CSE 140 Project – Due: 5/11 11:59PM

Work in a group of two members for the following tasks.

1. Single-cycle RISC-V CPU (80 pts)

Extend the decoder function implemented for HW3 to design a single-cycle RISC-V CPU in any language you like. Your single-cycle CPU program needs to be able to execute the following 10 instructions. We will test a RISC-V program that uses the following instructions on your single-cycle CPU code.

lw, sw, add, addi, sub, and, andi, or, ori, beq

Code Structure:

- **Program to run**

We will provide a program that contains machine code of multiple instructions in a text file. The main function of your program will open and read the text file and call Fetch() function to fetch one instruction per cycle.

- **Fetch()**

Review pages 13 to 15 of lecture slide "CSE140_Lecture-4_Processor-1".

Create a function named Fetch() that reads one instruction from the input program text file per cycle.

PC: As learned in the lecture, the CPU needs to fetch an instruction that is pointed by the PC value and the PC value needs to be incremented by 4. Declare a global variable named "**pc**" and initialize it as 0. Whenever you read one instruction from the input program text file, increment the pc value by 4 and read the *i*th instruction that is pointed by PC value, 4**i*. For example, if pc is 4, you will read 1st instruction, which is the second instruction when we count from 0.

Next PC: After fetching an instruction, fetch step will increment the PC value by 4 as can be seen in the lecture slide. Declare one variable named "**next_pc**" and store "pc" + 4. Later, we will need to choose the next cycle pc value among this next_pc value and the branch target address. We will discuss later but the branch target address will be updated by another function that produces "branch_target". So, add a logic in this Fetch() function that copies one of the three possible pc values (next_pc and branch_target) to pc variable. This updated pc value will be used for the next instruction fetch.

- **Decode()**

Review pages 16 to 21 of lecture slide "CSE140_Lecture-4_Processor-1".

Reuse your HW3 program to decode an instruction. The instruction that is fetched by Fetch() function will be sent to this function and decoded. Unlike HW3 program, this Decode() function will actually read values from a register file.

Register file: Create an integer array that has 32 entries and name it as "**rf**". This register file array will be initialized to have all zeros unless otherwise specified. Your Decode() function should be able to read values from this register file array by using the register file ID (e.g., zero will access rf[0]).

*Sign-*extension: The Decode() function should be able to do sign-extension for the offset field of I-type instructions as can be seen on page 20 of the lecture slide.

- **Execute()**

Review pages 22 to 28 of lecture slide "CSE140_Lecture-4_Processor-1" and pages 13 to 19 of lecture slide "CSE140_Lecture-4_Processor-2".

Create a function named Execute() that executes computations with ALU. The register values and sign-extended offset values retrieved/computed by Decode() function will be used for computation.

ALU OP: The Execute() function should receive 4-bit "**alu_ctrl**" input and run the operation indicated by the alu_ctrl as specified on **page 14 of "CSE140_Lecture-4_Processor-2"**. For each function, simply run the corresponding operation (you don't need to implement ALU on page 11). For example, if the input alu_ctrl is 0000, you will run "&" for the two operands which is "AND" operator of C/C++. Similarly, if alu_ctrl is 0010, you will run "+" for the two operands which is "ADD" operator of C/C++.

Zero output: As can be seen on **page 23 and 28 of "CSE140_Lecture-4_Processor-1"**, 1-bit zero output should be generated by Execute(). Declare a global variable named "**alu-zero**" that is initialized as 0. This variable will be updated by Execute() function and used when deciding to use branch target address in the next cycle. The zero variable will be set to 1 by ALU when the computation result is zero and unset to 0 if otherwise.

Branch target address: As can be seen on page 23 of "CSE140_Lecture-4_Processor-1", Execute() function should also calculate the branch target address. Declare a global variable named "**branch_target**" that is initialized as 0. This variable will be updated by Execute() function and used by Fetch() function to decide the PC value of the next cycle. The first step for updating the branch_target variable is to shift-left-1 of the sign-extended offset input. The second step is to add the shift-left-1 output with the PC+4 value.

