

Introduction to MARS Simulator

(MIPS Assembler and Runtime Simulator)

1. Introduction

MARS is a Java based Integrated Development Environment (IDE) for MIPS assembly language programming. Prominent features of MARS include the ability to evaluate many instructions at once using the command line. Register and memory values can also be modified easily. It can also control the speed of execution and can represent data in decimal or hexadecimal formats.

A list of syscalls are provided at this [link](#).

The “green sheet” with MIPS instructions is [here](#).

2. Installation

- The MARS simulator can be downloaded from the Missouri state university [link](#)
- The MARS simulator webpage:
<http://courses.missouristate.edu/KenVollmar/mars/Help/MarsHelpIntro.html>
- As it is a jar file, the user should have the Java J2SE in the system to run the simulator
 - If the Mars4_5.jar icon looks like this, Java is installed:



- If not, download the latest version of Java (for example here:
 - <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>)
- After download, the user should make sure that the file is on the desktop for launch
- Alternatively, it can be launched from the command line with: `java -jar Mars4_5.jar`
#assuming you didn't rename the download
- Once the application starts, the user should see an IDE like shown below (Fig. 1)

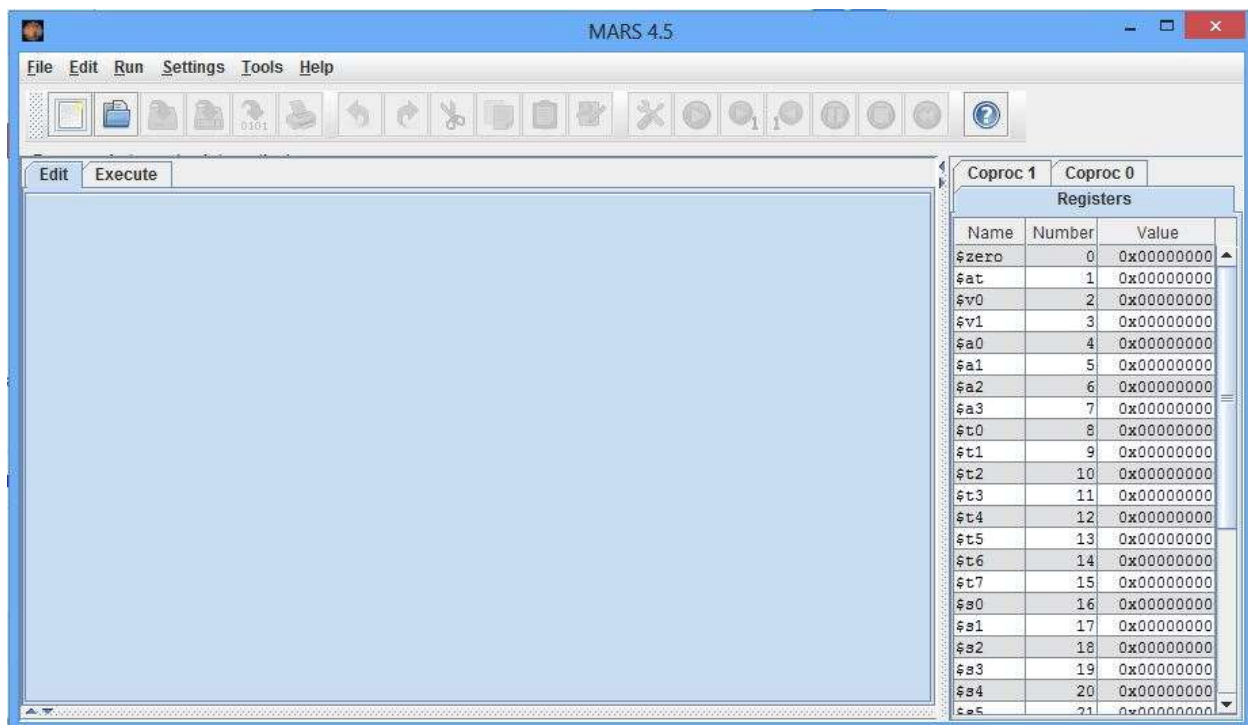


Fig 1: MARS Welcome screen

3. Parts of the simulator

The following are a few tools that are available in the simulator. It would be advantageous to also know the shortcuts to operate with ease. The user should note that most of these controls will be activated only after the written code is saved and assembled.



This is used to initiate the assembly operation. Shortcut key is F3



This is used to RUN the complete program. Shortcut key is F5



This is to step through the program one instruction at a time. Shortcut key is F7



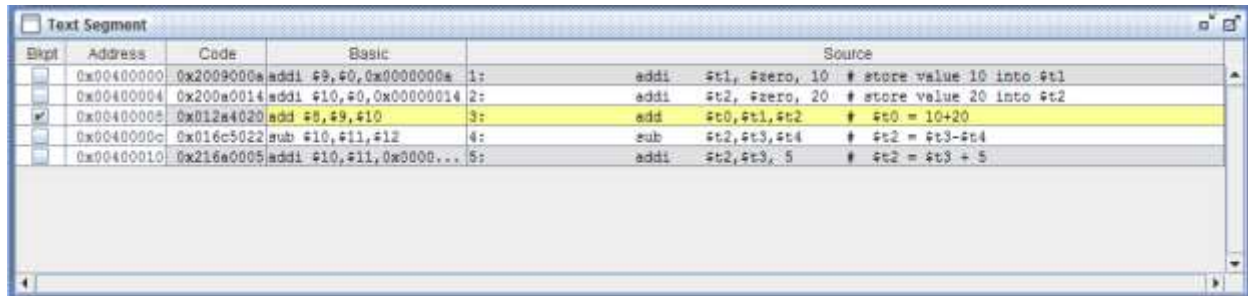
This is to reset the program to an idle point in the beginning. Shortcut key is F12



This is for backstepping to the previous instruction. Shortcut key is F8

The user can try a sample code to test out the simulator and verify if all the tools are working. This document describes a very simple program being executed below.

Once the code was written to the “Edit” region of the simulator, the “assemble” operation is performed and the screen switches to the “Execute” tab. The execute tab has multiple regions and the code text resides in the “Text Segment” as shown below.



Bkpt	Address	Code	Basic	Source
<input type="checkbox"/>	0x00400000	0x2009000a	addi \$9,\$0,0x0000000a 1:	addi \$t1,\$zero,10 # store value 10 into \$t1
<input type="checkbox"/>	0x00400004	0x200a0014	addi \$10,\$0,0x00000014 2:	addi \$t2,\$zero,20 # store value 20 into \$t2
<input checked="" type="checkbox"/>	0x00400008	0x012a4020	add \$8,\$9,\$10 3:	add \$t0,\$t1,\$t2 # \$t0 = 10+20
<input type="checkbox"/>	0x0040000c	0x016c5022	sub \$10,\$11,\$12 4:	sub \$t2,\$t3,\$t4 # \$t2 = \$t3-\$t4
<input type="checkbox"/>	0x00400010	0x216a0005	addi \$10,\$11,0x0000... 5:	addi \$t2,\$t3,5 # \$t2 = \$t3 + 5

Fig 2: Text Segment

The text segment describes each instruction in multiple ways. The “Address” tab holds the memory address at which the particular instruction resides. The “code” is a hexadecimal representation of the instruction in machine language. Note that this is a 32 bits long MIPS instruction. The “source” is the direct textual representation of the written code, along with comments. The “Bkpt” is the breakpoint toggle checklist. More about this will be discussed in the next section.

The area marked “Registers” contains all the system registers of a MIPS processor. The following figure shows some part of the register set.

Registers			Coproc 1	Coproc 0
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	0x0000000a		
\$t2	10	0x00000014		
\$t3	11	0x00000000		
\$t4	12	0x00000000		
\$t5	13	0x00000000		
\$t6	14	0x00000000		
\$t7	15	0x00000000		
\$s0	16	0x00000000		
\$s1	17	0x00000000		
\$s2	18	0x00000000		

Fig 3: Register set of MIPS

And lastly, the addresses (32 bit) and the values they hold can be found at the “Data Segment” region of the simulator. This is an array of memory locations which hold the respective data values and is updated at run time.


Data Segment								
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Fig 4: Data Segment

The user should note that all values are zero initially. They can see the values being updated if the program has load and store operations.

4. Getting Started

To create an assembly file, goto File->New. Once you have written some assembly you must save the file before you can assemble it (turn it into machine language). You can save with


either Ctrl+S or File-> Save As. To assemble your code press the  button on the top. If

there are any errors in the assembling process they will be in the MARS messages section. To

run your program click the  or  button. The latter is for single step execution.

