EASTERN MEDITERRANEAN UNIVERSITY
COMPUTER ENGINEERING DEPARTMENT

CMPE 324 - Computer Architecture and Organization

LAB 4:

1. Objective:

This lab will be an overview of Modular Programming in MIPS Using Jump-and-Link (jal) and Jump-Return (jr) Instructions.

2. Introduction:

We have discussed in the class that programmers use procedures or subroutines while writing programs. A modular program is easier to understand than a so called spagetthy or flat programs. Furthermore, modular programming provides more compact coding because re-using the modules is possible. To support the modular programming, an instruction set must provide a technique to call a procedure and then return from the procedure to the address right next to the call instruction.

MIPS instruction set has two instructions for modular programming:

A. Jump-and-link (jal)

B. Jump-register (jr).

```
      ...
      jal  ProcedureAddress
      add  $2,$5,$9 # next-instruction after the call
      ...
ProcedureAddress:
      ...
      jr   $31
AnotherProcedure:
      ...
```

The jal instruction changes the sequence of execution to the specified procedure address and simultaneously saves the address of the following instruction in register $31. The word "link" in the instruction name implies that a link is formed to calling address to resume the execution of the next instruction after the call. This link is called the return address and it is stored in register $31.

The jr $addr instruction is called jump-return or jump-to-register. It changes the sequence of execution of the instructions to the address that was stored in the $addr register, resulting in a jump to that address.

jr $31 terminates the execution of the procedure by jumping to the address that was stored by the jal instruction into $31.

In MIPS, the address of the next instruction to execute is kept in the Program Counter (PC). In sequential execution, PC is incremented automatically by 4, so that it points the next instruction in the memory as the next instruction to execute. A jump is accomplished by copying a new address into PC. Jump (j) and Jump-and-link (jal) are implemented by copying the specified address into PC, which directs the sequence of the execution to that address. Jump-register copies the contents of the specified register into PC register. jr $31 accomplishes return from subroutine by copying the contents of $31 into PC, that yields a jump to the return address.

If a procedure calls another procedure, then the old value of register $31 must be saved into a last-in-first-out (LIFO) stack. MIPS provides two conventions governing how to pass parameters and how to support the nesting of procedure calls. The registers are grouped for specific purposes.

- $0 contains 0, $1 is reserved for assembler-pseudocodes

- $2 and $3 are used in returning the value or the pointer of the return value of the procedure from the callee to the caller.
- $4, $5, $6 and $7 are used in transfering the arguments from the caller to the callee.
- $8 ... $15 are the callee saved registers, for local work in the callee.
- $16 ... $23 are the caller saved registers, for long range work across the callees.
- $28 is the global data pointer that points the static-data-segment.
- $29 is the stack-pointer that points the top-of-stack address.

And also, you can find more in the table below:

name	reg#	convention
$zero	0	**constant** 0
$at	1	reserved for compiler
$v0-$v1	2–3	**results**
$a0-$a3	4–7	**arguments**
$t0-$t7	8–15	(callee-saved) temps
$s0-$s7	16–23	**caller-saved**
$t8-$t9	24–25	(callee-saved) temps
$k0-$k1	26–27	reserved for OS
$gp	28	global pointer
$sp	29	**stack pointer**
$fp	30	**frame pointer**
$ra	31	**return address**

A compiler usually implements the calls in both callee and caller save convention, and selects the best one to have fast and compact coding.

In this experiment, we will use only the callee-save convention and the assembler with the pseudo-codes allowed option to implement.

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

3. Experimental Work

Part 1: Nested Calls:

In order to see how jal and jr instructions are employed in a structured program, study the following C-program, that is converted to MIPS assembly. The program finds the sum of an array from a specified start index to an end-index by recursive calls of addition. Type and execute the corresponding MIPS assembly program that writes the sum to the data segment.

```
// Function computes sum of the array elements which starts from first
element (0) and ends when reaches to (size-1)th index

int sum (int arr[ ], int size)
{
        If (size==0)
                return 0;
        else
                return sum(arr, size-1) + arr[size-1];
}
```

