CMPE 325 - Computer Architecture II

EXPERIMENT 3:

Modular Programming in MIPS Using Jump-and-Link (jal) and Jump-Return (jr) Instructions

1.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: *Jump-and-link* (`jal`) and *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 **$0 contains 0, $1 is reserved for** assembler-pseudocodes
   ① `$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.**
   ② `$4`**,** `$5`**,** `$6` **and** `$7` **are used in** transfering the arguments **from the caller to the callee.**
   ③ `$8` **...** `$15` **are the** callee saved registers**, for local work in the callee.**
   ④ `$16` **...** `$23` **are the** caller saved registers**, for long range work across the callees.**
   ⑤ `$28` **is the** global data pointer **that points the static-data-segment.**
   ⑥ `$29` **is the** stack-pointer **that points the top-of-stack 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.

2. Experimental Work:

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  3,5,6,2,0,4
      .text 0x00400000
      .globl main
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 f10-key after initializing `PC` to `0x00400000` (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 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.

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

EASTERN MEDITERRANEAN UNIVERSITY
COMPUTER ENGINEERING DEPARTMENT

Spring
2007-08

CMPE 325 - Computer Architecture II
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

1- C-program to translate into MIPS code is ....
2- Fill in here the contents of the calculated array-elements (in hexadec.):

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

Grading: Quiz Performance:
              Lab Performance:
              Asst. Observations: