Lab 2

Strings and Conditional Execution

Learning outcome of this lab

* Using syscall for character string output
* Conditional Execution using ‘branch’ and ‘jump’ instruction for implementing
  + Loops
  + If – else statements
* Pre-Lab

1. Displaying Strings
   1. Code Syntax for MIPS Assembler:

Memory in a computer is divided in to separate segments e.g. for user code, variables, stack and OS reserved space etc. Figure 2.0 shows a particular distribution in memory as an example. So, the code should include assembler directives to help assembler identify the type of content. MIPS assembler uses .text and .data directives to differentiate between code and data.

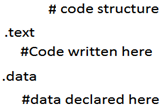
 

Fig 2.0 Fig 2.1

The order does not matter here. Data segment can be declared before code segment (.text). Inside .data, data can be defined using any of multiple directives shown in table below. Note that by default, without any directive (.text or .data) the assembler assumes that the given content is code.

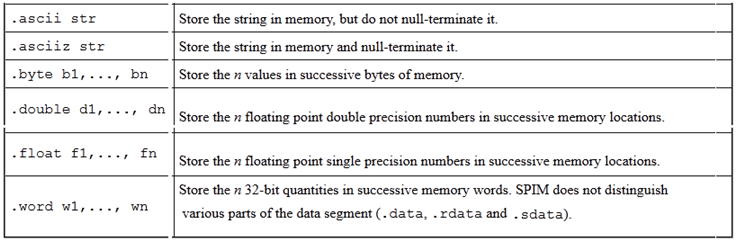


Table 2.1

* 1. Example

The code given below displays “Hello World" and “How are you?”. In order to display a string, “syscall” is used; but before that, $v0 is loaded with calling code 4 and the address of the string in $a0.

# Name Reg No. Date

# hello.asm-- A "Hello World" program.

# Registers used:

# $v0 - syscall parameter and return value.

# $a0 - syscall parameter-- the string to print.

.text

main:

la $a0, hello\_msg # load the addr of hello\_msg into $a0.

li $v0, 4 # 4 is the print\_string syscall.

Syscall # do the syscall.

la $a0, question # load the addr of question into $a0.

Syscall # do the syscall.

li $v0, 10 # 10 is the exit syscall.

Syscall # do the syscall.

# Data for the program:

.data

hello\_msg: .asciiz "Hello World\n"

question: .asciiz "How are you?\n"

# end hello.asm

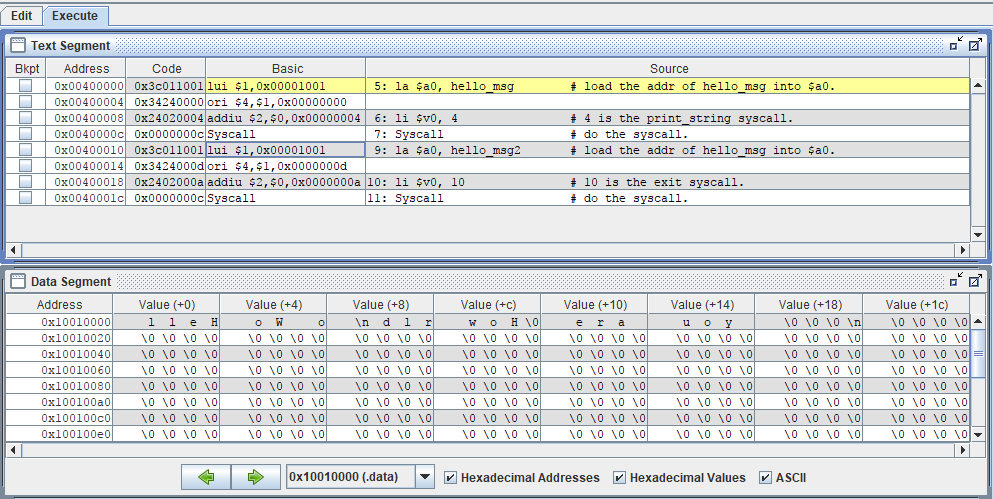


Figure 2.2

Observe the assembled code for instruction la $a0, hello\_msg. It comprises of two instructions lui and ori. lui is load upper immediate. It copies immediate value to upper 16-bits of a 32-bit register and setting lower 16-bits to zero. Executing ori after lui simply copies the immediate value to lower 16-bit since the lower 16-bits were previously set to zero by lui.

Explaining the code for second string “question” will clarify the concept better. In the data segment, which starts from 0x1001000, note that second string starts from 13th location i.e 0xd (hexadecimal). So the address 0x1001000d should be copied to $a0 ($4 in the list). So, the instruction

lui $1,0x00001001 (lui $1,0x1001 initial zeros are useless)

results in $1 = 0x10010000. Data is copied to upper 16-bits while lower 16-bits are set to zero

Now,

Ori $4,$1,0x0000000d

Results in $4=$a0=0x1001000d which is the address of second string.

Note that data in the data segment is stored in adjacent locations. Therefore, there are many ways that we could have declared the string "Hello World\n" and gotten the same exact output. For example we could have written our string as:

.data

hello\_msg: .ascii "Hello" # The word "Hello"

.ascii " " # the space.

.ascii "World" # The word "World"

.ascii "\n" # A newline.

.byte 0 # a 0 byte.