- **Mem()**

Review pages 26 to 27 of lecture slide "CSE140_Lecture-4_Processor-1".

Data memory: Create an integer array named "**d-mem**" that has 32 entries. Initialize the entries with zeros unless otherwise specified. Each entry of this array will be considered as one 4-byte memory space. We assume that the data memory address begins from 0x0. Therefore, each entry of the d-mem array will be accessed with the following addresses.

Memory address calculated at Execute()	Entry to access in d-mem array
0x00000000	d-mem[0]
0x00000004	d-mem[1]
0x00000008	d-mem[2]
…	…
0x0000007C	d-mem[31]

Mem() function should receive memory address for both LW and SW instructions and data to write to the memory for SW. For the LW, the value stored in the indicated d-mem array entry should be retrieved and sent to the Writeback() function that is explained below.

- **Writeback()**

Review pages 25 to 26 of lecture slide "CSE140_Lecture-4_Processor-1".

This function will get the computation results from ALU and a data from data memory and update the destination register in the rf array. Also, this is the last step of an instruction execution. Therefore, the cycle count should be incremented. Declare a global variable "**total_clock_cycles**" and initialize it with zero. And then increment it by 1 whenever one instruction is finished.

- **ControlUnit()**

Review lecture slide “CSE140_Lecture-4_Processor-2”

In the Decode() function, ControlUnit() will be called to generate control signals. This function will receive 7-bit opcode value and generate 7 control signals as can be seen on page 17-19 of lecture slide “CSE140_Lecture-4_Processor-2”. Declare one global variable per control signal (e.g., **RegWrite, Branch, and so on**) and initialize them with zeros. ControlUnit() will update these variables. Then, the following functions, Execute(), Mem(), and Writeback(), will use the control signals to use proper inputs as we learned in the lecture.

ALU Control: ALU Control receives ALUOp from Control Unit and function field values from the instruction to generate the proper alu_ctrl code that is used by Execute() function. You can create a separate function for ALU control that is called by ControlUnit() function or, merge the function of ALU control with the ControlUnit() function so that the ControlUnit() generates ALUOp signal for the Execute() function.

Code Execution:

Your program should show the **modified contents** of arrays **rf** and **d-mem** and the values of **total_clock_cycles** and **pc** whenever an instruction finishes, for the given program. Use the provided sample test program code and test your program execution.

Initialize rf array like below (all the other registers are initialized to 0):

x1 = 0x20, x2 = 0x5, x10 = 0x70, x11 = 0x4

Initialize d_mem array like below (all the other memory addresses are initialized to 0):

0x70 = 0x5, 0x74 = 0x10

Sample results (user input is marked in bold and italic font):

```
Enter the program file name to run:

sample_part1.txt

total_clock_cycles 1 :
x3 is modified to 0x10
pc is modified to 0x4

total_clock_cycles 2 :
x5 is modified to 0x1b
pc is modified to 0x8

total_clock_cycles 3 :
pc is modified to 0xc

total_clock_cycles 4 :
x5 is modified to 0x2b
pc is modified to 0x10

total_clock_cycles 5 :
x5 is modified to 0x2f
pc is modified to 0x14

total_clock_cycles 6 :
memory 0x70 is modified to 0x2f
pc is modified to 0x18
```

2. Extended RISC-V CPU (20pts)

Extend the single-cycle CPU to also support the following two instructions.

JAL and JALR

We did not design JAL and JALR instruction data path in the class. You should propose your own idea to extend the data path to support these two instructions. There are no restrictions for this implementation. Hints are like below:

1. Check the instruction types of JAL and JALR instructions from the RISC-V Reference Data sheet so that the proper fields are checked for indicating the instructions.
2. Add new control signals for these new instructions.
3. Configure the remaining control signals to run the functions that each instruction needs to finish such as
    a. JAL : 1) jump to PC + imm address and 2) update destination register with the PC+4 value
    b. JALR : 1) jump to RS1 + imm address and 2) update destination register with the PC+4 value.

