How to use MARS to program in TAL for EEC 70

-By Michael Sticlaru

ABSTRACT

The motivation for this guide is to help students who would like to use very useful tool in the creation of their assembly code for EEC 70 at UC Davis. This tool can help understanding of the assembly language, as well as give a visual aid and reference for concepts discussed in the class room and book. Traditionally, students use a text editor to generate lines of code for use in the SPIM/SAL simulator located on the ECE machines, or using other windows based simulators.

The problem with this approach is there is little to no feedback given to the student when writing the code. When loading the code into the simulator, feedback on any errors is difficult to discern or understand. This can create a problem for students who are new to the language, and frustration when trying to determine the cause of an error.

The use of the MARS MIPS simulator alleviates this problem by use of a power interactive development environment (IDE) that can help students understand the code they are writing.

INRODUCTION

MARS – What is it?

MARS (MIPS Assembler and Runtime Simulator)

An IDE for MIPS Assembly Language Programming

MARS is a lightweight interactive development environment (IDE) for programming in MIPS assembly language, intended for educational-level use [1]

Features

- GUI with point-and-click control and integrated editor
- Easily editable register and memory values, similar to a spreadsheet
- Display values in hexadecimal or decimal
- Command line mode for instructors to test and evaluate many programs easily
- Floating point registers, coprocessor1 and coprocessor2. *Standard tool: bit-level view and edit of 32-bit floating point registers (screenshot).*
- Variable-speed single-step execution
- "Tool" utility for MIPS control of simulated devices. *Standard tool: Cache performance analysis tool (screenshot).*
- Single-step backwards

It can be downloaded from the program's website along with more information about the program.

DOWNLOAD

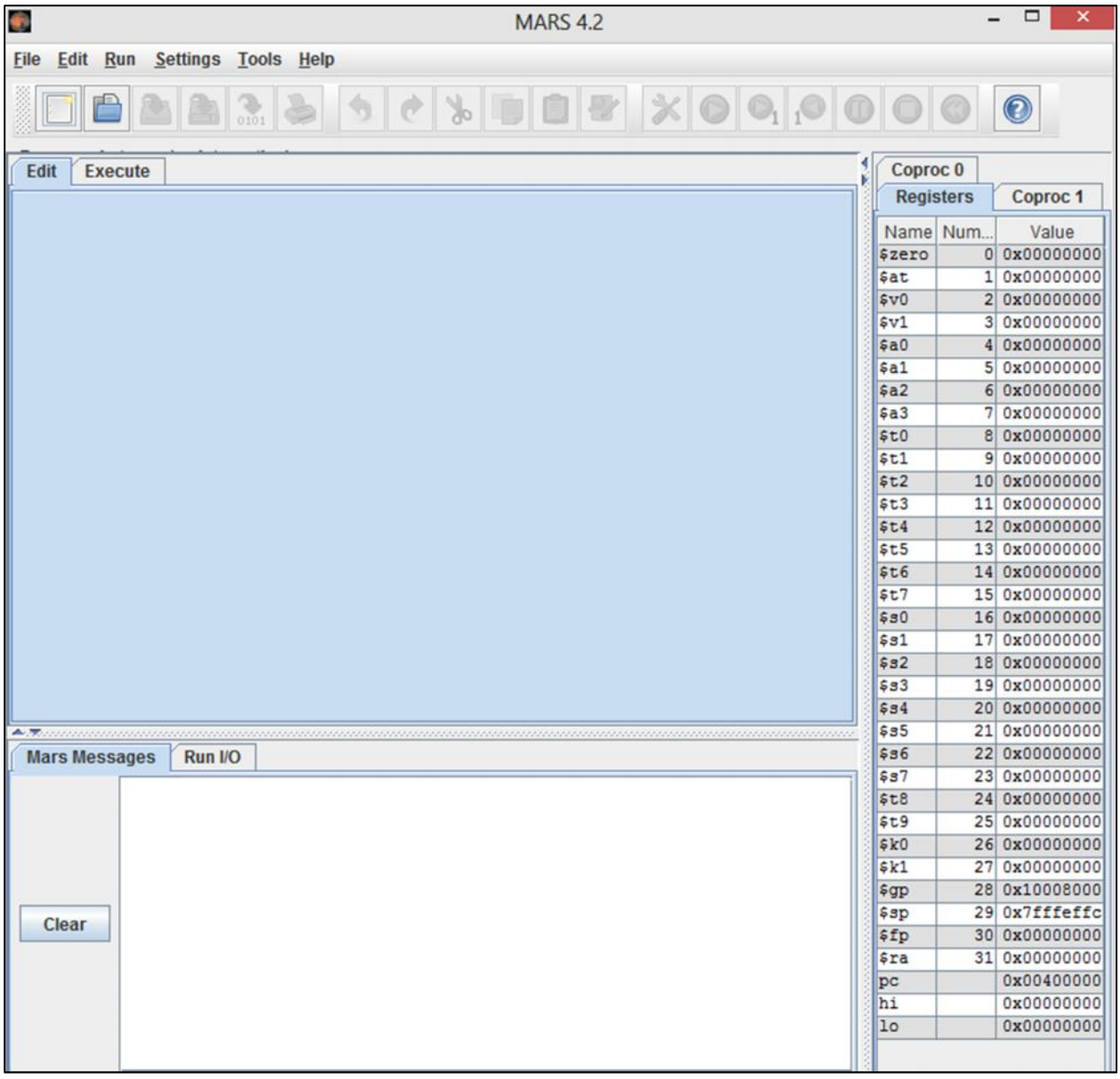
[1] http://courses.missouristate.edu/kenvollmar/mars/

HOW TO USE IT

I will leave the download and installation up to the student to accomplish. After installation run the program executable, and launch MARS.

You should now be looking at a screen like this.

INITIAL SCREEN



This is the main program layout. Instantly, you notice the Register stack on the right. It follows standard the MIPS convention that is discussed in chapter 8 and 9, of the text, and in the table on page 244.

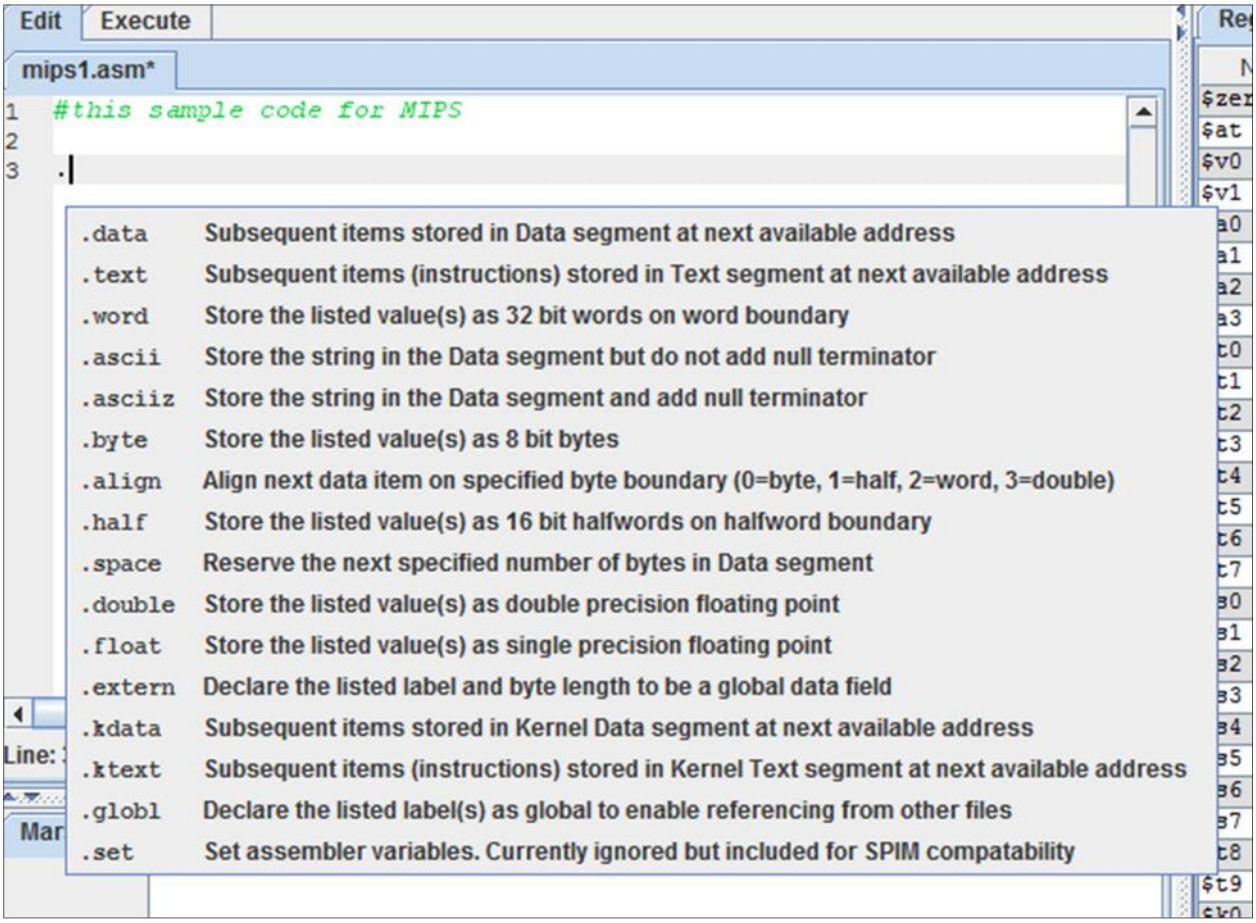
You will also notice the large blue area, this is where the code is typed out, and then the message window on the bottom, this is where print outs and interaction created by the program are presented.

WRITING CODE

Mars will open all files with .s extension as well as .asm extension. It will also save with both extensions. EEC70 requires the .s extension, so let us use that for our purposes.

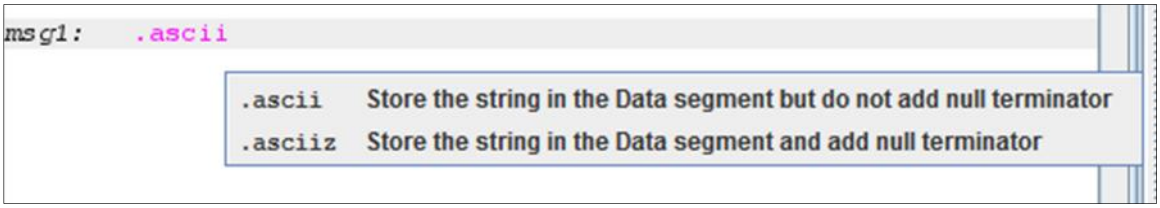
To open a file, click File -> new, or open if you are opening existing code.

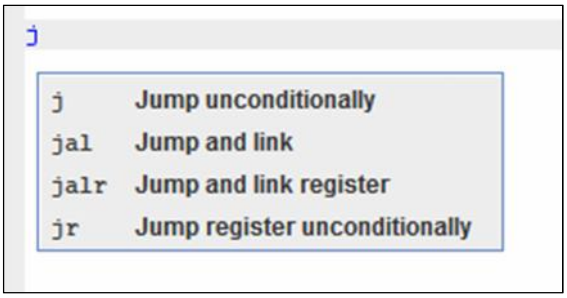
MARS is an IDE, this means that when you type code, there is interaction between you and the program that gives you information about the code you are writing, as you write it. For example, when I start my code, I will start with ".data", lets see what happens. Directly after pressing the period key to start .data, I get a list of all the possible commands I can use not all apply to our class, but a lot of them do.



As I continue to spell out data, I get the description of data, what it does, and how it works.

Here are a few more descriptions of commands, You can play with this on your own.





There is also command format descriptions, for example, if I want to use the Boolean" branch if not equal", I can start typing it to see how to use it.

As you can start to see, using MARS to write code is very beneficial and can help solidify the concepts taught in the class

MARS also have formatting that it uses in the text editor to identify commands, statements, comments, and so on, here is an example.

```
1  # this sample code for MIPS
2
3  .data
4
5  msg1:    .asciiz "this is my message"
6
7  .text
8
9  __start:
10
11         la $t0,msg1      # this is a comment
12         lw $t1,0($t0)
13
14         addi $t1,48      # commands are blue
15                          # vairables red
16                          # constants black.
17
```

REGISTER DISPLAY

Other than being a powerful editor for assembly, MARS also gives information about the registers. This is helpful when trying to understand the PC counter, Stack Pointer, or other register values that are used.

