CSC 204 *Computer Architecture and Organization* Lab 7

PROJECT Intro to MIPS assembler

Objective To examine various topics concerning the assembly language.

PROJECT DESCRIPTION

To get started with some of the basics, uses and programming with an assembler system thru the MARS IDE.

Information About this Project

Let's start with downloading and installing the MARS 4.5 assembler system.

Then on to executing a Fibonacci assembler program.

Also introduced will be some important assembler commands, syntax and logic!

Steps to Complete this Project

STEP 1 **Download and install MARS**

Go to the following URL to download the MARS written in java so any OS can use it.

http://courses.missouristate.edu/KenVollmar/mars/download.htm

Click the 

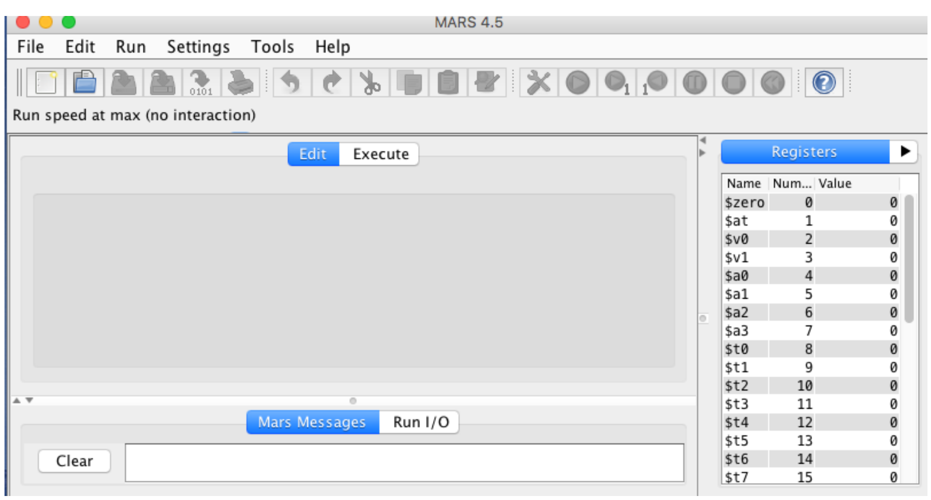
button to get the latest version 4.5.

You will notice the file downloaded is **Mars4_5.jar** and is ready for execution!

STEP 2 **MARS IDE**

Open up your assembler IDE by clicking on your Mars4_5.jar file.

You should see a window relative to the snapshot that follows. Take a look over the IDE and some of the features along the way to familiarize yourself with many of the great features an assembler system IDE can offer showing the true low level capabilities of a program at compile time and via any debug modalities!



PROJECT Intro to MIPS assembler

STEP 3 Creating a program in MARS!

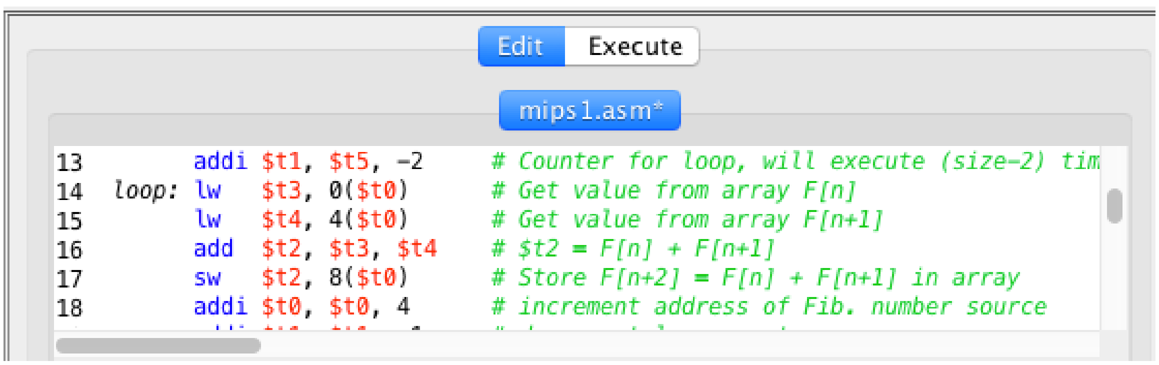
No other way to learn but to jump right into things.

From the link in Step 1 click on the Fibonacci.asm link as shown below.

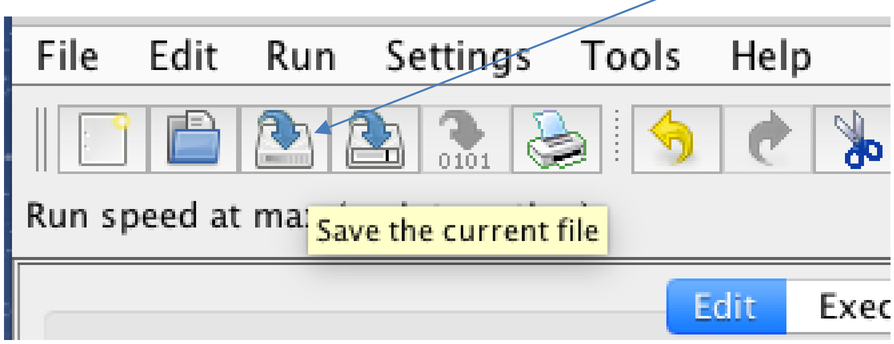
Sample MIPS assembly program to run under MARS Fibonacci.asm

Next in MARS go to your menu and start a new assembly (.asm) file by clicking on File > New.

Now paste your fibonacci code into your editor (top most inner window) as shown below.



Save your program with a formidable name by clicking on the save icon and naming your file **fibonacci.asm**.



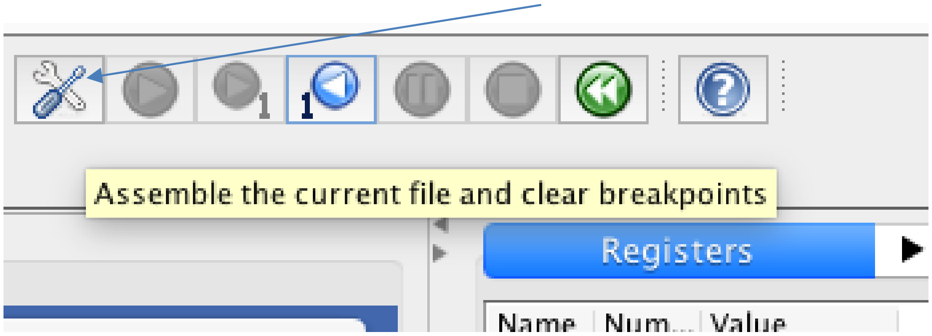
PROJECT Intro to MIPS assembler

STEP 4 **Running a program in MARS!**

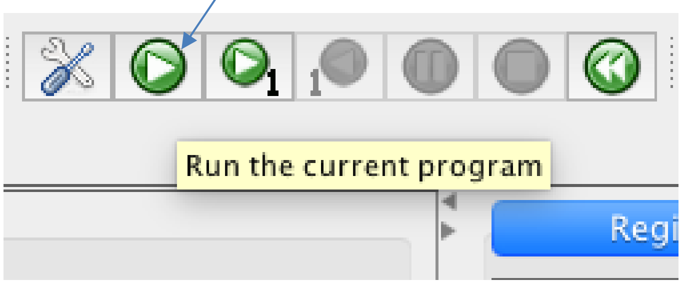
Running a program follows to steps in MARS namely by running the assembler to compile the program then actually executing it.

To assemble or compile the program merely go your menu and click on

Run > Assemble or by clicking on the  Assemble icon in your toolbar area.



Then if there are no errors either go to Run > Go or click the  in your toolbar area to execute your program.



STEP 5 **Snapshot your code for credit.**

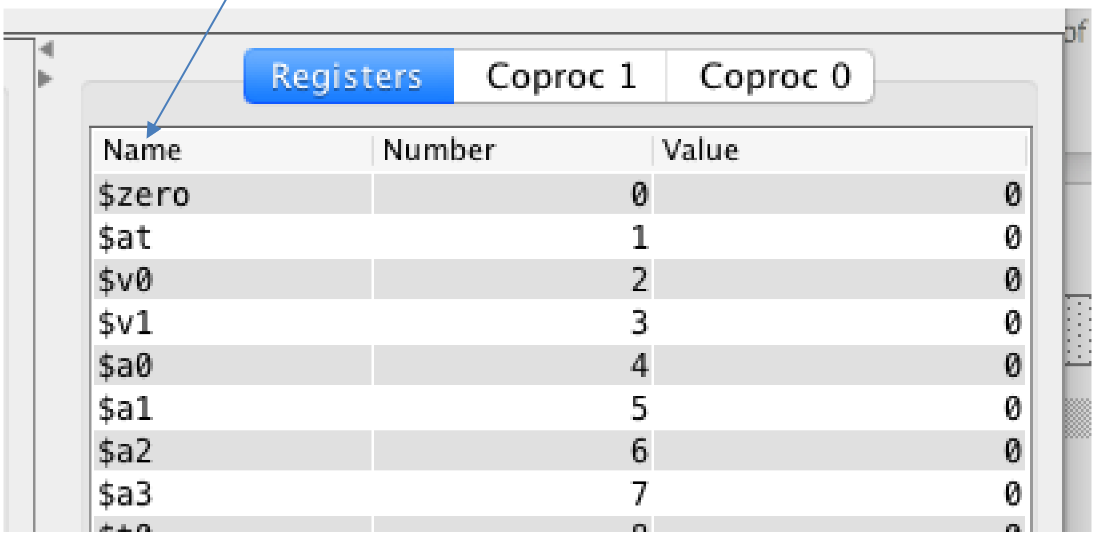
Take an initial snapshot of your first run in your output console at the bottom window of your IDE and paste it into a Word doc file for credit. Some modification of the code will come in Step 9!

STEP 6 **Checking over the MARS IDE**

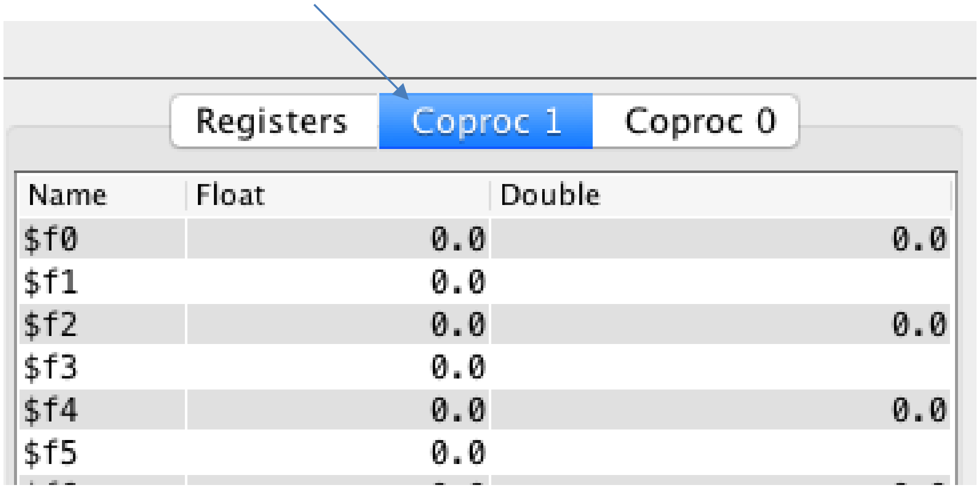
A quick overview of the MARS IDE worth noting at this point follows as noted by the snapshots.

PROJECT Intro to MIPS assembler

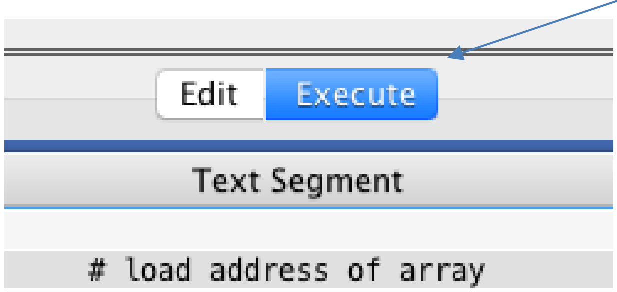
The memory **Registers/Coprocessors (Coprocs)** are to the right of the IDE. Registers are a way to hold and refer to data and thus assist in the processing of data. Check over page 274 of Chapter 4 of your text for intel on Registers and their naming conventions.



Coprocessors, (Coproc1 for ex.) display needed registers for floats.

