Getting Mars Simulator ON your System

1- Go to course website http://profile.iiita.ac.in/bibhas.ghoshal/teaching_coa_2020.html and download the zip file of Mars simulator (Mars4_5.zip).

2- Extract the zip file and go into the folder Mars4_5.

3- Open a terminal here and install jdk required to run the simulator by typing

   sudo apt-get install default-jdk.

   (If you are having problem in this then try update once by typing sudo apt-get update**)**

4- After successful installation of jdk, start the simulator by typing in the terminal

   java Mars

Directions for using the MARS simulator for MIPS assembly:

1. When the MARS simulator starts up, you can either:
   a. 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.

   b. 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.

2. When ready, you may begin running/testing the assembly code by:

   The environment of this simulator can be simplistically split to three segments:
   i. The *editor* at the upper left where all of the code is being written,
   ii. The *compiler/output* right beneath the editor and
   iii. The *list of registers* that represent the "CPU" for our program.

First assemble the code by selecting **Run -> Assembly** from the menu bar.
   After assembling (by simply pressing F3) the environment changes, with two new segments getting the position of the editor: the text segment where

   i) Each line of assembly code gets cleared of "***pseudo instructions***" at the "***basic***" column

   ii) The machine code for each instruction at the "code" column, and the data segment where we can have a look at a representation of the memory of a processor with little-endian order.

3. Run the compiled code:
   Run the code using either the **Run -> Go** option,
   Which will execute the program to completion, or
   The ***Run -> Step*** option, which will execute only one instruction at a time.
   You may likewise use the corresponding buttons for Go or Step available under the menu bar (or use the F5 or F7 keys, which correspond to Go and Step. These buttons and key

shortcuts are particularly useful for Step, since you need to push/execute Step each time you want to execute the next instruction in the program).

4. Debug: To see what's going on while the program is being executed, look at the registers in the **'Register'** window, which is the right-most window in the display.
Notice that this displays the current values of all the registers, including the $s and $t registers, and even the **program counter (pc).**

Note: By default, the register values are shown in hexadecimal. If you would prefer to see their decimal values, in the menu bar, de-select Settings -> Values displayed in hexadecimal.

If you create a new assembly program or make modifications to an existing assembly program, you can save the code to a .asm file using the ***File -> Save or File -> Save as.***.. options from the menu bar.

6. The registers:
i. $zero -Constant Register (vale 0)
ii. $a registers - to pass arguments
iii. $t registers – to store information
iv. $s registers – to store information (allow contents to be saved on the stack)
v. $k registers – kernel registers
vi. sp – stack pointer
vii. Frp- frame pointer
viii. $ra – return address register (will be used for subroutine calls)
ix. $v registers - to hold the arguments for system calls
x. $f registers – floating point registers available from the co-processor (will be used for floating point operations)

7. The first assembly program: Hello World

i. The hello_world.asm file will have two sections: data (to hold the data of the program) and text (to hold the instructions and pseudo instructions)

```
.data                           #data section
str: .asciiz "Hello world\n"    # we define a variable called str having data type ascii and holding the
                                string Hello world with a new line character at the end. This variable
                                called str is loaded in the data memory during compile time

 .text                          #code section

li      $v0, 4                  #system call for printing strings
la      $a0, str                #loading our string from data section to the $a0 register;
                                la stands for load the address, where address refers to the first
                                address of the string of characters named as  str
syscall                         # the system is instructed to print the value pointed to by the address
                                loaded in $a0
```

Before illustrating the results through MARS, a little more explanation about these commands is needed:

System calls are a set of services provided from the operating system. To use a system call, a call code is needed to be put to $v0 register for the needed operation. If a system call has arguments, those are put at the $a0-$a2 registers. Here are all the system calls.

li (load immediate) is a pseudo-instruction (we'll talk about that later) that instantly loads a register with a value.

la (load address) is also a pseudo-instruction that loads an address to a register.

With li $v0, 4 the $v0 register has now 4 as value, while la $a0, str loads the string of str to the $a0 register.

A word is (as much as we are talking about MIPS) a 32 bits sequence, with bit 31 being the Most Significant Bit and bit 0 being the Least Significant Bit.

lw (load word) transfers from the memory to a register, while sw (store word) transfers from a register to the memory. With the lw $s1, 0($t0) command, we loaded to $s1 register the value that was at the LSB of the $t0 register (that's what the 0 symbolizes here, the offset of the word), aka 256. $t0 here has the address, while $s1 has the value. sw $t2, 0($t0) does just the opposite job.

MARS uses the Little Endian, meaning that the LSB of a word is stored to the smallest byte address of the memory.

MIPS uses byte addresses, so an address is a part of its previous and next by 4.

By assembling the code from before, we can further understand how memory and registers exchange, disabling "Hexadecimal Values" from the Data Segment: or enabling "ASCII" from the Data Segment: