CMPS 3620 Comp Arch I: Assembly Language

Spring 2021

Lab 1 – Introduction to MIPS

Introduction

Goals

- Understand the structure of MIPS programs
- Understand how to use SPM
- Write a MIPS program to sum two integers and display the result

Resources

- Examples in the lab folder
- MIPS quick guide (Reference: MIP reference data PDF document on Blackboard)
- Multi-platform SPIM simulator: http://sourceforge.net/projects/spimsimulator/files/

Read this Entire Document Before Doing the Lab: It's always a good idea to read the lab before starting it. You will have a greater insight into the scope of the work you will be doing to at what points to save your work.

If You Do Not Finish: These are not intended to be extra homework. They are designed for you to complete in the two and a half hour period on lab day. Labs are submitted by turning in the source files on Blackboard. If you do not finish before the lab ends you have until the start of the next lab period. The instructor will give you feedback while you complete the lab, so it is in your best interest to complete the labs during the lab period. Even if you finish during the lab you must upload your code to Blackboard.

Background: We will use a MIPS simulator (SPIM) in this course. We must use a simulator because you are most likely using a machine with an x86-64 microprocessor. SPIM is installed on Sleipnir but you can also download SPIM for your home machine (see Resources above). SPIM is not an assembler; i.e., SPIM does not create an executable file in machine code. SPIM translates MIPS line by line, similar to how an interpreter runs code. Because of this, SPIM is a good learning environment.

Downloading Files Needed: Typically, you will be given some files to start the lab. Put them into your local directory on Odin.

Hello.S

You will learn MIPS well in the coming weeks. This lab introduces you to a few of the basics. The code below from hello.s is a good place to start:

```
# hello.s
# a sample MIPS program to demonstrate MIPS basics
# Usage: $ spim -f hello.s

        .data           # data segment begins here
stuff:  .asciiz  "Hello World!\n"

        .text           # code segment begins here
main:
        la $a0, stuff
        li $v0, 4       # 4 is syscall to print a string
        syscall         # execute the call

        li $a0, 10      # 10 is ascii value for linefeed
        li $v0, 11      # 11 is syscall to print char
        syscall

        li $v0, 10      # 10 is system call to exit
        syscall         # execute the call
```

There is no strict convention on the extension of a MIPS source code. You might see .s or .asm or .nasm or something else. We will use .s in this class since s is the first letter of both short and sweet.

The source code above has two sections, a data section, and a text section. Everything is in one section or the other.

Comments are prefixed with a hash # everything to the end of the line past the hash is ignored by the assembler. The data section holds literals such as the "Hello World!" string. The text section contains instructions and assembler directives. Instructions are in the form of an opcode (operation code) followed by zero, one, two or three arguments for that opcode. The amount of whitespace between opcodes and arguments is irrelevant. An argument for a MIPS instruction is either a register, a literal, or an address. An address is marked by a label. E.g.,stuff. A literal is a number, e.g. 4, 10 or 11.

Registers and Memory: A register is special storage on the microprocessor. This is opposed to variables stored in memory, which must be defined in the .data section. For more on this, read: http://cnx.org/content/m29470/1.1/.

In short, data is stored in either a register or in memory. Registers are very fast but limited in number and restricted to word-length. Memory is abundant for our purposes, is not necessarily restricted to word length, but is slower. Additionally, in MIPS, most operations operate on registers. If you want to add two values in memory you would need to issue commands to bring them from memory, place them in registers, then carry out your operation. In MIPS, a register can hold either a value or an address. The meaning of the data in the register is determined by its

context in the instruction. It is up to you to keep track of whether a register holds an address or a value.

Load Address: Some MIPS instructions are not part of the MIPS instruction set architecture (the hardware) but are provided by the assembler to make the programmer's life a little easier. These instructions are called pseudo-instructions. Each pseudo-instruction is a macro for 2 or more instructions from the ISA. Load address, la, is a MIPS instruction that stuffs the address of a label into a register; e.g.:

la $a0, format #load address of label format into $a0

The two arguments for la specify the movement of data from memory (the source) to a register (the target). The movement is right to left. Be careful not to confuse the two. The target comes before the source (but only for load operations). Thus a LOAD always reads like an assignment statement in a high-level language.