The registers display on the right side of the screen gives information about the register name, its number, and the value contained in that register at any point during the program execution.

This is a valuable tool that runs in real time as the program is being executed. We can see that the value of the stack pointer in this example, and also the value of the PC counter before its offset.

When a program is running, the values will that the program uses or creates will populate this list. If the program is run in real time, then the results shown will be the final results of the program.

However, MARS supports single step instruction execution. Using the single step, we can watch the values change, as our code is executed. This can help not only understand HOW an instruction works, but can also allow the student to determine where their code is not working.

For example, if I were writing a subroutine to extract digits from a number, I might expect that the command

Rem $t0,$t5,10 will extract the last digit from the integer stored in $t5 and store is in $t0, in fact that is exactly what we would see.

Registers

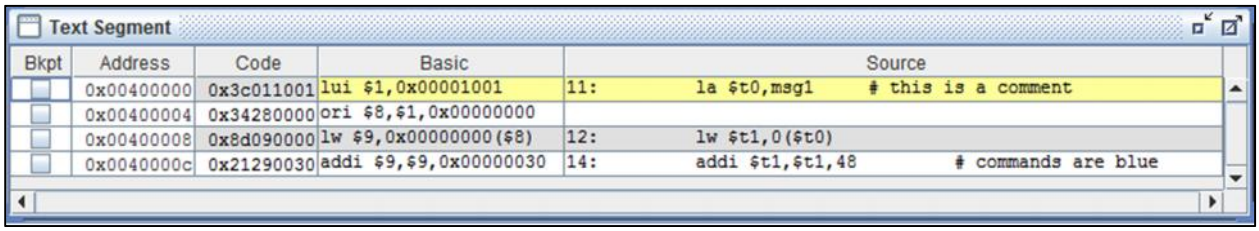
Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

COMPILING THE CODE

Now that the code we wanted to write has been written, we can compile it and run it. First we begin by compiling, or “assembling” the code. We can do this by pressing f3, or going to “run -> assemble” MARS will then attempt to assemble our code, and tell us about any errors we may have made. Here are a few examples.

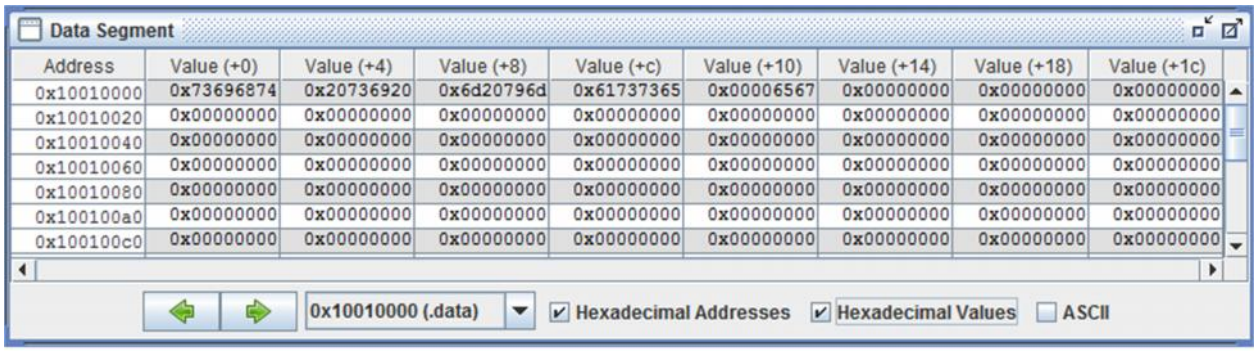
```
line 12 column 2: "lw": Too few or incorrectly formatted operands. Expected: lw $t1,-100($t2)
line 14 column 2: "addi": Too few or incorrectly formatted operands. Expected: addi $t1,$t2,-100
```

Once the errors are corrected, and the program will assemble and then a new screen is presented to us.



