MIPS Emulator
Project 2 – CS 3339

Due Date per Canvas. Friday before 11:59pm, 24 hour late period -10% after that no submissions accepted.

PROBLEM STATEMENT

In this project, you will write an emulator that reads MIPS machine instructions from the provided binary executable files and correctly simulates their behavior, i.e. emulates a MIPS CPU that executes the program.

Write your emulator code in the provided C++ code skeleton, which models several components of a CPU. I have provided the following files:

- A Memory class (`Memory.h`, `Memory.cpp`) that emulates a byte-addressed memory with load and store operations
- An ALU class (`ALU.h`, `ALU.cpp`) that performs arithmetic operations given an opcode and two 32-bit sources
- A CPU class (`CPU.h`, `CPU.cpp`) that simulates the behavior of a CPU, i.e. fetching instructions from instruction memory, decoding them, and executing them. The CPU includes a program counter and a register file. It can interact with instruction and data memory banks and instantiates an ALU object on which to execute instructions. Except for putting your name in the header, **all of your changes go in the file `CPU.cpp`, and only in the `decode()` function.**
- A driver (`Simulator.cpp`) that loads the instruction memory with words from the executable and invokes the CPU execution
- A debug macro (Debug.h) that will turn on additional print statements
- A Makefile

Although you only need to change `CPU.cpp`, you should read and understand all of the given code, as it will form the basis of all the remaining projects.

The `jal` and `trap` instructions are already (partially) implemented for you as an example. You need to implement all the rest of the instructions from Project 1. Note that depending on how you structure your code, you may also need to add statements to my partial implementation of `jal` and `trap`! ***The provided implementations do not give meaningful values to all of the control signals. Remember never to leave a control signal floating at an undefined or unsafe value.***

The CPU skeleton already includes code to increment the program counter (PC), to fetch the next instruction from instruction memory, and to invoke the ALU to execute a particular ALU op on two sources (as well as to conditionally use the ALU output as the address of a load or store operation or the next PC for a branch or jump instruction). It also includes code to update the register file with the ALU output or load data.

Your code, which belongs in the `decode()` function, needs to extract the bit-fields from the instruction (just like in Project 1), and then set up all the control signals used by the ALU, load and store mechanism, and the register file writeback.

Unlike the MIPS single-cycle CPU we will cover in class, you should resolve branch and decode instructions inside decode(). Do not use the ALU to compare the sources for conditional branch instructions. Rather, do the comparison and update the `pc` field (as is done in the `jal` example) inside decode().

Do not change any code outside of the `CPU::decode()` function. Your decode() code may not update the register file (e.g., no statements like `regFile[x] = y;` may go in decode) and may not interact with memory (e.g., no calls to loadWord or storeWord in decode). Use decode() to correctly set up the control signals so that my execute(), mem(), and writeback() functions behave correctly.

ASSIGNMENT SPECIFICS

This project is to be submitted individually, and you should be able to explain all code that you submit. You are encouraged to discuss, plan, design, and debug with fellow students.

The next 3 projects in this class will depend on the code for this project, so it is important to correctly solve it! To help with debugging, I have provided debug verbosity output for all inputs except nqueens. You can diff this output with the output created by your emulator (with debug verbosity turned on; see below) in order to identify where program execution begins to go wrong.

The source files are contained in **cs3339_project2.tgz** and the debug-verbosity expected output files (*.debug_out) are in **cs3339_project2_debug_out.tgz**.

After moving the compressed tar file to your home space on zeus, the following command will extract the files:

```
$ tar xzvf cs3339_project2.tgz
```

This will create a directory called project2 that contains several sample executables (`*.mips`) for testing your emulator, the corresponding expected output files (`*.out`), and all of the C++ files described above, including the CPU skeleton (`CPU.cpp`) to which all of your code must be added.

Once you have added the missing code to `CPU.cpp`, compile the project with this command:

```
$ make
```

Assuming the compilation succeeded, you can run your emulator on the provided `*.mips` files. For example, to simulate the execution of `sssp.mips`, run this command:

```
$ ./simulator sssp.mips
```

If your project gives you an odd compilation error (especially after moving files from one machine to another) try removing the executable and all the intermediate object (.o) files with this command:

```
$ make clean
$ make
```

Your emulator must generate precisely the same output as in the provided expected output files. See the Project 1 handout for a reminder about how to redirect console output to a file and use the 'diff' utility to compare it to the expected output.

Debugging

You can use your solution for Project 1 to disassemble the executables to find out what instructions should be executed. (Note that most of the Project 1 `*.mips` files cannot be executed, however).

To use the provided debug_out reference files:

- Turn on additional output by un-commenting out the '`#define DEBUG`' line in `Debug.h`.
- Recompile your program
- Run your program and redirect the output to a file such as rand_debug_temp.
- Start with the smaller rand.mips and pqueue.mips, don't run nqueens.mips with debug outputs enabled – it will fill up your drive! Even with the smaller files it's best to <ctrl-c> after a few seconds, there is usually no need to run them to completion.
- Using the same method as in project 1, 'diff' the provided rand.debug_out and your output file.
- Find the first miscompare; the associated instruction or the one prior is incorrect.
- Correct the instruction and repeat the process.

If you want to add debugging output to `CPU.cpp` please do so using the `D(x)` macro defined in `Debug.h` so that it will not print in normal operation. [And note that if you add output, you won't be able to cleanly diff the expected debug-verbosity output files with mine].

If you get an "unaligned [inst/data] memory access" error when testing your code, it means that your emulator attempted a load (either of an instruction or data) or store that was not to a word-aligned address. If you get a "[inst/data] memory access out of range" error, it means that your emulator attempted to access a memory location that does not hold either program instructions or data. In either case, your code either computes load/store or branch/jump addresses incorrectly, or the value in the base register associated with one of those operations is incorrect due to an earlier instruction's incorrect implementation. Use my debug-verbosity output files to help you debug why.

Additional Requirements:

- **Your code must compile with the given `Makefile` and run on zeus.cs.txstate.edu**
- Add your name to the CPU.cpp header but do not change any code outside of the `decode()` function in `CPU.cpp`
- Your code must be well-commented, sufficient to prove you understand its operation
- Make sure your code doesn't produce unwanted output such as debugging messages. (You can accomplish this by using the `D(x)` macro defined in `Debug.h`)
- Make sure your code is correctly indented and uses a consistent coding style
- ***Do not use TAB characters for indentation!*** (Use a consistent # of spaces per indent level)
- Clean up your code before submitting, i.e., make sure there are no unused variables, unreachable code, etc.

SUBMISSION INSTRUCTIONS

Make sure your name and netID is at the top of the `CPU.cpp` file in the header block.

All project files are to be submitted using Canvas. If you want to provide any special instructions or comments to the grader, including notes about features you know do not work, write them in a Canvas comment.

Submit only your final `CPU.cpp` file to Canvas. Do not submit any additional files (i.e., no other .cpp files, no .h files, no input or output files).

You may submit your file(s) as many times as you'd like before the deadline. Only the last submission will be graded. Late submissions during the 24-hour late submission period will be graded with a 10% penalty. ***Canvas will not allow submissions after the deadline***, so I strongly recommend that you don't come down to the final seconds of the assignment window. Assignments will not be accepted after closure.

