

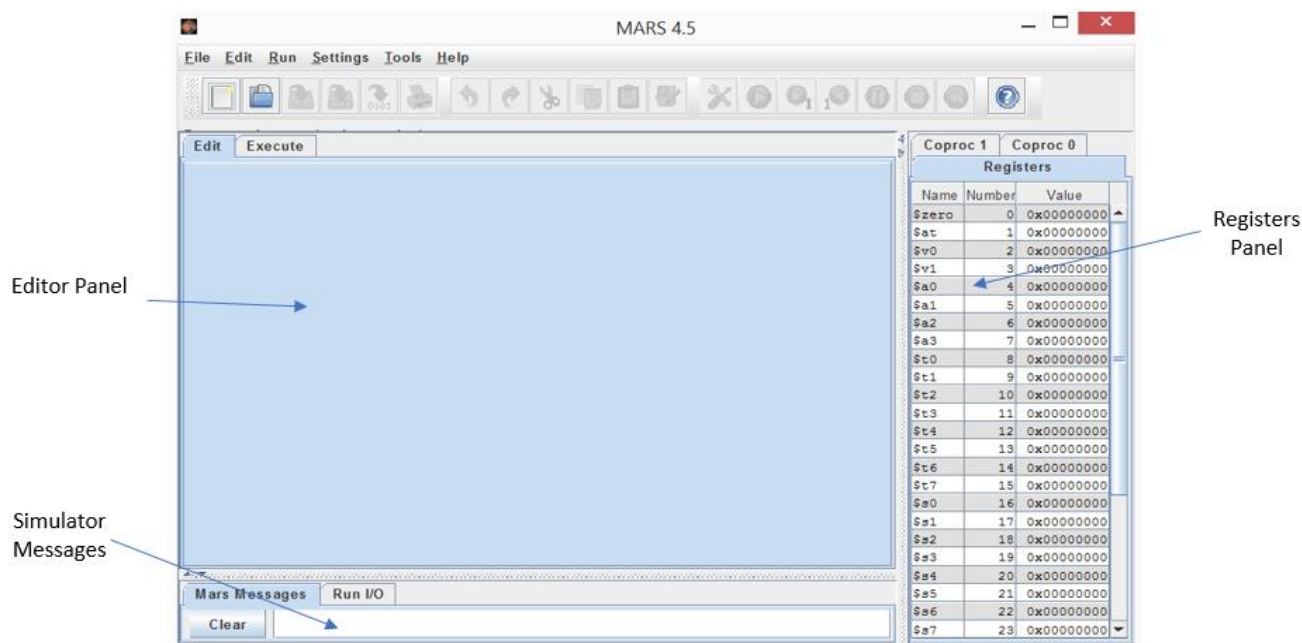
Software used: Mars4.5\_Windows

MARS may be downloaded from [www.cs.missouristate.edu/MARS](http://www.cs.missouristate.edu/MARS)

## A. Overview

Mars is a simulator that runs MIPS programs loaded from your computer. The Mars simulator has a number of features and requirements for writing MIPS assembly language programs. This includes a properly formatted assembly source file.

When you launch Mars, a window will open as shown in the figure below.



The window is divided into different sections:

- 1. Menus and Toolbar:** Most menu items have equivalent toolbar icons. If the function of a toolbar icon is not obvious, just hover the mouse over it and a tool tip will soon appear. Nearly all menu items also have keyboard shortcuts. Any menu item not appropriate in a given situation is disabled.
- 2. Editor:** The editor tab is used to write and edit the assembly code. The bottom border of editor includes the cursor line and column position and there is a checkbox to display line numbers. They are displayed outside the editing area.

**3. Messages Panel:** There are two tabbed message areas at the bottom of the screen.

- The **Run I/O tab** is used at runtime for displaying console output and entering console input as program execution progresses. You have the option of entering console input into a pop-up dialog then echoes to the message area.
- The **Mars Messages tab** is used for other messages such as assembly or runtime errors and informational messages. You can click on assembly error messages to select the corresponding line of code in the editor.

**4. MIPS Registers:** This panel display the content of all registers. MIPS registers are displayed at all times, even when you are editing and not running a program. While writing your program, this serves as a useful reference for register names and their conventional uses (hover mouse over the register name to see tool tips). There are three register tabs: the Register File (integer registers \$0 through \$31), Coprocessor 0 registers (exceptions and interrupts), and Coprocessor 1 floating point registers.

**B. Creating and Running a MIPS program in Mars:**

**1. Create or open MIPS file:**


- a. Begin creating a new MIPS assembly file by:

From the menu bar select **File -> New**. Then you can begin typing in MIPS assembly instructions into the Edit window, as desired.

- b. Open an existing MIPS assembly file by:

From the menu bar select **File -> Open...** and then select the desired assembly program (the desired .asm file) from the file chooser window.

**2. Running/Testing the assembly code**

- a. First assemble the program using the icon  (also available from the **Run** menu). Examine the Mars Messages panel, and notice that the message indicates the assembly was successful.

- b. Run the compiled code:

Run the code using either the **Run -> Go** option, which will execute the program to completion, or

the  run icon.

**3. Typical MIPS Program Layout**

```
.text          #code section
.globl main    #starting point: must be global
main:
    # user program code
.data          #data section
    # user program data
```

#### 4. MIPS Assembler Directives

##### a) Top-level Directives:

###### **.text**

Indicates that following items are stored in the user text segment, typically instructions

###### **.data**

Indicates that following data items are stored in the data segment

###### **.globl sym**

Declare that symbol sym is global and can be referenced from other files

##### b) Common Data Definitions:

###### **.word $w_1, \dots, w_n$**

Store n 32-bit quantities in successive memory words

###### **.half $h_1, \dots, h_n$**

Store n 16-bit quantities in successive memory half words

###### **.byte $b_1, \dots, b_n$**

Store n 8-bit quantities in successive memory bytes

###### **.ascii str**

- Store the string in memory but do not null-terminate it
- Strings are represented in double-quotes "str"
- Special characters, eg. \n, \t, follow C convention

###### **.asciiz str**

Store the string in memory and null-terminate it

##### c) Common Data Definitions:

###### **.float $f_1, \dots, f_n$**

Stores n floating point single precision numbers in successive memory locations

###### **.double $d_1, \dots, d_n$**

Stores n floating point double precision numbers in successive memory locations

###### **.space n**

Reserves n successive bytes of space

#### 5. Data declaration format

Format for declarations:

- name: storage\_type value(s)
  - ✓ create storage for variable of specified type with given name and specified value
  - ✓ value(s) usually gives initial value(s); for storage type .space, gives number of spaces to be allocated
  - ✓ *Note: labels always are followed by colon ( : )*

Example:

- var1: .word 3 # create a single integer variable with initial value 3
- var2: .word 3,4 # create 2-element integer array with elements initialized to 3 & 4
- array1: .byte 'a', 'b' # create a 2-element character array with elements initialized to a & b
- array2: .space 40 # allocate 40 consecutive bytes, with storage uninitialized  
# could be used as a 40-element character array, or a  
# 10-element integer array; a comment should indicate which!

## 6. System Conventions for Registers

0	zero	constant 0
1	at	reserved for assembler
2	v0	results from callee
3	v1	returned to caller
4	a0	arguments to callee
5	a1	from caller: caller saves
6	a2	
7	a3	
8	t0	temporary
...		
15	t7	

16	s0	callee saves
...		
23	s7	
24	t8	temporary (cont'd)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	pointer to global area
29	sp	stack pointer
30	fp	frame pointer
31	ra	return Address caller saves

## 7. System Calls

Method

- Load system call code into register \$v0
- Load arguments into registers \$a0...\$a3
- call system with instruction syscall
- After call, return value is in register \$v0

*System Call Codes:*

Service	Code (put in \$v0)	Arguments	Result
print_int	1	\$a0=integer	
print_float	2	\$f12=float	
print_double	3	\$f12=double	
print_string	4	\$a0=addr. of string	
read_int	5		int in \$v0
read_float	6		float in \$f0
read_double	7		double in \$f0
read_string	8	\$a0=buffer, \$a1=length	
sbrk	9	\$a0=amount	addr in \$v0
exit	10		

Examples:

- to read an integer  
*li \$v0,5*  
*syscall*  
*sw \$v0, 0(\$t1)*
- to print an integer  
*li \$v0,1*  
*move \$a0,\$t3*  
*syscall*

## 8. Some Instructions

Instruction		Description
<b>l&lt;type&gt;</b>	<b>Rdest, mem</b>	Load value from memory location into destination register.
<b>li</b>	<b>Rdest, imm</b>	Load specified immediate value into destination register.
<b>la</b>	<b>Rdest, mem</b>	Load address of memory location into destination register.
<b>s&lt;type&gt;</b>	<b>Rsrc, mem</b>	Store contents of source register into memory location.
<b>move</b>	<b>Rdest, RSrc</b>	Copy contents of source register into destination register.

Here is some basic operations and how can we do them in MIPS:

**wans1 = wnum1 + wnum2**

```
lw $t0, wnum1
lw $t1, wnum2
add $t2, $t0, $t1
sw $t2, wans1          # wans1 = wnum1 + wnum2
```

**wans2 = wnum1 \* wnum2**

```
lw $t0, wnum1
lw $t1, wnum2
mul $t2, $t0, $t1
sw $t2, wans2          # wans2 = wnum1 * wnum2
```

**wans3 = wnum1 % wnum2**

```
lw $t0, wnum1
lw $t1, wnum2
rem $t2, $t0, $t1
sw $t2, wans3          # wans = wnum1 % wnum2
```

**hans = hnum1 \* hnum2**

```
lh $t0, wnum1
lh $t1, wnum2
mul $t2, $t0, $t1
sh $t2, wans          # hans = wnum1 * wnum2
```

**bans = bnum1 / bnum2**

```
lb $t0, bnum1
lb $t1, bnum2
div $t2, $t0, $t1
sb $t2, bans          # bans = bnum1 / bnum2
```

## 9. Addressing Modes

### a. Direct Mode

It is when the register or memory location contains the actual values. For example:

```
lw $t0, var1
lw $t1, var2
```

Registers and variables \$t0, \$t1, var1, and var2 are all accessed in direct mode addressing.

### b. Immediate Mode

Immediate addressing mode is when the actual value is one of the operands. For example:

```
li $t0, 57
addi $t0, $t0, 57
```

The value 57 is immediate mode addressing. The register \$t0 is direct mode addressing.

### c. Indirection

The ()'s are used to denote an indirect memory access. For example, to get a value from a list

```
la $t0, list
lw $t1, ($t0)
```

The address, in \$t0, is a word size (32-bits). Memory is byte addressable. As such, if the data items in "list" (from above) are words, then four add must be added to get the next element.

A form of displacement addressing is allowed. For example, to get the second item from a list of long sized values:

```
la $t0, list
lw $t1, 4($t0)
```

The "4" is added to the address before the memory access. However, the register is not changed.

## 10. My First Program: Hello World

```
.data
msg: .asciiz "Hello World!\n"

.text    # All program code is placed after the .text assembler directive

.globl main # Declare main as a global function. The label 'main' represents the
starting point

main:    li $v0, 4      # Code for syscall 4 (print_str)
         la $a0, msg    # Pointer to string (load the address of msg)
         syscall

         li $v0, 10     # Code for syscall: exit
         syscall
```

## Exercises:

- 1) Write a program in MIPS that multiply two numbers and displays the result to the user. Use indirect access mode. Define them in the data section as:

```
.data
value: .word 30,20
```

- 2) Update program 1, and print a message *"The product is: "* before printing the value of multiplication.
- 3) Update program 2, and read the two values to be multiplied from the user.
- 4) Convert the following C++ program into a MIPS program. This program finds the maximum between two numbers:

```
void main ()
{
    int a, b;
    cout<<"Please enter two numbers:";
    cin>>a>>b;
    int r = (a < b) ? b : a;
    cout<<"the max is"<< r;
}
```

You can use the following instructions:

- **slt Rdest, Rsrc1, Src2**  
Set Less Than: Sets register Rdest to 1 if register Rsrc1 is less than to Src2 and to 0 otherwise
- **movn \$t3, \$t2, \$t0**  
Move \$t2 to \$t3 if \$t0 is set.