COSE222: Computer Architecture
Design Lab #3

Due: Nov 12, 2024 (Tuesday) 11:59pm on Blackboard

Total score: 20

In Chapter 4 of this course, we are exploring how to design the simple RISC-V processor. The baseline integer ISA of RISC-V is RV32I, which is a 32-bit instruction set for 32-bit computers. The textbook also covers the part of integer instructions (just 7 instructions: `lw, sw, add, sub, or, and, beq`) for the processor design. In this lab, we will design a simple RISC-V processor that support the following integer instructions: `lw, sw, add, addi, sub, xor, xori, or, ori, and, andi, beq, bne, blt, bge`. Note that our processor will support 3 more branch instructions (`bne, blt, bge`), xor instruction (`xor`), and I-type arithmetic/logical instructions (`addi, xori, ori, andi`) compared to the processor in the textbook. The design labs provided for COSE222 will follow the processor design steps described in the textbook. For this design lab assignment, you will be requested to design datapath elements (instruction memory, register file, ALU, and data memory) of the RISC-V processor.

During the class we explained about the typical RTL design process. For a single datapath element design, you will be provided the information regarding the interfaces (inputs and outputs) and functional operations of the corresponding hardware unit. You should build the RTL design based on such information and then verify your design with testbenches. If the verification is done successfully, designing the datapath element is complete.

For this design lab, you should design the following RTL codes using *SystemVerilog*.

1. Instruction memory (imem.sv)

Instruction memory stores 32-bit RISC-V instruction. The instruction memory is accessed by the address pointing instructions (i.e. program counters). Once the program counter is given, the instruction memory provides the corresponding instructions.

In this design you are requested to design the instruction memory that includes **1024 entries** of 32-bit width. In the textbook the width of program counter is 32-bit since we are exploring 32-bit computer systems. However, the designed instruction memory will receive 10-bit address since the memory have 1024 entries.

The required input/output ports are already provided in the incomplete code of "**imem.sv**". Please complete the instruction memory design.

2. Register file (regfile.sv)

Register file means a group of registers. As a single RISC-V instruction include two source registers and one destination register at most, the register file needs to provide interfaces for these requirements. The register file receives two register numbers for the source registers and one register number for the destination register. Based on the input register numbers, the data can be read from and written to the register file.

In this design lab you are requested to design the RISC-V register file. In the textbook, the register file does not receive a clock signal. However if so, the register file can generate an infinite loop. Think about

the case of "`add x1, x1, x1`" – the `x1` is repeatedly updated by new doubled value by `x1`. In order to avoid this undesired situation, we will make the write to the register file is triggered by the rising edge of the clock signal.

The required input/output ports and the detailed descriptions are also provided in the incomplete "**regfile.sv**". Please complete the design.

3. ALU (alu.sv)

ALU receives two operand data and generates the result data by performing the specified operations using the input operands. We will design ALU that supports the following 15 instructions: `lw, sw, add, addi, sub, xor, xori, or, ori, and, andi, beq, bne, blt, bge`. We will design the ALU as described in shown in the below table. The behavior of ALU is similar to the descriptions in the textbook, however the interface of ALU is modified slightly to support the additional three branch instructions. Anyway, our ALU will support *add*, *subtract*, *bitwise xor, bitwise or*, and *bitwise and* operations. The operations of ALU are selected by the ALU control signal. The interface of the ALU is already described in the incomplete ALU design file "**alu.sv**". Please complete the ALU design.

Instruction	ALUOp	Operation	Opcode field	ALU function	ALU control
Load (lw)	00	Load	0000011	ADD	0010
Store (sw)	00	Store	0100011	ADD	0010
Branch	01	Branch	1100011	SUB	0110
R-type, I-type arithmetic/logic	10	Add	0110011 (R-type) 0010011 (I-type)	ADD	0010
		Subtract		SUB	0110
		Bitwise AND		AND	0000
		Bitwise OR		OR	0001
		Bitwise XOR		XOR	0011

4. Data memory (dmem.sv)

Two memory reference instructions (`lw` and `sw`) accesses the data memory. The data memory design is similar to the instruction memory design, but the data memory receives two control signals: `MemRead` and `MemWrite`. You are requested to design the data memory which has **1024 entries** of 32-bit width.

In the textbook, the data memory does not receive a clock signal. However, without the clock signals we need to implement the data memory using latches, which must be avoided in usual RTL designs. Latches can be exploited for special-purpose blocks. However, **if latches are generated after synthesis, it usually means your design includes critical bugs**. Hence, in this design lab you are requested to design the data memory working with the clock signal.

Another design issue of the data memory is the misaligned memory accesses supported by RISC-V ISA. Even though RISC-V supports misaligned memory accesses, the data memory for this design lab will support only aligned accesses for word data in order to simplify the hardware structure. The detailed information is provided in "**dmem.sv**". Please complete the data memory design.

5. What to do

(a) Complete the design of the datapath elements (i.e. imem.sv, regfile.sv, alu.sv, and dmem.sv).

(b) Verify your design with the provided testbenches. The testbench file for each design module is provided, thus you can use it. For instance, you can use tb_imem.cpp as the testbench for imem.sv. The testbench will generate a report file (*.rpt file) under your project folder.

You should use a provided cpp file for the testbench of each design. See the below example for verifying imem with Verilator.

```
$ verilator -Wall --trace --cc imem.sv --exe tb_imem.cpp
$ make -C obj_dir -f Vimem.mk Vimem
$ ./obj_dir/Vimem
```

(c) Compress your design files (imem.sv, regfile.sv, alu.sv, and dmem.sv) and the generated report files for designs (imem.rpt, regfile.rpt, alu.rpt, and dmem.rpt) in **one zip file**. You must name your zip file as "`FirstName_LastName.zip`". (e.g. `Gildong_Hong.zip` for Gildong Hong) If your submission file does not meet this rule, we will reduce 1 point from your score. 😟