CSC258H Week 9 Lab: Intro to Assembly

1 Introduction

It's the week many of you have been waiting for: we're leaving Quartus behind and starting something new – programming in assembly. We will be learning about a specific architecture called MIPS. MIPS is a RISC (reduced instruction set computer) family of processors from the early 1980's. The MIPS assembly language is well-known for being concise and logically structured, and because of their simplicity and energy efficiency, MIPS processors are still in use as embedded processors in devices like cell phones and portable game players.

Since the PCs in the lab are not MIPS machines, we will need to simulate one. The simulator we are using is called MARS. It's already installed in the CS labs, but you can also install it at home:

`http://courses.missouristate.edu/kenvollmar/mars/download.htm`

We'll be using MARS in all of the remaining labs, so take your time on this lab and ask your neighbors or the TAs questions whenever you see something you don't understand.

2 An Assembly Program

Every assembly program we write will look very similar. Here is an outline of a typical program:

```
.data
# Add your constant and variable declarations here.

.globl main
.text
main:
    # Add your program code here.
    li $v0, 10            # "Exit" is syscall 10. The next line will invoke a
                          # syscall based on the value in $v0.
    syscall               # Always end your program with an exit.
```

Every program is separated into two parts: a data section and a code section. The declaration *.data* indicates that beginning of the data section and the declaration *.text* indicates the start of code. You may add as many constant and global variable declarations as you like in the data section. Your code should be placed in the main block. The *main* **label** specifies where the program's main function starts (where MARS should start executing code). You may create other labels as you like; each label is used to name a specific line of code so that you can branch or jump to it. We'll use labels a lot when we implement control flow like branches, loops, and function calls.

The keyword *syscall* asks the operating system (or the simulator, in this case) to intervene to run a privileged instruction. You could also print a message on the screen (syscall #4) or get input from the user (syscall #5). The syscall in the example above exits the program (syscall #10). The constant ("immediate") *10* is loaded into register *$v0* before invoking the syscall; this tells the systems which syscall number to perform.

As you program in assembly, you may find it helpful to have a list of the syscalls and instructions at hand. We don't want to memorize all of the instructions and the registers they use. (Some instructions implicitly work with specific registers (e.g., *syscall* implicitly uses *$v0* and *$a0*).) Here is a MIPS reference card that has all the information that you will need while programming MIPS assembly.

`https://mcs.utm.utoronto.ca/~peters43/258/practicals/w9/mipsref.pdf`

And here is a more detailed description of MIPS assembly that explains, among other things, the function call conventions:

`http://pages.cs.wisc.edu/~larus/HP_AppA.pdf`

3 MARS Basics

Download the sample MIPS assembly program from the course webpage:

`https://mcs.utm.utoronto.ca/~peters43/258/practicals/w9/sample.s`

Load the file into MARS and then "Assemble" it (under the "Run" menu or on the toolbar). You cannot run your code until it assembles correctly. To test this, add a few random characters to the code in the "Edit" window, then try to assemble again. You will see an error message printed in the "MARS Messages" window at the bottom of the screen.

Now, take a moment to familiarize yourself with the layout of MARS. Fix your code, and then re-assemble it. You'll be taken to an "Execute" series of windows.

- The top left window contains the text segment – the code in your assembly program. That window provides you with the addresses of the various instructions, the machine-code value that is stored at that address, the assembly equivalent, and finally the line of code in the source file that generated that assembly instruction. You'll see that some lines of source code generate multiple lines of assembled code. In some cases, the original assembly instruction is a pseudoinstruction. In other cases, extra operations are required because of the simulated machine's architecture (hardware).
- The middle left window (above the message window) contains a window into memory. Originally, it is set to the data segment – the constants defined in your assembly program. However, the pulldown bar lets you select other segments including the heap or stack. Note that you can also scroll through memory and select how the values are interpreted. ASCII mode is particularly useful for checking strings.
- The pane on the right contains information about all of the registers in the machine. By default, the registers in the processor or displayed. "Coproc 1" (co-processor 1) is the floating point unit, and we will not use it in this course. "Coproc 2" supports the execution of interrupts, which we will investigate in the last week of lab.

Now, step through the code line by line. You can also execute the entire program, if you just wish to see the result, or step backward, if you wish to investigate a particular instruction more carefully. Take a few moments to familiarize yourself with how MARS uses highlighting to indicate the currently executing instruction and the registers that are being accessed.

As you are stepping through the program, you will stop at source line 22. That line of code executes a syscall that requests user input. Look in the "MARS Messages" window, and you'll see that a previous syscall printed a prompt. If you enter an integer, as requested, you will be able to continue execution.

Reset the simulator through the "Run" menu or by clicking on the "rewind" button in the toolbar. Now, step through the program line by line and try to answer the questions embedded in comments in the source file. (The questions are prefaced by the text "TODO". If you run into an instruction that doesn't behave like you expect, make sure to ask your neighbor or the TA for help.

4 Your First Program

Using sample.s as a guide, write an assembly program that prompts the user for a maximum string length, allocates space for a string of that length, and then prompts the user for a string and then prints the string. You will want to look at a list of available system calls to solve this problem. Pay special attention to sbrk (which allocates a chunk of memory much like malloc) and to the two parameters required by read_string. (**Note: The mips reference card has an incorrect argument for the sbrk command. It should be $a0, instead of $a.**)

Test your program using different inputs. Pay special attention to the memory you allocate to hold the string. When you type strings that are close the maximum length, what happens? What do you need to do to the max length to make sure that the end of line character and null terminator are stored appropriately? What happens if you type a string that is too long? How does MARS react?

5 Summary of TODOs

Below is a short summary of the steps to be completed for this lab:

1. Investigate MARS. Set it up on your own machine, if you would like.
2. Use MARS to step through the sample program and answer the "TODO" questions.
3. Write an assembly program that allocates space for a string, prompts the user for the string (placing it in that space), and then prints the string.
4. Investigate what occurs when you overrun your string buffer.

Evaluation (3 marks in total): 1 mark for answering a question your TA asks about one of the TODO lines, 1 mark for demonstrating that your string-reading assembly program works, and 1 mark for explaining what occurs when a buffer is overrun.

In next week's lab, we'll look at ways to implement basic control flow constructs like branches and loops.