Lab 1: Single Cycle MIPS

In this Lab assignment you will implement an instruction-level simulator for a single cycle MIPS processor in C++. The simulator supports a subset of the MIPS instruction set and can model the execution of each instruction.

An example MIPS program is provided for the simulator as a text file “imem.txt” file, which is used to initialize the Instruction Memory. Each line of the file corresponds to a Byte stored in the Instruction Memory in binary format, with the first line at address 0, the next line at address 1, and so on. Four contiguous lines correspond to one whole instruction. Note that the words stored in memory are in “Big-Endian” format, meaning that the most significant byte is stored first.

We have also defined a “halt” instruction as 32’b1 (0xFFFFFFFF), which is the last instruction in every “imem.txt” file. As the name suggests, when this instruction is fetched, the simulation is terminated. We will provide a sample “imem.txt” file containing a MIPS program. You are strongly encouraged to generate other “imem.txt” and “dmem.txt” files to test your simulator.

The Data Memory is initialized using the “dmem.txt” file. The format of the stored words is the same as the Instruction Memory. As with the instruction memory, the data memory addresses also begin at 0 and increment by one in each line.

The instructions that the simulator supports and their encodings are shown in Table 1. Note that all instructions, except for “halt”, exist in the MIPS ISA. The *MIPS Green Sheet* defines the semantics of each instruction.

Name	Format Type	Opcode (Hex)	Func (Hex)
addu	R-Type	00	21
subu	R-Type	00	23
addiu	I-Type	09	
and	R-Type	00	24
or	R-Type	00	25
nor	R-Type	00	27
beq	I-Type	04	
j	J-Type	02	
lw	I-Type	23	
sw	I-Type	2B	
halt	J-Type	3F	

Table 1. Instruction encodings for a reduced MIPS ISA

Skeleton Code

The file "MIPS.cpp" contains a skeleton code for the assignment. You need to fill in the missing code. In this section, we provide descriptions for each of the components in the skeleton code.

Classes

We have defined four C++ classes that each implement one of the four major blocks in a single cycle MIPS machine, namely RF (to implement the register file), ALU (to implement the ALU), INSMem (to implement instruction memory), and DataMem (to implement data memory).

1. **RF class:** contains 32 32-bit registers defined as a private member. Remember that register $0 is always 0. Your job is to implement the *ReadWrite()* member function that provides read and write access to the register file.
2. **ALU class:** implements the ALU. Your job is to implement *ALUOperation()* member function that performs the appropriate operation on two 32 bit operands based on ALUOP. See Table 1 for more details.
3. **INSMem class:** a Byte addressable memory that contains instructions. The constructor *InsMem()* initializes the contents of instruction memory from the file imem.txt (this has been done for you). Your job is to implement the member function *ReadMemory()* that provides read access to instruction memory. An access to the instruction memory class returns 4 bytes of data; i.e., the byte pointed to by the address and the three subsequent bytes.
4. **DataMem class:** is similar to the instruction memory, except that it provides both read and write access.

Main Function

The main function defines a 32 bit program counter (PC) that is initialized to zero. The MIPS simulation routine is carried out within a while loop. In each iteration of the while loop, you will fetch one instruction from the instruction memory, and based on the instruction, make calls to the register file, ALU and data memory classes (in fact, you might need to make two calls to the register file class, once to read and a second time to write back). Finally, you will update the PC so as to fetch the next instruction. When the halt instruction is fetched, you are to break out of the while loop and terminate the simulation.

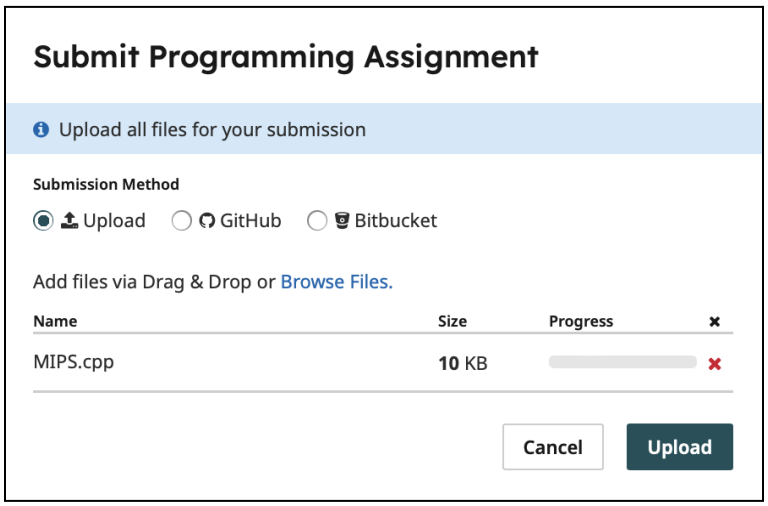
Make sure that the architectural state is updated correctly after the execution of **each** instruction. The architectural state consists of the Program Counter (PC), the Register File (RF), and the Data Memory (DataMem). We will check the correctness of the architectural state after *each* instruction.