Store: In the coming weeks we will also cover STORE operations. Unlike 80x86 (which we will cover in week 8), MIPS is a LOAD/STORE architecture. This means that computations cannot be done directly to values in memory. Data must always be read from memory into a register (LOAD) or written to memory from a register (STORE). This makes instructions simple since the first argument in a load instruction is always the target register. The first argument in a store is the source register. While the flow of data in a LOAD instruction is logically right to left (load the address of "Hello World!\n" into register a0), the flow of data in a STORE instructions is logically left to right:

sw $a0, $t0 # store value in $a0 to address in $t0

Load Immediate The load immediate, li, is also a pseudo-instruction that loads the numeric constant 10 (source) into the target register $v0.

syscall The syscall instruction executes a system call identified by the integer value in register $v0. In this case '10' is the exit call. Different values placed in $v0 will cause different behavior. syscall can print strings, read strings, print integers, exit, etc.

Run hello.s

Understanding the above concepts will be enough to complete this lab. If you desire more details, browse through A-43 through A-81 of appendix-A link on our home page, for a table of syscall codes, assembler directives and a reference of MIPS R2000 Assembly Language instructions. Get SPIM if you don't already have it and run hello.s. Make sure you understand what each line does in the code. You do not need to check this off with the instructor.

Modifying the Program

Add Two Integers: Your job is to create an assembly file called sum.s that takes two hard-coded integer values (hard-coded means the values are not read in from the keyboard but literal constants that you code into the program), adds the two values together together and displays the result. The guts of the code (the header and the syscall to exit) will resemble hello.s so you can start by modifying the program. Before making any changes review hello.s until you understand every line of code. You will need code to load the values of two integers into registers and sum the two integers together. You will use the li, addu and move instructions shown below. Note that source register(s) are the right operands and the destination register is the left operand in these instructions:

```
li     dest, integer      # load integer into dest register
addu   dest, src1, src2   # add values in src1 and src2 registers
                          # and store result in dest register
move   dest, src          # copy value in src reg to dest reg

You can also use fewer instructions if you want to use the add immediate
instruction:

addi  dest, src, integer  # store result of value in src + integer
                          # in dest
```

Which Registers to Use? Since all 32 MIPS registers are general purpose, you can generally get away with using any registers that you want. However, there are conventions that good coders should use. For example, temporary values (such as the integers you are adding together) should be loaded into temporary registers $t0 - $t7. Function arguments are loaded into $a0 - $a3. The results of computations are loaded into the value registers $v0 - $v1. To make sure you have added the values correctly you need to display the result. Display it like this (your integers should differ from mine):

```
The sum of 5 and 9 is 14.
```

Input/Output is provided by the assembler via system calls. In a high-level language such as C++ you can sometimes get away with letting the compiler figure out what data type you are displaying; e.g.,

```
int x = 5;
cout << x << endl;
char stuff[20] = "hello\n";
cout << stuff << endl;
```

will display an integer first and then a character string next. In assembly you need to specify precisely what type of data you wish the bit string in the register to be displayed as. In this lab you will make calls to print strings (printable text) and integers. The syscall service you need to print the integers is print int. To use this call load the value you want to display into $a0. Then load the value 1 into $v0. Then make the print int call.

The syscall service you need to print the text strings is the same as noted in hello.s. You will need to create strings with labels in the data segment for each string you want to print. For example:

```
.data
str1:     .asciiz "The sum of"

Then make a syscall to print each string in the correct order. For example,

la $a0, string1         # load the address of the string into a0
li $v0, 4               # 4 is syscall to print a string
syscall                 # make the call
```

Hint: Since you need to print three integers, you may need to use three registers for these values. Move each integer as you print it into $a0.

What to turn in

Upload your lab program sum.s to Blackboard.