5. Debugger

After the code is assembled, the user may proceed to debug mode by executing it one step at a time or by executing a fixed number of instructions per second. The user may step through their

code with the  button. In the register panel, the one highlighted in green is the one that has most recently been changed. The user may also change the values in the register by double clicking the value and setting it to the value they want. In the execute tab the most recently executed instruction will be highlighted in yellow. To change the number of instructions executed per second, the user should drag the slide on the top hotbar to the desired value.

If the user wishes to run the program only to a particular instruction to check for intermediate program state, they can do so by using the breakpoint facility. The “Bkpt” column check box should be ticked to run the program continuously to that particular instruction. The user should note that the marked instruction will NOT be executed as part of the first run. This means that if a breakpoint is placed at a particular instruction, the program will run until the previous step. The user can rerun the program to finish complete execution. The user also has freedom to add more than one breakpoints.

Summary of Mars

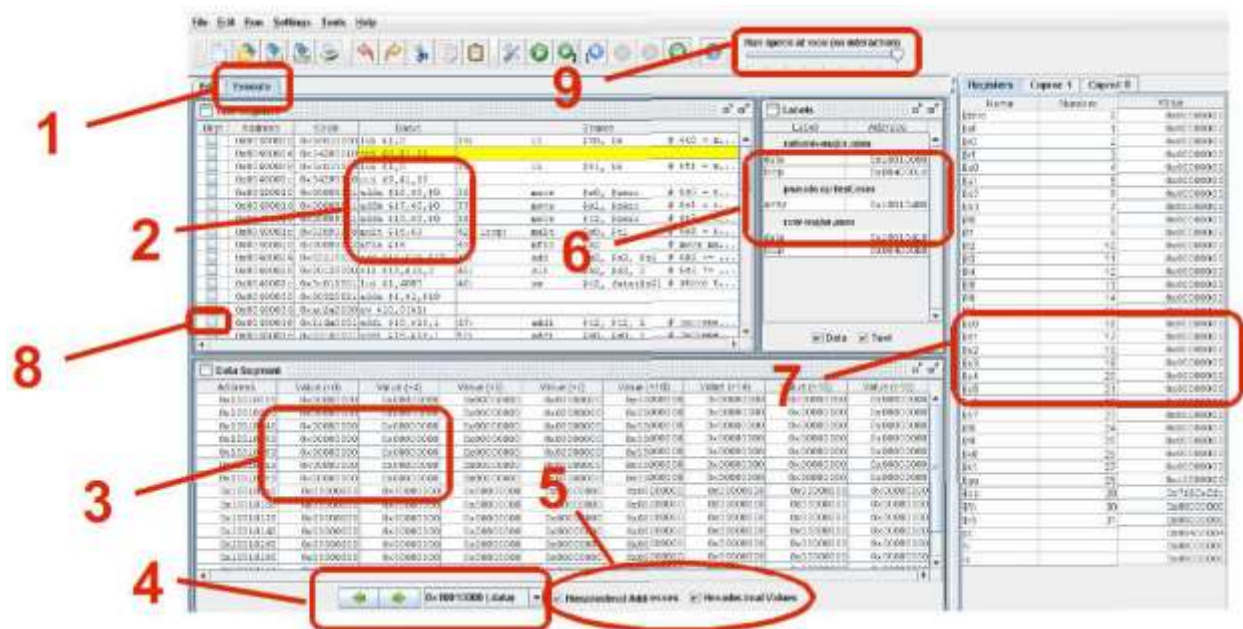


Fig. 5 MARS Execute window's panes

The above figure shows the MARS Execute window's panes, and emphasizes the following features:

1. The Execute window's tab.
2. Assembly code displayed with addresses and machine code.
3. Values stored in the data segment. These are directly editable.
4. Controls for navigating the data memory area. Allows switching to view the stack segment.
5. Switching between decimal and hexadecimal addresses and values in memory and registers.
6. Labels and their corresponding memory addresses.
7. Values stored in registers. These are directly editable.
8. Checkboxes used to setup breakpoints for each MIPS instruction. Useful in debugging.
9. Execution speed selection. Useful in debugging.

At all times, the MIPS register window appears on the right-hand side of the screen, even when you are editing and not running a program. While writing a program, this serves as a useful reference for register names and their use. Move the mouse over the register name to see the tool tips.

There are three register tabs:

- The Register File: integer registers \$0 through \$31, HI, LO, and the Program Counter PC.
- Coprocessor 0: exceptions, interrupts, and status codes.
- Coprocessor 1: floating point registers.

A MIPS assembly language program template

```
# Title:
# Author:
# Date:
# Description:
# Input:
# Output:
##### Data segment #####
.data
. . .
##### Code segment #####
.text
.globl main
main:          # main function entry
. . .
li $v0, 10
syscall        # system call to exit program
```

There are three types of statements that can be used in assembly language, where each statement appears on a separate line:

1. Assembler directives: These provide information to the assembler while translating a program. Directives are used to define segments and allocate space for variables in memory. An assembler directive always starts with a dot. A typical MIPS assembly language program uses the following directives:

- .data** Defines the data segment of the program, containing the program's variables.
- .text** Defines the code segment of the program, containing the instructions.
- .globl** Defines a symbol as global that can be referenced from other files.

2. Executable Instructions: These generate machine code for the processor to execute at runtime. Instructions tell the processor what to do.

3. Pseudo-Instructions and Macros: Translated by the assembler into real instructions. These simplify the programmer task. In addition, there are comments. Comments are very important for programmers, but ignored by the assembler. A comment begins with the # symbol and terminates at the end of the line. Comments can appear at the beginning of a line, or after an instruction. They explain the program purpose, when it was written, revised, and by whom. They explain the data and registers used in the program, input, output, the instruction sequence, and algorithms used.

Declaring constants and variables

Parts of a declaration: (in data segment)

1. **label**: memory address of variable
2. **directive**: “type” of data (used by assembler when initializing memory, not enforced)
3. **constant**: the initial value

```
.data
# string prompt constant
prompt: .asciiz "What is your favorite number?: "

# variable to store response
favnum: .word 0
```

No real difference between constants and variables!

System Calls

Programs do input and output using system calls. On a real-system, the operating system provides system call services to application programs. The MIPS architecture provides a special **syscall** instruction that generates a system call exception, which is handled by the operating system.

System calls are operating-system specific. Each operating system provides its own set of system calls. Because MARS is a simulator, there is no operating system involved. The MARS simulator handles the **syscall** exception and provides system services to programs. The following table shows a small set of services provided by MARS for doing basic I/O.

Before using the **syscall** instruction, you should load the service number into register \$v0, and load the arguments, if any, into registers \$a0, \$a1, etc. After issuing the **syscall** instruction, you should retrieve return values, if any, from register \$v0.

Service	Code in \$v0	Arguments	Result
Print Integer	1	\$a0 = integer to print	
Print String	4	\$a0 = address of null-terminated string	
Read Integer	5		\$v0 = integer read
Read String	8	\$a0 = address of input buffer \$a1 = maximum characters to read	
Exit program	10		Terminates program
Print char	11	\$a0 = character to print	
Read char	12		\$v0 = character read

Task to do

1. Test a simple MIPS program. Consider the following program shown below:

```
# Description: A simple hello world program!
.data                                # add this stuff to the data segment
hello: .asciiz "Hello, world!\n"    # load the hello string into data memory
                                     # a null terminated string
.text                                # now we're in the text segment
    li    $v0, 4                    # set up print string syscall
    la    $a0, hello                # argument to print string
    syscall                          # tell the OS to do the syscall
    li    $v0, 10                   # set up exit syscall
    syscall                          # tell the OS to do the syscall
```

- Type the program shown in the Figure above. (new → save it as “hello_world.asm”)
- Find out how to show and hide line numbers.
- Assemble and run the program.
- What output does the program produce? and where does it appear?

2. Explore the MARS simulator:

- Open and assemble the `Fibonacci.asm` program.
- Identify the locations and values of the initialized data.
- Toggle the display format between decimal and hexadecimal.
- Run the program at a speed of 3 instructions per second or less.
- Single-step through the program and watch how register and memory values change.
- Observe the output of the program in the Run I/O display window.
- Set a breakpoint at the first instruction that prints results. What is the address of this instruction?
- Run the program at full speed and watch how it stops at the breakpoint.
- Change the line:

space: .asciiz " " # space to insert between numbers

to:

space: .asciiz "\n" # space to insert between numbers

Run the program again. What do you notice?

3. Write a simple assembly program that calculates the area of a triangle with height=10 and width=5. At the end of each line, write a comment about your code.

4. Write an assembly program that calculates the following.

Original value in memory = 7

Operation to do: 1) double the org. value

2) add the org. value to the result of 1)

3) save result in memory

4) print the result

At the end of each line, write a comment about your code.

NOTE: Write a report according to the LAB questions and submit a word (or pdf) file.

For 3 and 4, screen capture of your Mars screen must be included.