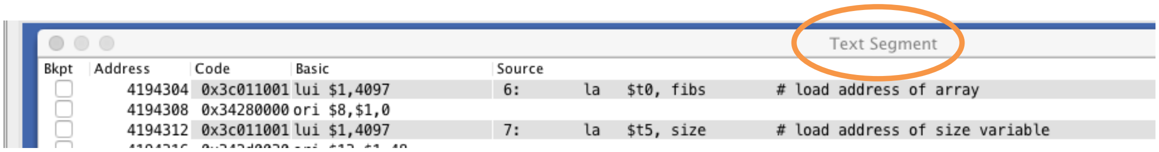


The following segments can be seen when in runtime or execution mode.

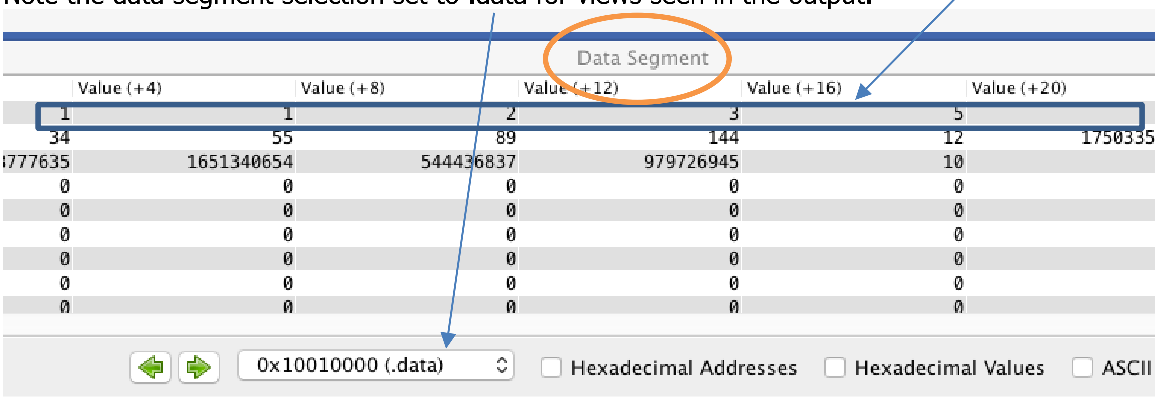


PROJECT Intro to MIPS assembler

Text Segment, sitting towards the top of the IDE, provides for a bird's eye view of the source relative to addresses and breakpoints (Bkpt) that can be set for debug modes.

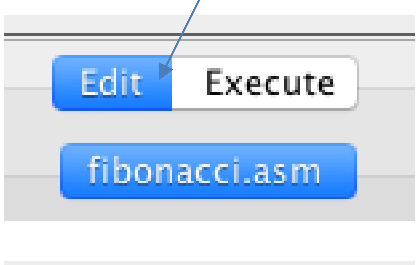


The **Data Segment** sitting beneath the Text Segment can depict various data values at memory addresses throughout a run. Nice for tracing data values! Notice fibonacci sequence results. Note the data segment selection set to .data for views seen in the output.



STEP 7 **Familiarizing yourself with some MIPS syntax and code logic.**

When in Edit mode, coding in MIPS involves 2 major sections namely **.data** and **.text.**



.data section holds data sets in variables and .text will be the instructions your program needs.

So for example – a typical Hello World app may include the following code

```
.data
   myMessage:  .asciiz "Hello World... \n"   # hi

.text
    li $v0, 4
    la $a0, myMessage
   syscall
```

PROJECT **Intro to MIPS assembler**

Hello World breakdown:

Note **myMessage** string variable holds a message in ascii character format or text that will be output with the help of the syscall command which executes the code block.

The **li** and **la** help load instructions and data sets into RAM memory registers for specific data types or values. This is the backbone behavior of assembler.

The **li** stands for load immediate (as perhaps you'll see with the built in intellisense during coding) and **$v0, 4** is the naming *convention* used (**$v0** part) to allow a *string* data value (**4** part) to be assigned.

la stands for the load address and will execute what value it is pointing to via the *argument* set by myMessage into registry **$a0**.

You'll notice a lot of the lines of code follow a similar structure when assigning values in the registry.

Ex.

Command / **Register name** / **Value**

li	**$v0,**	**4**

Things differ a bit though when performing arithmetic assignments. For example when you add values you can just use the **add** command to point to locations in memory (always 2 arguments) and store the result into a dedicated register which (always 1 argument).

Ex. add command

Command / **Single Register storage** / **Adding 2 registers**

add	**$t2,**	**$t3, $t4**

STEP 8 **Fibonacci.asm code walk through**

Study carefully, line by line, the fibonacci code as it is replete with comments that walk thru every line of code! Notice the code has an array to contain the fibonacci values and a loop to add the fibonacci sequence of values that eventually are printed out.

Further notice use of routines or labels defined and called throughout the code namely **loop**, **print** and **out**. Routines as you can see can be called numerous times throughout the program and thus executed. Jump commands such as those shown in code namely **jal** and **jr** are common to call routines.

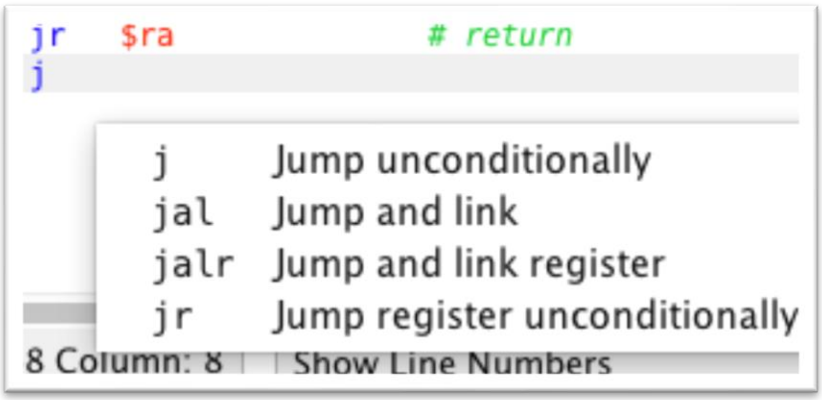
PROJECT **Intro to MIPS assembler**

MIPS common conventions to deal with functions are:

- $a0-$a3 (registers 4 to 7) arguments 1-4 of a function.

- $v0-$v1 (registers 2 and 3) results of a function.

Snapshot of intellisense of jump commands



Fibonacci source code break down*:*

As you will see code in assembler is like any language and is followed in a linear fashion. In a nutshell the code is setting up an array to store fibonacci values tabulated inside a loop up to array size – 2 then ultimately printed out.

The following shows a breakdown of some of the syntax and logic in the fibonacci code.

Line 3

```
fibs: .word   1: 12       #"array" of 12 words to contain fib values
```

.word allows for up to 32 bits of storage (the max for MIPS) for numeric types to be contained.

lw short for load word, allows for information to read from memory as lines 16 & 17 depict for the start of the loop routine as shown below.

```
loop: lw   $t3, 0($t0)      # Get value from array F[n]
      lw   $t4, 4($t0)      # Get value from array F[n+1]
```

Note storage for MIPS data varies either by using exclusive registries denoted by $s (for saved value) or $t registries (for temp values). **sw** short for store word is used to execute storage choices. Example of various lines which are storing initial fibonacci data values follows:

```
sw   $t2, 0($t0)       # F[0] = 1
sw   $t2, 4($t0)       # F[1] = F[0] = 1
```

PROJECT Intro to MIPS assembler

Finally you will note in the *loop* and *out* routines the command **bgtz** which stands for "branch if greater than 0" which allows for a condition to keep occuring unto the result is 0. A counter stored in **$t1** is checked for each call to the various routines to see if it will be necessary to continue in the code block.

Code of counter actions for loop logic follows highlighted in bold:

```
      addi $t1, $t5, -2      # Counter for loop, will execute (size-2) times
loop: lw   $t3, 0($t0)       # Get value from array F[n]
      lw   $t4, 4($t0)       # Get value from array F[n+1]
      add  $t2, $t3, $t4     # $t2 = F[n] + F[n+1]
      sw   $t2, 8($t0)       # Store F[n+2] = F[n] + F[n+1] in array
      addi $t0, $t0, 4       # increment address of Fib. number source
      addi $t1, $t1, -1      # decrement loop counter
      bgtz $t1, loop         # repeat if not finished yet.
      la   $a0, fibs         # first argument for print (array)
      add  $a1, $zero, $t5   # second argument for print (size)
      jal  print             # call print routine.
      ::
```

STEP 9 Modify Fibonacci.asm code

Modify the Fibonacci code by creating a **new line** for each Fibonacci value limited though to the first 10 values which should be 1 1 2 3 5 8 13 21 34 55.

Snapshot your resulting output display and paste it into your Word doc again for credit. Also include a copy of your completed source code into your Word doc file following your snapshots.