Lab 1 : An Introduction to MARS, MIPS Assembler and Runtime Simulator

Name:

Kraig Orcutt

Sign the following statement:

On my honor, as an Aggie, I have neither given nor received unauthorized aid on this academic work

Kraig Orcutt

1 Objective

The objective of this lab is to give you hands on experience with MARS; a MIPS simulator.

2 Setup

For those of you unfamiliar with Linux (or Unix) and the command line interface (CLI), you should review the basics on the following website:

http://www.ee.surrey.ac.uk/Teaching/Unix/index.html

For this lab you will be using the *M*IPS *A*ssembler and *R*untime *S*imulator (MARS). MARS will serve as an integrated development environment (IDE) for writing MIPS as-

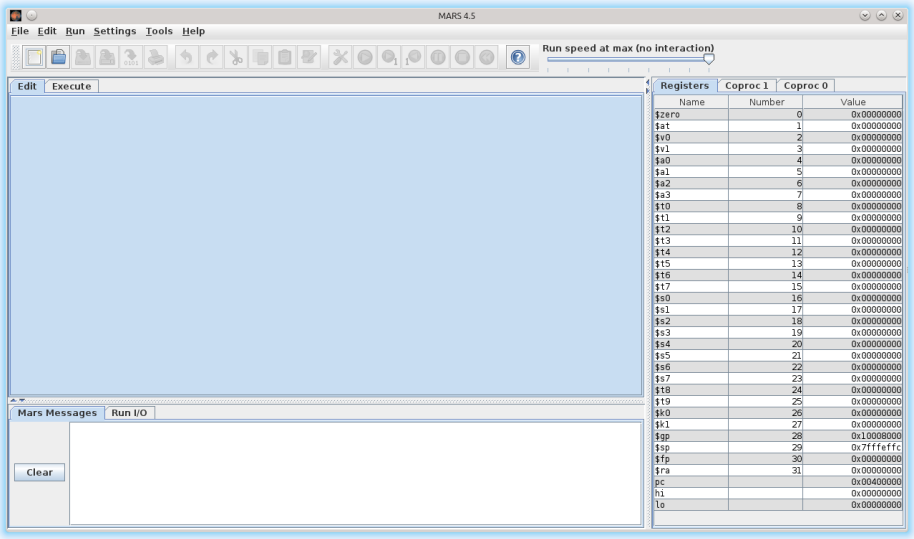

Fig. 1: The *M*IPS *A*ssembler and *R*untime *S*imulator (MARS) frontend IDE.

sembly, an assembler (tool which converts assembly to an executable binary), and a runtime architectural-level simulator for a MIPS processor.

Before you begin you must first download and setup MARS. MARS is a java program so it should work on many different machine architectures, as long as they have a JAVA runtime environment installed. Start by downloading the MARS Java archive (.jar file) from either eCampus or from the following website:

http://courses.missouristate.edu/KenVollmar/MARS/download.htm

In most cases (assuming JAVA is setup on your machine and your operating system is windows), you should be able to click on this file and start up the MARS IDE frontend. On Linux start MARS by first opening up a terminal window, then invoking the following command:

> java -jar /softwares/courses/350/Mars4_5.jar

Assuming everything works the window shown in Figure 1 should appear. If you have problems getting the MARS executable running, ask for assistance from your TA.

At this point you should familiarize yourself with the MARS IDE. Read the information and tutorial on MARS found at the following website:

http://courses.missouristate.edu/KenVollmar/MARS/tutorial.htm

Run through the tutorial before attempting the rest of the lab.

3 Loading and executing a simple MIPS assembly program

Consider the program described below.

```
1       .text                   # text section
2       .globl main             # call main by MARS
3       main:
4       addi $t1, $0, 10            # load immediate value (10) into $t1
5       addi $t2, $0, 11            # load immediate value (11) into $t2
6       add $t3, $t1, $t2       # add two numbers into $t3
7       jr $ra                  # return from main; return address stored in $ra
```

The MIPS assembly language has a few simple rules:

- Text that follows after a sharp sign (#) is a comment.
- A line that starts with a period (.) is an assembler directive. Assembler directives are a way of instructing the assembler how to translate a program but do not produce machine instructions. For example at line 6 in the code below, *.globl main* defines main as a global variable.
- Text followed by a colon (:) defines a label. It is similar to defining variables in any high-level language, for example *int var* in C-language.

Use the MARS IDE to type the program and save it as "*Lab1a.s*". **Make sure you do not enter the line numbers**.

Note: MARS accepts files with either ".asm" or ".s" extensions. We will be using '.s' notation. You may refer to page B-47 in the Appendix of the Text Book for more information on Assembler Syntax and Directives.

1. Assemble the program;
2. Execute the program step by step;
3. Examine the content of registers `$t1,$t2`, and `$t3`;
4. Copy the hexadecimal representation of the instruction on line 6 (`add $t3, $t1, $t2`);
   0x00000015
5. Modify the program such that in the end the register `$t3` is equal to `4*$t1+2*$t2` (use multiple `add` instructions);
6. Run and validate the modified program step by step and examine the content of the registers `$t1,$t2`, and `$t3`. Demonstrate your progress to the TA;

4 Loading and executing another sample MIPS assembly program

Consider the program described below. The following program takes two values, X and Z, and computes 2 values.

```
1   .text
2   .globl main
3   main:
4   # replace X with any number you like
5   addi $t0, $0, X
6   # make sure you replace Z with the first digit of your UIN
7   srl $t1,$t0, Z      #computation 1, result is in $t1
8   sll $t2,$t0, Z      #computation 2, result is in $t2
9   # check the content of $t1 and $t2
10  jr $ra              # return from main; return address stored in $ra
```

1. Type the program above into a file called "*Lab1b.s*".
2. Enter an integer of your choice in X and Z.

   Value of X=[ ];
   Value of Z=[ ];
   Content of `$t1`=[ ];
   Content of `$t2`=[ ].

3. Can you figure out what computation the program is doing? Write the mathematical representation of the computations in terms of X and Z.

   `$t1` := [ ]

   `$t2` := [ ]

5 Loading and executing a sample MIPS assembly program with Input/Output operations

We would like to execute this program multiple times with different values for X. Execution will be much more convenient if the user is prompted for the value of X and the program prints out the result. To achieve this, we will use system calls to interact with the user. System calls are a form of function call that runs routines defined by the Operating System. In MARS, the register $v0 defines which function will be executed, and $a0 is its argument. For example, function 4 will print text to the console, using $a0 as the address of the string to print. You can refer to page B-43 in the appendix of the Text Book for a list of System Calls that MARS supports.

Enter the program below into a file called "*Lab1c.s*"

```
1   .data
2   msg1:       .asciiz "Please enter an integer number: "
3   msg2:       .asciiz "\tFirst result: "
4   msg3:       .asciiz "\tSecond result: "
5   .text
6   .globl main
7   # Inside main there are some calls(syscall) which will change the
8   # value in register $ra which initially contains the return
9   # address from main. This needs to be saved.
10  main:
11  addu $s0, $ra, $0   # save $31 in $16
12  li $v0, 4           # system call for print_str
13  la $a0, msg1                # address of string to print
14  syscall
15  # now get an integer from the user
16  li $v0, 5           # system call for read_int
17  syscall             # the integer placed in $v0
18
19  # do some computation here with the integer
20  addu $t0, $v0, $0   # move the number in $v0 to $t0
21  # make sure you replace Z with the first digit of your UIN
22  sll $t1,$t0, Z      #computation 1, result is in $t1
23  srl $t2,$t0, Z      #computation 2, result is in $t2
24
25  # print the first result
26  li $v0, 4           # system call for print_str
27  la $a0, msg2                # address of string to print
28  syscall
29  li $v0, 1           # system call for print_int
30  addu $a0, $t1, $0   # move number to print in $a0
31  syscall
32  # print the second result
33  li $v0, 4           # system call for print_str
34  la $a0, msg3                # address of string to print
35  syscall
36  li $v0, 1           # system call for print_int
37  addu $a0, $t2, $0   # move number to print in $a0
38  syscall
39  # restore now the return address in $ra and return from main
40  addu $ra, $0, $s0   # return address back in $31
41  jr $ra                      # return from main
```

Make sure you put the first digit of your UIN, instead of Z in lines 22 and 23

1. Enter an integer of your choice and the program should print two results for you. Keep repeating the process for 5 different integers, and fill the table below:

   Note: You can re-run the program without reloading it by selecting "Simulator/Clear

Registers" prior to re-running. Clear Registers resets all the register values to their defaults (including PC).

Value of Z =		
Integer	Result 1	Result 2

2. Next, you learn how to use the simulator to debug your program. First, make sure that the checkbox "Settings/Show Label Window" is checked and fill out the following table. This is the list of where parts of your program are located in memory.

Symbol	Address

3. Step through your program using the "Step" menu button (step one instruction at a time). With each step, you will see lines in the *Text* window become highlighted. The highlighted data contains the Address, Machine Instruction, and the line of code currently executing. Step through the first 5 lines of *your* code and fill in the following table:

Address (PC)	Native Instruction	Source code

4. Re-initialze your program, and set a breakpoint at line 26 (*li $vo, 4*). Find the appropriate line with this instruction in the *Text* window and note its memory address. Right click on this line and select "Set Breakpoint". Run your program. When it hits the breakpoint, examine registers $8, $9, and $10. Try entering different numbers into the program to fill in the following table:

Number you entered	$8	$9	$10

5. Execute and validate the assembly program created in prelab that swaps the contents of two variables stored in registers $4 and $5.

   Note: You can save the info in the "Mars Messages" and "Run I/O" by highlighting it and cut and pasting it into a text editor.

6 Deliverables

- Submit completed copy of this lab manual.
- Include the following in a compressed file (.zip format) to your TA:
  - The source code for all .s files.
  - All log files (cut and pastes of the "Mars Messages" and "Run I/O" windows).