CS447 Lab #2

Introduction

By this point, you should know how to use MARS to implement very simple MIPS programs, and also to inspect the contents of the register file and memory. In Lab #2, you will learn how to perform more sophisticated operations related to memory and control flow.

Note: This Lab must be submitted online by 11:59 pm tonight. Please see the instructions on the final page.

Part 1: Use MIPS to selectively manipulate individual bytes of memory.

Create a new file in MARS, and save it with the name **Lab2a.asm**. For this and all subsequent lab files, please add your name, recitation section, and the file name at the top of the source code as comments.

As a warm up, let's recreate the hello_world program from Lab #1. You can either download the hello_world.asm file from the course website, or type in the code pasted below:

```
.data
        msg: .asciiz "Hello, world!"
.text
        # Print the message
        la $a0, msg # Put the message address in a0
        li $v0, 4   # Put the "print string" syscall code in v0
        syscall     # Call it!
        # Terminate the program
        li $v0, 10  # Put the "exit" syscall code in v0
        syscall     # Call it!
```

Assemble the program, and switch to the **Execute** tab. As in Lab #1, we'll focus on the initial cells of the memory array, where the **Hello, world!** ASCII string has been stored. In particular, you will use MIPS instructions to selectively modify individual bytes of it. For the next steps, the goal is to replace ***Hello, world!*** with ***Hello: World.\n***, solely by changing the values of four bytes of memory (note: \n is the newline character). Ultimately, we'll want to see the **syscall** print this modified string to the **Run I/O** window.

If you already have a good sense of how to do this, please feel free to try whatever specific instructions you like – just make sure that you're directly modifying bytes in memory (as opposed to, for example, changing the string in the **.data** segment). If you'd like a tip on how to get started, proceed through the following guidelines for how you can change the ***w*** to a ***W***:

1. Determine the offset of the character **w** relative to the start of the ***Hello, world!*** string.
    - Stated differently, determine the index that would be assigned to this character if the string were stored in a character array in a language such as Java or C (which start array indices at 0). Don't forget the whitespace!
    - This offset value maps directly onto the number of bytes that intervene between the address that **la** will return for **msg** and the memory location of the character **w.**

2. Place in a temporary register the ASCII value for 'W'.
    - You can do this using **li**, as in the previous lab, or if you prefer to use a basic (not pseudo) instruction, then you can use **addi** (add immediate) and add the desired value to the contents of register **$zero**.
    - If you know the decimal or hexadecimal representation of 'W', you can use that in the immediate field. For hex, be sure to precede the value with '0x'.
    - MARS will also accept 'W' as an immediate value, and do the conversion to a number for you.

3. Once you have stored the value for 'W' in a temporary register, use **sb** (store byte) to store that value in the memory location that currently contains 'w'.
    1. This command can be constructed using the following formula:
    sb $register-with-value-to-store, offset-number($register-with-base-address-of-string)
    2. Be sure to include the parentheses. Only the underlined parts need to be replaced. Remember, the **la** command retrieved the base address, so that instruction should tell you where the base address information is located.

The above steps can be repeated to change , to :, ! to ., and the NUL character after the string to \n.

To test your program, make one more modification: After the **syscall** that is used to print the string, add a second syscall. This will make it easy to tell if the end-of-line character was placed correctly.

When your program is done, your output should look like this:

Hello: World.
Hello: World.

-- program is finished running --

Part 2: Contruct loops using labels, branching, and comparison instructions

If you have experience programming in other languages, you have doubtlessly had experience with different kinds of loops, such as *for* loops and *while* loops. In MIPS, you will construct your own loops out of more basic instructions. In this exercise, emphasis will be placed on the instructions listed in the following table:

purpose	instruction	name
branching	beq	Branch On Equal
branching	bne	Branch On Not Equal
comparison	slt	Set Less Than
comparsion	slti	Set Less Than Immediate

Since some of these instructions may have not yet been covered in lecture, they are explained in some detail below. Additional information is also available in the MARS **Help**.

Create a new file in MARS, and save it with the name **Lab2b.asm**. For the first part of this exercise, just copy the code below (comments optional) to write a very basic loop. Note that the **.data** section can be omitted for this program:

```
# Lab2b.asm

.text

        # set $v0 to syscall code

        li $v0, 1


        # use a loop to print all digits 0-9 to the I/O window

        add $t0, $zero, $zero # initialize $t0 --> acts as counter

        addi $t1, $zero, 10   # initialize $t1 --> termination condition

        top:

        add $a0, $t0, $zero   # set syscall argument to the value of $t0

        syscall               # print to screen

        addi $t0, $t0, 1      # increment $t0

        bne $t0, $t1, top     # do not branch if $t0 = termination condition


        # Terminate the program
```

```
li $v0, 10  # Put the "exit" syscall code in v0
syscall     # Call it!
```

The output of the program should appear as below:

0123456789

-- program is finished running --

Once you have this working, you can review this explanation of how the loop works:

The goal is to mimic the behavior of a very simple *for* loop which prints the integers 0 to 9 to the screen (without line breaks). In Java, the analogue of the above MIPS instructions might look something like this:

```
for (int i = 0; i != 10; i++) {
    System.out.print(i);
}
```

In other words, we will iterate through this loop, incrementing the integer *i* by 1 each time, and printing it out to the screen, and leave the loop when *i* reaches a value of 10. Now we can examine the individual lines of the loop in MIPS:

add $t0, $zero, $zero # initialize $t0 --> acts as counter

Here we are initializing **$t0** to the value 0 (which is good practice, even if **$t0** may already be set to 0 when you open MARS). This is similar to our initialization **int i = 0** in the *for* loop.

addi $t1, $zero, 10 # initialize $t1 --> termination condition

Here we are setting **$t1** to the value of *i* that should trigger termination of the loop.

top:

This is a **label**, and it directly precedes the instructions that form the body of the loop.

add $a0, $t0, $zero # set syscall argument to the value of $t0

syscall # print to screen

These lines are the analogue of **System.out.print(i).**

addi $t0, $t0, 1 # increment $t0

This performs the same role as **i++.**

bne $t0, $t1, top # do not branch if $t0 = termination condition

This "Branch on Not Equal" line implements the check against the termination condition (in other words, does **i != 10** evaluate to **true** or **false**). The instruction tests the contents of **$t0** and **$t1** for equality. If **$t0** and **$t1** are not equal, then MARS will return ("branch") to **top** and run the body of the loop again. If **$t0** and **$t1** are equal, then the branch will not be performed, and MARS will just proceed to the subsequent lines of code – in this case, the **syscall** to gracefully exit the program.

MIPS also includes a **Branch On Equal (beq)** condition, which does exactly what the name implies: It branches when the register values are equal, rather than when they are not.

If you have experience with Java (or C/C++), you may have noticed something unconventional about the *for* loop header line that we are attempting to mimic:

for (int i = 0; i != 10; i++)

The loop is terminated when the termination condition (**i != 10**) evaluates to **false**. A loop of this style is more conventionally written with a **<** operator:

for (int i = 0; i < 10; i++)

or, equivalently, with a **≤** operator:

for (int i = 0; i ≤ 9; i++)

This distinction in loop headers becomes important if we decide that we instead want to increment by 3 each time, rather than 1. If we tried to combine this increment with the strict **i != 10** termination condition (as illustrated below):

for (int i=0; i !=10; i=i+3)

then the program would run indefinitely, because **i** would never be equal to 10. We could avoid this problem by using a **<** operator in the termination condition. So how could we do something similar in MIPS?

Fortunately, MIPS offers instructions to evaluate these kinds of inequalities. The **Set Less Than** operation will **set** the value of a destination register to **1** (think of this as a Boolean **true**) if the value in the first source register is less than the value in the second source register, and to **0** otherwise. For example, if **$s0** = 1 and **$s2 =** 2, then **slt $s1, $s0, $s2** would update the **$s1** register to contain a value of **1**. Frequently, the result of an **slt** instruction will be used by a subsequent **beq** or **bne** operation to decide whether or not to branch. In this case, one operand of **beq/bne** will be the destination register of the **slt** instruction (which stores the 0/1 outcome of the comparison) and the other operand should be a register that contains either 0 or 1 (depending on whether we want **beq/bne** to compare the **slt** outcome to 'true' or 'false').

The **Set Less Than Immediate** instruction (**slti**) performs the same operation as **slt**, except the second source register is replaced with an immediate value. More information on other MIPS comparison and branching instructions/pseudoinstructions may be found in the **Help**.

To complete **Lab2b.asm**, you should modify the loop, so it now performs as specified below:

- Initializes a counter at 0
- Increments the counter value **by 3** on each iteration
- Prints the value to the screen
- Terminates the loop once the counter value is **no longer less than 10.**

This can be accomplished by using a combination of the branching and comparison operators that we have discussed. → **A word of caution:** Since a bug in your program could easily create an inifinite loop, you might want to initially try testing your program using the **single-step** button that we discussed in **Lab #1.** If you run the full program, and do get caught in an infinite loop, you may have to abort MARS using whatever method is standard for your OS (for example, Alt-Ctrl-Delete to pull up Task Manager in Windows).

When your program is finished, the output should look like this:

0369

-- program is finished running --

Submission:

If you finish during recitation, please notify the TA, so that your credit for this assignment can be confirmed right away.

Regardless of whether you finish during recitation or not, you must submit your work online. Add **Lab2a.asm** and **Lab2b.asm** to a zip file named **YourPittUserName_lab2.zip** (other extensions, such as .rar, are also OK) and submit this file via the appropriate link in Courseweb (see Course Documents/Week 3 Lab). For this and future recitations, the lab must be submitted by **11:59 pm** the **same day** on which the recitation is held.

Optional:

A survey has been posted to gather opinions regarding the kinds of lab exercises that might be most helpful. This survey is anonymous and optional. A link is provided in the recitation slides.