Computer Architecture
&
Organization (EEL 4768)

Lab #1

Introduction to MARS

MARS, the Mips Assembly and Runtime Simulator, will assemble and simulate the execution of MIPS assembly language programs. It can be used either from a command line or through its integrated development environment (IDE). MARS is written in Java and requires at least Release 1.5 of the J2SE Java Runtime Environment (JRE) to work. It is distributed as an executable JAR file.

1) How to install and run MARS?

Windows User:

Download and install JRE through below link:
http://javadl.sun.com/webapps/download/AutoDL?BundleId=76860

Download MARS from following link and double click the icon for Mars.jar and MARS environment will open:
http://courses.missouristate.edu/kenvollmar/mars/download.htm

Mac User:

Download and install JRE through below link:
http://www.java.com/en/download/index.jsp

Download MARS from following link and double click the icon for Mars.jar and MARS environment will open:
http://courses.missouristate.edu/kenvollmar/mars/download.htm

2) Tutorial - Basic Intro into MARS

http://www.youtube.com/watch?v=z3ltaJ5UU5I

3) Basic MARS Use

There are two main windows in MARS. The Edit window is used to create and modify your program. The Execute window is used to run and debug your program. The tabs at the top of the windows are used to switch between the two.

Part 1: Editing and Assembling

Creating a new program

Select "File => New" from the Mars menu to open a blank editor window. Enter your program. The example below shows the format of a Mars program. Select "File => Save As" to save your program to disk. The ".asm" extension is recommended. Once the file has been saved for the first time, you may select "File => Save" to save changes without having to specify the file name.

```
.data
out_string: .asciiz "\nHello, World!\n"
.text
li $v0, 4
la $a0, out_string
syscall
li $v0, 10
syscall
```

Fig 1. HelloWorld program

Opening an existing program
To open an existing program, select "File = > Open" from the Mars menu. Enter the name of the program file and click the Open button.

Assembling the program
Once the program has been created and saved to disk it may be assembled (translated into MIPS machine language). Select "Run => Assemble" from the Mars menu.
If there are no errors the Execute pane will appear, showing memory and register contents prior to execution. Click the "Edit" tab if you wish to return to the Edit pane. If there are syntax errors in your program, they will appear in the Mars messages window at the bottom of the Mars screen. Each error message contains the line and position on the line where the error occurred.

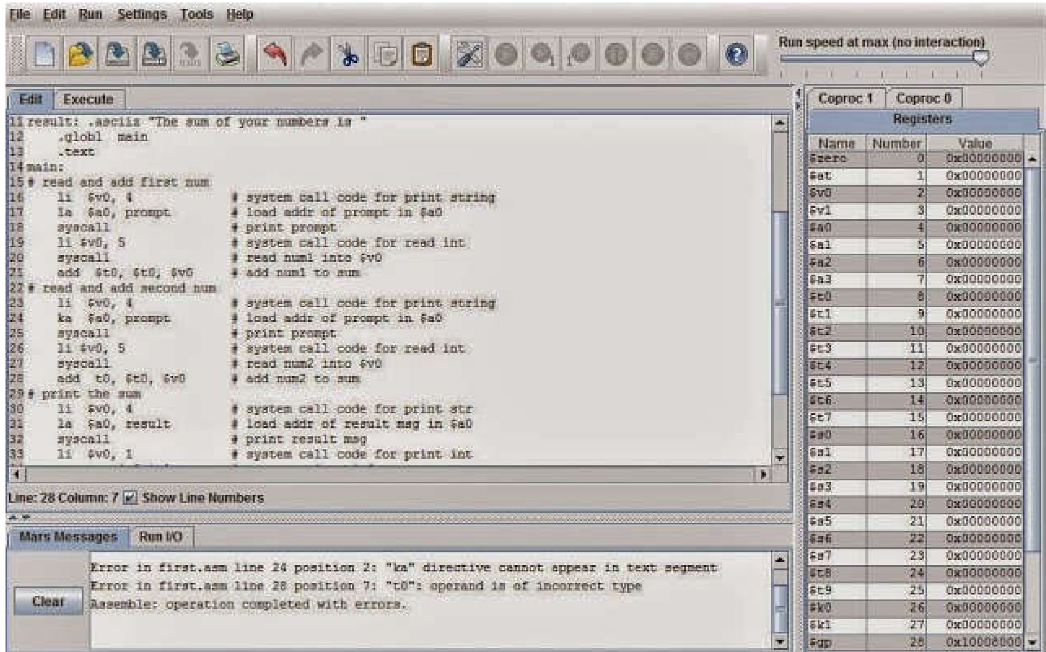

Fig 2. Errors on lines 24 and 28 shown in Mars messages window

Part 2: Executing Your Program
If there are no syntax errors when you assemble, the display will switch to the execute window. The execute window contains panes for the text segment and the data segment (which holds the variables) as shown in Fig. 3.

The **Text Segment** window displays the instructions. Each line contains 5 columns:
- Column 1 displays a checkbox for setting breakpoints.
- Column 2 displays the address of an instruction in hexadecimal.
- Column 3 displays the machine encoding of the instruction in hex.
- Column 4 is a mnemonic description of the machine instruction.
- Column 5 contains the assembly source that corresponds to the instruction.

Running the program
Once you have removed any syntax errors you can run your program. The Run menu and the toolbar contain the follow execution options:

- Go runs the program to completion.
- Run speed at max (no interaction) The Run Speed Slider allows you to run the program at full speed or slow it down so you can watch the execution.
- Step executes a single statement.
- Reset resets the program to its initial state, so that you can execute again from the beginning using the initial variable values.
- Pause suspends execution at the currently executing instruction when are running your program.
- Stop terminates a running program.
- Backstep "unexecutes" the last instruction when you are paused or stepping.

You can also set a breakpoint at any statement by clicking the checkbox in front of the statement in the text segment pane. During execution you can see which statement is being executed (highlighted in yellow), which register was last modified (highlighted in green) and which variable was last changed (highlighted in blue). It's usually only possible to see the highlighting when you are stepping or running at less than full speed. Below figure shows the environment of “HelloWorld” program after completion.

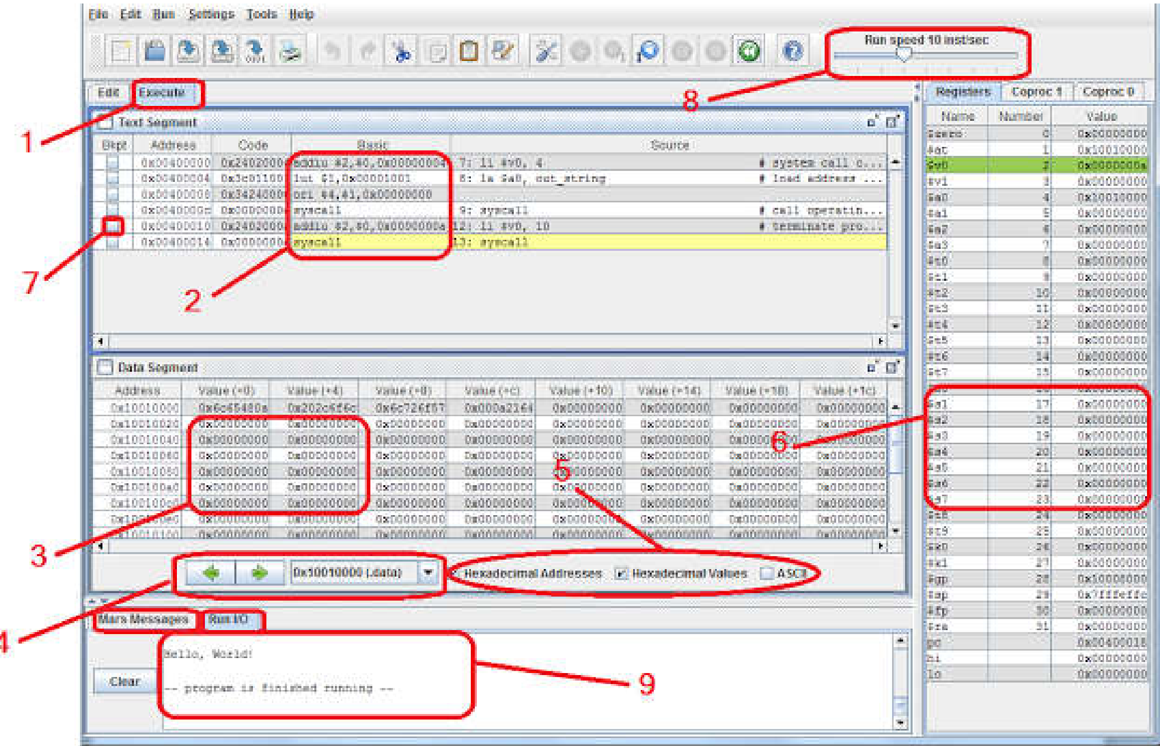

Fig 3. The environment of MARS after executing “HelloWorld” program

1. Execute display is indicated by highlighted tab.
2. Assembly code is displayed with its address, machine code, assembly code, and corresponding line from the source code file.
3. The values stored in Memory are directly editable.
4. The window onto the Memory display is controlled in several ways: previous/next arrows and a menu of common locations (e.g., top of stack).
5. The numeric base used for the display of data values and addresses (memory and registers) is selectable between decimal and hexadecimal.
6. The values stored in Registers are directly editable.
7. Breakpoints are set by a checkbox for each assembly instruction. These checkboxes are always displayed and available.
8. Selectable speed of execution allows the user to “watch the action” instead of the assembly program finishing directly.
9. MARS messages are displayed on the MARS Messages tab of the message area at the bottom of the screen. Runtime console input and output is handled in the Run I/O tab.

Closing a program

Select "File => Close" to close the current program. Always close the program before removing the program disk or exiting Mars. Exit Mars with "File => Exit.

MIPS Language

The Format of a MIPS Program
The components of a MIPS program are as follows:

Comments
Comments start with a # sign. Everything is a comment from the # to the end of the line.

Labels
Labels are user defined names, assigned to statements and variables. A label starts with a letter, followed by letters and/or digits, and ends with a colon, I,e. ":" symbol.

Variables
Variables are defined at the beginning of the program. The directive .data starts the variable section.

Code
The code comes after the variables. There are two directives for the code. The .globl directive specifies the external name of the function. For now, that name will be main. Later we will discuss how to create additional functions. This is followed by the .text directive, which starts the code section. The label main comes right after the .text directive to indicate where execution should begin.
The layout of a MIPS program is as follows:

```
# comments describing the program
#
      .data
# variable declarations to be stored in the memory
      .text
       main:
# program assembly code
```

The .data and .text segment identifiers are required, but the main: label technically is optional.

Constants
Character
Character constants are enclosed in single quotes, for example 'a', 'Q', '4', '&'

Numeric
Numeric constants are written in base 10, with an optional leading sign, for example 5, -17

String
String constants are enclosed in double quotes, for example "this is a string"

1. Getting Started: add.asm
To get our feet wet, we'll write an assembly language program named add.asm that computes the sum of 1 and 2, and stores the result in register $t0.

1.1 Commenting
Before we start to write the executable statements of program, however, we'll need to write a comment that describes what the program is supposed to do. In the MIPS assembly language, any text between a pound sign (#) and the subsequent newline is considered to be a comment.

Three forms of comments are absolutely essential:

1) **Program Header:** this is the overall description of the program or project, which appears at the front of the code listing.
2) **Code Block:** comments identifying a block of code inside the program such as a loop or a subroutine.
3) **Individual Instruction:** one comment per each assembly language instruction is required.

Comments are vital for assembly language programs because they are notoriously difficult to read unless they are properly documented. Therefore, we start by writing the following Program Header:

```
# Your Name -- DATE
# add.asm-- A program that computes the sum of 1 and 2,
# leaving the result in register $t0.
# Registers used:
# t0  - used to hold the result.
# end of add.asm
```

Even though this program doesn't actually do anything yet, at least anyone reading our program will know what this program is supposed to do, and who to blame if it doesn't work. We are not finished commenting this program, but we've done all that we can do until we know a little more about how the program will work.

1.2 Finding the Right Instructions

Next, we need to figure out what instructions the computer will need to execute in order to add two numbers. Since the MIPS architecture has relatively few instructions, it won't be long before you have memorized all of the instructions that you'll need, but as you are getting started you'll need to spend some time browsing through the lists of instructions, looking for ones that you can use to do what you want. Documentation for the MIPS instruction set can be found in the textbook. Luckily, as we look through the list of arithmetic instructions, we notice the add instruction, which adds two numbers together. The add operation takes three operands:

1. A register that will be used to store the result of the addition. For our program, this will be $t0.
2. A register which contains the first number to be added. Therefore, we're going to have to place a value of 1 into a register before we can use it as an operand of add. Checking the list of registers used by this program (which is an essential part of the commenting) we select $t1 for this purpose and make note of this in the comments.
3. A register which holds the second number, or a 32-bit constant. In this case, since 2 is a constant that easily fits in 32 bits, we can just use 2 as the third operand of add.

We now know how we can add the numbers, but we have to figure out how to get 1 into register $t1. To do this, we can use the li (load immediate value) instruction, which loads a 32-bit constant into a register. Therefore, we arrive at the following sequence of instructions:

```
# Your Name -- DATE
# add.asm-- A program that computes the sum of 1 and 2,
#  leaving the result in register $t0.
# Registers used:
#  t0 - used to hold the result.
#  t1 - used to hold the constant 1.
       li $t1, 1           # load 1 into $t1.
       addi $t0, $t1, 2    # $t0 = $t1 + 2.
# end of add.asm
```

1.3 Completing the Program

These two instructions perform the calculation that we want, but they do not form a complete program. Much like C, an assembly language program must contain some additional information that tells the assembler where the program begins and ends. The exact form of this information varies from assembler to assembler (note that there may be more than one assembler for a given architecture, and there are several for the MIPS architecture). This tutorial will assume that MARS is being used as the assembler and runtime environment.

1.3.1 Labels and main

To begin with, we need to tell the assembler where the program starts. In MARS, program execution begins at the start of the .text segment, which can be identified with the label main. A label is a symbolic name for an address in memory. In MIPS assembly, a label is a symbol name (following the same conventions as C symbol names), followed by a colon. Labels must be the first item on a line. A location in memory may have more than one label. Therefore, to tell MARS that it should assign the label main to the first instruction of our program, we could write the following:

```
# Your Name -- DATE
# add.asm-- A program that computes the sum of 1 and 2,
#  leaving the result in register $t0.
# Registers used:
#  t0 - used to hold the result.
#  t1 - used to hold the constant 1.
       main: li $t1, 1    # load 1 into $t1.
       addi $t0, $t1, 2    # $t0 = $t1 + 2.
# end of add.asm
```

When a label appears alone on a line, it refers to the following memory location. Therefore, we could also write this with the label main on its own line. This is often much better style, since it allows the use of long, descriptive labels without disrupting the indentation of the program. It also leaves plenty of space on the line for the programmer to write a comment describing what the label is used for, which is very important since even relatively short assembly language programs may have a large number of labels. Note that the MARS assembler does not permit the names of instructions to be used as labels. Therefore, a label named add is not allowed, since there is an instruction of the same name. (Of course, since the instruction names are all very short and fairly general, they don't make very descriptive label names anyway.) Giving the main label its own line (and its own comment) results in the following program:

```
# Your Name -- DATE
# add.asm-- A program that computes the sum of 1 and 2,
#  leaving the result in register $t0.
# Registers used:
#  t0 - used to hold the result.
#  t1 - used to hold the constant 1.
       main:               # MARS starts execution at main.
       li $t1, 1           # load 1 into $t1.
       addi $t0, $t1, 2    # $t0 = $t1 + 2.
# end of add.asm
```

1.3.2 Syscalls

The end of a program is defined in a specific way. Similar to C, where the exit function can be called in order to halt the execution of a program, one way to halt a MIPS program is with something analogous to calling exit in C. Unlike C, however, if you forget to "call exit" your program will not gracefully exit when it reaches the end of the main function. Instead, in actual practice it may blunder on through memory, interpreting whatever it finds as instructions to execute. Generally speaking, this means that if you are lucky, your program will terminate immediately; if you are unlucky, it will do something random and then

crash. The way to tell MARS that it should stop executing your program, and also to do a number of other useful things, is with a special instruction called a syscall.

The syscall instruction suspends the execution of your program and transfers control to the operating system. The operating system then looks at the contents of value register $v0 to determine what it is that your program is asking it to do. Note that MARS syscalls don't actually transfer control to the operating system. Instead, they transfer control to a very simple simulated operating system that is part of the MARS program. In this case, what we want is for the operating system to do whatever is necessary to exit our program. Looking in Syscall functions available in MARS, we see that this is done by placing a 10 (the number for the exit syscall) into $v0 before executing the syscall instruction. We can use the li instruction again in order to do this:

```
# Your Name -- DATE
# add.asm-- A program that computes the sum of 1 and 2,
#  leaving the result in register $t0.
# Registers used:
#  t0 - used to hold the result.
#  t1 - used to hold the constant 1.
       main:            # MARS starts execution at main.
       li $t1, 1        # load 1 into $t1.
       addi $t0, $t1, 2  # $t0 = $t1 + 2.
       # put it into $t0.
       li $v0, 10       # syscall code 10 is for exit.
       syscall          # make the syscall.
# end of add.asm
```

Exercise 1:

1- a) Write a program which sums of two numbers specified by the user at runtime

We'll write a program named add2.asm that computes the sum of two numbers specified by the user at runtime, and displays the result on the screen.
The algorithm this program will follow is:
1. Read the two numbers from the user. We'll need two registers to hold these two numbers. We can use $t0 and $t1 for this.
    a. Get first number from user, put into $t0.
        i. load syscall read_int into $v0,
        ii. perform the syscall,
        iii. move the number read into $t0.
    b. Get second number from user, put into $t1
        i. load syscall read_int into $v0,
        ii. perform the syscall,
        iii. move the number read into $t1.
2. Compute the sum. We'll need a register to hold the result of this addition. We can use $t2 for this.
3. Print the sum.
4. Exit. We already know how to do this, using syscall.

```
# Your Name -- DATE
  # add2.asm-- A program that computes and prints the sum
  # of two numbers specified at runtime by the user.
  # Registers used:
  #  $t0 - used to hold the first number.
  #  $t1 - used to hold the second number.
  #  $t2 - used to hold the sum of the $t1 and $t2.
  #  $v0 - syscall parameter and return value.
  #  $a0 - syscall parameter.
  main:
   ## Get first number from user, put into $t0.
   |---------------------------------------|
   |Put your code here                     |
   |---------------------------------------|

   ## Get second number from user, put into $t1.
   |---------------------------------------|
   |Put your code here                     |
   |---------------------------------------|

   # compute the sum.
   |---------------------------------------|
   |Put your code here                     |
   |---------------------------------------|

   ## Print out $t2.
   |---------------------------------------|
   |Put your code here                     |
   |---------------------------------------|

   li $v0, 10   # syscall code 10 is for exit.
   syscall      # make the syscall.
  # end of add2.asm.
```

1-b) Modify the above program to show sum of two numbers specified by the user at runtime in Hexadecimal.