A Detail Description of *TB_RISCV_** Files

/ *EE312 Lab 3 Auxiliary Document* */*

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 *SB, SH, SW, LB, LH, LW, LBU, LHU* instructions in RISC-V, *D_MEM_BE* must be properly set to match the correct semantics of the instruction. For example, *SB: b0001, SH: b0011, SW: b1111, LB: b0001, LH: b0011, 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, so 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 check the correctness of the output value for every instruction execution. Testbench files check *NUM_INST*, and if the number of executed instructions

matches the predefined number of instructions, then it compares *OUTPUT_PORT* with the test case value, grading the score.

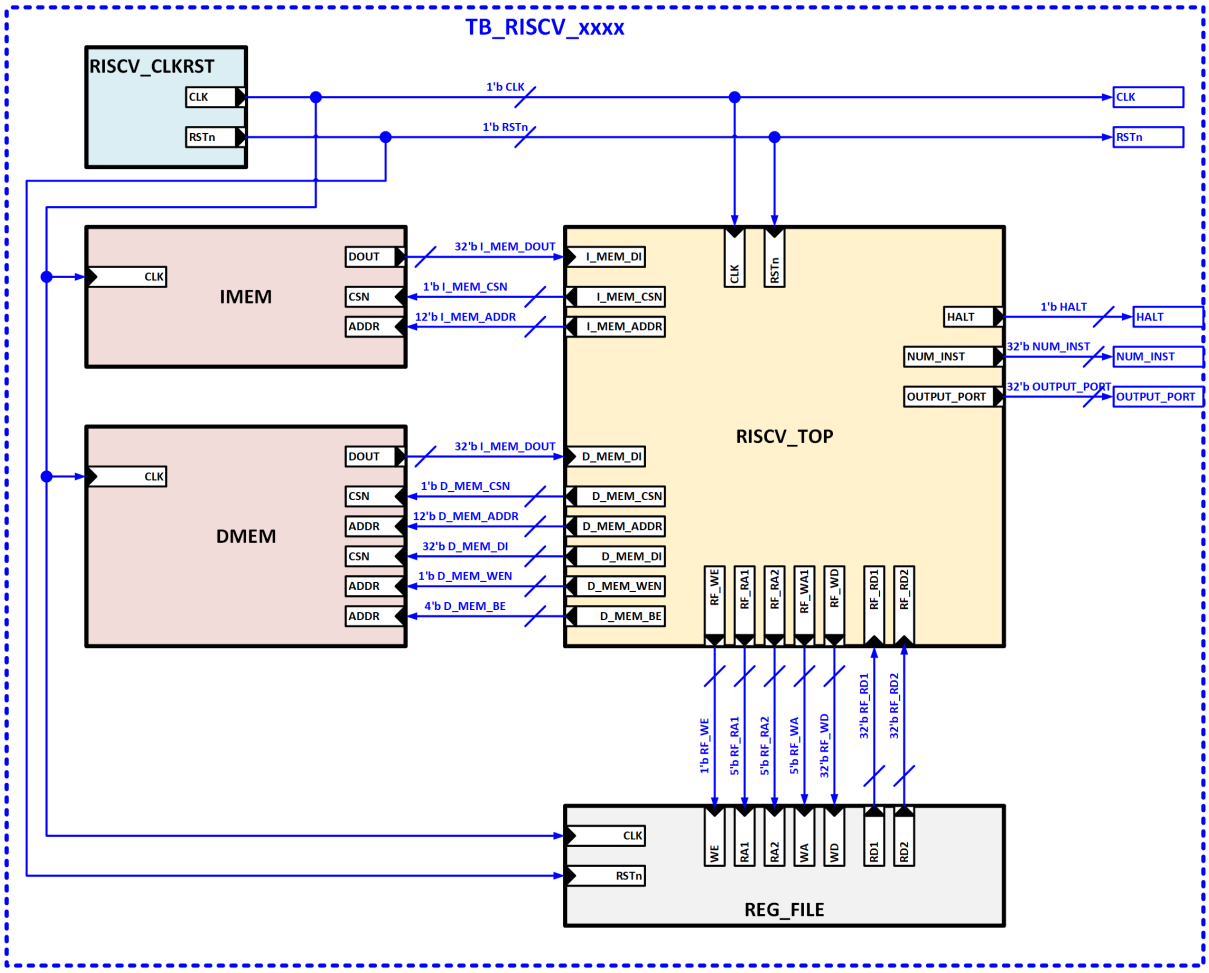

Figure 1 TB_RISCV Module