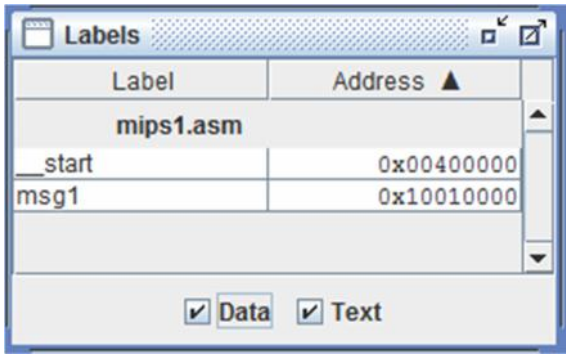
This portion of the next screen shows the code that we wrote on the right, the True assembly version of the code we wrote, the HEX representation of our code, and then finally, the address that each instruction is located at.

Next, we get to see the data table created by our code. Displayed by memory address and value.



This is the data segment of our code, it shows the variables we defined m in this case, 0x100 10000 is the beginning of the label “msg1” and goes from 0(msg1) through address 10(msg1) where +10 is the offset of the address segment.

Next we have the address of each label we have defined in our program, this will hold the address of all labels that we used in our program. You can use this table to calculate the offsets for jump calls.



Notice that the label for msg1 corresponds to the appropriate data segment, starting at address 0x1001 0000

Now let’s run our program and see what happens.

RUNNING THE PROGRAM

I am going to run the program in step by step mode. By clicking the green arrow with the '1' on it. This will step forward 1 instruction at a time.

Coproc 1 | Coproc 0

Registers

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x10010000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400008
hi		0x00000000
lo		0x00000000

I stepped forward 4 times, For the sake of saving pages, This is the result of running the steps. We can see that $t0 now contains the value that we expected, 0x1001 0000, and $t1 has the result of the addition. Also notice that $a1 also has the value of 0x1001 0000. This is because register 1 is used by the operating system as discussed in chapter 8. The reason it is used follows from the TAL load instruction

	Basic	Source		
01	lui $1,0x00001001	11:	la $t0,msg1	# this is a comment
00	ori $8,$1,0x00000000			
00	lw $9,0x00000000($8)	12:	lw $t1,0($t0)	
30	addi $9,$9,0x00000030	14:	addi $t1,$t1,48	# commands are blue

So, instruction lui loads the upper address into $1, then ori loads the lower address plus the upper address into $t0. I have stopped executing on line 12, so the lw command has not yet executed.

t	Address	Code	Ba
	0x00400000	0x3c011001	lui $1,0x00
	0x00400004	0x34280000	ori $8,$1,0
	0x00400008	0x8d090000	lw $9,0x000
	0x0040000c	0x21290030	addi $9,$9,

This is a continuation to the left of the screen shot above. Notice that we are stopped on line 12: if we look to the left, we see that the address of that line is 0x 0040 0008. Now, if we look at the register stack, we see that PC has the value of 0x 0040 0008.

Using these tools, it is easy to see what is going on in your code, why its going on, and how to manipulate your code to become more efficient.