If we were in a particularly cryptic mood, we could have also written it as:

.data

hello\_msg: .byte 0x48 # hex for ASCII "H"

.byte 0x65 # hex for ASCII "e"

.byte 0x6C # hex for ASCII "l"

.byte 0x6C # hex for ASCII "l"

.byte 0x6F # hex for ASCII "o"

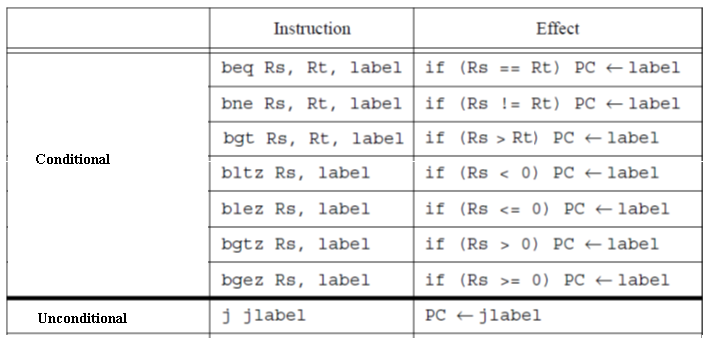
... # and so on...

.byte 0xA # hex for ASCII newline

.byte 0x0 # hex for ASCII NUL

1. Conditional Execution

Conditional executions such as ‘for loops’ and ‘if-else’ in C, are accomplished in assembly using conditional and unconditional jump statements. Jump statements break the normal sequence of execution and code starts executing from the jump address instead of the next instruction.



PC stands for ‘program counter’, which hold the address for the next instruction. means if the condition is true, PC will be filled with address of the instruction after ‘label’, instead of the next instruction in sequence.

* 1. Loops

Loops are implemented using conditional branch statements, or, with the combination of conditional and unconditional branch statements.

# Name Reg No. Date

# loop1.asm-- A "Loop" program using bne.

# Registers used:

# $v0 - syscall parameter and return value.

.text

#######Loop#########

anylabel:

li $t1,5 #initialize $t1 with 5

li $t0,0 #initialize $t0 with 0

#insert any useful code here

add $t0,$t0,1 #increment counter

bne $t0,$t1,anylabel # Jump back to anylabel while #$t0 < $t1

#################

li $v0, 10 # 10 is the exit syscall.

Syscall # do the syscall.

# Name Reg No. Date

# loop2.asm-- A "Loop" program using beq and j.

# Registers used:

# $v0 - syscall parameter and return value.

.text

########Loop######

anylabel:

li $t1,5 #initialize $t1 with 5

li $t0,0 #initialize $t0 with 0

#insert any useful code here

add $t0,$t0,1 #increment counter

beq $t0,$t1,out # Jump to label out when $t0=$t1

j anylabel #unconditional jump to label

#################

out:

li $v0, 10 # 10 is the exit syscall.

Syscall # do the syscall.

* 1. Branching (If – else)

Branching is implemented using combination of conditional and unconditional jump instructions. Example Code given below is (over simplified) grading code. It gets student marks and outputs GPA. It is oversimplified in the sense that it displays

4 when input is greater than or equal to 90

2 when input is greater than 49 and less than 90

0 when input is less than 50

# Name Reg No. Date

# simplegrading.asm-- A grading program

# Registers used:

# $v0 - syscall parameter and return value.

# $t1 – contains input marks

# $t4 – contains output grade

.text

li $t1, 50 #initializing with the marks for comparison

li $t2, 90

li $v0, 5 # Calling Code for integer input (Getting Marks)

syscall

move $t0,$v0 #Move input marks to $t0

bge $t0, $t2, GradeA #If input is greater than 89, jump to label GradeA

bge $t0, $t2, GradeC #If this instruction runs it means Input #is less than 90. Now if input is greater #than 49 jump to label GradeC

li t4, 0 #If two instructions above are not true, it #automatically means marks are less than 50

j bypass\_to\_end #Unconditional jump to end label bypass #inbetween code

GradeA:

li $t4,4 # Loading with output grade 4

j bypass\_to\_end

GradeC:

li $t4,2 # Loading with output grade 4

bypass\_to\_end:

move $a0,$t4 #Moving grade to $a0

*li $v0, 1 # Call code for displaying integer is 1*

*syscall # make the syscall*

*li $v0, 10 # syscall code 10 is for exit.*

*syscall # make the syscall.*

* In-Lab

**Task**

* Write the third code of previous lab i.e. student total marks, but this time display strings, asking for sessiona1, sessional2, QA and Terminal marks. For example, first string will be; “Enter marks secured in Sessional 1:”
* Repeat the Total Marks code . This time get four marks in loop, displaying the single string “Enter Marks:” each time.
* Modify the first code (with each input having separate string). If the entered marks are greater than the limit, get the input again. The limits for Sessional1, Sessional2, QA and Terminal are 10,15,25 and 50 respectively.
* Post-Lab
* Write the Total Marks program with separate strings for each input by using loop which runs four times. So there should only be two syscalls inside the loop.
* **Your lab report, a .doc file, should contain properly commented Post-Lab task code, MARS screenshot for each code which clearly show the register contents after execution and I/O panel, and critical analysis.**
* **The report must have a title page.**
* **Name the .doc file RegNo.docx; eg SP14-BCE-99.docx**
* **Upload it on portal**
* **Deadline for Lab Report submission is before start of the next Lab.**