CSE 140 Project – Due: 5/6 11:59PM

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

1. Single-cycle MIPS CPU (80 pts)

Extend the decoder function implemented for HW3 to design a single-cycle MIPS CPU in C or C++. Your single-cycle CPU program needs to be able to execute the following 10 instructions just like the CPU that we designed at the lectures. We will test a MIPS program that uses the following instructions on your single-cycle CPU code.

LW, SW, ADD, SUB, AND, OR, SLT, NOR, BEQ, J

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 6 to 9 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 and jump target addresses. We will discuss later but the branch target and jump target addresses will be updated by the other functions to "branch_target" and "jump_target". So, add a logic in this Fetch() function that copies one of the three possible pc values (next_pc, branch_target, and jump_target) to pc variable. This updated pc value will be used for the next instruction fetch.

- **Decode()**

Review pages 10 to 15 of lecture slide "CSE140_Lecture-4_Processor-1" and pages 2 to 4 of lecture slide "CSE140_Lecture-4_Processor-3".

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 "**registerfile**". 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 registerfile[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 15 of the lecture slide.

Jump target address: Jump instruction target address can be calculated in the Decode() function. Declare a global variable named "**jump_target**" and initialize it with zero. To update the jump_target variable, take the last 26 bits from the instruction. Run shift-left-2 (you may want to use multiplication) on it. And then, merge it with the first 4 bits of the next_pc variable.

- **Execute()**

Review pages 16 to 22 of lecture slide "CSE140_Lecture-4_Processor-1" and pages 10 to 17 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_op**" input and run the operation indicated by the alu_op as specified on page 10 of "CSE140_Lecture-4_Processor-2". For each function, simply run C/C++ operations (you don't need to implement ALU on page 11). For example, if the input alu_op is 0000, you will run "&" for the two operands which is "AND" operator of C/C++. Likely, if alu_op is 0010, you will run "+" for the two operands which is "ADD" operator of C/C++. For set-on-less-than, you may want to run a combination of C/C++ lines to compare two values and generate the set/unset output.

Zero output: As can be seen on page 22 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 zero. This variable will be updated by Execute() function and used when deciding PC value of the next cycle. The zero variable will be set by ALU when the computation result is zero and unset if otherwise.

Branch target address: As can be seen on page 22 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 zero. 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-2 of the sign-extended offset input. The second step is to add the shift-left-2 output with the PC+4 value.

- **Mem()**

Review pages 20 to 21 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 19 to 21 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 registerfile 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" and pages 1 to 6 of lecture slide "CSE140_Lecture-4_Processor-3".

In the Decode() function, ControlUnit() will be called to generate control signals. This function will receive 6-bit opcode value and generate 9 control signals as can be seen on page 3 of lecture slide "CSE140_Lecture-4_Processor-3". Declare one global variable per control

signal (e.g., **RegWrite, RegDst, and so on**) and initialize them with zeros. ControlUnit() will update these variables and 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 inputs from Control Unit and generates the proper ALU OP 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 ALU OP input for the Execute() function.

Code Execution:

Your program should show the **modified contents** of arrays **registerfile** 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 registerfile array like below (all the other registers are initialized to 0):

$t1 = 0x20

$t2 = 0x5

$s0 = 0x70

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 :
$t3 is modified to 0x10
pc is modified to 0x4

total_clock_cycles 2 :
$t5 is modified to 0x1b
pc is modified to 0x8

total_clock_cycles 3 :
$s1 is modified to 0x0
pc is modified to 0xc

total_clock_cycles 4 :
pc is modified to 0x1c

total_clock_cycles 5 :
memory 0x70 is modified to 0x1b
pc is modified to 0x20

program terminated:
total execution time is 5 cycles
```

2. Extended MIPS CPU (20pts)

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

JAL and JR

Review pages 8 to 13 of lecture slides "CSE140_Lecture-3_ISA-4" and MIPS Reference Data sheet used for HW 1.

There is no restrictions for this implementation. Hints are like below:

1. Check the instruction types of JAL and JR instructions from the MIPS Reference Data sheet so that the proper fields are checked for indicating the instructions.
2. Add new control signals for these three instructions.
3. Configure the remaining control signals to run the functions that each instruction needs to finish such as
    a. JAL : 1) jump to the target instruction and 2) update $ra register with the PC+4 value
    b. JR : 1) read $ra register value and 2) jump to the address contained in $ra.

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

$a3 = 0xa

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 :
$t0 is modified to 0x7
pc is modified to 0xc

total_clock_cycles 3 :
$v0 is modified to 0x3
pc is modified to 0x10

total_clock_cycles 4 :
pc is modified to 0x4

total_clock_cycles 5 :
pc is modified to 0x14

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

program terminated:
total execution time is 6 cycles

Demo:

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

1) In the week of **3/30**, 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 JR.
2) In the week of **4/27**, 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 JR**.
3) In the week of **5/4**, you will show the **final demo** of your project. In this demo, you will show the single-cycle CPU that supports all 12 instructions.

Submission Guideline

- Write a report of your program in a pdf or MS 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 JR 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: **5/6 11:59PM (the same deadline for all sessions)**