Mike Zhong
CS575 Operating Systems
Professor Yuting Zhang
12/12/18
Final Project Paper

Development of a MIPS Machine Simulator in C

Table of Contents

README

For quick start, simply navigate to the directory and type "make" in the terminal. After compilation, type "./main test.txt" to start the simulator using the "test.txt" file I have provided. Details of the "test.txt" file can be found under the Testing section.

Introduction

MIPS is a reduced instruction set computer instruction set architecture (RISC ISA) originally developed by MIPS technologies. Early MIPS architectures were 32-bit and were primarily used in embedded systems such as routers and game consoles. The basic stages of a MIPS machine involve fetching the instruction, decoding the instruction, executing, accessing memory, and writing back. MIPS has become a very popular tool for teaching computer architecture and assembly languages due to all MIPS instructions being encoded in 32-bits, and also the simple system of registers used. This project aims to build a MIPS simulator in C, whereby programs to be loaded and executed are saved as text files of '0' and '1' characters. Each instruction will consist of 32 '0' or '1' characters and a '\n' character will denote the end of an instruction. Two '\n' characters will separate the text portion of the program from the data portion. As a default, the application will dedicate 8 MB of memory for the whole program, allocating 8 KB for the text portion, and the remaining to be used for data.

Mike Zhong
CS575 Operating Systems
Professor Yuting Zhang
12/12/18
Final Project Paper

Data Types

We begin by defining the data types this simulator will use in the “mips_types.h”. Two data structures will be used, one for storing instructions and one for the machine’s memory. The instruction data structure will consist of a union of three different structs, all of which will be 32-bit in size. Because all MIPS instructions are 32-bit, we can encode R-, I-, and J-type instructions in the same size block of memory. Therefore, a union is appropriately used and specific number of bits are assigned to each component of the instruction, this will make referencing those components much easier. The memory data structure consists of 8 MB of uint32_t. Within this union, the memory is further broken up into text and data sections which can be accessed directly. The memory data structure also contains 32 registers, represented by a 32 element array of uint32_t. The special registers are denoted in the union allowing for direct access to those registers. The prototypical instruction is a function that returns void, and takes two arguments, an instruction, and a pointer to a memory object. Because the input program is a text file of ‘0’ and ‘1’ characters, a helper function, my_atoi() was created to translate lines of ‘0’ and ‘1’ representing instructions into uint32_t. The function takes a character pointer and will scan until it reaches a ‘\n’ character. It will then return the uint32_t value of the character string. NOTE: The function will treat any non-‘1’ character as a 0.

Design

A top-down approach was taken in designing the MIPS simulator. In main(), we first allocate the machine’s memory in the heap and set all bits to 0. Next, we open the file/program specified by the command line argument and get a handle for it, we can use this handle and the memory allocated to load the program into memory. Now with all the instructions and data loaded, we set our program counter to 0, and begin the fetch, decode, execute cycle. This cycle will continue until an instruction of 0 is detected, the contents of the registers will print and the program will end. The basic design can now be further broken down into smaller pieces. We will need to load the program, that sub-routine will need a function to convert strings of ‘0’ and ‘1’ into uint32_t. We will need a function to decode an instruction and execute the appropriate MIPS instruction. Lastly, we will need a function to print the contents of the registers. In deciding how to separate one instruction from the next, I decided to use a ‘\n’ character. Of course, a real machine does not need this but since we will be creating our programs using ‘0’ and ‘1’ characters, it would be unwieldy and infeasible to have our instructions one after the other since we as humans could not easily decode them, or tell where one instruction ends and the other begins. The end of the text section will be denoted by an additional ‘\n’ character, this will signal the start of the data section.

Mike Zhong
CS575 Operating Systems
Professor Yuting Zhang
12/12/18
Final Project Paper

Implementation

All individual instructions were implemented with the guidance and assistance of reference 1. Each instruction shares the same prototype, lending itself to use with file pointers. MIPS instructions take instruction objects and a pointer to the machine's memory. The decoding process takes place across two functions. First, the raw instruction pointed to by the program counter and the memory pointer are passed to get_instruction(). Get instruction decodes the opcode and then passes either the opcode or the function (depending on the instruction type) to a second function. This second function decodes the specific instruction type and returns a function pointer to the specific MIPS instruction encoded to the calling function. This function pointer then gets passed back to main() and assigned to the generic function pointer, execute(). Execute() now holds the exact MIPS instruction encoded so we can call execute() and pass it the raw instruction and the pointer to the machine's memory. In main(), we first use malloc() and memset() to allocate the machine's memory, set all bits to 0, and obtain a pointer/handle to that memory. We also use open() to get a handle to the file/program specified in argv[1]. We now pass both the program and memory handle to load_program(), which translates the programs' text and data sections into uint32_t. The program has now effectively been loaded into the machine's memory, which is represented as an array of uint32_t. From here, a for() loop is used to advance the program counter until an instruction of 0 is encountered. At which point, the program will end. Each iteration of the for() loop involves calling get_instruction(), which will return the appropriate function pointer for the MIPS instruction encoded. This pointer gets assigned to a generic execute() function which is called with the raw instruction and memory pointer.

Testing

The file "test.txt" contains a non-exhaustive test of the MIPS simulator. In total, there are 15 instructions. The instructions are decoded below in the order they are executed. Each instruction will be presented as the integer value, the type, and the operation. Press "ENTER" to step through each cycle.

1) 16416, R, $8 = $0 + $0
2) 537141248, I, $4 = $0 + 8192
3) 537206824, I, $5 = $0 + 8232
4) -1935212544, I, $7 = mem[(8232 + 0)/4] ($5 holds 8232, so mem[2058]), given the 8kB text section, this index will land us at the eleventh line of the data section since the text section ends at index 2047. The value there is 10, or 0xa)
5) 567148545, I, $14 = $14 + 1
6) 280704, R, $9 = $4 << 2
7) 8998944, R, $10 = $4 + $9
8) -1387986944, I, mem[($10)/4] = $5
9) 554172417, I, $8 = $8 + 1
10) 25714720 , R, $12 = $12 + 8
11) 17127466, R, $11 = ($8 < $5)

12) 359530553, I, if($11 != $14) $pc = $pc + (57 << 2)
13) 8415264, R, $13 = $4 + $0
14) 1007616007, I, $15 = (7 << 16)
15) 27748384, R, $13 = $13 + $7

Additional Notes

Due to time constraints, the program was not extensively tested. I cannot ensure every instruction behaves as expected but have included a small file called "test.txt" which contains some basic instructions for operation. Additionally, after every fetch, decode, execute cycle, the program will pause, to continue, press "ENTER" to fetch, decode, and execute the next instruction. After each cycle, the contents of the registers will be printed and a breakdown of the instruction will be displayed.

References

1) http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html
2) https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Mips/format.html
3) http://alumni.cs.ucr.edu/~vladimir/cs161/mips.html
4) https://en.wikibooks.org/wiki/MIPS_Assembly/Instruction_Formats#R_Format
5) http://homepage.cs.uiowa.edu/~ghosh/1-24-06.pdf
6) http://www.skidmore.edu/~pvonk/mips/lessons/registers.html