CSCB58 Lab 8: Intro to Assembly

1 Introduction

From this week, we are leaving Logisim 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 computers we use today are not MIPS machines, we will need to simulate one. The simulator we are using is called MARS. You can download it from the following link. The filename is `Mars4_5.jar` and you can save on the desktop of your computer (a lab PC or your own).

`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 on Piazza or to your TAs questions whenever you see something you don't understand.

Note: Your will submit your solution code to Quercus before the start of your practical section, please read the "Summary of TODOs" section for more details.

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.

3 MIPS Reference

Before you become an assembly programming guru, you almost always need a reference card at hand, since some instructions implicitly work with specific registers (e.g., *syscall* implicitly uses *$v0* and *$a0*); and some register values are assigned to specific operations (like different system calls), which we don't want to memorize. Download the MIPS reference card from the link below — it has all the information that you need while programming MIPS assembly.

`https://cscb58f20.ml/docs/assembly_reference.pdf`

Below is the detailed documentation for MIPS assembly.

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

4 MARS Basics

Download the sample MIPS assembly program from the course website:

`https://cscb58f20.ml/labs/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 pseudo-instruction. 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 on Piazza so we can address your questions. We will put out a video over the weekend walking you through the `sample.s` file. However, we want you to attempt this yourself first.

5 Your First Programs

Write your first assembly program by modifying **sample.s**. You will create the two following assembly program files, with filenames being “`lab8a.s`” and “`lab8b.s`”.

1. `lab8a.s`: Prompt the user for A and B, and output results A + 42 and B - A.
2. `lab8b.s`: Prompt the user for A, B and C and output A + B + C.

This should be pretty simple to do after you fully understand all the lines in **sample.s**. In the practical section, show the TA that these programs work as expected.

6 Summary of TODOs

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

1. Read through the handout and get familiar with the procedure.
2. Go through **sample.s**, finish the TODO parts in the code and ask us for help on the course forum if you don’t understand any of the parts.
3. Implement the first variation: A + 42 and B - A. Save this program file as `lab8a.s`.
4. Implement the second variation: A + B + C. Save this program file as `lab8b.s`.
5. Before the start of your practical session, submit your `lab8a.s` and `lab8b.s` to Quercus. (Don’t worry, if you have any issues and get them resolved before the end of the practical and show your TA, we will mark you accordingly.)

6. In the practical section, demonstrate both your programs to your TA.

Evaluation (5 marks in total):

1. 2 mark for making A+42/B-A work
2. 2 mark for making A+B+C work
3. 1 mark for answering any questions your TA has.