Computer Architecture 2021 Fall Homework 4

TA: 陳炫均

1. Problem Description

In this homework, we are going to implement a simple RISC-V CPU with Verilog. The CPU in this homework is single-cycle, all operations are assumed to be completed equally in one cycle.

This CPU has 32 registers, and 1KB instruction memory. It should take 32-bit binary codes as input, and should do the corresponding RISC-V instructions, saving the result of arithmetic operations into the corresponding registers. Only arithmetic operations are covered in this homework, there will be no load/store instructions in this homework. We will examine the correctness of your implementation by dumping the value of each registers after each cycle.

1.1. Data path

Figure 1 describes the data path you should follow in this homework. The registers, PC, and instruction memory are provided by TAs, so you do not have to (and are also not allowed) modify them. The modules you have to implement is introduced in detail in section 1.6. And after you finish these modules, you have to connect these modules in your CPU.v.
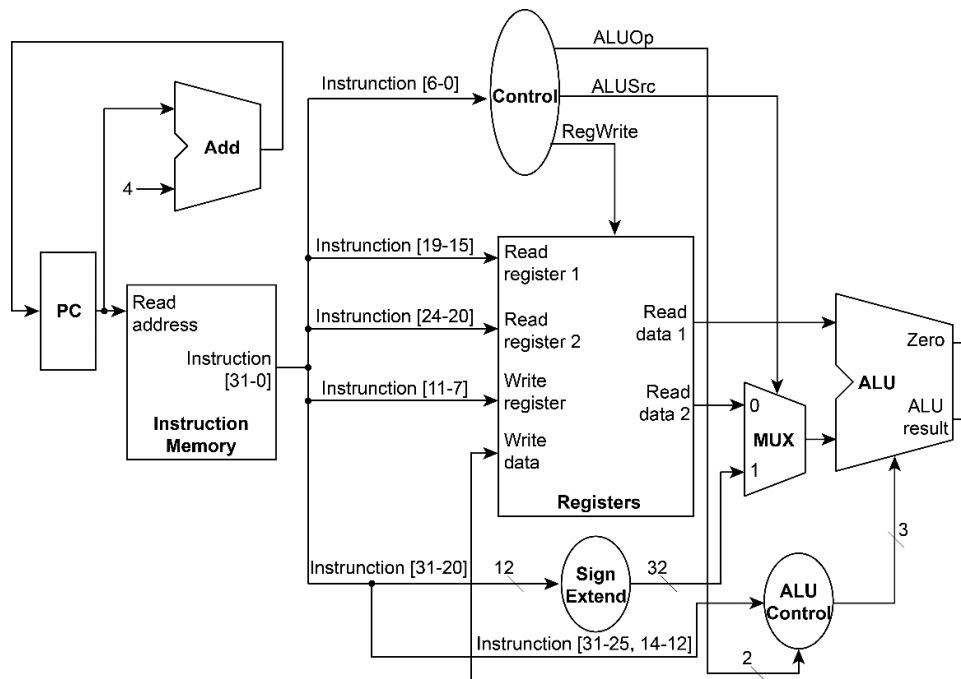

Figure 1 Data path of the CPU in this homework

1.2. Instructions

Your CPU should be able to support following instructions:

- `and rd, rs1, rs2` (bitwise and)
- `xor rd, rs1, rs2` (bitwise exclusive or)
- `sll rd, rs1, rs2` (shift left logically, rs2 will always be positive in this hw)
- `add rd, rs1, rs2` (addition)
- `sub rd, rs1, rs2` (subtraction)
- `mul rd, rs1, rs2` (multiplication)
- `addi rd, rs1, imm` (addition)

- `srai rd, rs1, imm` (shift right arithmetically, imm will always be positive in homework)

That is, to get full credits, your CPU have to function correctly upon encountering these instructions in any order. Instructions other than specified above will not be involved at evaluation.

1.3. Instruction Format

Instructions involved in this homework are 32-bit following RISC-V standard. Table below lists the specification of instructions your CPU should support in this homework.

funct7	rs2	rs1	funct3	rd	opcode	function
0000000	rs2	rs1	111	rd	0110011	and
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	001	rd	0110011	sll
0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
0000001	rs2	rs1	000	rd	0110011	mul
imm[11:0]		rs1	000	rd	0010011	addi
0100000	imm[4:0]	rs1	101	rd	0010011	srai

1.4. Input / Output Format

Besides the modules listed above, you are also provided "testbench.v" and "instruction.txt". After you finish your modules and CPU, you should compile all of them including "testbench.v". A recommended compilation command would be

$ iverilog -o CPU.out *.v

Then by default, your CPU loads "instruction.txt", which should be placed at the same directory as hw4.out, into the instruction memory. This part is written in "testbench.v". You don't have to change it. "instruction.txt" is a plain text file which consists of 32 bits (ASCII 0 or 1) per line, representing one instruction per line. For example, the first 3 lines in "instruction.txt" are

```
0000000_00000_00000_000_01000_0110011 //add  $t0,$0,$0
000000001010_00000_000_01001_0010011  //addi $t1,$0,10
000000001101_00000_000_01010_0010011  //addi $t2,$0,13
```

Note that underlines and texts after "//" (i.e. comments) are neglected. They are inserted simply for human readability. Therefore, the CPU should take "00000000000000000000010000110011" and execute it in the first cycle, then "00000000101000000000010010010011" in the second cycle, and "00000000110100000000010100010011" in the third, and so on.

Also, if you include unchanged "testbench.v" into compilation, the program will generate a plain text file named "output.txt", which dumps value of all registers at each cycle after execution. The file is self-explainable.

1.5. Machine code

Both R-type and I-type instructions are included in this homework. As we all know that they have different formats in RISC-V.

Name (Field size)	Field						Comments
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic

1.6. Modules

Following are modules you are required to implement and their function.

1.6.1. Control

Reads opcodes in the instruction and output control signals to control the behavior of ALU and registers.

1.6.2. ALU control

Reads funct7 and funct3 in the instruction and tell ALU what kind of operation it should do.

1.6.3. Sign extend

Signed extends the immediate value into 32-bit number. You have to make sure both positive and negative immediate values can work correctly.

1.6.4. ALU

According to the control signal, do the corresponding arithmetic operation on two input fields.

1.6.5. PC, Instruction memory, Registers, testbench

These modules are provided by TA. You are not allowed to modify them. (Except for adding debug information in testbench)

1.6.6. Others

You can add more modules than listed above if you want.

1.6.7. CPU

Use structure modeling to connect input and output of modules following the data path in Figure 1.

1.7. Reminder

Lab 1 and 2 will be strongly related to this homework. Please make sure you can fully understand how to write this homework; otherwise you may encounter difficulties in your future labs. Also, because this homework is rather simple, it is recommended for you to get familiar to waveform visualization tool (e.g. gtkwave), which is much more useful than "display method" (or printf 大法 in C, cout 大法 in C++, which only prints messages on terminal rather than using debugger to trace, stepwise execution, etc.). You may not have enough time to learn this tool at the labs, or hard to learn it from scratch because of the difficulty of the labs.

2. Report

2.1. Modules Explanation

You should briefly explain how the modules you implement work in the report. You have to explain them in human-readable sentences. Either English or Chinese is welcome, but no Verilog. Explaining Verilog modules in Verilog is nonsense. Simply pasting your

codes into report with no or little explanation will get zero points for the report. You have to write more detail than Section 1.6.

Take "PC.v" as example, an acceptable report would be:

> PC module reads clock signals, reset bit, start bit, and next cycle PC as input, and outputs the PC of current cycle. This module changes its internal register "pc_o" at positive edge of clock signal. When reset signal is set, PC is reset to 0. And PC will only be updated by next PC when start bit is on.

And following report will get zero points.

> The inputs of PC are clk_i, rst_i, start_i, pc_i, and ouput pc_o.
> It works as follows:
>
> ```
> always@(posedge clk_i or negedge rst_i) begin
>     if(~rst_i) begin
>         pc_o <= 32'b0;
>     end
>     else begin
>         if(start_i)
>             pc_o <= pc_i;
>         else
>             pc_o <= pc_o;
>     end
> end
> ```

2.2. Development Environment

Please specify the OS (e.g. MacOS, Windows, Ubuntu 20.04) and compiler (e.g. iverilog) or IDE (e.g. ModelSim) you use in the report, in case that we cannot reproduce the same result as the one in your computer.

3. Submission Rules

Put all your Verilog codes into a directory named "codes", then put "codes" and your report (should be named in format "studentID_hw4_report.pdf", e.g. "b08902000_hw4_report.pdf) into a directory named "studentID_hw4". **Note that you have to REMOVE testbench.v, Instruction_Memory.v, Registers.v, PC.v, instruction.txt, output.txt, which are provided by TA, in your submission.** Finally, zip this directory, and upload this zip file onto NTU COOL before **11/23/2021 (Tue.) 23:59.**

In short, we should see a single directory like following structure after we type

```
$ unzip b08902000_hw4.zip
```

in Linux terminal:
- b08902000_hw4/
  - b08902000_hw4/codes
    - b08902000_hw4/codes/CPU.v
    - b08902000_hw4/codes/ALU.v
    - …
  - b08902000_hw4/b08902000_hw4_report.pdf

Make sure you remove following files before submission: Instruction_Memory.v, PC.v, Registers.v, testbench.v, instruction.txt, output.txt.

4. Evaluation Criteria

We will compile your program in following command:

```
b08902000_hw4/codes/ $ iverilog -o CPU.out *.v ../../*.v
```

, where `../../*.v` includes `Instruction_Memory.v`, `PC.v`, `Registers.v`, `testbench.v`, and the working directory is in your "codes/" directory. That is, some modules are provided outside the directory you submit.

    4.1. Programming Part (80%)

      We will test your program by following set of instructions file.

        o instructions.txt contains only single type of instructions (registers will be initialized to non-zero values) (6% each, 48% in total)

        o all R-type instructions in any order (10%)

        o all I-type instructions in any order (10%)

        o all instructions in any order (12%)

    4.2. Report (20%)

    4.3. Other

Minor mistakes (examples below) causing compilation error: -10 pts

        o wrong usage of "`include"

        o submitting unnecessary files (those provided by TA except CPU.v)

        o other mistakes that can be fixed within 5 lines

Major mistakes causing compilation error: 0 pts on programming part

Wrong directory format: -10 pts

Late submission: -10 pts per day

email to eclab.ca.ta@gmail.com if you have any questions.