

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 ability to evaluate many instructions at once using 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.

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 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)

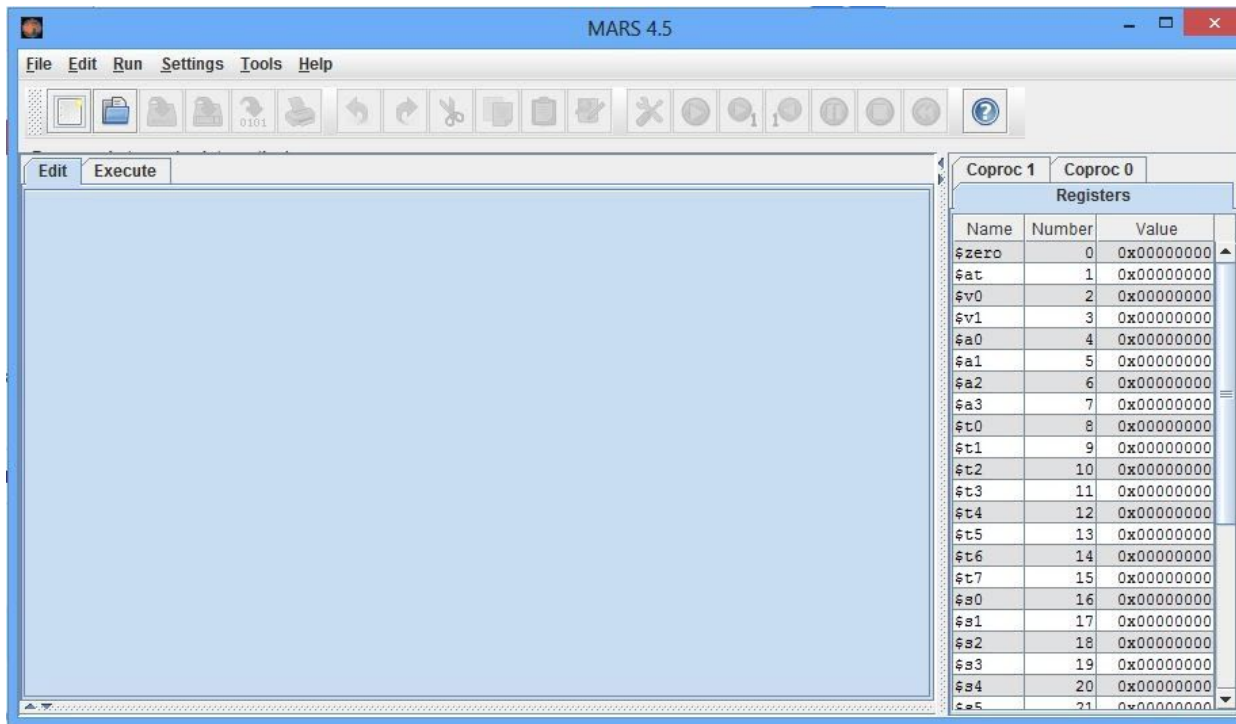


Fig 1: MARS Welcome screen

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.

Text Segment				
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 segregates the instructions and each instruction is identified with multiple attributes. 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 their holding values of the program based memory transactions can be found at the “Data Segment” region of the simulator. This is an array of memory locations which hold the respective values and is updated in run time.


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.

6. Runtime IO and Arrays

A list of syscalls are provided at this [link](#). You will need to make the appropriate syscalls to do any form of IO, any console IO will appear in the Run I/O tab in Mars.

MIPS also provides a few ways to handle data in arrays. Strings are a special case of arrays, where the elements are one byte characters (in the ASCII encoding scheme), terminated with the null character (ASCII character 0). The null character is necessary in many languages so that functions handling the string know where it ends.

Below you will find MIPS code demonstrating one way to accomplish this in MARS, using statically allocated memory. If you are curious to know how to use dynamically allocated memory (“heap memory”) in MIPS/MARS, there is a decent tutorial on dynamic memory here: https://chortle.ccsu.edu/AssemblyTutorial/Chapter-33/ass33_1.html Although it was written for the SPIM MIPS program, the syscalls and instructions are the same in MARS.

The program below makes use of the ‘.align’ MARS directive, which tells the assembler to place the next data element on an appropriate memory boundary so that it is handled appropriately with stores and loads, according to the byte size of the elements you expect to store there.

The reason it’s a good idea to use this directive is complicated. Suffice to say that you will avoid a lot of trouble by using it as shown. The text below can be copied directly into MARS for an interactive demonstration. Note that using the instruction `la $reg, <label>` instructions MARS to load the address of the first byte of data following that label into the register, giving you easy access to the statically allocated data addresses.

.data

```
# Use the .align directive to align data elements to appropriate memory boundaries,  
# .align 0 for bytes, .align 1 for half-words (shorts), .align 2 for words (ints) and  
# .align 3 for doubles (longs, and if you're a sadist, double precision floating)
```

```
# The labels and sizes of these buffers were chosen arbitrarily. The size  
# of space could be any size that fits in memory, and the name of the label  
# can be whatever combination of words and letters you choose followed  
# immediately by a semi-colon
```

```
# call this array A
```

```
.align 0      # byte-align the 40 byte space declared next
```

```
buffer1:      .space 40          # allocate 40 bytes as a read buffer for string input
```

```
# call this array B
```

```
.align 0# byte-align the 96 byte array declared next
```

```
buffer2:      .space 96          # allocate 96 bytes as a read buffer for string input
```

```
# call this array C
```

```
.align 2# word-align the 16 byte array declared next
```

```
buffer3:      .space 12          # make room for 3 ints (3 words)
```

```
# declare null terminated ascii strings at address labels
```

```
# directly in .data so we can print it later using syscalls
```

```
prompt:       .ascii "Enter up to 39 characters to re-print: "
```

```
description:  .ascii "We wrote these vowels straight to the buffer in MIPS: "
```

```
prompt2:      .ascii "Enter 3 integers to be printed one after the next: "
```

space: .asciiz " "

.text

This is how you read from standard in to a character array

This is a syscall that writes 39 characters (+ null char is 40) into
the read buffer at the address in \$a0 from standard input

When we execute a syscall with the value 8 in \$v0, it tells MARS that
we are reading a string from standard input. MARS expects the address
of the buffer to write to in \$a0, and the number of bytes to read in
\$a1. Note that MARS will actually read 1 fewer bytes than indicated
in \$a1 because it automatically null terminates the string

li \$v0, 4 # syscall reads this reg when called, 4 means print string
la \$a0, prompt # load the address of the string declared at .data
syscall # print the string declared at label prompt in .data

li \$v0, 8 # syscall reads this reg when called, 8 means read string
la \$a0, buffer1 # load the address of the buffer to write into from stdin
li \$a1, 40 # load the length of the buffer in bytes in \$a1
syscall

This is how you would write to standard out what you just read from
standard in

This is a syscall that reads the characters that are in the buffer
at address \$a0 and prints them to standard output. We set \$a0 equal

to the location of the label buffer1

When we execute a syscall with the value 4 in \$v0, it tells MARS that
we are writing a string to standard out. MARS expects the address of
the buffer to read from in \$a0.

```
li $v0, 4           # syscall reads this reg when called, 4 means write string
la $a0, buffer1     # load the address of the buffer to read from to stdout
syscall
```

Here is an example of how to write the string 'aeiou\n' to a buffer and
null terminate it, then print it to standard out. Without the null
termination the syscall wouldn't know when to stop reading characters from
memory

```
li $v0, 4
la $a0, description
syscall             # print the string declared at label description in .data
```

```
la $t0, buffer2     # load the base address of array A declared above into $t0
```

Write 'a' to A[0]

```
li $t1, 97          # load 97 into reg $t1 (ascii value of 'a')
sb $t1, 0($t0)      # store 97 into the first byte of readbuffer
```

You can also use hex immediates if you prefer

Write 'e' to A[1]

```
li $t1, 0x65        # load 0x65 into reg $t1 (hex ascii value of 'e')
sb $t1, 1($t0)      # load 0x65 into the second byte of readbuffer
```

```
# Write 'i' to A[2]
li $t1, 0x69          # load 0x69 into reg $t1 (hex ascii value of 'i')
sb $t1, 2($t0)         # load 0x69 into the third byte of readbuffer
```

```
# Write 'o' to A[3]
li $t1, 0x6f          # ... 'o'
sb $t1, 3($t0)         # ... 4th byte
```

```
# Write 'u' to A[4]
li $t1, 0x75          # ... 'u'
sb $t1, 4($t0)         # ... 5th byte
```

```
li $t1, 0x0A          # ... '\n'
sb $t1, 5($t0)         # ... 6th byte
```

```
# Write the null terminator (value 0) to the end of the string
```

```
li $t1, 0              # load 0 into reg $t1 (ascii value of null terminator)
sb $t1, 6($t0)         # store 'null' as the last byte of the buffer
```

```
# This syscall prints the contents of buffer2 to stdout as detailed above
li $v0, 4
la $a0, buffer2
syscall
```

```
# This is an example of how to read 3 integers from the prompt and write them
# to memory in the pre-allocated buffer3 using a loop
```

```
li $v0, 4
la $a0, prompt2
syscall
```

```
addi $s0, $0, 0          # i = 0
la $t1, buffer3          # load base address of array C declared above under
                          # label buffer3. This array is word-aligned, meaning
                          # we can load and store whole words to it safely
```

```
li $s1, 3                # stopping condition is i == 3
```

input_loop:

```
beq $s0, $s1, end_input  # if i == 3, stop taking input
```

```
li $v0, 5                # tell MARS to read an integer from the user
syscall                  # the integer is returned in register $v0
```

```
sll $t0, $s0, 2          # store i*4 to t0, call this offset
add $t2, $t0, $t1        # t2 = Base address of c + offset, this is address of c[i]
sw $v0, 0($t2)           # store the integer read into memory at c[i]
```

```
addi $s0, $s0, 1         # i++
j input_loop
```

end_input: # 3 integers (4 bytes each) have now been read and stored to memory

```
# This is an example of how to print 3 integers from the pre-allocated
# buffer 3 using loops, which should now have 3 integers written into it by
# the previous lines of codes
```

li \$s0, 0	# i = 0
la \$t1, buffer3	# load base address of array C declared above under label # buffer 3. The array is word-aligned, meaning we can load # and store whole words to it safely
li \$s1, 3	# stopping condition is i == 3
output_loop:	
beq \$s0, \$s1, end_output	# if i == 3, stop printing out
sll \$t0, \$s0, 2	# store i*4 to t0, call this offset
add \$t2, \$t0, \$t1	# t2 = base address of c + offset, this is address of c[i]
lw \$a0, 0(\$t2)	# load the integer to print into register \$a0 from c[i]
li \$v0, 1	# tell MARS to print an integer to standard out
syscall	# the integer to print is in \$v0
li \$v0, 4	
la \$a0, space	
syscall	# tell MARS to print the space character at label space in .data
addi \$s0, \$s0, 1	
j output_loop	
end_output:	

7. An Example Program

The following program declares variables, does some basic math, then prints a value with a system call.

```
.data    # This declares data elements that will be placed in the global data region.
        # The global data region starts at address 0x10010000. The default memory snapshot
        # shown in MARS starts at this point.
label1: .word 5 # This declares a 32-bit entity that is initialized to 5. After assembling, make
               # sure you see "5" in the first word of the memory region starting at 0x10010000
               # To access this word, your program will use the label "label1". Notice that the
               # assembler will convert references to "label1" with the appropriate address.
label2: .word 7 # This declares a 32-bit entity that is initialized to 7.
label3: .asciiz "The answer is " # This declares a string with label "label3".

.text    # I'm now entering the text/code region. Note that you can go back and forth between
        # .text and .data any time.
lw $t0, label1 # This loads the value at address "label1" into register $t0. As you step through
               # the program, make sure that $t0 now has the value 5. Also observe how the
               # assembler translates label1 into the appropriate address sequence. In many
               # instances, you'll see that the assembler converts your pseudo-instructions
               # into multiple MIPS instructions.
lw $t1, label2 # This loads the value at address "label2" into register $t1.
add $t2, $t1, $t0 # At the end of this, $t2 should have the value 12.
li $v0, 4        # I am getting ready to do a system call. I use the pseudo-instruction "li",
               # which means load-immediate. This puts the immediate operand "4" into
               # register $v0. I'm doing this to specify the "print string" system call.
la $a0, label3   # I'm using pseudo-instruction "la" which refers to load-address. I'm loading
               # the address of the string "The answer is: " into register $a0. This will serve
               # as the argument to the upcoming system call. Note that I referred to the
               # string by its label.
syscall         # This invokes the system call. The system call examines $v0 and $a0 and
               # goes on to print the string.
```

```
li $v0, 1      # I'm now going to do a system call to print an integer.
move $a0, $t2  # I'm using pseudo-instruction move to copy the answer in $t2 into the
               # argument register $a0 for the system call.
syscall       # Print the integer answer.
```

Here's another [example program](#) that is a tad more complicated.

8. Running MARS from the Command Line in Windows

MARS supports running from the command line and can take many arguments with various effects, a list of which can be found here: <https://courses.missouristate.edu/KenVollmar/mars/Help/MarsHelpCommand.html>. Perhaps the most useful (from a grading perspective at the least) is the ability to run MIPS .asm files from the command line, using input/output as expected in the command terminal.

To do this conveniently without fully qualifying the path to the java virtual machine installed on your windows machine, you may want to set your PATH environment variable to include your java installation directory. Directions for doing this can be found here: <https://stackoverflow.com/questions/1672281/environment-variables-for-java-installation>.

Once that is done, it is as simple as moving your MIPS .asm files into the same directory as your MARS .jar, opening a terminal there and typing

```
java -jar <MARS .jar name> <MIPS .asm assembly file name>
```

There are details on the Missouri State website linked above on how to pass arguments, etc. if you are inclined to investigate.

9. Common Problems -- FAQ

If MARS freezes when running your program it is very likely that the user has written an infinite loop. To debug an infinite loop the number of instructions per second have to be changed to a value less than run at max speed. This can be adjusted using the slider to the right of the action buttons. (to not freeze MARS) Proceed to normal debugging.



"I can't assemble my program!" -- Please make sure you have saved the file you have created.

On Linux machines, sometimes, double clicking the .jar MARS file might throw an error. Instead, executing the command "**java -jar MARS_SIM.jar**" (MARS_SIM being the name of the file) in terminal will work. An easy way to run the simulator is to make it executable. One way to do it: Right click->Properties->Permissions->Select "Allow executing file as program".

10. Common Terminologies

Integrated Development Environment - A software application for a developer to build their code using features like code editor, compiler, debugging and many more.

J2SE - Java 2 Standard Edition is a platform used to deploy particular Java based softwares like MARS.

Simulator - The programs that would be written in the lab are not being exported to a product yet. Therefore these codes have to be run in a simulation environment that would mimic the functionality.

Assembly - The process of converting the written code into machine understandable instructions.