Lab# 2 Basic Structure

Instructor: I Putu Danu Raharja.

Objectives:

- Describe the general structure of MIPS assembly language programs.
- Learn to read and modify assembly language programs.
- Simulating the program using MARS.

Method:

Run and modify a simple MIPS assembly language program. Load and run a more complex program.

Preparation:

Read the chapter 2 of lecture textbook.

Files to use:

lab1-01.asm.

What to Hand In:

Modified copy of lab1-01.asm.

2.1 Download and Install MARS

Download the program from the following URL:

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

to the Desktop on your computer. Download also the accompanying help files.

2.2 The Components and Structure of an Assembler Program

Now we are ready to start writing assembler programs. Let's start by looking at the minimum requirements of a working assembler program. Even a simple assembler program requires quite a few lines. For instance, consider the following program:

```
            .data
prompt:     .asciiz     "\n Please Input a value "
bye:        .asciiz     "\n ** Have a good day **"
            .globl      main
            .text
main:
            li          $v0, 4          # system call code for print string
            la          $a0, prompt     # loads address of prompt into $a0
            syscall                     # print the prompt message
```

```
        li      $v0, 5          # system call code for read integer
        syscall                 # reads the value into $v0
        beq     $v0, $0, end    # branch to end if $v0 equals 0
        addu    $a0, $0, $v0
        li      $v0, 1          # system call code for print integer
        syscall                 # print
        b       main            # branch to main
end:
        li      $v0, 4          # system call code for print string
        la      $a0, bye        # loads address of bye into $a0
        syscall                 # print the string
        li      $v0, 10         # terminate program run and
        syscall                 # return control to the OS.
```

This program contains the directives .DATA, .TEXT, .ASCIIZ, and .GLOBL. Directives are required in every assembler program in order to define and control memory space usage.

Directives only provide the framework for an assembler program, though; you also need lines in your source code that actually DO something, lines like

```
beq $v0, $0, end
```

and

```
addu $a0, $0, $v0
```

,these are instruction mnemonics, corresponding to the instruction set of the MIPS32. In addition, there are some pseudo-instruction that were created by the assemblers to simplify translation and programming. These pseudo-instructions are not part of the MIPS32 processor. Before you can use either instructions, pseudo-instructions or directives, however you must first learn about the format of a line of assembler code.

2.3 THE FORMAT OF A LINE

Assembly language source code lines follow this format:

[*label*:] [*instruction*/*directive*] [*operands*] [#*comment*]

where [*label*] is an optional symbolic name; [*instruction*/*directive*] is either the mnemonic for an instruction or pseudo-instruction or a directive; [*operands*] contains a

combination of one, two, or three constants, memory references, and register references, as required by the particular instruction or directive; [*#comment*] is an optional comment.

A. Labels

Labels are nothing more than names used for referring to numbers and character strings or memory locations within a program. Labels let you give names to memory variables, values, and the locations of particular instructions. For instance, the above code mentioned in Page 1 uses several labels:

The labels *prompt* and *bye* are equivalent to the addresses of two 8-bit variables; they're used to refer to those variables later in the code. The label *main* is equivalent to the address of the first instruction

```
li $v0, 4
```

and the label *end* is equivalent to the address of another instruction

```
li $v0, 4
```

Labels can consist of the following characters:

A-Z a-z _ 0-9

The digits 0-9 cannot be used as the first character of a label. Each label must be defined only once; that is, labels must be ***unique***. Labels may be used as operands any number of times.

A label may appear on a line by itself, that is, on a line without an instruction or directive. In this case, the value of the label is the address of the instruction or directive on the next line in the program. For instance, in the code

```
                .
                .
                .
                b DoSubtract
                .
                .
                .
DoSubtract:
                sub $s0, $zero, $t8
```

The next instruction executed after the **b** pseudo-instruction, which branches to the label *DoSubtract*, is **SUB $s0, $zero, $t8**. The preceding example is exactly the same as

```
                .
                .
                .
                b DoSubtract
                .
                .
                .
DoSubtract:     sub $s0, $zero, $t8
                .
                .
```

A label cannot be the same as any of the built-in symbols used in expressions. This includes the register names ($at, $t2, $3, and so on), the instructions and the pseudo-instructions used in expressions (*add, sub, addi*, and so on). You also should avoid using any directives as label names.

