Informatics 2C Computer Systems - Lab 1

1 Introduction

1.1 MIPS Programming and the MARS IDE

The objective of this lab is to introduce you to the basics of MIPS assembly programming using MARS, an IDE (Integrated Development Environment) for MIPS assembly language programming.

1.2 Background Material on MIPS

Beyond the lecture slides and notes, you should refer to the appendix of the Patterson and Hennessy course textbook on MIPS. This is Appendix A in 5/e[1], and Appendix B in 4/e. Appendix A from 3/e is also available online. Sections 1-6 of this appendix provide useful background material, while Sections 10 and 9 (the part on System Calls) serve as a valuable reference on the MIPS assembly language assembler syntax.

Note that various parts of the appendix, particularly Section 9, refer to the SPIM software simulator for MIPS. MARS is designed as an extension of SPIM, so all discussions of SPIM are fully relevant to MARS.

1.3 Installing and Running MARS

MARS is a Java application and therefore should run on any platform supporting Java. The latest version (currently V4.5) can be downloaded from the Download page on on the MARS website. This version of MARS requires Java 1.5 or later.

The download page gives various options for running the downloaded JAR (Java ARchive) file. Option A does not work on DICE machines. Option B does. While the descriptions of Options B and C refer to a DOS shell, they work equally well when using a Unix-flavour shell on Linux or Mac OS X.

1.4 Alternative MIPS Simulators

The SPIM simulator referred to in the book is installed on all DICE computers and are available as the command-line commands `spim` and `xspim`. MARS is a much nicer tool however, and its use is strongly encouraged for this class.

2 Using MARS

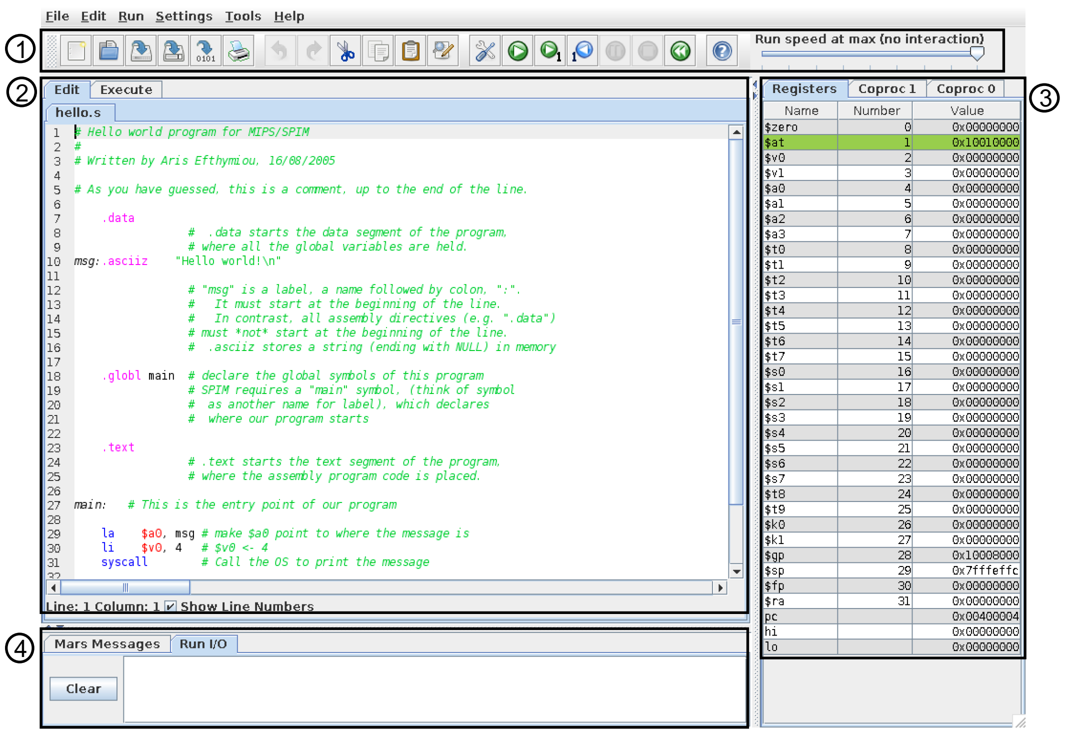
After launching MARS you should see a screen similar to Figure 1.

1. Top Menu Here you will find the most commonly used tools and operations. The most important ones are shown in Figure 2.

2. Edit and Execute The Edit tab is where the currently open files can be edited. The Execute tab shows the Text Segment and Data Segment during execution. The Text Segment shows generation Machine code instruction after assembly. The Data Segment shows memory contents at the corresponding addresses. You can view the contents as hexadecimal values or ASCII using the checkboxes at the bottom of the tab.

3. Registers You can use this pane to follow register contents during execution.

[1]Also Appendix A in 2/e and 3/e.



```
# Hello world program for MIPS/SPIM
#
# Written by Aris Efthymiou, 16/08/2005

# As you have guessed, this is a comment, up to the end of the line.

    .data
                #  .data starts the data segment of the program.
                # where all the global variables are held.
msg: .asciiz    "Hello world!\n"

                # "msg" is a label, a name followed by colon, ":".
                #   It must start at the beginning of the line.
                #   In contrast, all assembly directives (e.g. ".data")
                # must *not* start at the beginning of the line.
                #  .asciiz stores a string (ending with NULL) in memory

    .globl main  # declare the global symbols of this program
                # SPIM requires a "main" symbol, (think of symbol
                #  as another name for label), which declares
                #  where our program starts

    .text
                # .text starts the text segment of the program.
                # where the assembly program code is placed.

main:   # This is the entry point of our program

    la    $a0, msg # make $a0 point to where the message is
    li    $v0, 4   # $v0 <- 4
    syscall        # Call the OS to print the message
```

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400004
hi		0x00000000
lo		0x00000000

Figure 1: A snapshot of the MARS startup window.

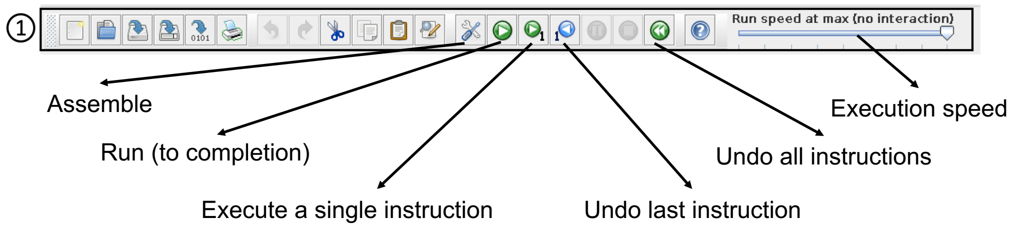


Figure 2: Commonly used operations in MARS.

4. MARS Messages and Run I/O The Mars Messages tab is used for displaying assembly or runtime errors and informational messages. The Run I/O tab is used at runtime for displaying console output and entering console input as program execution progresses.

Breakpoints Breakpoints are a very useful feature of MARS. Breakpoints, together with single-stepping and backward single-stepping are very powerful when trying to understand or debug a program. To create a breakpoint, navigate to the Execute tab (after assembling) and check the box labelled "Bkpt" to the left of the desired instruction. You can create multiple breakpoints throughout your program in order to ease the debugging process.

3 MIPS Assembly Examples

Download the example soure files provided and open them using the "Load file" option in MARS. Assemble the program using the "Assemble" button and then run using "Run" and/or "Step" through the generated instructions.

Note: You must check the box labelled "Initialize program counter to global main if defined" under the Settings menu.

hello.s A small assembly program that prints out "hello world!" and then exits. The program uses MIPS assembly pseudo-instruction `la $a0, msg`, which loads register `$a0` with the memory address where msg is held. Printing (and exiting) is done by the special instruction `syscall`. For now, think of `syscall` as providing a mechanism for making calls to special procedures built-in to MARS. You will get a more complete picture when exceptions and interrupts are discussed later in this course.

The exact task that `syscall` performs depends on the value stored into register `$v0`. To find out what other `syscall` options are available, see page A-44 of Appendix A of the Patterson and Hennessy book (3rd ed). We will be using some of these for input/output. We will not be using the `syscall` options provided in MARS that are not provided in SPIM.

hexOut.s This program converts a 32-bit decimal number that the user inputs into hexadecimal format (8 hex digits). The first two `syscalls` prompt the user for a number and get the number from the keyboard. Then a message is printed. The rest of the program extracts each hexadecimal digit from the provided number, converts it into the corresponding ASCII character and prints it (using yet another `syscall`). This is done eight times for the eight hex digits.

Assemble and step through the program to find out exactly what each instruction is doing. Try setting a breakpoint at the instruction labelled `loop`, so that you can let the program run through quickly the part where it asks for input and focus on what happens in the loop.

4 Tasks

You can now practise writing your own assembly programs. Here are some suggestions:

- Modify hexOut.s so that leading zeros are not printed.
- Modify hexOut.s so that when a negative number is entered, it is printed as '-', followed by the two's complement of the number (in hex again).
- Based on hexOut.s write a program, binOut.s that asks for a number and prints it in binary format, digit by digit.