Code Execution:

Use the provided sample test program code and test your program execution. The expected results are like below:

Initialize registerfile array like below (all the other registers are initialized to 0):

s0 = 0x20, a0 = 0x5, a1 = 0x2, a2 = 0xa, a3 = 0xf

Initialize d_mem array to all zero's.

Sample results (user input is marked in bold and italic font):

```
Enter the program file name to run:

sample_part2.txt

total_clock_cycles 1 :
ra is modified to 0x4
pc is modified to 0x8

total_clock_cycles 2 :
a0 is modified to 0xc
pc is modified to 0xc

total_clock_cycles 3 :
t5 is modified to 0x3
```

```
total_clock_cycles 4 :
ra is modified to 0x14
pc is modified to 0x4

total_clock_cycles 5 :
ra is modified to 0x8
pc is modified to 0x14

total_clock_cycles 6 :
memory 0x20 is modified to 0x3
pc is modified to 0x18

program terminated:
```

Demo:

In all demos, both team members should present if you are working in a group.

1) In the week of **4/7**, you will show the first intermediate demo of your project. In this demo, you will **show the code** of **Fetch()**, **Decode()**, **Execute()**, **Mem()**, and **WriteBack()** of the first part (supporting 10 instructions). You should also **explain your plan to finish** the first part and to extend the first part code to also support JAL and JALR.
2) In the week of **4/28**, you will show the second intermediate demo of your project. In this demo, you will show the code of **ControlUnit()** and the **execution of the completed first part** (supporting 10 instructions). You should also **show the code** that you added for **JAL and JALR**.
3) In the week of **5/5**, you will show the **final demo** of your project. In this demo, you will show the single-cycle CPU that supports all 12 instructions.

3. Extra Credit: Pipelined RISC-V CPU (20pts)

Once you finish the part 1 and part 2 successfully, you can also apply for gaining extra credit. Extend your CPU to support 5-stage pipeline.

Review lecture slide “CSE140_Lecture-4_Processor-3” & “CSE140_Lecture-4_Processor-4”

We will check if your CPU supports the following. There are no other restrictions.

1. Support pipeline stage registers, namely if_id, id_ex, ex_mem, mem_wb. You may define them as struct or class that has member variables corresponding to individual stage’s output signals. At each stage function, the values in these registers should be read as inputs and the results should be buffered in the next stage register.
    a. For example, in decode function, you will read inputs from if_id and store the outputs to id_ex.
2. Control signals should be buffered in the extended pipeline stage registers. See “CSE140_Lecture-4_Processor-4” page 4. The control signals should be propagated along the pipeline stage registers.
3. You do not need to support forwarding unit or branch predictor. But, you should put proper number of stalls when needed. See “CSE140_Lecture-4_Processor-4” page 32, 48, and 50.
4. The clock cycle should be incremented when each stage is finished; not when one instruction finishes.
5. You can demo with sample_part1.txt code. Show the print output from the cycle when the first instruction finishes execution, that time should be already cycle 5.

<u>Submission Guideline</u>

- Write a report of your program in a pdf or a word format.
  - The report should have a cover page that specifies the project title and team member names
  - On the first page, if you worked in a group, write how you partitioned tasks (e.g., contributions and list up of the assigned work per member) between the two members. Also state if you agree with getting the same score in a group. If most of the work was done by one of the members, you should clearly state as such.
  - The report should explain the code structure for the first part (that supports 10 instructions) and the extended code to support the JAL and JALR instructions separately, if possible.
  - Each function and variable should have proper explanations so that the TAs and the Instructor can understand your code.
  - Show the execution result screen shots for the first and the second parts, separately.
- Submit the code(s) with the report to the CatCourse.
- Deadline: **5/11 11:59PM (the same deadline for all sessions)**