

PROJECT An Introduction to the 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 through the MARS IDE.

Information About this Project

You will be using the MIPS Assembler and Runtime Simulator (MARS) , which serves as an integrated development environment (IDE) for writing MIPS assembly.

Let us 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**

Visit the following URL to download the MARS written in Java so any OS can use it.

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

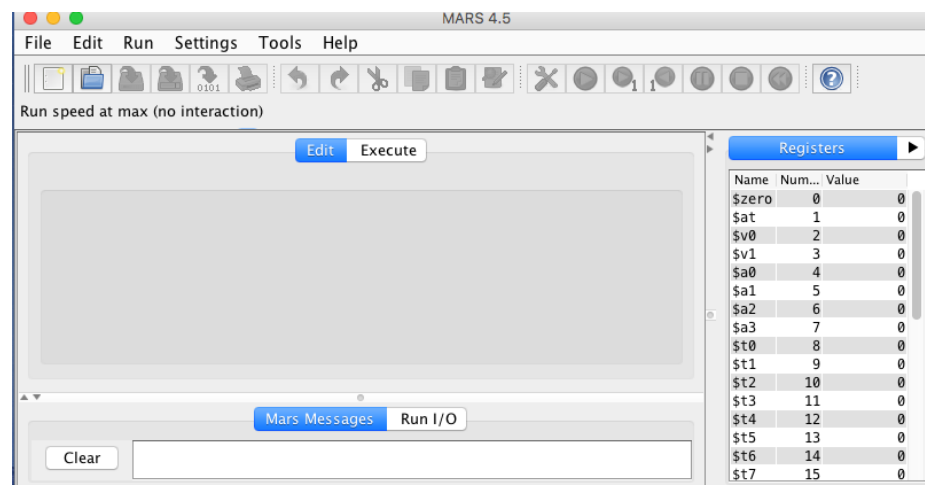
Click the **Download MARS** button to get the latest version 4.5.

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

STEP 2 The MARS IDE

Open 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 An Introduction to the MIPS Assembler**STEP 3** Creating a Program in MARS!

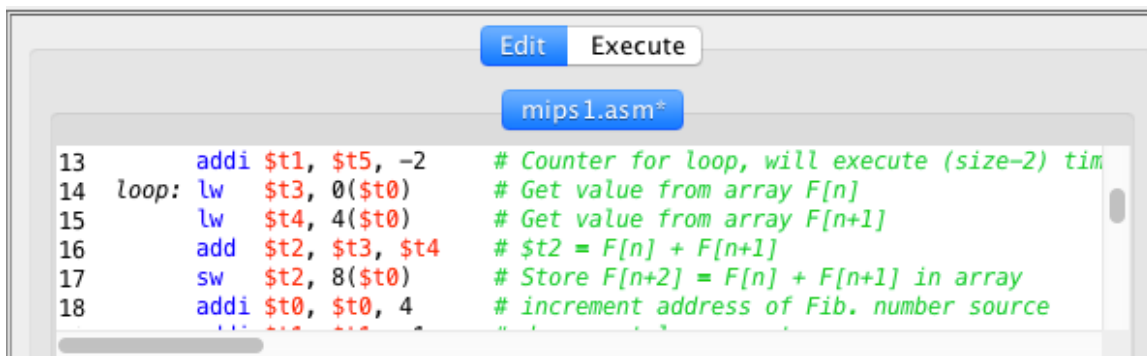
No other way to learn but to jump right into things.

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] .

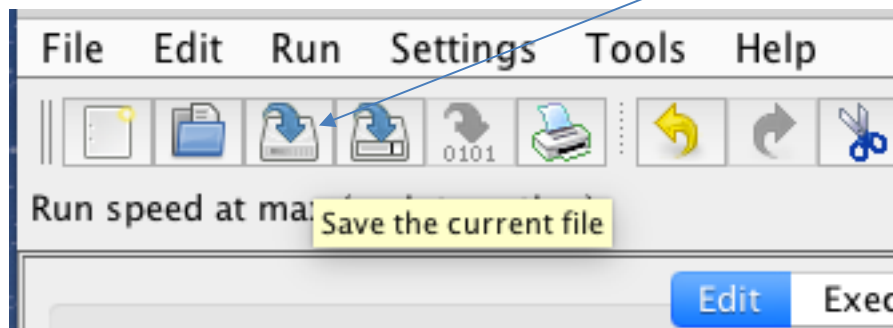
Now paste your Fibonacci code into your editor (top most inner window) as shown below.



The screenshot shows the MARS MIPS Assembler editor window. At the top, there are 'Edit' and 'Execute' buttons. Below them is a tab labeled 'mips1.asm*'. The main editor area contains the following assembly code:

```
13      addi $t1, $t5, -2      # Counter for loop, will execute (size-2) times
14 loop: lw   $t3, 0($t0)      # Get value from array F[n]
15      lw   $t4, 4($t0)      # Get value from array F[n+1]
16      add  $t2, $t3, $t4     # $t2 = F[n] + F[n+1]
17      sw   $t2, 8($t0)      # Store F[n+2] = F[n] + F[n+1] in array
18      addi $t0, $t0, 4      # increment address of Fib. number source
```


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

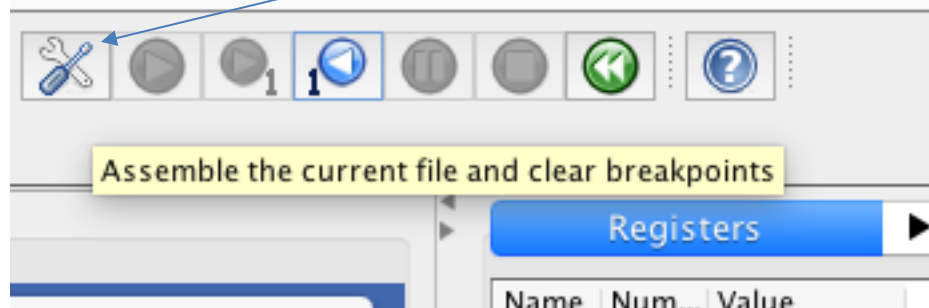



PROJECT An Introduction to the MIPS Assembler**STEP 4 Running a Program in MARS!**

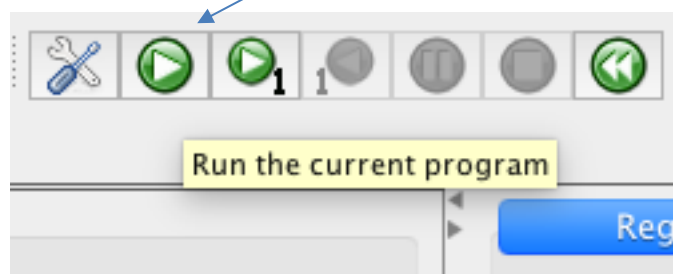
Running a program follows to steps in MARS namely by running the assembler to compile the program then actually executing it.

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**

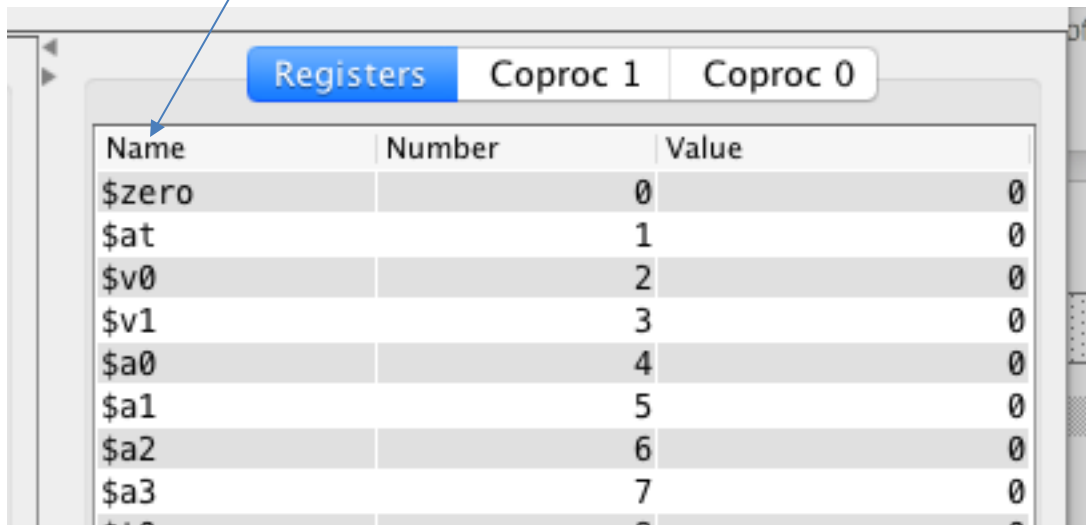
Take an initial snapshot of your first run in your output console at the bottom window of your IDE and paste it into a Word document file for credit. Some modification of the code will come in **Step 9** !

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.

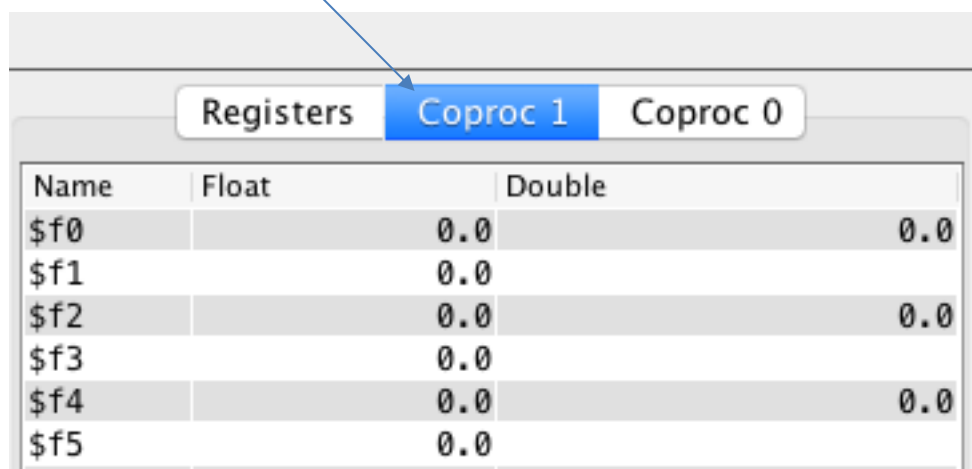
PROJECT An Introduction to the MIPS Assembler

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.



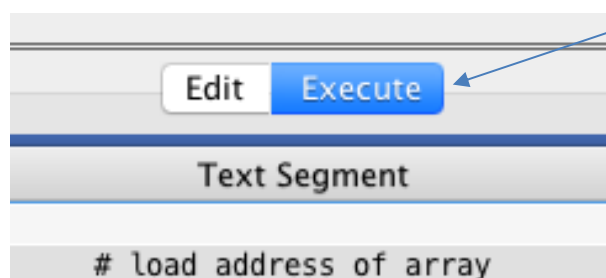
Name	Number	Value
\$zero	0	0
\$at	1	0
\$v0	2	0
\$v1	3	0
\$a0	4	0
\$a1	5	0
\$a2	6	0
\$a3	7	0

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



Name	Float	Double
\$f0	0.0	0.0
\$f1	0.0	
\$f2	0.0	0.0
\$f3	0.0	
\$f4	0.0	0.0
\$f5	0.0	

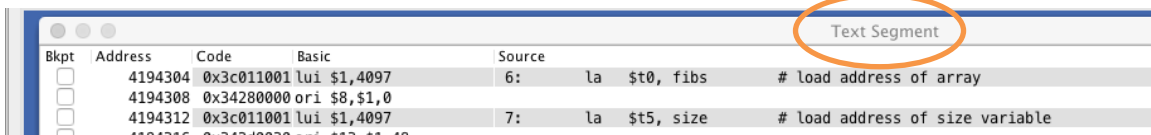
The following segments can be seen when in runtime or execution mode.



Text Segment
load address of array

PROJECT An Introduction to the 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.

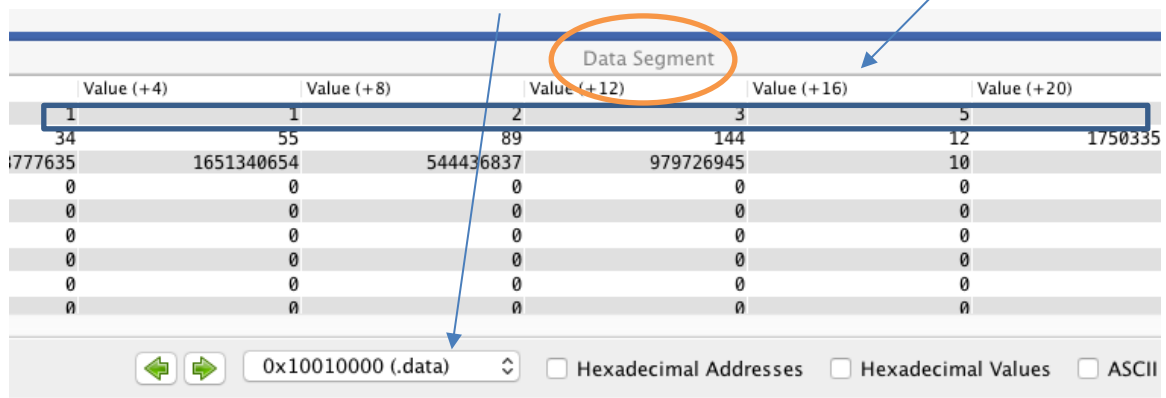


Bkpt	Address	Code	Basic	Source
<input type="checkbox"/>	4194304	0x3c011001 lui \$1,4097		6: la \$t0, fibs # load address of array
<input type="checkbox"/>	4194308	0x34280000 ori \$8,\$1,0		
<input type="checkbox"/>	4194312	0x3c011001 lui \$1,4097		7: la \$t5, size # load address of size variable

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 also the Fibonacci sequence results.