Specifically, the OutputRF() function is called at the end of each iteration of the while loop, and will add the new state of the Register File to "RFresult.txt". Therefore, at the end of the program execution "RFresult.txt" contains all the intermediate states of the Register File. Once the program terminates, the

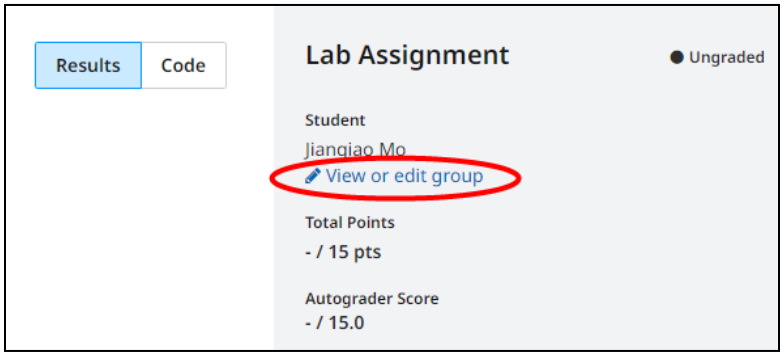
OutputDataMem() function will write the final state of the Data Memory to "dmemresult.txt". These functions have been implemented for you. Do not modify them.
(Note: **You should delete the "RFresult.txt" and "dmemresult.txt" files before re-executing your program**, otherwise the new results will append to the previous results.)

Your Assignment:

1. We have provided the skeleton code in the file MIPS.cpp. Implement the functions where you see "TODO: implement".
    a. A Makefile has been provided for you to compile the source code (do not modify the Makefile)
    b. We will be compiling your code with: `g++ version 11.3.0.` **Please make sure that your code is compatible with g++. See 4. for more detail about compiling.**
    c. You can compile your design by typing "make" which will create an executable called "MIPS". Run the executable by calling: "./MIPS**"**.
    d. As mentioned above, calling the binary "./MIPS" expects both an "imem.txt" and "dmem.txt" file in the same directory and produces two output files (RFresults.txt and dmemresults.txt).
    e. We have provided an "imem.txt" and "dmem.txt" files containing a sample program (loading two variables and adding them) and initialized data. These files must be in the same directory as the source code.
    f. We have provided the correct output results for the provided test case in the "expected_results" directory. Make sure you pass this test case before submitting your code. However, we encourage you to write your own MIPS programs and check your design for different cases.
2. You will be submitting your assignments on Gradescope. **You must upload ONLY one file on Gradescope: "MIPS.cpp". Do not zip the file or upload any other file.**



3. You are allowed to work with one other student as a two-person group. If you choose to work with another student, only one of you needs to submit the files on Gradescope **and you must add your group partner after submission.** The screenshot below shows how to add another student to your submission after you have submitted the assignment.



You can form your group (add a group member) after you submit your assignment.

4. When you submit your code to Gradescope, we will check that your code compiles without errors. If your code compiles, you will see the following message as the output:

```
test_compile (test_all_cases.Test) (0/0)
```

In case your compilation is unsuccessful, you will receive the following message

```
test_compile (test_all_cases.Test) (0/0)
Test Failed: False is not true : Compilation process failed. Please check your code and try again.
```

You may submit your code as many times as you like before the submission deadline, but please make sure you pass the test_compile case. Otherwise, you will receive a zero. All other test cases will be hidden until the submission deadline has passed. We will be testing your code against a series of MIPS programs that test the instructions supported in Table 1. We expect your uploaded file to be called **exactly** "MIPS.cpp", and your binary executable **must** output "RFresult.txt" and "dmemresult.txt" for the test cases to run correctly. (They are already specified in the code, so don't change them).

5. On Gradescope, there is a Similarity Test so don't copy answers from other groups. We also set up 20 test cases and you earn 0.5 points by passing each.

Due Date:

Friday Sept 22, 2023, 23:59 Eastern Time.

Some useful references for this lab:

1. A brief introduction to C++ (https://web.eecs.umich.edu/~sugih/pointers/c++.pdf)
2. A reference for the C++ bitset class (http://www.cplusplus.com/reference/bitset/bitset/)
3. A reference to the C++ string class (http://www.cplusplus.com/reference/string/string/)
4. Makefile (https://earthly.dev/blog/g++-makefile/)
5. Get Started with C++ and MinGW-w64 in Visual Studio Code.