ECE 200: Project 2 ISA Emulator

Due Date: April 17, 2020 11:59:00 PM

1 Introduction

In this project you are to create a MIPS emulator, written in C, that is capable of executing the MIPS-I subset of the MIPS Instruction Set Architecture. Starting with the test binaries provided to you, you will fetch, decode, and execute instructions to mimic an actual MIPS processor, effectively implementing a toy version of MARS.

To that end, you may work in a group of **at most two** students or you may elect to work alone at your own discretion. If you choose to work in a pair, please make sure that both you and your partner do the same amount of work. You are encouraged to collaborate. Do not, however, share your code (unless within the same group, of course).

2 Getting Started

Download and unpack the project files using the command

```
tar -xvzf Project2.tar.gz
```

The extracted directory will have the following file structure:

- `Instruction_Subset.pdf`

  Documents a list of all the instructions your emulator is expected to decode and execute successfully. Note that this is a list of all MIPS-I instructions that do not make use of the floating point co-processor. Additionally, **you do not need to implement the *break* instruction**. For more information, refer to the first bullet in the *Task* section.

- `MIPS_ISA.pdf`

  The technical specifications of the MIPS ISA. This document contains detailed descriptions of each instruction in the MIPS32 instruction set (of which MIPS-I is a subset).

- `Project2_Instructions_S20.pdf`

  The document you are currently reading.

- `code/`

  This directory contains the source files you will use to complete this project.

Within said **code/** directory you will find the following files:

- `Makefile`

  Used to compile the project to generate an executable. Running the *make* command in your command line (at this directory) generates an executable called *eMIPS*.

- `src/`

  This directory contains the files that constitute the emulator.

- `tests/`

  Contains three sub directories, each corresponding to one tier of tests, containing binaries of each test program. You are required to have all these tests running. Refer to the section on grading for the point distribution. Note that tests are compiled for a Big-endian machine.

Within the **src/** directory you will find the following files:

- `PROC.c`

  Contains the *main()* function of the program, which is where you will be implementing your emulator. Note that this is **the only file that needs to be altered**. Refer to the *Submission* section for additional information.

- `RegFile.c/h`

  Contains the declaration of the register file. The register file is declared as an array of 34 *int32_t* (32 general purpose registers as well as *HI* and *LO*), and can be accessed as a normal array (e.g., int x = RegFile[2];, RegFile[2] = 10;).

- `Syscall.c/h`

  Contains an abstracted implementation of the relevant Linux syscalls. In order to execute a syscall, simply call the function *void SyscallExe(uint32_t SID)* where *SID* is the syscall ID found in register 2.

- `elf_reader/`

  Contains the source files necessary to interpret compiled binaries. This directory also gives you access to functions which interact with main memory. These functions are

  ```
  void writeByte(uint32_t ADDR, uint8_t DATA, bool DEBUG)
  void writeWord(uint32_t ADDR, uint32_t DATA, bool DEBUG)
  uint8_t readByte(uint32_t ADDR, bool DEBUG)
  uint32_t readWord(uint32_t ADDR, bool DEBUG)
  ```

  where *ADDR* is the address in main memory you wish to interact with, *DATA* is the data you wish to write to that address, and the *DEBUG* flag, set to true, will print a small debug message when the function is called.

- `utils/`

  Contains files that manage the heap. **You do not need to read or understand anything in this directory to complete the project.**

3 Task

You may use whatever methodology you would like to implement your emulator. The basics have been set up for you already in the *PROC.c* file. A variable called *ProgramCounter* stores the current PC of the program and is initialized to the start of the program binary for you. Within the *for* loop, the variable *CurrentInstruction* is set equal to the contents of main memory at address *ProgramCounter*. The program then prints the contents of the register file. Your task is to decode and execute any arbitrary, MIPS-I *CurrentInstruction*. Note that the execution of the instruction also involves setting the program counter to the correct next value. There are three additional things to note:

- Your emulator does not need to include either co-processor. Therefore, **you do not need to implement exceptions**. For instructions that include exceptions, ignore this functionality. For example, the *add* instruction would **not** raise an exception on overflow as specified in the ISA.
- Be sure to implement the branch delay slot.
- In case an unsupported instruction arises, treat it as a *NOP* and proceed (continue with the next instruction).

4 Running Your Code

Typically, we would require that this project properly execute on the ECE cluster. However, given the current situation, we recognize that many of you will not be able to create an account or be unable to *ssh* into your account due to not having the University's VPN. Accordingly, we will provide several options for how you may work on this project:

1. Use the ECE servers: if you have access to these servers, you can (and should) make use of them to develop your project. The servers have all of the necessary tools already installed on them and have been tested to work with this project. Note that in order to *ssh* into these servers from outside the University's network, you will have to have installed the University's VPN, which can only be done while on the University's network.
2. Use your local machine: if you have either a Linux or a macOS machine, you will likely be able to run this project. You will need to install *make* (Linux: *sudo apt-get install make*) and *gcc* (Linux: *sudo apt-get install gcc*)—macOS users should look into installation via a package manager such as Homebrew. In the case of Brew, *make* and *gcc* may be installed via *sudo brew install make* and *sudo brew install gcc*, respectively. If you have a Windows machine, you will likely have difficulties getting the above dependencies to install properly. Accordingly, we recommend that Windows users make use of option 3.
3. Install a virtual machine: we recommend installing VirtualBox and running the Ubuntu 18.04 operating system. There are an abundance of tutorials available online that can

assist you in this process. Additionally, you may make use of office hours where we will provide you with assistance. You will need to install *make* and *gcc* upon setting up your virtual machine. Refer to option 2 for details.

Make sure to compile your code using the *Makefile*. Once compiled, you can execute one of the test programs on your emulator by running

```
./eMIPS path/to/test number_of_instructions
```

For example, to run a million instructions of the *hello* program, you would execute

```
./eMIPS test/cpp/hello 1000000
```

Note that the *asm_tier2* and *cpp* tests will eventually terminate via a syscall (this will never take more than a million instructions).

During the execution of a program, any stdout and stderr messages will be dumped in the *stdout.txt* and *stderr.txt* files.

5 Advice

- Start early. Irrespective of your proficiency with C this project will take time and patience.
- The *printf()* function is your best friend. Use it whenever in doubt.
- The best place to start is to try and break down your entire process of execution and put that down on paper. This will make it easier for you to keep track of the various aspects of your implementation.
- The *make* program is intended to only recompile the aspects of your project that have changed since the last compilation. However, **this does not always work as intended**. We therefore recommend executing *make clean* prior to executing *make*. This will remove the existing compilation as well as your diagnostic *stdout.txt* and *stderr.txt* files.
- Exercise proper version control, especially if working in a pair. We recommend using a service such as GitHub.
- Consult TA office hours. It is in our interest to help you help yourselves.

6 Grading

	Test Cases	Points
Tier 1	arith, branchtest, hilo, linktest, systest, zero	40 Points
Tier 2	MatrixMultiplication, MergeSort, BinarySearch	30 Points
Tier 3	class, hello	30 Points

7 Submission

Via Blackboard, you are to electronically turn in a compressed project folder containing **only** the following:

1. A **README** file (in Markdown or plaintext) containing:
   (a) A brief write-up of the specifics of your implementation.
   (b) Your partner's name (if you worked in a pair) and the distribution of your workload.
   (c) Any idiosyncrasies of your implementation that may be relevant for partial credit.
   (d) A list of the testbenches your implementation has cleared as well as a list of the testbenches your implementation may have failed, if any (on our end, each test will be evaluated either manually or automatically and a code review will be performed in addition).
   (e) You are welcome (and encouraged) to facilitate grading by providing an exposition of any bugs or errors you may have been unable to resolve.
2. Your **PROC.c** file (your implementation, commented liberally).

Compress and archive your project folder using the following command and naming convention:

```
tar -cvzf FirstInitialLastName.tar.gz Project2
```

For example, Mark Randolph would compress his project folder using

```
tar -cvzf MRandolph.tar.gz Project2
```

You are **required** to adhere to this naming convention. Note that only one person per pair need submit the project (name the project after the person in charge of submitting)—both collaborators will be given the same grade.