Both labels that appear on lines without directives or instruction mnemonics and labels that appear on lines with instructions must be end with a colon. The colon merely ends the label, and is not part of the label itself.

By default all labels are *local*. It means that the label referring to an item that can be used only within the file in which it is defined. A label can be made as external or *global* by using the directive **.globl**. It is a label referring to an item that can be referenced from files other than the one in which it is defined.

B. Instruction Mnemonics and Directives

The key field in a line of assembler code is the [*instruction/directive*] field. This field may contain either an instruction mnemonic or a directive, two very different beasts.

MARS assembles each instruction mnemonics directly to the actual MIPS32 machine-language code. Whenever you insert one instruction mnemonic in an assembler program, the result is one corresponding machine-language instruction in the executable code.

Directives generate no executable code at all, but rather control various aspects of how MARS32 operates. They are responsible for providing high-level features of MARS32 that make assembly language programming much easier.

▪**TEXT** tells the MARS that the subsequent items are stored in the user text segment. This directive tells MARS32 exactly which text segment to place your instructions in.

▪**DATA** tells MARS that the subsequent items are stored in the data segment. You should place your memory variables in this segment. For example,

```
                .
                .
                .DATA
First:          .space 100
Second:         .word 1, 2, 3
Third:          .byte 99, 2, 3
                .
                .
```

C. Pseudo-instructions

Pseudo-instructions give MIPS a richer set of assembly language instructions than those implemented by the hardware. For example, one of the frequent steps needed in programming is to copy the value of one register into another register. This actually can be solved easily by the instruction:

```
add $t0, $zero, $t1
```

However, it is more natural to use the pseudo-instruction:

```
move $t0, $t1.
```

The assembler converts this pseudo-instruction into the machine language equivalent of the prior instruction.

2.4 BYTE ORDERING AND ENDIANNESS.

Bytes within larger CPU data formats—halfword, word, and doubleword—can be configured in either big-endian or little-endian order. **Endianness** defines the location of byte 0 within a larger data structure (in MIPS, bits are always numbered with 0 on the right).

A. Big-Endian Order:

When configured in big-endian order, byte 0 is the **most-significant (left-hand) byte**. For example, the following figure illustrates the byte order in a word:

Word Address	Bit# 31 24	23 16	15 8	7 0
12	12	13	14	15
8	8	9	10	11
4	4	5	6	7
0	0	1	2	3

B. Little-Endian Order

When configured in little-endian order, byte 0 is always **the least-significant (right-hand) byte**. For example, the following figure illustrates the byte order in a word:

Word Address	Bit# 31 24	23 16	15 8	7 0
12	15	14	13	12
8	11	10	9	8
4	7	6	5	4
0	3	2	1	0

2.5 SOME MIPS INSTRUCTIONS

Instructions		Description
lb	Rdest, address	**Load Byte**. Loads the byte at address in memory into the LSB of Rdest. The byte is treated as a signed number; sign extends to the remaining three bytes of Rdest.
lbu	Rdest, address	**Load Byte Unsigned**. This instruction is similar to lb except that the byte is treated as an unsigned number. The upper three bytes of Rdest are filled with zeros.
lh	Rdest, address	**Load Halfword**. Loads the half-word (two bytes) at address in memory into the least significant two bytes of Rdest. The 16-bit data are treated as a signed number; sign extends to the remaining two bytes of Rdest.
lhu	Rdest, address	**Load Halfword Unsigned**. Same as lh except that the 16-bit halfword is treated as an unsigned number.
lw	Rdest, address	**Load Word**. Loads the word (four bytes) at address in memory into Rdest.
lui	Rdest, imm.	load constant in upper 16 bits of Rdest.
sb	Rdest, address	**Store Byte**. Stores the least significant byte of Rsrc at the specified address in memory.
sh	Rdest, address	**Store Halfword**. Stores the least significant two bytes (halfword) of Rsrc at the specified address in memory.
sw	Rdest, address	**Store Word**. Stores the four-byte word in Rsrc at the specified address in memory.

Instructions		Description
add	Rdest,Rsrc1,Rsrc2	adds the contents of Rsrc1 and Rsrc2 and stores the result in Rdest. The numbers are treated as signed integers. In case of an overflow, an overflow exception is generated.
addu	Rdest,Rsrc1,Rsrc2	Same as add except that no overflow exception is needed.
addi	Rdest,Rsrc1, imm	Rdest = Rsrc1 + 16-bit signed immediate. In case of an overflow, an overflow exception is generated.
addiu	Rdest,Rsrc1, imm	Same as addi except that no overflow exception is needed.
sub	Rdest,Rsrc1,Rsrc2	Rdest = Rsrc1 − Rsrc2. The numbers are treated as signed integers. In case of an overflow, an overflow exception is generated.
subu	Rdest,Rsrc1,Rsrc2	Same as sub except that no overflow exception is needed.
Assembler pseudoinstructions:		
la	Rdest, var	**Load Address**. Loads the address of var into Rdest.
li	Rdest, imm	**Load Immediate**. Loads the immediate value imm into Rdest.

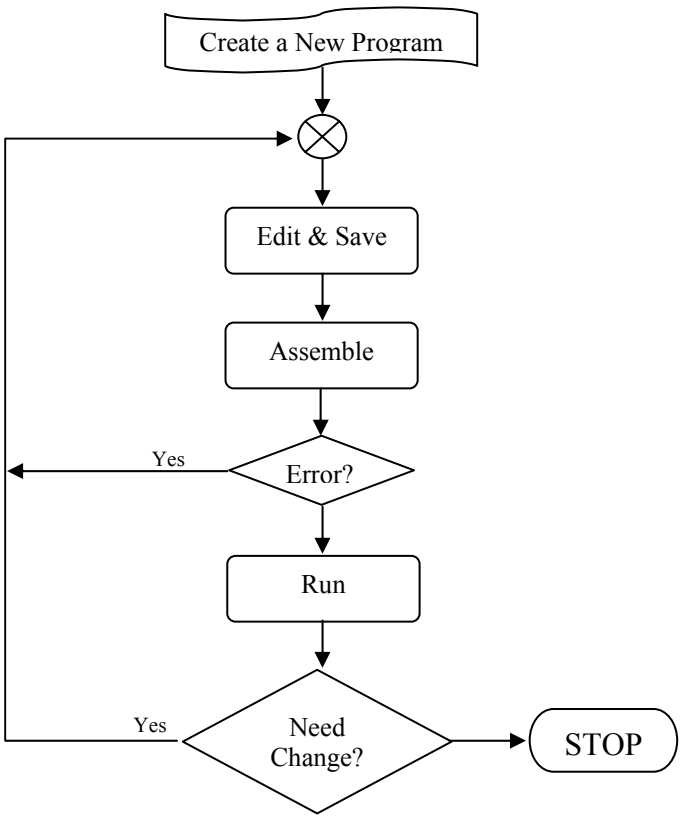
2.6 SYSTEM I/O (INPUT/OUTPUT)

The developers of the MARS simulator wrote simple I/O functions. Access to these functions is accomplished by generating a software exceptions. The MIPS instruction a programmer uses to invoke a software exception is ***syscall***. There are 10 different services provided. In your programs, you specify what services you want to perform by loading register ***$v0*** with a value from 1 to 10. The following table describes some system services:

Service	Code in $v0	Argument(s)	Result(s)
Print integer	1	$a0 = number to be printed	
Print String	4	$a0 = address of string in memory	
Read Integer	5		Number returned in $v0.
Read String	8	$a0 = address of input buffer in memory. $a1 = length of buffer (n)	
Exit	10		

2.7 ASSEMBLING, LINKING, AND EXECUTING PROCESSES

Before you can run the program, though, you have to convert the source code into an executable (able to be run or executed) form. This requires two additional steps, assembling and linking as shown in the following figure.

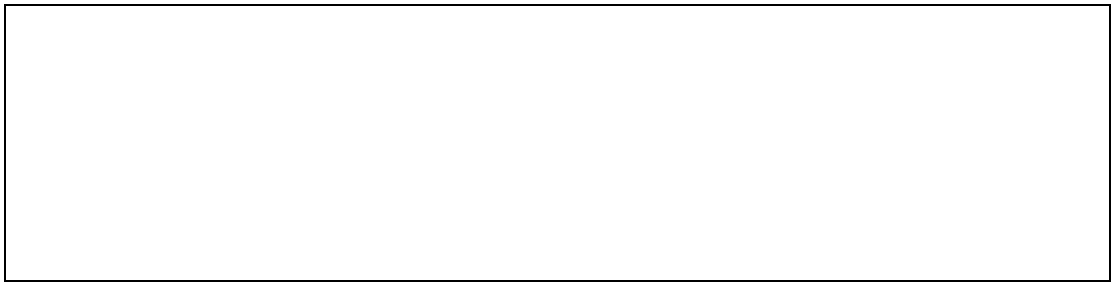


2.8 LAB EXERCISES:

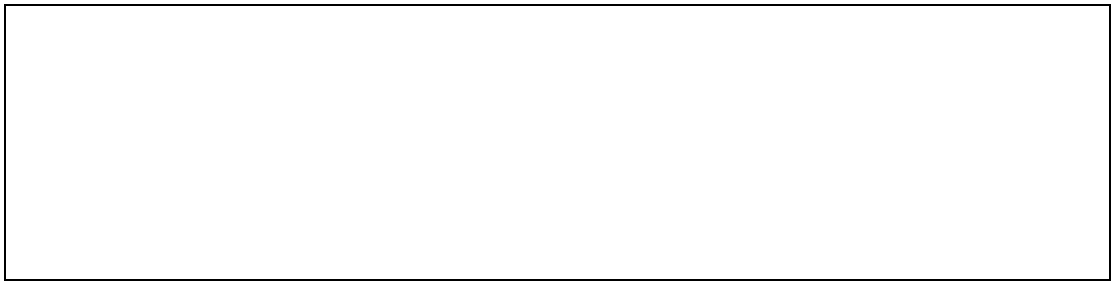
1. Start the MARS simulator and load the ***lab1-01.asm*** file into the simulator. Try running the program with both the run command and the step command.
2. Where (to which window) is the output data displayed?
3. Write down the address of the first instruction of the program (see the text window)
4. Write down the address of the first data of the program (see the data window)
5. Write down the value of the register **$sp** just before you start the program.
6. What is the representation of the newline symbol "\n" in memory (write down the hexadecimal value)?
7. What is the hexadecimal representation in memory of the string "abcd"?
8. Single step the program and write down the memory addresses and hexadecimal codes that represent the instruction sequence corresponding to this line:

```
la $a0, endl
li$v0, 4
```

9. Delete the quotation in the declaration of endl. Save the file and then assemble again the file. Run the program. What happens? Describe?



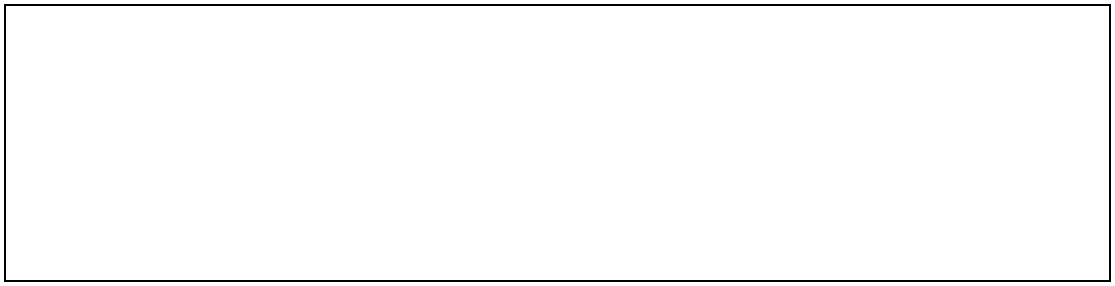
10. Deliberately insert a syntax error. How does MARS help you in locating the source of the errors? Describe below from your limited experience.



11. Change the code in line 21 and 22 to:

```
li      $t0, 0x70000000
li      $t1, 0x10000000
```

12. Save the file and then assemble again the file. Run the program. What happens? Describe?



13. Change the code in the line 23 to:

```
addu    $a0, $t0, $t1
```

Save the file and then assemble again the file. Then set a breakpoint at the instruction after the instruction addu $a0,$t0,$t1. Run the program. What is the value of $a0? Continue to run the program.

14. Insert the following codes in the line 34:

```
la      $a0, endl
sh      $t1, 0($a0)
```

15. Save the file and then assemble again the file. Run the program. What happens? Describe?

16. Remove the codes you inserted in step 14. Change the data description in the line 41 to:

```
msg1:   .ascii   "abcdef"
```

    Save the file and then assemble again the file. What is the hexadecimal representation in memory of the data in the line 43-44?

17. Exit from MARS.

2.9 EVALUATION

Review the material for any evaluation questions.