Note the data segment selection set to .data for views seen in the output.

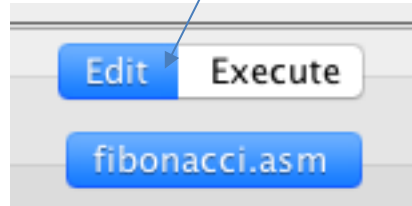


Data Segment					
Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	
1	1	2	3	5	
34	55	89	144	12	1750335
1777635	1651340654	544436837	979726945	10	
0	0	0	0	0	
0	0	0	0	0	
0	0	0	0	0	
0	0	0	0	0	
0	0	0	0	0	
0	0	0	0	0	
0	0	0	0	0	

0x10010000 (.data) ☐ Hexadecimal Addresses ☐ Hexadecimal Values ☐ ASCII

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 An Introduction to the 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 will notice a lot of the lines of code follow a similar structure when assigning values in the registry.

Ex.

Command / Register name / Value



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

**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.

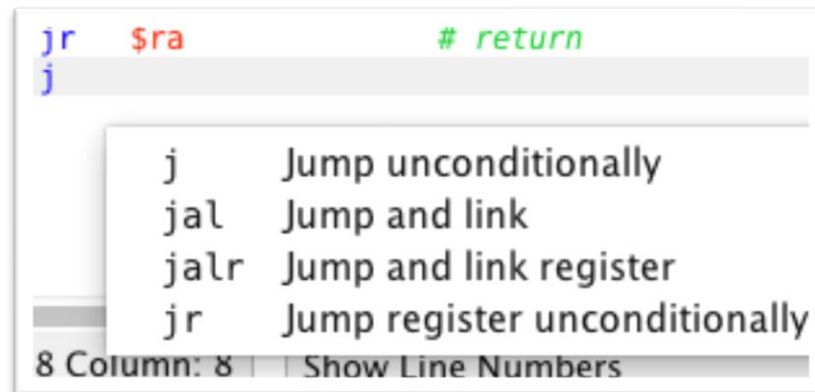
Further the notice use of routines or labels defined and called throughout the code namely **loop**, **print** and **out**. Routines as you can see can be called numerous times throughout the program and thus executed. Jump commands such as those shown in code namely **jal** and **jr** are common to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

PROJECT An Introduction to the 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 An Introduction to the 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 occurring 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 the 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.

STEP 10 Questions and Answers Concerning this Computer Laboratory Project

Open MS Word and, within a new document, place your responses to these questions. Submit your completed MS Word document for credit.

(1) (Assembly Language Commands)

Describe the function and purpose of each of these Assembly language commands.

li , la , lw

(2) (Assembly Language Statements)

Explain what is accomplished by these assembly language statements, taken from the **fibonacci.asm** file.

```

      .data
fibs: .word 0 : 12
size: .word 12

```


PROJECT An Introduction to the MIPS Assembler**(3) (Assembly Language Statements)**

Explain what is accomplished by this assembly language statement, taken from the **fibonacci.asm** file.

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

(4) (Assembly Language Statements)

Explain what is accomplished by this assembly language statement, taken from the **fibonacci.asm** file.

```
addi $t1, $t1, -1
```

(5) (The Fibonacci Sequence)

Often studied in Computer Science for its importance in recursive function design, the Fibonacci Sequence has these elements:

```
{ 0 , 1 , 1 , 2 , 3 , 5 , 8 , 13 , 21 , 44 , 55 , . . . }
```

Open this computational and intellectual Web site:

<https://www.wolframalpha.com/>

Enter and execute this statement.

Is 55 a Fibonacci number?

Also, enter and execute this statement.

Is 251 a Fibonacci number?

Now, test whether your own name generates a Fibonacci Number.

Write the individual characters that comprise both your first name and last name and total the number of letters, as shown in this example.

First Name

<i>letters</i>	S	A	M	M	Y		<i>count</i>	5
----------------	---	---	---	---	---	--	--------------	---

Last Name

<i>letters</i>	S	T	U	D	E	N	T		<i>count</i>	7
----------------	---	---	---	---	---	---	---	--	--------------	---

Concatenate the two count values and place a random single digit number on the right.

579

Now test whether the number you generate is or is not a Fibonacci Number. Place a screen snapshot of the result in your answers document.