



**EASTERN MEDITERRANEAN UNIVERSITY
COMPUTER ENGINEERING DEPARTMENT**

**CMPE 324 - Computer Architecture and
Organization**

LAB 3:

1. Objective:

Introduction to the usage of the Data Segment, and Arrays in functional single-module assembler programs. Development of understandable assembly programs using variable symbols.

2. Introduction:

Unlike programs in high-level languages, the operands of arithmetic instructions in the MIPS assembly cannot be variables. They must be from a limited number of general purpose registers. There are 32 g.p. registers (\$0,\$1,\$2,...,\$31) available in the MIPS architecture.

One of the main differences of the registers from the variables of a programming language is their limited number. The restricted number of registers provide higher speed in reading and writing their contents.

Higher the number of the registers, more gates are necessary in address decoding, resulting in higher delays, and therefore raises the clock cycle time because of delays in address decoding gates.

In most programming languages there are simple data types that contain single data elements, and also more complex data structures such as arrays. These complex data structures can contain many more data elements than available registers of the processor. A processor has only a restricted number of registers to store data. Then, how can a processor access and process such large structures?

The answer is, a processor can also access to a memory system, which contains millions of data elements. Hence large data structures, like arrays, are kept in memory. Only a part of data that has to be processed is loaded to the registers.

MIPS architecture provides a set of memory-transfer instructions. A data transfer from memory to a register is possible using the memory reference instruction load word (lw), **lw \$destination, offset (\$base)** Which loads \$destination with the data word starting from memory address offset+\$base.

For example, **lw \$20, 10000 (\$0)** means that the constant (or data) in the memory location at address 10000+\$0 is stored into register \$20. A data transfer from a register into a memory is obtained using a store word (sw) instruction, **sw \$source, offset (\$base)** which stores the \$source register into the memory to the address starting from offset+\$base. For example, **sw \$10, 20000(\$2)** stores the contents of register \$10 into the memory location at address 20000+\$2.

Both **in lw and sw instructions**, offset is a 16-bit signed integer. However, if the pseudo-instructions are allowed, they accept 32-bit addresses, and implement the pseudo-instruction using lui, add, and sw or lw with a 16-bit-offset.

Note: Now let's begin our first program for LAB3

3. Experimental Work

Part 1: Tracing Exercise

In order to understand the operation of memory reference instructions (lw and sw), type and execute the following MIPS assembly program giving it filename "exp3.1.s". The program contains pseudo-instruction such as la \$dest, <label> and the lw and sw instructions have 32-bit offsets.

Thus you have to turn on the pseudo-instruction option of SPIM in settings dialog-box (keysequence alt-s,s). Ask your particular data-set from your assistant, and fill in the related part of the report after tracing your program with that data-set. Don't forget to set the value in PC to 0x00400000 to start the execution from this address.

```

.data 0x10000000
# A[] is array with a trailing zero.
A:
    .word 128 100 42 16 5 2 0 0
Count:
    .word 0
Sum:
    .word 0
Result:
    .word 0
Remainder:
    .word 0
EndofData:
    .word -1
# End of data segment, code starts here from address 0x00400000
# Turn-off bare-machine, and turn-on pseudo-instruction options.
.text
.globl main
main:
# finding average of A[]
la $2,A # pseudo-instruction load-address A[.]
or $8,$0,$0 # 0 -> $8 , count
or $10,$0,0 # 0 -> $10, sum
slp:
lw $11,0($2) # A[i] -> $11
beq $11,$0,slx
add $10,$10,$11 # A[0]+...+A[i] -> $2
addi $8,$8,1 # ++count
addi $2,$2,4 # address of(A[i+1]) -> $2

```

```

j slp # loop until getting the zero
slx:
# save the count
sw $8,Count($0)
sw $10,Sum($0)
# divide $10 by $8, use count of repeating subtractions.
div $10,$8 # quotient is in LO, remainder is in HI
mflo $11 # move from LO to $destination
mfhi $10 # move from HI to $destination
sw $11,Result($0) # mean
sw $10,Remainder($0) # and this is the remainder of
division
syscall # for the sake of SPIM add below one empty line

```

Part 2 - Programming Exercise

Write a program to calculate the ratio of four words (20 40 60 80) of [A] to its total value and then store the four ratio results in to [res] memory location.

After you debug your program trace your program with the given particular data and fill in the data segment to your report sheet. $res[0]=180/20$ $res[1]=180/40$ $res[2]=180/60$ $res[3]=180/80$

Note: Modify your program according to your assistant direction and fill in the data segment to your report sheet.

```
.data 0x10000000
A:
.word 20,40,60,80
res:
.word 0,0,0,0
.text
.globl main
main:
.
.
COMPLETE THE PROGRAM
.
.
.
syscall # for the sake of SPIM add below
one empty line
```

4. Reporting

Before the Lab-time is over, fill in the following report page as soon as you complete the laboratory work, and submit it to your assistant. Your report is important for your grading.

Name: _____ Student Number: _____
 Submitted to (Asst.): _____ Date: dd/mm/yy ___/___/___

**EASTERN MEDITERRANEAN UNIVERSITY
 COMPUTER ENGINEERING DEPARTMENT**

2019 Fall

**CMPE 324 -Computer Architecture and Organization
 EXPERIMENT - Reporting Sheet**

Part1: Your data segment lines must have

A: .word _____, _____, _____, _____, _____, _____, _____, _____

_____:

.word 0

_____:

.word 0

_____:

.word 0

_____:

.word 0

EndofData:

.word -1

The data segment readings in Hexadecimal after the execution

0x10000000				
0x10000010				
0x10000020				
0x10000030				
0x10000040				
0x10000050				
....				

Part 2: Programming Exercise

1- Your data set is:

<code>0x10000000</code>				
<code>0x10000010</code>				

2- After modification:

<code>0x10000000</code>				
<code>0x10000010</code>				
<code>0x10000020</code>				
<code>0x10000030</code>				
<code>0x10000040</code>				
<code>0x10000050</code>				
<code>....</code>				

Grading:

Lab Performance:

Asst. Observations: