CSCI 2500 final project: MIPS CPU simulator

March 24, 2014

1 Introduction

Your assignment is to finish the implementation of a provided MIPS-1 CPU simulator. It will function much like SPIM but simpler; there is no debug functionality, memory view, register view, etc. It also will take compiled machine code, rather than assembly-language source, as input.

The goal of this assignment is to test your understanding of both C programming and the MIPS instruction set architecture by writing a C program that pretends to be a MIPS CPU. Given a 32-bit instruction and the current state of the machine (memory, pc, registers) you will need to update the current system state appropriately, as well as performing I/O operations.

2 Inputs

2.1 Introduction

The input to your program is a MIPS executable file in ELF format generated by gcc. Note that the "gcc" command on the class VM is a normal C compiler which produces outputs for the x86 instruction set used by the class VM. We will also provide a "mipsel-linux-gnu-gcc" compiler which produces outputs for the MIPS instruction set. This is known as a *cross-compiler*: the compiler runs on one CPU architecture (in this case x86) but produces output for another (in this case MIPS). Use of a cross-compiler is necessary for this project since we do not have a computer with an actual MIPS CPU available.

In case you are wondering the "el" in "mipsel" means "little-endian". Most MIPS processors can run in either big- or little-endian mode; since x86 is little-endian we chose to use this mode to make instruction decoding simpler.

2.2 ELF format

The ELF file format is the standard Linux executable file format and can contain machine-language instructions for any CPU supported by Linux. Any program you compile on an Linux system, including the class VM (which is an x86 system), will produce an output in ELF format. The ELF format is also used for .o files and shared

libraries however these cannot be executed directly since they don't have a main() function.

An ELF file is divided into one or more *segments* which specify a region of memory required by the program to execute. Each segment is defined by a *program header* in the executable. The program header specifies the starting address of the segment, the length of the segment, and optionally values to be placed in the segment. (Some segments may contain data such as uninitialized global variables, in which case the program header will specify addresses but not contents). The ELF format also contains additional data used for linking object files or shared libraries which is not necessary for this assignment.

We have provided a simple ELF loader in main.c which reads a MIPS ELF executable from disk, finds the table of program headers, and loads their contents into a linked list in memory.

More information about the ELF format is available online (just google "ELF format") if you are curious, however since the ELF loader in the simulator is already written for you detailed understanding of the file structure is not necessary for the assignment.

2.3 Testing

You may test your simulator with any language that can be cross-compiled to a MIPS ELF executable without depending on external shared libraries (since the simulator does not include a dynamic linker); sample makefiles and dummy code for C and assembly-language input will be provided.

A significant amount of extra credit will be provided if you are able to demonstrate successfully cross-compiling code written in another language (besides C/C++ or assembly) and running it. Please be warned that this is a fairly major undertaking since you will have to configure a cross-compiler (most likely building from source code with custom compile options) and most likely write some initialization code in assembly language to call class constructors etc.

3 Outputs

Your simulator should simulate all *syscall* instructions in the input program and perform I/O using C standard library functions as appropriate.

After the program has completed, create a file on disk called "simlog.txt" and print the following statistics to it:

- Number of instructions simulated
- Elapsed wall-clock time spent simulating. (You may use the *clock_gettime()* function for this purpose. Include time spent simulating normal instructions, but not waiting for input from the user.)
- (optionally) number of instructions of each type simulated

This file must be human-readable text but details of formatting are up to you.

4 Code structure

The bulk of your work for the simulator should be in the SimulateInstruction() and SimulateRTypeInstruction() functions in sim.c. You are free to change any other functions, or add more of your own, as necessary.

The provided FetchWordFromVirtualMemory() and StoreWordToVirtualMemory() functions read and write 32-bit words to a given virtual address within the simulated memory space; if the address is invalid they will print an error message and terminate the program. For the "lb" and "lh" instructions you will need to read the entire word and ignore some bytes; "sb" and "sh" will require a read-modify-write cycle.