Lab Assignment 2: Writing MIPS simulator

100 points (10% of entire grade)

Each team only allows one/two member(s)

Due: 11:59pm, Monday, April 4th, 2022

The goal of this assignment is to understand how MIPS datapath is designed and performed. To achieve this, **you will write a cycle-accurate instruction-level simulator that supports a subset of the MIPS ISA.** This instruction-level simulator will model the behavior of each instruction and will allow the user to run MIPS programs and see their outputs.

What You Should Do

Your job is to implement the process_instruction() function in sim.c. The sim.c file is part of code template you must use, and details about the template will be explained later. The process_instruction() function should be able to simulate the instruction-level execution of the following subset of MIPS instructions:

R-Type	add, addu, sub and
I-Type	addi, addiu
J-Type	J

When there is no instruction to process, the process_instruction() function that you write to set the global variable RUN BIT to 0 so that the program terminates.

Your simulator should simulate each instruction's behavior precisely by updating the proper register and memory location correctly after executing each instruction. TA will evaluate your simulator using **several input cases (included in this lab materials)** that cover the instruction sets you are supposed to implement.

Each MIPS instruction you are supposed to implement belongs to one of the MIPS instruction types: R-Type, I-Type, and J-Type. Refer to the MIPS reference data[1] for details of each instruction type. While the table has many instructions, there are actually only a few unique instruction behaviors with a number of minor variations.

Finally, note that your simulator does not have to handle instructions that we do not include in the table above or any other invalid instructions. We will only test your simulator with valid code that uses the instructions listed above.

The simulator will take an input file that contains a MIPS instruction or program. Each line of the input file corresponds to a single MIPS instruction written in a hexadecimal string. For example, "add $t1, $t2, $t3" corresponds to 0x014b4820 in a hexadecimal representation.

The simulator will execute the input program one instruction at a time. After each instruction, the simulator will update the MIPS architectural state: values stored in registers and memory. The simulator is partitioned into two main sections: the (1) shell and the (2) simulation routine. Your job is to implement the simulation routine.

Logistics: The code template for this assignment is provided in the Canvas. Download the zip file in your local directory and unzip it. The unzipped directory contains 4 files: Makefile, shell.c, sim.c, and shell.h.

[1] https://inst.eecs.berkeley.edu/~cs61c/resources/MIPS_Green_Sheet.pdf

The shell.c (including shell.ch) implements an interactive shell for running the simulator. DO NOT modify both files unless you want to debug a shell feature. In the src/ directory, we provide two files (shell.c and shell.h) that already implement the shell. There is a third file (sim.c) where you will implement the simulator routine - this is the only file that you are allowed to change.

Interactive shell:
The provided shell.c implements several command lines to control the execution of the simulator. You can think it as a command line version of the MARS simulator like loading and running program, examining register and memory values, etc. The shell accepts one program file as a command line argument and loads it into the program memory. The shell supports the following commands:

- g | G | go: simulate the program until it indicates that the simulator should halt.
- r | R | run <n>: simulate the execution of the machine for n instructions.
- mdump <low> <high>: dump the contents of memory, from location low to location high to the screen <low> and <high> address should be provided as hexadecimal numbers.
- rdump: dump the current instruction count, the contents of $0- $31, FLAG N, Z, C, V, and the PC to the screen.
- i | I | input reg_num reg_val: set register (denoted reg_num) to value (reg_val).
- ? | help: print out a list of all shell commands.
- q | Q| quit: quit the shell.

The Simulation Routine
The simulation routine carries out the instruction-level simulation of the input MIPS program in machine code. During the execution of an instruction, the simulator should take the current architectural state and modify it according to the ISA description of the instruction in https://inst.eecs.berkeley.edu/~cs61c/resources/MIPS_Green_Sheet.pdf. The architectural state includes the PC, the registers, FLAGS, instruction format, and the memory. The state is contained in the following global variables:

```
#define MIPS_REGS 32
typedef struct CPU_State {
        uint64_t PC;                /* program counter */
        uint64_t REGS[MIPS_REGS];   /* register file. */
        int FLAG_N;        /* negative flag or sign flag*/
        int FLAG_Z;        /* zero flag */
        int FLAG_V;        /* overflow flag */
        int FLAG_C;        /* carry flag */
} CPU_State;

/* STATE_CURRENT is the current rch. state */
/* STATE_NEXT is the resulting arch. state after the current instruction is processed */
CPU_State STATE_CURRENT, STATE_NEXT;

int RUN_BIT; /* initialized to 1; need to be changed to 0 if the HLT instruction is encountered */
```

Furthermore, the simulator models the simulated system's memory. You need to use the following functions, which we provide, to access the simulated memory:

```
uint32_t mem_read_32(uint64_t address);
void     mem_write_32(uint64_t address, uint32_t value);
```

As mentioned in the lecture, memory is byte-addressable whereas registers are word-addressable. Furthermore, you will implement a big-endian architecture where bytes are ordered from the big end (most significant bit).

The provided simulator template, sim.c includes an empty function named process_instruction(), which called by the shell and simulates one machine instruction at a time. You have to write the code for process_instruction() to simulate the execution of instructions. You can also write additional functions to make the simulation modular. (Keep in mind that you will be using the code that you write in later labs in order to validate your work.)

Lab Files
The lab2.zip contains all code template for this assignment. You can compile the simulator with the provided Makefile. You will be provided with the same input files for testing your implementation. To test whether your implemented simulator supports all ISAs you are supposed to implement, you should create your own test files, which should be in hexadecimal formats. The simulator won't accept files in other formats.

We assume your programming environment should be in Linux or similar. If you have access to Linux machine (either native or in a virtual machine), make sure the system includes the GCC compiler. The provided template should work with any version of GCC. If your code uses a different compiler than GCC, you should provide enough information (how to compile and run) in your submission. If you use Macintosh, you should also be able to compile and run C code using Terminal. Lastly, if your system is Windows, Cygwin (https://www.cygwin.com/) is strongly recommended. After installing Cygwin, you will have a Linux-like prompt where you can compile and run C programs using GCC.

Submission Guideline:

- All program submissions should be made to Canvas.
- One copy of the code needs to be submitted for one group
- Write down team member's name at the top of the sim.c file.
- Document your code well so it is easily readable.
- You need to any additional document or test files/cases that are used in your implementation.

Useful Tips:

- Please refer to MIPS green sheet. https://inst.eecs.berkeley.edu/~cs61c/resources/MIPS_Green_Sheet.pdf.
- Use the MARS simulator to generate corresponding machine code for an individual instruction or a MIPS program.
- Use the MARS simulator also to convert decimal to binary or hexadecimal, or vice versa.
- Again, you are allowed to modify sim.c only and leave the other files as it.

BONUS (10 points)

In addition to the given MIPS ISAs, implement the following additional ISAs.

- lb, sb
- jal

Note that you MUST use the provided mem_read_32 and mem_write_32 functions to implement loading and storing 8-bit (byte). Note that you need to call mem_read_32 and mem_write_32 with only 32-bit-aligned addresses (i.e., the bottom two bits of the address should be zero). Note also that you need to modify only the appropriate part of a 32-bit word.

Grading Guidelines

- Your simulator compiles without errors. (and appears on the surface to be correct): 30 pts.
    - Submit only your implemented sim.c file including all your comments within it.
- Program executes correctly: 45 pts.
    - Should pass provided test cases.
- Documentation and comment on the program: 25 pts
    - Have a commented program header with each group member's name and lab assignment
    - The program should be well commented so much so the TA can understand what you are doing without you there to explain it.