The following code is the corresponding MIPS assembly source.

```
.data 0x10000000
A: .word 3 5 6 2 04
.text 0x00400000
main:
la $a0, A
li $a1, 6
jal fun
move $s0, $v0
syscall
fun: addi $sp, $sp, -8 # Adjust sp
addi $s0, $a1, -1 # Compute size - 1
sw $s0, 0($sp) # Save size - 1 to stack
sw $ra, 4($sp) # Save return address
bne $a1, $zero, L1 # branch ! ( size == 0 )
li $v0, 0 # Set return value to 0
addi $sp, $sp, 8 # Adjust sp
jr $ra # Return
L1: move $a1, $s0 # update second arg
jal fun
lw $s0, 0($sp) # Restore size - 1 from stack
li $t7, 4 # t7 = 4
mult $s0, $t7 # Multiple size - 1 by 4
mflo $t1 # Put result in t1
add $t1, $t1, $a0 # Compute & arr[ size - 1 ]
lw $t2, 0($t1) # t2 = arr[ size - 1 ]
add $v0, $v0, $t2 # retval = $v0 + arr[size - 1]
lw $ra, 4($sp) # restore return address from stack
addi $sp, $sp, 8 # Adjust sp
jr $ra # Return
```

Load and trace the code step-by-step using f1Ø-key after initializing PC to ØxØØ4ØØØØØ (it is a good idea to prepare the file before the lab hour to save time in the Lab). Write the contents of the stack-list to your report, indicating from which register the value is originated when the jr $31 instruction is executed the first time. You have to note down the register number each time when a register is pushed to stack so that you will be able to fill in from which register the values are originated.

Part 2: Tracing Exercise

In this section you can trace the code, and find where jal and jr instruction are used?

```
# Simple routine to demo functions
# NOT using a stack in this example.
# thus, the function does not preserve values of calling function!
# ---------------------------------------------------------------
                .text
        main:
        # Register assignments
        # $s0 = x
        # $s1 = y
        # Initialize registers
        lw      $s0, x          # Reg $s0 = x
        lw      $s1, y          # Reg $s1 = y
        # Call function
        move    $a0, $s0        # Argument 1: x ($s0)
        jal     fun             # Save current PC in $ra, and jump to fun
        move    $s1,$v0         # Return value saved in $v0. This is y ($s1)
        # Print msg1
        li      $v0, 4          # print_string syscall code = 4
        la      $a0, msg1
        syscall
        # Print result (y)
        li      $v0,1           # print_int syscall code = 1
```

```
move    $a0, $s1            # Load integer to print in $a0
        syscall
        # Print newline
        li      $v0,4               # print_string syscall code = 4
        la      $a0, lf
        syscall
        # Exit
        li      $v0,10              # exit
        syscall
# ---------------------------------------------------------------
                # FUNCTION: int fun(int a)
        # Arguments are stored in $a0
        # Return value is stored in $v0
        # Return address is stored in $ra (put there by jal instruction)
        # Typical function operation is:
fun:    # Do the function math
        li $s0, 3
        mul $s1,$s0,$a0                 # s1 = 3*$a0  (i.e. 3*a)
        addi $s1,$s1,5              # 3*a+5
        # Save the return value in $v0
        move $v0,$s1
        # Return from function
        jr $ra                      # Jump to addr stored in $ra
        # ---------------------------------------------------------------
        # Start .data segment (data!)
        .data
x:      .word 5
y:      .word 0
msg1:   .asciiz   "y="
lf:    .asciiz      "\n"
```

Part 2 - Programming Exercise

In this part, your assistant will give you a C-program source containing a function or procedure in it. You have to write the corresponding code, and verify that your program works correctly.

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 1 - Reporting Sheet

Part1: The contents of the stack at the first execution of the jr $31 instruction.

Address		word0	word1	word2`	word3
addr.					
register					
addr.					
register					
addr.					
register					
addr.					
register					
addr.					
register					

What is the depth of the nested calls? .....

What is the number of words occupying the stack? .....

Part 2: Programming Exercise

C-program to translate into MIPS code is:

```
int islower(char ch)
 {
if (ch>='a' && ch<='z')
return 1;
else
 return 0; }
```

..... , ... , ... , ...

..... , ... , ... , ...

..... , ... , ... , ...

..... , ... , ... , ...

..... , ... , ... , ...

Grading: Quiz Performance:

Lab Performance:

Asst. Observations: