What you will learn

In this chapter you will learn:

1. to download, install, and run the MARS IDE.
2. what are registers, and how are they used in the CPU.
3. register conventions for MIPS.
4. how memory is configured for MIPS.
5. to assemble and run a program in MARS.
6. the syscall instruction, and how to pass parameters to syscall.
7. what immediate values are in assembly language.
8. assembler directives, operators, and instructions.
9. to input and output integer and string data in MIPS assembly.

Chapter 2 First Programs in MIPS assembly

This chapter will cover a first program that is often implemented when writing in a new language, *Hello World*. This program is significant in any language because it covers the most fundamental concepts any program can achieve; creating and executing a program that can read data in and print results out. This exercise is important because it covers an Integrated Development Environment (IDE) which will allow the programmer to edit the program and to create resultant execution of that program. Being able to input data and output results covers a basic understanding of registers, I/O mechanisms, and provides a mechanism to test algorithms by allowing users to enter data and see if the result is what is expected.
This first program is particularly important because the concepts of statements and variables require a much more in-depth knowledge of the language and platform. This chapter is intended to prepare the reader for the rigors of programming MIPS assembly by leading the reader step-by-step into a first working program.

Chapter 2. 1 The MARS IDE

This text will use an IDE called the *MIPS Assembler and Runtime Simulator* (MARS). There are several MIPS simulators available, some for educational use, and some for commercial use. However, in the opinion of this author, MARS is the easiest to use, and provides the best tools for explaining how MIPS assembly works.

MARS was written by Pete Sanderson and Kenneth Vollmar, and is documented at the site http://courses.missouristate.edu/kenvollmar/mars/index.htm, and should be downloaded from this site. MARS is an executable jar file, so you must have the Java Runtime Environment (JRE) installed to run MARS (https://www.java.com/en/download/manual.jsp).

Instructions for running MARS are on the download page. When it is started, a page which is like the following should come up. This will be the starting point for the material in the rest of the chapter.

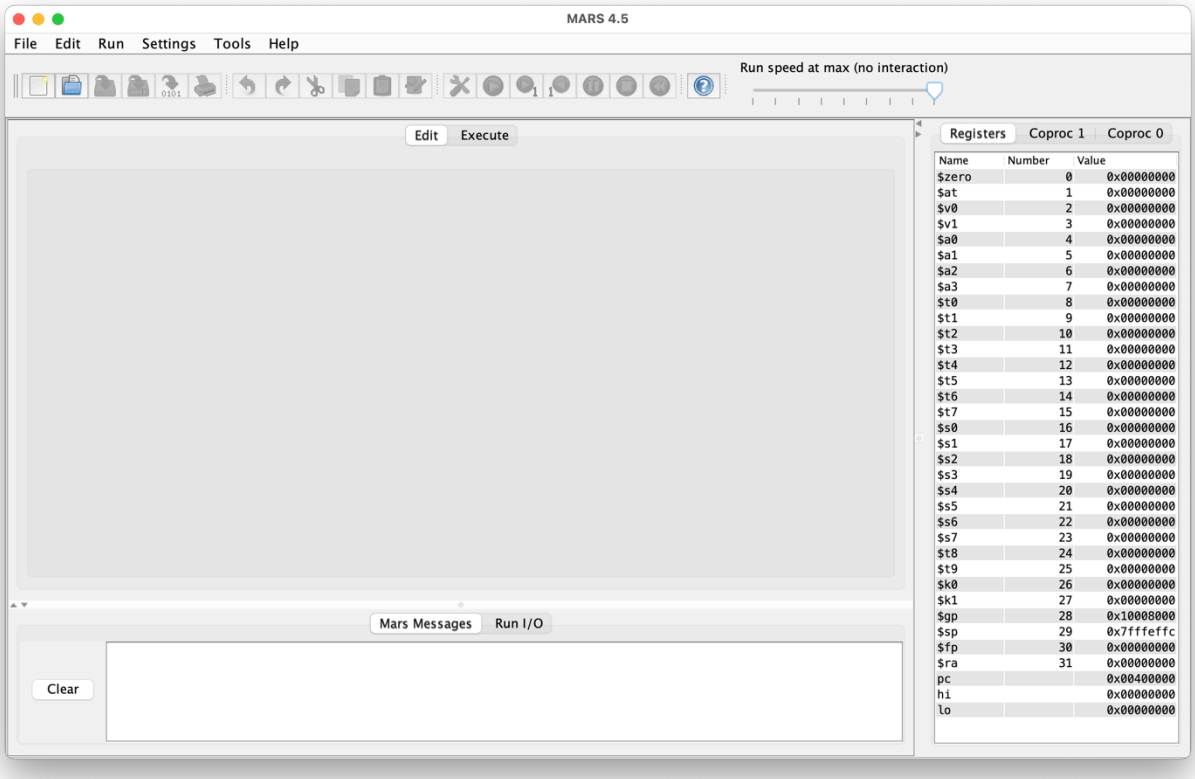


Figure 0-1: Initial Screen of the MARS IDE

Chapter 2. 2 MIPS and memory

(Note: the next two sections provide background material to assist the reader in understanding the MIPS programs later in the chapter. It might be helpful to first read lightly through this material, then implement the programs. This material is difficult to understand, even for some experienced programmers. It is anticipated that the reader will have to refer to this section throughout the reading of the rest of the book, and quite possibly for future reference in programming in other languages. How memory is implemented and used is a complex and interesting topic, so at least some level of understanding is foundational for the study of Computer Science.).

It is not unusual for novice programmers to have no concept of memory except as a place to store variables. For a novice this is sufficient, but any real program will require that a programmer have at least a basic knowledge of the types of memory that are used, and the characteristics of each. For example, programs that use concurrency are difficult to implement without problems if memory is not understood. Some very powerful design patterns, such as an Immutable Objects, Singletons, or a State Pattern, cannot be understood properly without a knowledge of the characteristics of different types of memory. So, every programmer should have at least a basic understanding of how the different types of memory that are used in nearly every computer platform.

One advantage of learning assembly language programming is that it directly exposes many of the types of memory (heap, static data, text, stack, and registers) used in a program, and forces

the programmer to deal with them. Some memory concepts are more appropriately covered in other courses, such as virtual memory (cache, RAM, and disk) which are generally covered in an OS class.

Chapter 2.2. 1 Types of memory

To a programmer, memory in MIPS is divided into two main categories. The first category, memory that exists in the Central Processing Unit (CPU) itself, is called *register memory* or more commonly *registers*. Register memory is very limited and contained in what is often called a *register file* on the CPU. This type of memory will be called *registers* in this text.

The second type of memory is what most novice programmers think of as memory and is often just called *memory*. Memory for the purposes of this book is where a HLL programmer puts instructions and data. HLL programmers have no access to registers, and so generally have no knowledge of their existence. So, from a HLL programmer's point of view, anything stored on a computer is stored in memory.

The non-register memory space of a modern computer is divided into many different categories, each category having different uses. The different areas of memory studied in detail in this text will be the text, static data, heap, and stack sections. Other areas also exist, though this text will not cover them.

Caveats about memory

Students are always complaining that in Computer Science the same terms refer to different things. For example, a binary heap and heap memory are both heaps, but they are completely unrelated terms. As with any study of a complex organization, definitional problems will exist in the study of memory. Therefore, it is important to be flexible and understand the contextual meaning or a term and not simply the words. Also keep in mind that external sources of information, like the WWW, may use different terminology, or even the same words with different meanings. So, when researching memory, keep the following points in mind.
First is that the view of memory in this text is the programmer's view. The actual implementation of the memory is likely to include virtual memory and several layers of cache. All of this will be hidden from the programmer, so the complexities of the implementation of memory are not considered in this book.

The second thing to keep in mind is that this text will present an older model of memory which is a single threaded process and does not have virtual program execution (such as the Java Virtual Machine). Memory can and does become much more complex than the model given here, but this model is already complex enough, and meets the needs of our assembler programs. So, it is a good place to begin understanding memory.

Chapter 2.2. 2 Overview of a MIPS CPU

The following diagram shows a simple design for a 3-Address Load/Store computer, which is applicable to a MIPS computer. This diagram will be used throughout the text to discuss how MIPS assembly is dependent on the computer architecture. To begin this exploration, the components of a CPU and how they interact is explained.

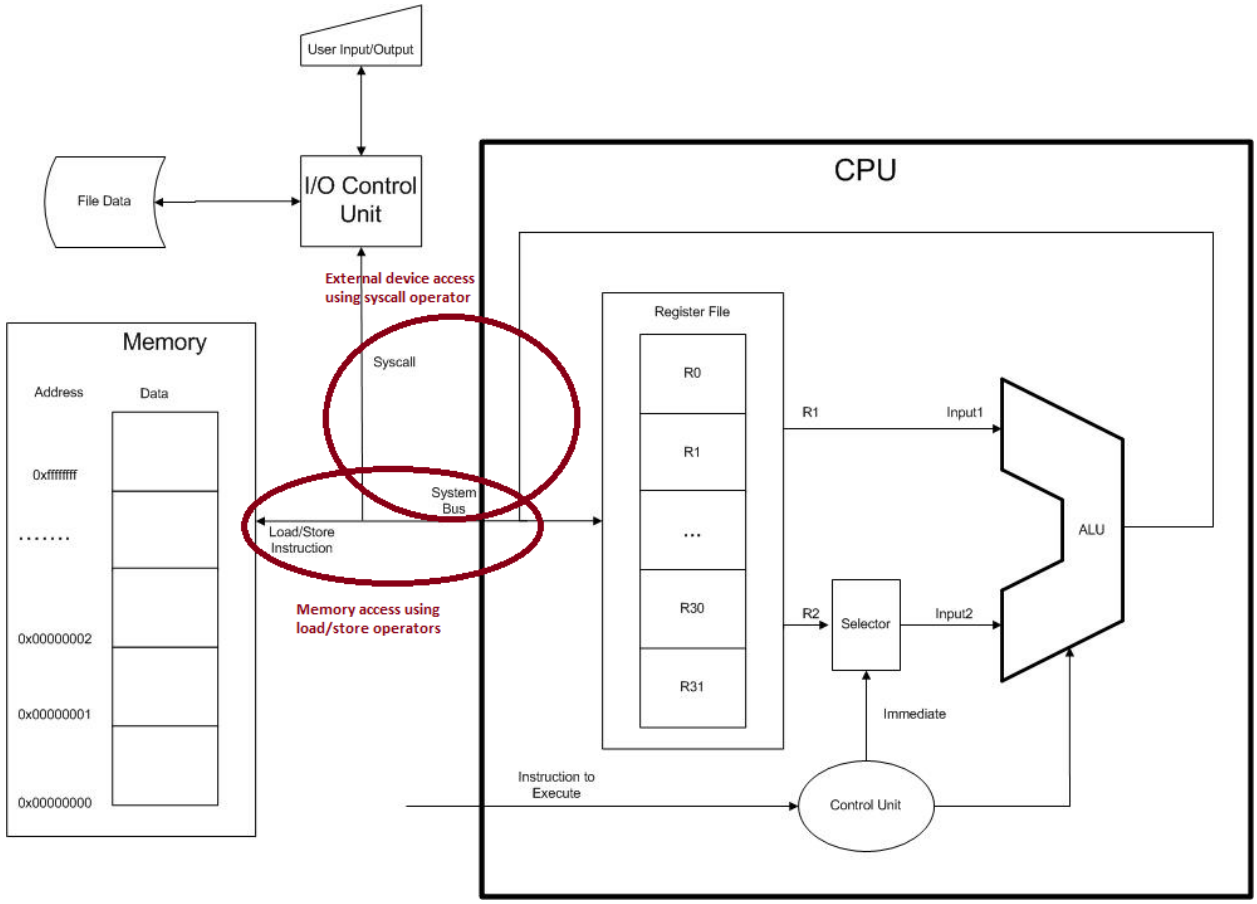


Figure 0-2: 3-address store/load computer architecture

All CPU architectures contain 3 main units. The first is the ALU, which performs all calculations such as addition, multiplication, subtraction, division, bit-shifts, logical operations, etc. Except for instructions which interface to units not on the CPU, such as memory access or interactions with the user or disks, all operations use the ALU. In fact, it is reasonable to view the basic purpose of the CPU as doing some sort of ALU operation on values from two registers and storing the result into a third register.

This interaction of the registers and the CPU helps to explain the purpose of the registers. Registers are a limited amount of memory which exists on the CPU. No data can be operated on in the CPU that is not stored in a register. Data from memory, the user, or disk drives must first be loaded into a register before the CPU can use it. In the MIPS CPU, there are only 32 registers, each of which can be used to store a single 32-bit value. Because the number of these registers is so limited, it is vital that the programmer use them effectively.

To use data from memory, the address, and data to be read/written is placed on the system bus using a load/store command and transferred to/from the memory to the CPU. The data and address are normally placed on the system bus using a Load Word, `lw`, or Store Word, `sw`, operation. The data is then read/written from/to memory to/from a register. To use more than 32 data values in a program, the values must exist in memory, and must be loaded to a register to use.

There is a second way to read/write data to/from a register. If the data to be accessed is on an external device, such as a user terminal or disk drive, the `syscall` operator is used. The `syscall` operator allows the CPU to talk to an I/O controller to retrieve/write information to the user, disk drive, etc.

The final part of a CPU is the Control Unit (CU). A CU controls the mechanical settings on the computer so that it can execute the commands. The CU is the focus of a class which is often taught with assembly language, the class being Computer Architecture. This class will not cover the CU in anything but passing detail.

Chapter 2.2. 3 Registers

Registers are a limited number of memory values that exist directly in the CPU. To do anything useful with data values in memory, they must first be loaded into registers. This will become clearer in each subsequent chapter of this text, but for now it is just important to realize that registers are necessary for the CPU to operate on data, and that there are a limited number of them.

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

Table 0-1: Register Conventions

Because the number of registers is very limited, they are carefully allocated and controlled. Certain registers are to be used for certain purposes, and the rules governing the role of the register should be followed. The preceding list is the 32 registers (numbered 0…31) that exist in a MIPS CPU, and their purposes. As with much else in this chapter, the meaning of each of the registers will become clear later in the text.

The conventions for using these registers is outlined below. Note that in some special situations, the registers will take on special meaning, such as with exceptions. These special meanings will be covered when they are needed in the text. Also note that in MARS only the lower-case name of the register is valid (for example `$t0` is valid, `$T0` is not).

- `$zero` (`$0`) – a special purpose register which always contains a constant value of 0. It can be read but cannot be written.
- `$at` (`$1`) – reserved for the assembler. If the assembler needs to use a temporary register (e.g., for pseudo instructions), it will use `$at`, so this register is not available for programmer use.
- `$v0-$v1` (`$2-$3`) – normally used for return values for subprograms. `$v0` is also used to input the requested service to `syscall`.
- `$a0-$a3` (`$4-$7`) – used to pass arguments (or parameters) into subprograms.
- `$t0-$t9` (`$8-$15, $24-$25`) – used to store temporary variables. The values of temporary variables can change when a subprogram is called.
- `$s0-$s7` (`$16-$23`) – used to store saved values. The values of these registers are maintained across subprogram calls.
- `$k0-$k1` (`$26-$27`) – used by the operating system and are not available for programmer use.
- `$gp` (`$28`) – pointer to global memory. Used with heap allocations.
- `$sp` (`$29`) – stack pointer, used to keep track of the beginning of the data for this method in the stack.
- `$fp` (`$30`) – frame pointer, used with the `$sp` for maintaining information about the stack. This text will not use the `$fp` for method calls.
- `$ra` (`$31`) – return address: a pointer to the address to use when returning from a subprogram.

Chapter 2.2. 4 Types of memory

MIPS implements a 32-bit flat memory model. This means as far as a programmer is concerned, memory on a MIPS computer starts at address 0x00000000 and extends in sequential, contiguous order to address 0xFFFFFFFF. The actual implementation of the memory, which is far from sequential and contiguous, is not of interest to the programmer. The operating system will reliably give the programmer a view of the memory which is flat.

A 32-bit flat memory model says that a program can *address* (or find) 4 Gigabytes (4G) of data. This does not mean that all that memory is available to the programmer. Some of that memory is used up by the operating system (called *kernel data*), some of it used by the I/O subsystem, etc. Figure 2.3 diagrams how the 4G of memory is configured in a MIPS computer. In this chapter, only static data and program text memory will be used. Later chapters will cover data such as stack and heap memory. The types of memory used by MIPS are the following:

- **Reserved** – reserved for the MIPS platform. Memory at these addresses is not useable by a program.
- **Program Text** (addresses 0x0040 0000 - 0x1000 0000) – This is where the machine code representation of the program is stored. Each instruction is stored as a *word* (32 bits or 4 bytes) in this memory. All instructions fall on a *word boundary*, which is a multiple of 4 (0x0040 0000, 0x0040 0004, 0x0040 0008, 0x0040 000C, etc.).
- **Static Data** (addresses 0x1001 0000 - 0x1004 0000) – This is data which will come from the *data segment* of the program. The size of the elements in this section are assigned when the program is created (assembled and linked) and cannot change during the execution of the program.
- **Heap** (addresses 0x1004 0000 – until stack data is reached, grows upward) – Heap is dynamic data which is allocated on an as-needed basis at run time (e.g. with a *new* operator in Java). How this memory is allocated and reclaimed is language specific. Data in heap is always globally available.
- **Stack** (addresses 0x7FFF FE00 – until heap data is reached, grows downward) – The program stack is dynamic data allocated for subprograms via *push* and *pop* operations. All method local variables are stored here. Because of the nature of push and pop operations, the size of the stack to create must be known when the program is assembled.
- **Kernel** (addresses 0x9000 0000 - 0xFFFF 0000) – Kernel memory is used by the operating system, and is not accessible to the user.
- **MMIO** (addresses 0xFFFF 0000 - 0xFFFF 0010) – Memory Mapped I/O, which is used for any type of external data not in memory, such as monitors, disk drives, consoles, etc.

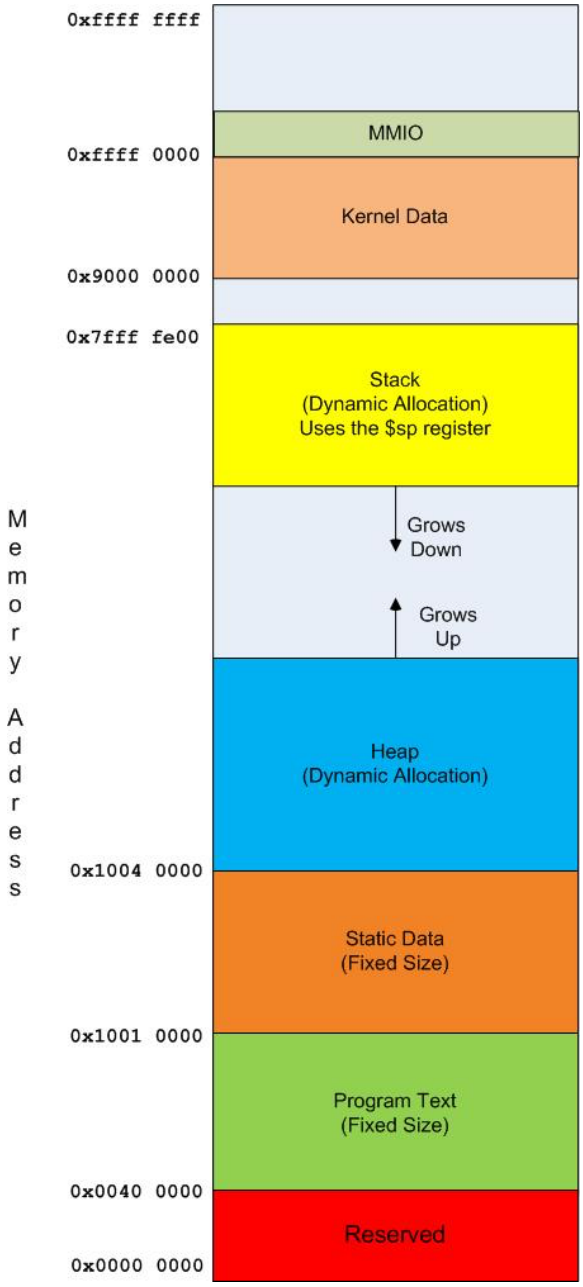


Figure 0-3: MIPS memory configuration

Most readers will probably want to bookmark this section of the text and refer to it when new memory types or access methods are covered.

Chapter 2. 3 First program in MIPS assembly

The following is a first MIPS assembly program. It prints out the string "`Hello World`". To run the program, first open the MARS program/app. Choose the `File->New` menu option, which will open an edit window, and enter the program. How to run the program will be covered in the following the section. You can cut and paste the program into your MARS app.

```
# Program File: Program2-1.asm
# Author: Charles Kann
# Purpose: First program, Hello World
.text                    # Define the program instructions.
main:                    # Label to define the main program.
    li $v0,4             # Load 4 into $v0 to indicate a print string.
    la $a0, greeting     # Load the address of the greeting into $a0.
    syscall              # Print greeting.  The print is indicated by
                         # $v0 having a value of 4, and the string to
                         # print is stored at the address in $a0.
    li $v0, 10           # Load a 10 (halt) into $v0.
    syscall              # The program ends.
.data                    # Define the program data.
greeting: .asciiz "Hello World" #The string to print.
```

Program 0-1: Hello World program

Once the program has been entered into the edit window, it needs to be **saved**, then assembled. The option to assemble the program is shown circled in red below.

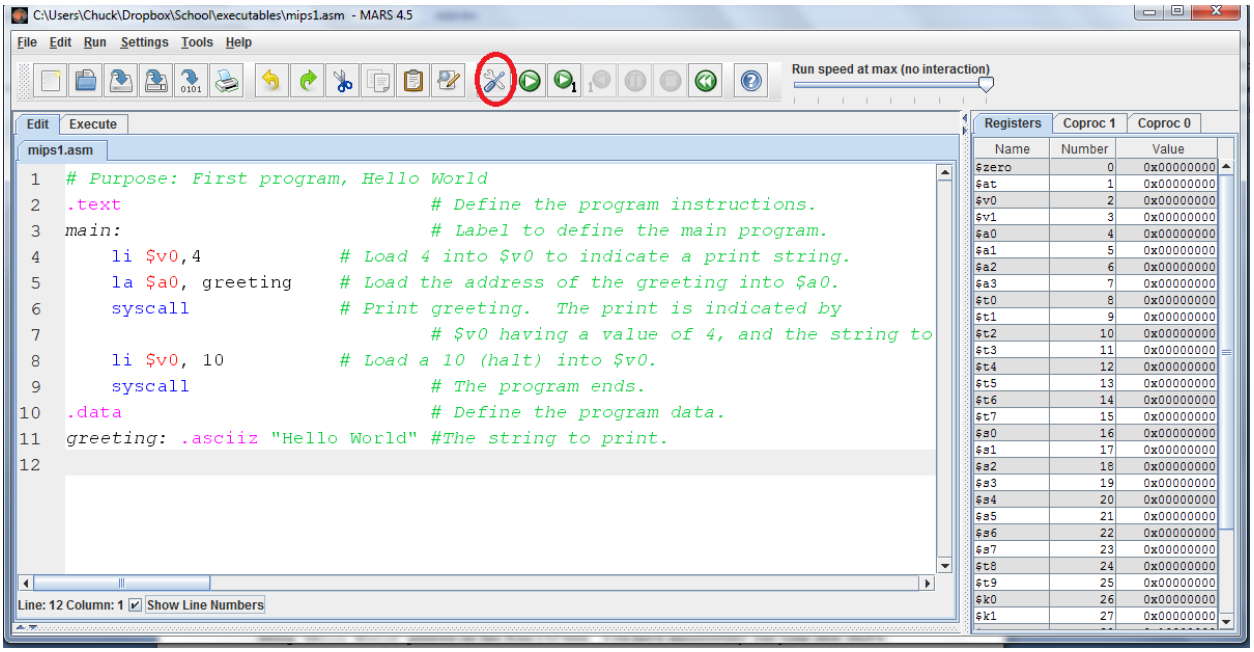


Figure 0-4: Assembling a program

If you entered the program correctly you should get the edit screen below. If you made any errors, the errors will be displayed in a box at the bottom of the screen. **If the option to assemble the program is greyed out, make sure to save your program to a file first.**
To run the program, click the green arrow button circled in the figure below. You should get the string "`Hello World`" printed on the Run I/O box. You may need to go back to the Edit tab and pull up from the bottom to view the Run I/O box. You have successfully run your first MIPS program.

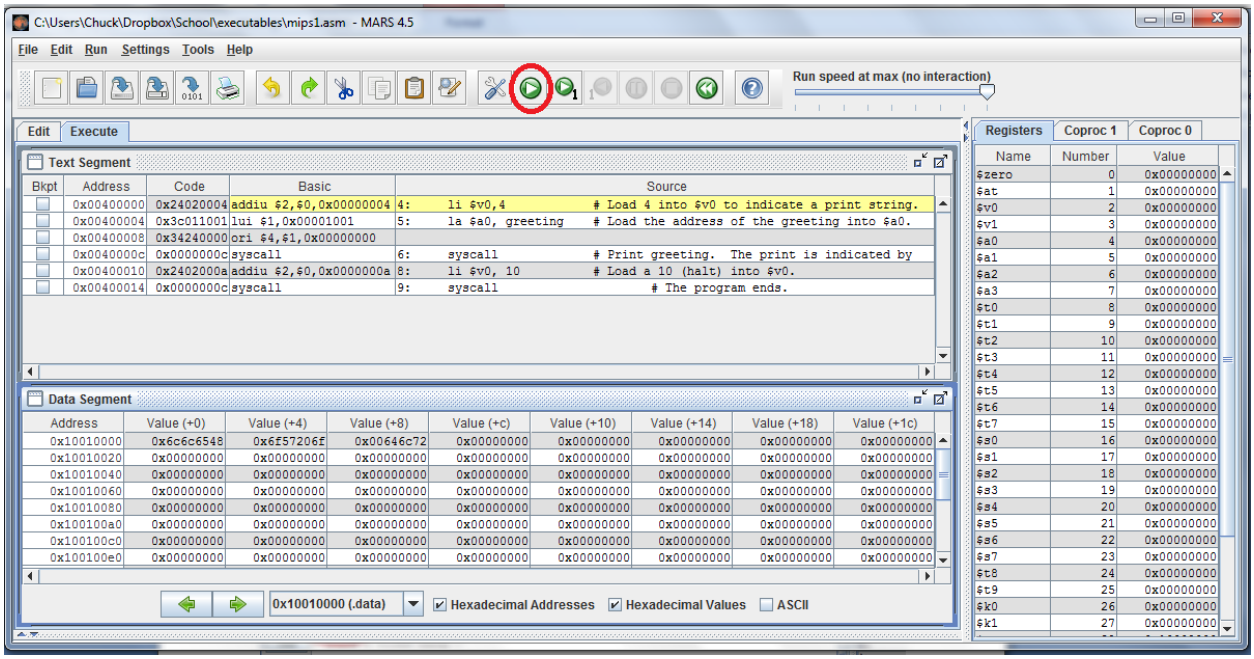


Figure 0-5: Running a program

Chapter 2.3. 1 Program 2-1 Commentary

Program 2-1 was used to show how to compile and run a program. This section will go over the details of this first program and help the reader understand it.

- MIPS assembler code can be indented as left white space on a line is ignored. All instructions must be on a single line, The # means any text from the # to the end of a line is a comment and to be ignored. Strings are denoted by " " marks around the string.

- Note the comments at the start of the file. These will be called a *file preamble* in this text. At a minimum all programs should contain at least these comments. The name of the file should be documented, as unlike some HLL such as Java, the name of the file is nowhere implied in the program text. It is very easy to lose files in this situation, so the source code should always contain the name of the file. The file preamble should also contain the programmer who created the code, and a short description of why this program was written.

- Assembly language programs are not *compiled*, they are *assembled*. So, a program does not consist of statements and blocks of statements as in a HLL, but several instructions telling the computing machine how to execute. These instructions are very basic, such as `li $v0, 4` (put the value 4 in `$v0`). Because this is a complete change of perspective from a HLL, it is useful to explicitly make this point, and for the reader to take note of it.

- In MIPS and most assembly languages in general, a "." before a text string means the token (string) that follows it is an as*sembler directive*. In this program the `.text` directive means the instructions that follow are part of a program text (i.e., the program),

and to be assembled into a program and stored in the text region of memory. The `.data` directive means that what follows is program data, and to be stored in the static data region of memory. The `.asciiz` directive tells the assembler to interpret the data which follows it as an ASCII string. Typing a "." in the MARS edit window will give you a tooltip with all the MIPS assembler directives.

- In MIPS assembler any text string followed by a ":" is a label. A label is just a marker in the code that can be used in other statements, as the `la $a0 greeting` instruction says to load the text at the label greeting into the `$a0` register. Note that data labels are not equivalent to variables. Labels are just markers in the program, and nothing more. Variables have a type, and there is no typing of any data done in assembler.

- The label `main:` does not need to be included as MARS assumes the program begins at the first line in the assembled program. But it is nice to label the starting point, and generally most runtimes will look for a global symbol name *main* as the place to begin execution. So, in this text it will just be included.

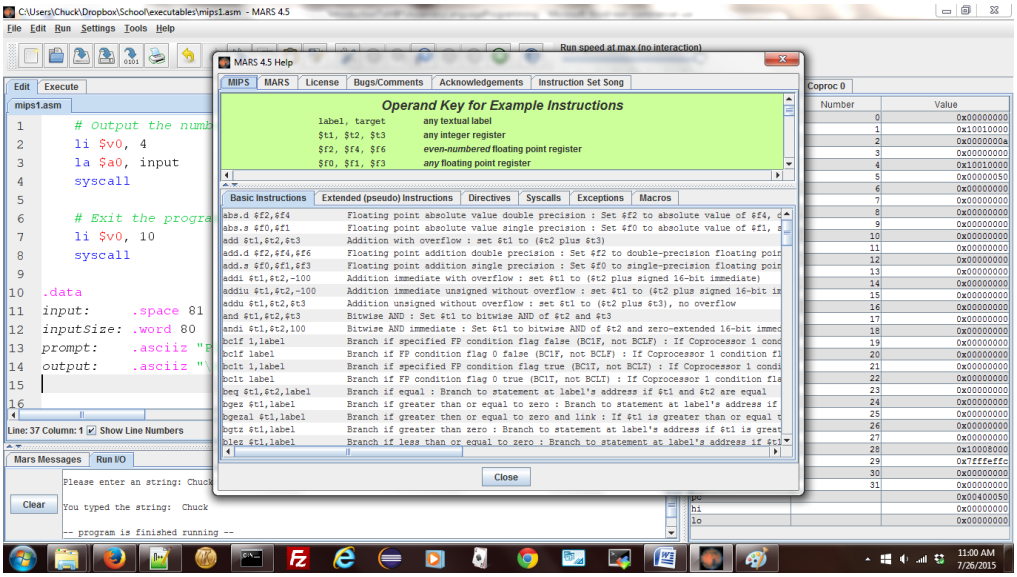
- Any time a constant is included in an instruction, it is called an *immediate* value. The constant must be in the instruction itself, so the value 4 in the instruction `li $a0,4` is an immediate value, but the string "Hello World" is a constant but not an immediate value.

- Only instructions and labels can be defined in a text segment, and only data and labels can be defined in a data segment. You can have multiple data segments and multiple text segments in a program, but the text must be in a text segment and the data in a data segment.

- Operators are text strings like `li`, `la`, and `syscall`. `li` means load the immediate value into the register (e.g., `li $v0,4` means `$v0` <- 4). `la` means to load the address at the label into the register. `syscall` is used to request a system service. System services will be covered in more detail later in this chapter. A list of all MIPS operators used in this text can be found in the MIPS Green Sheet or by using the help option in MARS.

- Instructions are operators and their arguments. So `li` is an operator; `li $v0,4` is an instruction.

- The `syscall` operator is used to call system services. System services provide access to the user console, disk drives, and any other external devices. The service to be executed is a number contained in the `$v0` register. In this program there are two services that are used: service 4 prints a string starting at the address of the memory contained in the `$a0`; and service 10 halts, or exit, the program. A complete list of all `syscall` can be found by using the help menu option (or F1) in MARS. This will bring up the following screen, which gives all the possible options in MARS.



Chapter 2. 4 Program to prompt and read an integer from a user

The next program that will be studied prompts a user to input a number, then reads that number into a register and prints it back to the user.

```
# Program File: Program2-2.asm
# Author: Charles Kann
# Read an integer from a user, and prints that number back to the console.
.text
main:
    # Prompt for the integer to enter
    li $v0, 4
    la $a0, prompt
    syscall

    # Read the integer and save it in $s0
    li $v0, 5
    syscall
    move $s0, $v0

    # Output the text
    li $v0, 4
    la $a0, output
    syscall

    # Output the number
    li $v0, 1
    move $a0, $s0
    syscall

    # Exit the program
    li $v0, 10
    syscall

.data
prompt: .asciiz "Please enter an integer: "
output: .asciiz "\nYou typed the number "
```

Program 0-2: Program to read an integer from the user

Chapter 2.4. 1 Program 2-2 Commentary

The following commentary covers new information which is of interest in reading Program 2-2.

- In this program, blocks of code are commented, not each individual statement. This is a better way to comment a program. Each block should be commented as to what it does, and if it is not obvious, how the code works. There should not be a need to comment each line, as a programmer should generally be able to understand the individual instructions.

- A new operator was introduced in this program, `move`. The `move` operator moves data from one register to another. In the first instance, the result of the `syscall`, read integer service 5, is moved from register `$v0` to a save register `$s0`.

- Two new `syscall` services have been introduced. The first is service 5. Service 5 synchronously waits for the user to enter an integer on the console, and when the integer is typed returns the integer in the return register `$v0`. This service checks to see that the value entered is an integer value and raises an exception if it is not.

- The second new `syscall` service is service 1. Service 1 prints out the integer value in register `$a0`. Note that with service 4 the string that is at the address `$a0` (or referenced by `$a0`) is printed. With the service 1 the value in register `$a0` is printed. This difference between a reference and a value is extremely important in all programming languages, and is often a difficult subject to understand even in a HLL.

- In this program, an escape character "`\n`" is used in the string named `output`. This escape character is called the new line character and causes the output from the program to start on the next line. Note that the sequence of characters written to the output can vary based on the operation system and environment the program is run on, but the escape character will create the correct output sequence to move to the start of a new line.

Chapter 2. 5 Program to prompt and read a string from a user

The programs to read a number from a user and read a string from a user looks very similar but are conceptually very different. The following program shows reading a string from the user console.

```
# Program File: Program2-3.asm
# Author: Charles Kann
# Reads a string from a user and prints it back to the console.
.text
main:
    # Prompt for the string to enter
    li $v0, 4
    la $a0, prompt
    syscall

    # Read the string.
    li $v0, 8
    la $a0, input
    lw $a1, inputSize
    syscall

    # Output the text
    li $v0, 4
    la $a0, output
    syscall

    # Output the number
    li $v0, 4
    la $a0, input
    syscall

    # Exit the program
    li $v0, 10
    syscall

.data
input:     .space 81
inputSize: .word 80
prompt:    .asciiz "Please enter an string: "
output:    .asciiz "\nYou typed the string:  "
```

Program 0-3: Program to read a string from the user

Chapter 2.5. 1 Program 2-3 Commentary

The following commentary covers new information which is of interest in reading Program 2-3.

- There were two new assembler directives introduced in this program. The first is the `.space` directive. The `.space` directive allocates `n` bytes of memory in the data region of the program, where n=81 in this program. Since the size of a character is 1 byte, this is equivalent to saving 80 characters for data. Why 81 is used will be covered in the discussion of strings later in this section.

- The `.word` directive allocates 4 bytes of space in the data region. The `.word` directive can then be given an integer value, and it will initialize the allocated space to that integer value. Be careful as it is incorrect to think of a `.word` directive as a declaration for an integer, as this directive simply allocates and initializes 4 bytes of memory, it is not a data type. What is stored in this memory can be any type of data.

- As was discussed earlier in this chapter, the `la` operator loads the address of the label into a register. In HLL this is normally called a reference to the data, and this text will use both terms when referring to reference data. This will be shown in the text as follows, which means the value of the label (the memory address) is loaded into a register.

  `$a0` <= label

- A new operator, `lw`, was introduced in this section. The `lw` operator loads the value contained at the label into the register, so in the preceding program `lw $a1, inputSize` loaded the value `80` into the register `$a1`. Loading of values into a register will be shown in the text as follows, which means the value at the label is loaded into a register.

  `$a1` <= M[label]

- In MIPS assembly, a string is a sequence of ASCII characters which are terminated with a null value (a null value is a byte containing 0x00). So, for example the string containing "Chuck" would be 0x436875636b00 in ASCII. Thus, when handling strings, an extra byte must always be added to include the null terminator. The string "Chuck", which is 5 characters, would require 6 bytes to store, or to store this string the following `.space` directive would be used.

  ```
  .space 6
  ```

  In the preceding program the string input, which was 80 characters, required a space of 81, including the null value. This is also the reason for the assembler directives `.ascii` and `.asciiz`. The `.ascii` directive only allocates the ASCII characters, but the `.asciiz` directive allocates the characters terminated by a null. So the `.asciiz` allocates a string.

- Reading a string from the console is done using the `syscall` service 8. When using `syscall` service 8 to read a string, there are two parameters passed to the service. The first is a reference to the memory to use the string (stored in `$a0`), and the second is the maximum size of the string to read (stored in `$a1`). Note that the size is 1 less than the number of characters available to account for the null terminator. If the string the user enters is larger than the maximum size of the string, it is truncated to the maximum size. This is to prevent the program from accessing memory not allocated to the string.

  The parameters passed to the method are the string reference in `$a0` and the maximum size of the string in `$a1`. Note that in the case of the string in `$a0`, the value for the string is contained in memory and only the reference is passed to the function. Because the reference is passed, the actual value of the string can be changed in memory in the function. This we will equate to the concept of *pass-by-reference*[1] in some higher-level languages like C++. In the case of string size, the actual value is contained in `$a1`. This corresponds to the concept of *pass-by-value* in a language like Java. A Java program to illustrate this is at the end of this chapter. This topic of value and reference types will be covered in much greater details in the chapters on subprograms and arrays.

[1] It would be more exact to call this a *pass-by-reference-value*, as it is not a true *pass-by-reference* as is implemented in a language like C or C#. But this parameter passing mechanism is commonly called *pass-by-reference* in Java, and the difference between the two is beyond what can be explained in assembly at this point.

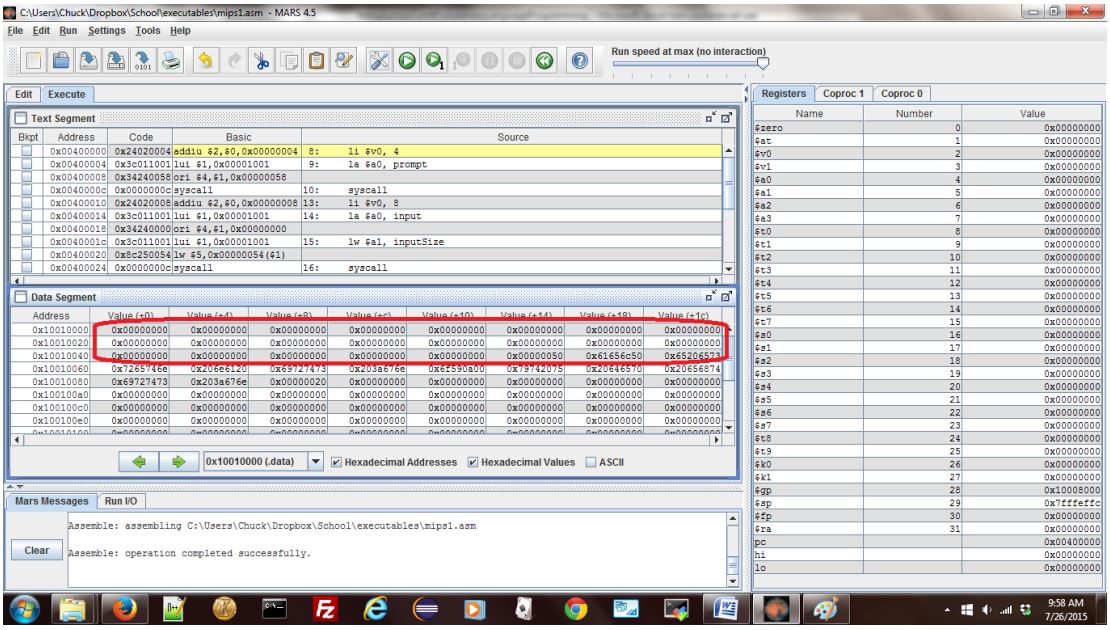
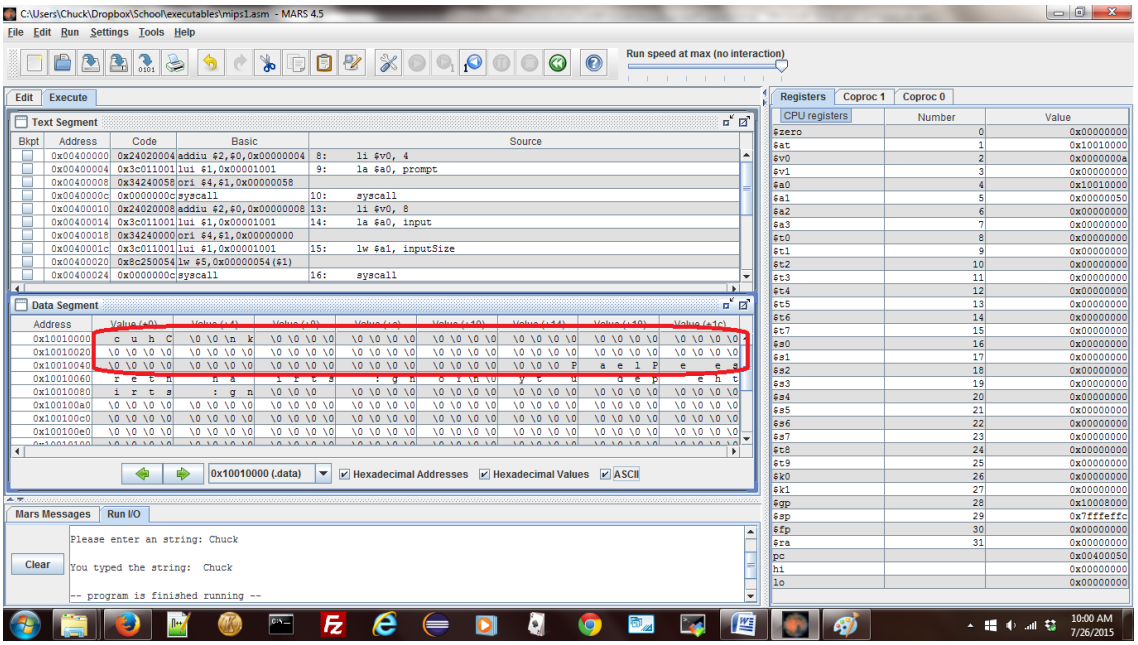


Figure 0-6: Memory before entering a string

- When using `syscall` service 8, the `syscall` changes the memory in the data region of the program. To understand this, the preceding figure shows the program execution string immediately before the program is run. Note that the memory circled in red is the space which was saved for the input string, and it is all null values.

  Run the program and enter "Chuck" at the prompt for a string. The memory for the input string has been changed to store the value "Chuck", as shown in the circled text in the figure below (be sure to select the ASCII checkbox, or the values will show up in hex).



In this figure there are 8 bytes containing the characters "cuhC \0\0\nk". This is the string "Chuck", plus a new line character which is always returned by service 8, the null terminator and an extra byte of memory which was not used. This shows that the `$a0` parameter to service 8 was a memory reference, and the service updated the memory directly.

The second thing to note in this figure is that the letters are stored backwards each grouping of 4 bytes, or a memory word. In this example, the string "Chuck\n" was broken into two strings "Chuc" and "k\n". The characters were then reversed, resulting in "cuhC" and "\nk". This is a common format in computer hardware referred to as *little endian*. Little endian means that bytes are stored with the least significant byte in the lowest address, which reverses the 4 bytes in the memory word. Big endian is the reverse, and in a big-endian system the string would appear in memory as it was typed. The choice of big-endian verses little-endian is a decision made by the implementers of the hardware. You as a programmer just must realize what type of format is used and adjust how you interpret the characters appropriately.

Note from this figure that the service 8 call always appends a "\n" to the string. There is no mechanism to change this in MARS, and no programmatic way to handle this in our programs. This is an annoyance which we will be stuck with until strings are covered at the end of this text.

Finally see that while the string which is returned has 6 character, "Chuck\n", the other 80 characters in memory have all be set to zero. The space allocated for the string is still 80, but the string size is 6. What determines the string size (the actual number of characters used) is the position of the first zero, or null. Thus, strings are referred to as "null terminated". Many HLL, like C and C++[2], use this definition of a string.

Chapter 2. 6 Summary

The purpose of this chapter was to help the reader understand how to create and run simple programs with I/O in MIPS assembly using the MARS IDE. However, even a simple "Hello World" program has complexities when written in assembly language, and this chapter attempted to give at least a casual explanation of these. Covered in this chapter were the following concepts:

- How to comment a MIPS program.
- Registers and memory in MIPS computers.
- Assembler directives such as `.text`, `.data`, `.asciiz`, `.space`, and `.word`.
- Labels in MIPS assembly.
- MIPS assembly operators, such as `li`, `la`, `lw`, and `move`.
- System services for interacting with the user console, in particular services 1, 4, 5, and 8.
- The difference between references and values of data.

[2] C++ uses null terminated strings but can also define strings as instances of the String class. Instances of the String class are very different from null terminated strings, so do not confuse the two.

Chapter 2. 7 Java program for call by value and reference

To better explain the concept of call by value and call by reference, the following Java program is used.

```
public class CallingConventions {
    public static void func(int a, final int b[])
    {
        a = 7;
        b[0] = 7;
    }

    public static void main(String... argv) {
        int a = 5;
        int b[] = {5};

        System.out.println("Before call, a = " + a + " and b[0] = " + b[0]);
        func(a, b);
        System.out.println("After call, a = " + a + " and b[0] = " + b[0]);
    }
}
```

Note that when this program is run, the value of `a` is not changed by the function, but the value of `b[0]` is changed. What is happening is very much analogous to the previously presented MIPS program which prompted for a string. In this case the variable `a` is a *value type*, e.g., the variable stores the value. In the case of the variable `b`, the values are stored in an array, and those values cannot be stored in a single data value (or register). So, what is stored in `b` is the `a` reference to the array, or simply the address of `b` in memory. Thus, `b` is a reference type.
Note that in both cases when calling the function something is essentially *copied* into a memory location (in our MIPS program registers). But if the value is copied, it cannot be changed. If the reference is copied, while the reference cannot be changed, the value which is referred to can. This will become very important when discussing subprograms and arrays later in the text.

Chapter 2. 8 Exercises

1) Write a program which prompts the user to enter their favorite type of pie. The program should then print out "`So you like _____ pie`", where the blank line is replaced by the pie type entered. What annoying feature of `syscall` service 4 makes it impossible at this point to make the output appear on a single line?

2) Using the `syscall` services, write a program to play a middle "C" for 1 second as a reed instrument using the Musical Instrument Digital Interface (MIDI) services. There are two services which produce the output. What is the difference between them?

3) Write a program to print out a random number from 1...100.

4) Write a program which sleeps for 4 seconds before exiting.

5) Write a program to open an input dialog box and read a string value. Write the string back to the user using a message box.