COS/ELE 375
Spring 2020

Project 1: MIPS ISA High-Level Simulation.
Submit via blackboard April 2 AoE.

Introduction

The purpose of the lab component of the class is to give you hands-on experience with designing microprocessors and microprocessor-based systems.

Much architecture research is performed using high-level simulations of processor behavior, in order to assess functionality or performance. Your work in Project 1 will be to create such a simulator for the main functionality of the MIPS ISA as documented in the frontispiece of your textbook (formerly called the "green card" because it used to actually be a green card in the textbook). The simulator is to be written in C++.

Your task in this first lab assignment is to write a C++ language program which will read a binary machine code file, interpret the bits in that file as instructions for the MIPS instruction set, and simulate the execution of those instructions. Such a program is called a functional simulator.

As discussed in class, you will work in groups of two for the lab assignments.

Overview

The basic idea of this simulator is to execute repeatedly a loop in which instructions are "fetched" by reading them from a simulated memory, then decoded and executed. You will be reading instructions that are encoded using the MIPS bit encodings as documented in the frontispiece of the textbook. YOU DO NOT NEED TO DEAL WITH THE FLOATING POINT INSTRUCTIONS. There is a MIPS ISA pdf that has been annotated to cross out the instructions you don't need to do.

The instruction decode portion of the simulator should use bit manipulations to discern the instruction type and relevant fields and values from the binary encoding of the instructions.

Once the simulator has decoded an instruction, the simulator reads input, sets output, or modifies state as "instructed" to do so by the instruction.

All of the state (aside from the state of memory, see "Memory" section for details of the provided memory abstraction) should be mimicked by having appropriate program variables. For example, a bank of general purpose registers could be simulated using an array of integers (if the integers on the system the simulator is running on are as wide as or wider than the registers of the simulated processor):

```
uint32_t regs[NUMREGS];
```

The frontispiece of the book offers a summary of the MIPS instruction set. In particular, it clusters the instructions into different "classes". Your simulator should support any program comprised of any instructions except those in the Floating Point portion of the

spec. A marked-up pdf in the project folder specifically X's out the unneeded FP instructions.

The MIPS ISA does not contain a HALT instruction, so we use the code 0xfeedfeed to signify the end of the code section of a program. When the pseudo-halt instruction 0xfeedfeed is read by the simulator, it **must** dump the register and memory state before exiting. (See the "Provided Files" and "Simulator Skeleton" sections for details on how to accomplish this.)

We suggest that you start your work by implementing the pseudo-halt instruction first. Then implement easier instructions first, gradually adding more challenging instructions. As you add instructions, write test cases (in MIPS assembly) for those instructions. We have provided an initial test case, but you should certainly do more.

Provided Files:

To allow you to concentrate on building the core of the simulator, we provide you with a number of files containing abstractions, utilities, and utility functions. These files are contained in the project tarball on Blackboard. The following is a list of the relevant files ordered by directory structure:

bin/
- mips-linux-gnu-as
- mips-linux-gnu-objcopy
- mips-linux-gnu-objdump

src/
- EndianHelpers.h
- example.cpp
- MemoryStore.h
- RegisterInfo.h
- UtilityFunctions.o

test/
- fib.asm
- fib_mem_state.out
- fib_reg_state.out

Below is a short description of each of the above files.

mips-linux-gnu-as – MIPS assembler. Converts MIPS text assembly to binary ELF files.

mips-linux-gnu-objcopy – MIPS object file copier/translator. Used to convert ELF files to flat binary files that are read by your simulator.

mips-linux-gnu-objdump – MIPS object file disassembler. Used to inspect the instructions in a binary ELF file.

EndianHelpers.h – Signatures of the functions to convert unsigned integers from little-endian to big-endian.

example.cpp – An example C++ file showing use of the memory abstraction.

MemoryStore.h – The interface to the memory abstraction.

RegisterInfo.h – Signature of the register file dump function and definition of the RegisterInfo struct passed to it as an argument.

UtilityFunctions.o – A binary file containing the implementations of the memory abstraction and other utility functions. This file is not intended to be human-readable.

fib.asm – An example MIPS assembly test program.

fib_mem_state.out – The memory state dump of running the compiled version of fib.asm.

fib_reg_state.out – The register state dump of running the compiled version of fib.asm.

Test case structure

Test cases for this project are MIPS assembly files consisting entirely of a single .text section. Any data values that must have specific values for the program to work must be manually listed at the end of the file (after 0xfeedfeed) using .word directives. For example, consider the following assembly program:

```
la   $t1, 20
la   $t2, 24
lw   $t3, 0($t1)
sw   $t3, 0($t2)
.word 0xfeedfeed
.word 0xac
.word 0xdb
```

Each instruction in the above program is 32 bits = 4 bytes. Thus, the last instruction is located at bytes 12-15.

A ".word" directive fills the corresponding location in the binary file with the raw data provided to the directive. For instance, the .word directive for 0xfeedfeed is at address 16 = 0x10, so the directive causes bytes 16-19 to get filled with 0xfeedfeed. Likewise, bytes 20-23 are filled with 0xac, and bytes 24-27 are filled with 0xdb. (This can vary slightly if the MIPS assembler adds a word or two of padding – see the "Compilation" section.)

Thus, when the third instruction reads from address 20, it will read the value 0xac (since no other instruction has written to it beforehand). Likewise, when the fourth instruction stores to address 24, it will overwrite the value of 0xdb. Note that it is quite possible for a program to access memory locations in its execution other than those locations initialized by .word directives.

At some point, you may also need the .align directive in order to skip bytes till you reach a required alignment. For instance, if the previous lines of the file reach address 0xa, then a ".align 4" directive will skip bytes (filling them with 0) until it reaches an address aligned for 4 bytes (in this case, address 0xc).

Compilation

To compile test cases, you should use a combination of mips-linux-gnu-as and mips-linux-gnu-objcopy. For example, to compile the example test case (assuming you have the provided binaries in your PATH), run the following commands:

```
mips-linux-gnu-as fib.asm -o fib.elf
mips-linux-gnu-objcopy fib.elf -j .text -O binary fib.bin
```

The first command assembles the text assembly file into a binary ELF file. This ELF file then needs to be translated into a flat binary file that only contains the code and any .word directives you may have added. This is done by calling mips-linux-gnu-objcopy on the .text section of the ELF file as shown in the second command. The fib.bin file created by the second command should be the input to your simulator.

The MIPS assembler may add an extra word or two of zero padding after the code and before your .word directives, and after your .word directives as well. Make sure any offsets you use to read/write values to the .word directive locations take these offsets into account.

To inspect the code in an ELF file, you can use the mips-linux-gnu-objdump utility. For example, running the following command

```
mips-linux-gnu-objdump -D -j .text fib.elf
```

disassembles the .text section of the ELF file, allowing you to see the instructions in it. If you want to examine the contents of your flat binary file, you can use hexdump. For instance, running

```
hexdump -e '"%08_ax :" 1/4 "%08x " "\n"' fib.bin
```

will print out the contents of the binary file fib.bin in 4-byte chunks with offsets on the left, as shown below:

```
00000000 :44000824
00000004 :74000d24
00000008 :0000ad8d
0000000c :01000a24
…
```

To compile your simulator, you will need to link your code with the implementation of the memory abstraction and utility functions provided in UtilityFunctions.o as follows:

```
g++ -o sim sim.cpp UtilityFunctions.o
```

Likewise, to compile the example.cpp use of the memory abstraction, you can use

```
g++ -o example example.cpp UtilityFunctions.o
```

Simulator Skeleton

A potential (not the only possible) outline of your program would be:

```
main (argc, argv) {
    Create a memory store called myMem
    Initialize registers to have value 0
    Read bytes of binary file passed as parameter into appropriate memory locations
    Point the program counter to the first instruction
    while (TRUE) {
        Fetch current instruction from memory@PC
        Determine the instruction type
        Get the operands
        switch (instruction type) {
        case 0xfeedfeed:
            RegisterInfo reg;
            Fill reg with the current contents of the registers
            dumpRegisterState(reg);
            dumpMemoryState(myMem);
            return 0;
        case INSTR1:
            Perform operation and update destination
            register/memory/PC
            break;
        ...
        default:
            fprintf(stderr,"Illegal operation...");
            exit(127);
        }
    }
}
```

Note the following points:

-Your simulator will be provided with one argument – the name of the binary file it is to run.
-You must create a MemoryStore object at the beginning of your simulator and use it to read and write memory.
-For consistency, initialize all registers to have a value of 0 upon startup. (The MemoryStore already internally initializes all memory locations to have a value of 0 upon startup.)
-You must read the bytes of the binary file to run into memory at address 0. Then point the program counter to address 0 and begin execution.
-When the program finishes, you **must** call dumpRegisterState with a RegisterInfo struct containing the current values of all registers, and you **must** also call dumpMemoryState with the MemoryStore object that you created at the beginning of the simulator. The dumpMemoryState function will dump the contents of a region of memory that will **vary** on a **test by test basis**. It is your responsibility to ensure that all of memory is appropriately updated by the running of your program, as you will not know which part of memory we will be examining for correctness.

The above calls will generate mem_state.out and reg_state.out files in the directory where the code is run. We will grade your code by comparing these outputs to the correct outputs

for these test cases. The implementation provided to you dumps memory from address 0 to address 500, which is the region inspected for the example test case provided to you. For the provided test case, the correct outputs are provided in the test/ directory of the tarball on Blackboard.

Warning: since the file you'll be reading from is a form of binary file, not a text file, do not treat it as a text file. In other words, scanf is not the proper function to use....

Memory Abstraction

We provide a memory abstraction for the ISA simulator you are to implement for Project 1 so that you do not need to implement memory on your own. You must use this abstraction to model memory in your simulator.

The memory abstraction is 64 KB (0x10000 bytes) large. It has three main functions (getMemValue, setMemValue, printMemory). Each of these functions returns 0 on success and a nonzero value on failure (an error message is also printed on failure).

The getMemValue and setMemValue functions take in a memory address, value, and size (which can be byte size (8 bits), half word size (16 bits), or word size (32 bits)). The getMemValue function returns a value from memory by reference through its "value" parameter, since the function's return value is the status of the operation (0 on success and nonzero on failure, as mentioned above). The printMemory function takes in a start and end address and prints the values of memory from the start address to the end address in 32-bit (4 byte) chunks, with five such chunks (20 bytes) per line. For example, the following three lines of printMemory(..) output each show the value of 20 bytes of memory, starting from addresses 0x0, 0x14, and 0x28 respectively.

```
0x00000000: 0x00000000 0x00000000 0x00000000 0x00000000 0xff000000
0x00000014: 0x00000000 0x0000c000 0x00000000 0x00000000 0x00000000
0x00000028: 0x00000000 0x00000000 0x00000000 0x0a000000 0x00000000
```

If the address range to print is not strictly divisible by 32, the value of a few bytes beyond the end address will also be printed at the end of the output.

Computers

We have tested the files for this project on the Nobel cluster at Princeton OIT. That is where we recommend that you work on the project. If you choose to try to do this elsewhere (e.g. your own computer) then it is your responsibility to ensure that it will work on Nobel when we grade your submission after you hand it in. For access to Nobel start with this link: https://researchcomputing.princeton.edu/systems-and-services/available-systems/nobel

What to Hand in

Your simulator should be written in C++. The compiler for grading will be g++ .

Your simulator should take a command-line argument indicating the name of the binary file to read in.

Use only one C++ source file and call it sim.cpp.

Please use the blackboard site to turn in your sim.cpp file.

You must turn in the sim.cpp file which you have written. It must compile properly on the Nobel cluster using:

g++ -o sim sim.cpp UtilityFunctions.o

If we cannot compile your source file, we cannot grade your assignment and you will have to resubmit; the delay will be taken as late days.

Grading

Grading will consist mainly of running test MIPS binaries (including but not limited to any sample binaries we give you) through your simulator. These test binaries will thoroughly test the operation of instructions. Particular attention will be paid to "corner cases". You will get back a "score sheet" indicating what instructions had problems and what kinds of problems. This represents 75 points of your grade.

We will also look at your source code to determine whether instructions are implemented correctly (i.e. we will not rely solely on test cases). Code which is difficult for us to understand (i.e. uncommented or incorrectly commented) will lose some points. Clarity of code and proper commenting represents 15 points of your grade.

Finally, efficiency matters. Excessively slow or needlessly inefficient simulators will be penalized. This represents 10 points of your grade.

A note on Endianness:

Endianness refers to the order in which bytes are stored in memory with respect to reading and writing multiple-byte words from memory. Big-endian processors use the leftmost or "big end" byte as the word address, while little-endian processors use the rightmost or "little end" byte as the word address. Thus, for example, if storing the value 0xAABBCCDD at address 0, big-endian and little-endian processors would store the value as shown in the following table:

Address	Big-endian	Little-endian
0x0	0xAA	0xDD
0x1	0xBB	0xCC
0x2	0xCC	0xBB
0x3	0xDD	0xAA

MIPS is a big-endian architecture, while x86 processors (such as the ones on the Nobel cluster and most laptops) are little-endian. So, in this assignment you are simulating a big-endian machine by executing the simulator on a little-endian machine. Thus, if on an x86 machine, you read a 32-bit instruction from one of your binary test programs (which are generated in big-endian format), the bytes will be reversed because the underlying x86 machine is little-endian. To assist you with changing the read data back to big-endian format, we provide two functions (signatures in EndianHelpers.h) to convert 16-bit and 32-bit integers from little-endian to big-endian:

```
extern uint32_t ConvertWordToBigEndian(uint32_t value);
extern uint16_t ConvertHalfWordToBigEndian(uint16_t value);
```

You do not need to worry about endianness when reading data from or writing data to the memory abstraction. Any endianness issues that arise with the memory abstraction are handled internally by the provided code.

Integer Types

We recommend you use the types for unsigned integers defined in inttypes.h (such as uint32_t, uint16_t, etc) to represent quantities so as to avoid some issues related to overflow and sign extension. The types in inttypes.h describe both the type and size of the integer in question, which is very useful when dealing with code that requires bit manipulation. For example, **u**int**32**_t is an **u**nsigned **32**-bit integer, and **u**int16_t is an **u**nsigned **16**-bit integer.