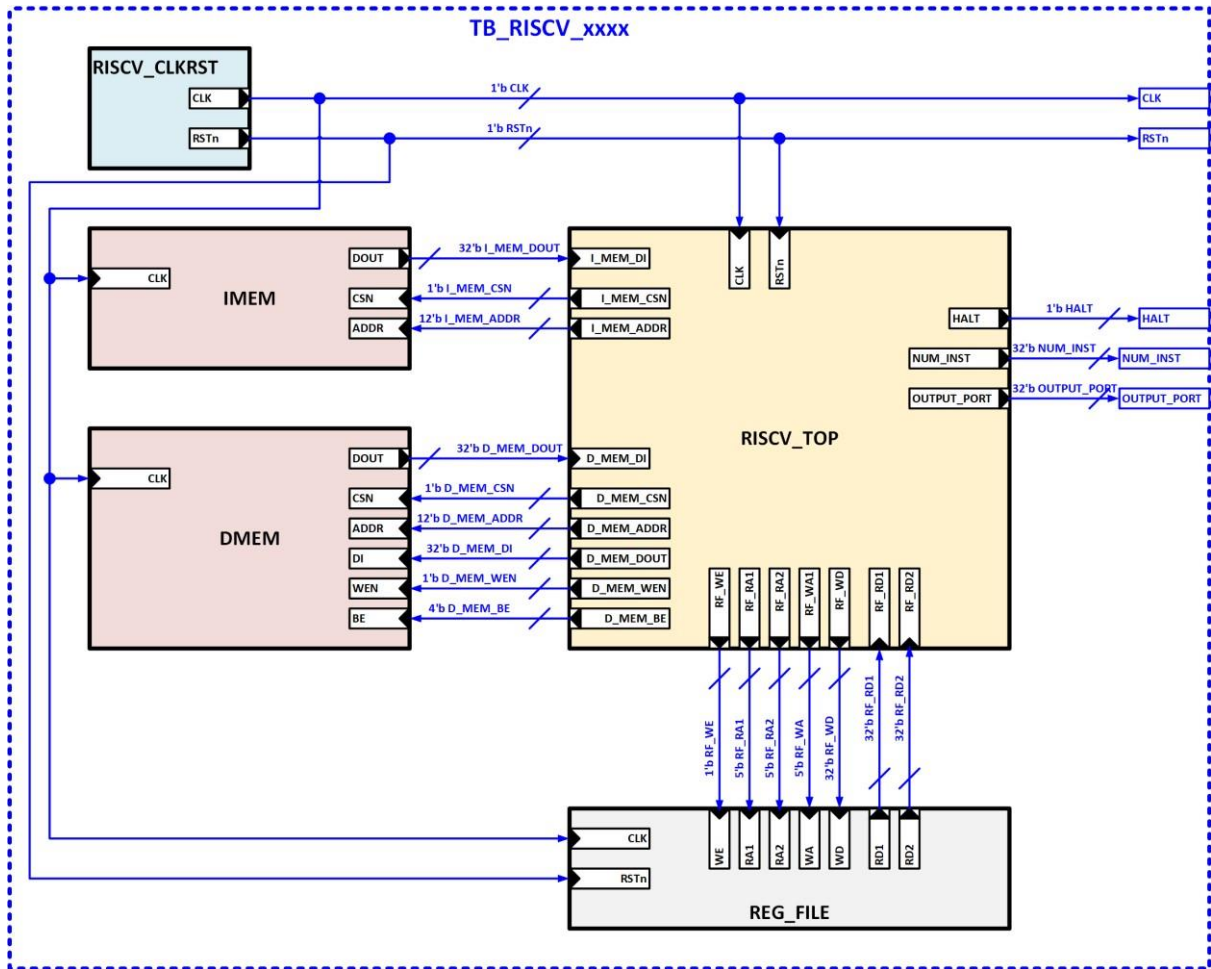
Figure 5. TB_RISCV Module

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.

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

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 (
```
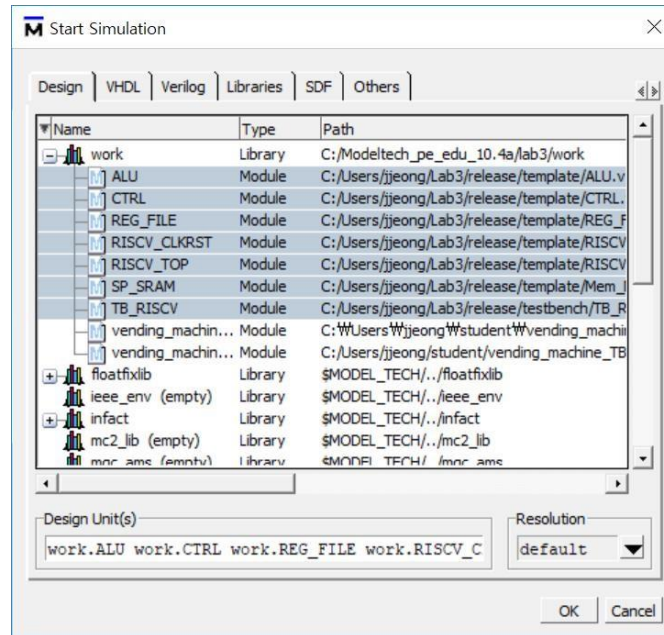
3. Compile all the files.

4. Choose all files relevant to your implementation when you start simulation.

5. Run the simulation.

6. You get the simulation result.