COIS 2300H Lab 2

This lab goes with lecture 2 on Assembly programming

The first Lab you learned how to set up and install MIPS

This week is about learning to do some programming. This lab is in part taken from <https://www.engr.colostate.edu/~sudeep/teaching/resources_452/cs50-asm.pdf> which has a quite a long tutorial. Don’t bother trying to read it all, it’s from 1994. But I’m citing my sources.

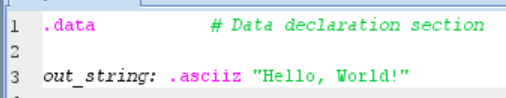
You will need to type out some of this code – I’ve put in screenshots rather than copy/pasteable text so you make mistakes and see what the typos are.   
  
Begin by creating a new program in Mars

1. **Start with comments**  
   # Name

# Username

# Whatever else you need as a comment  
  
Now that you’ve spent the last hour typing out code that doesn’t do anything, lets move on to something useful!

1. MIPS programs are broken into a few sections, the first is data, that section is defined by typing



1. The next thing we need is to define the start of the actual assembly program (this is a directive, we’ll talk about them later in the lab)

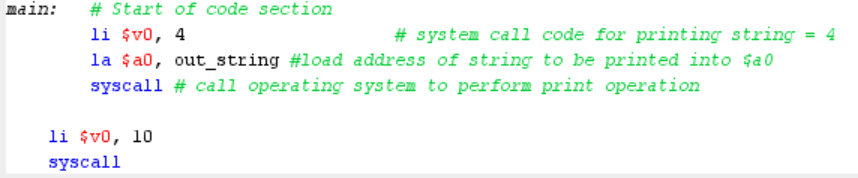
Put this before main:

.text # Assembly language instructions

You need that .text line to tell the compiler where the actual program itself starts.

1. **Hello World!**

that says “main” which looks like it didn’t copy well from the screenshot



1. Change the line from step 2 to be “\nHello, World!\n” and compare. Now you know how \n works!
2. **Save** your program as helloworld.asm somewhere. You’ll need it again later.
3. Ok now you can see what hello world does, lets work from a blank program, and we’ll do some **basic Registers and Addition**. Create a new file  
     
   If we want to do any operation we need to look up (or memorize) the basic commands. For our purposes lets stick to an easy one to see how it works.

We want to add two numbers together, and put the result in a register ($t0)

So what we want is something like

add $t0, $t1, $t2   
  
Which should add the contents of t1 and t2, and put the result in t0

B**efore** we can do that, we need to put some numbers in $t1 and $t2

1. Let’s **set our Register values** to something, these need to be above the add instruction. You may need a couple of other things as well

.text

main:   
li $t1, 1

li $t2, 2  
  
add $t0, $t1, $t2   
Feel free to experiment with other simple functions like subtract and multiply and other variables.  
  
add can also be used as, for example, add $t0, $t1, 2 # $t0 = $t1+ 2  
You can see after assembling the program that the values of t0, t1 and t2 have been set in the registers.

1. **Labels**   
     
   main: is an example of labels, and they serve to tell MARS or SPIM or whatever some stuff about what the text in your program is going to mean.   
     
   Labels are a bit like function names. These are similar to directives (.data and .text) but we’ll get to those in a couple of sections.   
   Labels cannot have the same name as MIPS instructions for obvious reasons. Try and see what happens!

(This is so you learn to recognize error messages)

1. **Syscalls**  
     
   MIPS emulators are generally smart enough to stop when they reach the end of their instructions. But that’s because they are special development environments. Real assembly programs can do all sorts of weird stuff, like continuing to try and read memory after the program is done as though it is a series of instructions. You can imagine this tends to not work and is a massive security and operating system design challenge.   
     
   in MIPS there is a feature called syscall, which simulates the transfer of control to the host operating system, rather than to an OS it transfers control to the program but it behaves the way it should

Recalling your macro from lab one, you should add that to your program now  
  
Your program will look something like  
  
main: # SPIM starts execution at main.

li $t1, 1 # load 1 into $t1.

add $t0, $t1, 2 # compute the sum of $t1 and 2, and put it into $t0.

li $v0, 10 # syscall code 10 is for exit.

syscall # make the syscall.  
Note how syscall must have a system of codes that mean something, and I’ve told you 10 is the particular one for exit but you have no real idea what any others are.

1. Now we’re going **to read in a number** from the keyboard  
     
   main:

li $v0, 5  
syscall   
move $t0, $v0  
  
This loads the syscall that is for reading an integer (code #5) into a variable, reads in that value from the keyboard to that same location, then moves the data out.

1. **Printing** your output

This is another syscall, 1. Which prints the contents of $a0

move $a0, $t2 # move the number to print into $a0.

li $v0, 1 # load syscall print\_int into $v0.

syscall # make the syscall.

1. **Directives**

To start making your code readable you should break it into sections. This is good practice in other languages as well, but we need explicit terms for the compiler to know what is happening. The relevant directives for us are .data, where you usually define the data of a program in advance, and .text where the body of your text goes

Save your previous program and reload your Hello World Example  
  
it should be something like

.text

main:

la $a0, hello\_msg # load the addr of hello\_msg into $a0.

li $v0, 4 # 4 is the print\_string syscall.

syscall # do the syscall.

li $v0, 10 # 10 is the exit syscall.

syscall # do the syscall.

# Data for the program:

.data

hello\_msg: .asciiz "Hello World\n"  
  
(You can copy paste this since it’s for illustration purposes) Replace the hello\_msg with   
  
hello\_msg:

.byte 0x48 # hex for ASCII "H"

.byte 0x65 # hex for ASCII "e"

.byte 0x6C # hex for ASCII "l"

.byte 0x6C # hex for ASCII "l"

.byte 0x6F # hex for ASCII "o"

.byte 0xA # hex for ASCII newline

.byte 0x0 # hex for ASCII NUL

There now you see how individual ascii works

1. **Conditionals** (in C/C#/C++/Java you would think of these as if statements)  
     
   While there are a few possible instructions let’s stick with a simple one, bgt. This instruction takes 3 arguments, two numbers and a label. If the first number is greater than the second, the program jumps to the label.

# If $t0 > $t1, branch to t0\_bigger,

bgt $t0, $t1, t0\_bigger

move $t2, $t1 # otherwise, copy $t1 into $t2.

b endif # and then branch to endif

t0\_bigger:

move $t2, $t0 # copy $t0 into $t2

endif:

Using what you have in this tutorial write a program that reads in two numbers from the user and then prints out the larger one. (Only have this execute once, ignore error checking etc.)

1. **Loops**

A loop is a piece of code that is executed multiple times. A loop needs some sort of counter or condition that should be modified each step through the loop so it doesn’t try and run forever. (E.g. for a loop that runs 10 times, you have, in C, for (int i=0; i< 10; i++){ some code} each time through the loop i is incremented by 1, and when it reaches 10 the loop doesn’t execute and execution continues after the loop)  
  
 Loops, like conditionals are implemented with branch instructions. The difference is in how you use the branching logic. If you branch ‘up’ the code, to somewhere that will eventually get back to the branch instruction you have a loop. If you have a branch that goes down the code it’s conditional.

|  |  |
| --- | --- |
| Loop | Conditional |
| Startofloop:  some code  Some code  Some code  Branch – if true go back to to Startofloop  If branch not taken execute code here | Branch if true go to C1  code that executes if not true  you need to be careful here that you don’t accidentally run both cases of the condition  C1:  Code that executes if the branch was true. |

The most useful possible branch instructions are beq (branch if equal), bne (branch if not equal) and bgt (branch if greater than), there are some others too which aren’t relevant right now.

Modify the program from the previous step to use a loop that keeps running until the two user inputs are equal  
  
here is a simple loop (this is more of a while loop than a for loop)  
  
loop:

beq $t2, $t3, endloop # if $t2 == $t3, exit

endloop:   
 done # as in your done macro from lab1

Now modify your program so that it asks the user for two numbers, and prints off the larger one in a loop, make the loop executes 5 times. (Use a temporary variable, add 1 to it each time, and then if it equals 5 times endloop)

1. Recursion will be in a future lab.