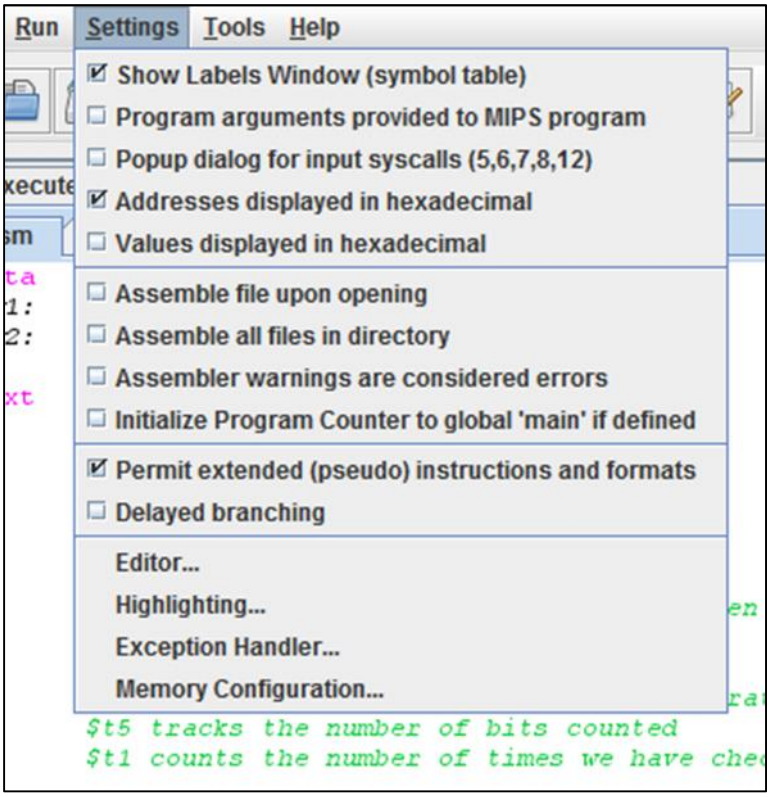
TIPS

Because MARS uses only MAL and TAL commands (mostly TAL) you will need to use the proper syntax for things like "get" and "put".

In MAL, the commands "putc" and "getc" are used, but there are no such commands in TAL. As such, the MARS simulator will not use them.

So how can you print?

Chapter 10 introduces TAL, and all instructions presented in the text work in the MARS simulator. The table on page 262 and explanation of system calls will help you tremendously.



In the settings tab there are several things of interest.

You can see the options for “values displayed in hexadecimal” . Try toggling this on and off when you are looking at your registers. It is helpful when trying to understand how characters are inputted through the keyboard. Using “getc” system call, you can write a simple subroutine that will take input from the user, you can watch in real time as the value of the register value changes. Toggling the values button, you can change from hex to decimal, and see the ascii values.

EXAMPLE CODE

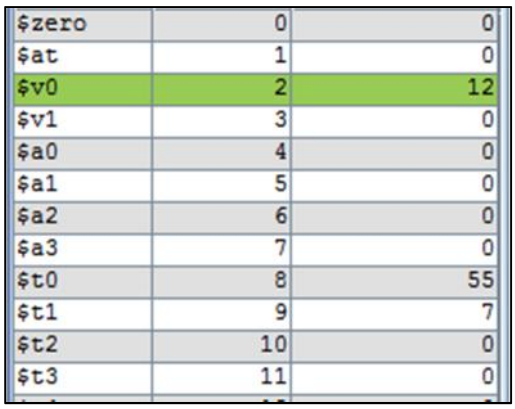
In this example, the program will ask for an input from the user, the ascii value of the key is then placed into register $2, copied to register $t0, and then the

```
1  # sample get input function
2  .data
3  .text
4  loop:   addi    $2,$0,12        # "getc" synthasized in TAL the
5                  syscall         # syscall asks for the key, the
6                                  # value of the key is placed in $v0
7          add     $t0,$0,$v0      # this places the ascii value into $t0
8          addi    $t1,$t0,-48     # $t1 holds the non biased value
9          j       loop
```

ascii bias is removed. Let's take a look at the registers to see what is going on.

Pressing the "7" on my keyboard resulted in the ascii value 55 is placed into register $t0, 7 is placed into $t1, and $v0 holds the value of 12.

Looking at this, it is clear how the system call works. The value in $v0 used to tell the system what to get, then the results are stored according to how the code is written, in this case $t0, and $t1. We also get immediate feedback on the values we type into the program. (note: I used run speed at 6 inst/sec)



$zero	0	0
$at	1	0
$v0	2	12
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	55
$t1	9	7
$t2	10	0
$t3	11	0

Conslusion

Using the MARS simulator provides some great benefits for students who wish to learn how assembly is written, and the underlying concepts that are discussed in the class. It provides a powerful tool for debugging, and "watching" how code is working. Students can see the true assembly code that is generated by their MAL commands.

Not all MAL commands are used "get, put, done, start.. etc" however, this is not a limitation, as synthesizing TAL commands are part of the learning experience. While the current solution being used is functional, it is often frustrating for students who cannot get feedback on how their code is working. There is no easy way to debug code in a text editor. At the very least, MARS can be used to generate the code for use on the SPIM/SAL simulators in use now. I have verified that code written with does compile and run perfectly on the SPIM/SAL sim.