Lab# 6 PROCEDURE

Instructor: I Putu Danu Raharja.

Objectives:
Learn how to write a procedure in MIPS assembly language programs.

Method:

Translate an algorithm from pseudo-code into assembly language.

Preparation:

Read the chapter 2 of lecture textbook.

6.1 INTRODUCTION

So far, we've only looked at programs consisting of a single long chunk of code. Each program has started at the top of the code, executed each instruction in turn (with an occasional detour for looping or decision-making), and then ended at the bottom of the code. That's fine for small programs, but larger programs require a programming construct known as a *subroutine*/*procedure*.

You've probably familiar with subroutines from a high-level language. In C, subroutines are known as functions, and in Pascal and Basic, they're known as procedures and functions. Subroutines, procedures, and functions all amount to the same thing – a separate section of code that optionally accepts well-defined inputs, promptly performs a certain action, and optionally returns a specific result value.

Subroutines let you build programs in a modular fashion, with the subroutines hiding the details of specific tasks so you can focus on the overall flow of the program. Subroutines can also make programs far more compact, since a single subroutine can be called from many places in a program, and can even perform different functions when passed different values. In a large program (whether written in assembler, C, Pascal, or some other language), subroutines are essential to creating orderly, maintainable code.

6.2 HOW PROCEDURES WORK

There are six steps that need to be accomplished in order to call and return from a procedure.

1. Place parameters in a place where the procedure can access them.

2. Transfer control to the procedure
3. Acquire the storage resources needed for the procedure.
4. Execute the procedure
5. Place the result value in a place where the calling program can access it.
6. Return control to the point of origin.

MIPS software follows the following convention in allocating its 32 registers for procedure calling:

- **$a0 – $a3**: four argument registers in which to pass parameters
- **$v0 – $v1**: two value registers in which to return values
- **$ra**: one return address register to return to the point of origin

The code that calls the procedure executes a **`jal`** instruction, which saves the address of the following instruction (PC+4) in register **$ra** and then loads PC with the address of the desired subroutine, thereby branching to the subroutine. The subroutine then executes just as any other code would. Procedures can – and often do – contain calls to other procedures; in fact, properly designed subroutines can even call themselves, a practice known as recursion.

When the subroutine has finished its task, it executes a **`jr $ra`** instruction, which jumps to the address stored in register $ra. This causes execution of the calling routine to resume at the instruction following the **`jal X`** instruction.

However, since the procedure may utilize any registers needed by the caller, those registers must be preserved before the procedure called and then be restored back after the procedure completed the tasks.

The ideal data structure for spilling registers is a *stack*. MIPS software allocates **`$sp`** to track as the top of stack (TOS). The stack grows from higher address to lower address.

6.3 STACK MANIPULATION

The MIPS architecture does not explicitly support stack operations. In MIPS, we have to manipulate the stack pointer register to implement the stack.

A. PUSH operation

We have to decrement **$sp** to make room for the value being pushed onto the stack. For example, if we want to push the contents of **$a0**, we have to reserve four bytes of stack space and use the **sw** instruction to push the value as shown below:

```
addiu    $sp,$sp,-4      # reserve 4 bytes of stack
sw       $a0,0($sp)      # save the register
```

B. POP operation

The operation can be implemented by using the load and add instructions. For example, to restore the value of $a0 from the stack, we use the lw instruction to copy the value and increment $sp by 4 as shown below:

```
lw       $a0,0($sp)      # restore the two registers
addiu    $sp,$sp,-4      # clear 4 bytes of stack
```

6.4 How To Preserve Registers

To preserve registers efficiently, MIPS software separates 18 of the registers into two groups:

- **$t0 – $t9**: 10 temporary registers that are NOT preserved by the called procedure on a procedure call. It is the caller's responsibility to preserve any of them.
- **$s0 – $s7**: 8 saved registers that must be preserved on a procedure call (if used, the called procedure saves and restores them).

6.5 Nested Procedures

Procedures that do not call others are called *leaf* procedures. Non-leaf procedures must push all necessary registers before calling other procedures.

6.6 Example

The following is a complete program consisting of a main program and a procedure.

```
            .data
array:      .word   -4, 5, 8, -1
msg1:       .asciiz  "\n The sum of positive values= "
msg2:       .asciiz  "\n The sum of negative values= "
            .globl   main
            .text
```

```
main:
            li      $v0, 4
            la      $a0, msg1
            syscall
            la      $a0, array      # Initialize address parameter
            li      $a1, 4          # Initialize length parameter
            jal     Sum             # Call sum function
            addu    $a0, $v0, $0    # sum of positive returned in $v0
            li      $v0, 1
            syscall
            li      $v0, 4
            la      $a0, msg2
            syscall
            addu    $a0, $v1, $0    #sum of negative returned in $v1
            li      $v0, 1
            syscall
            li      $v0, 10
            syscall
Sum:        addu    $v0, $0, $0
            addu    $v0, $0, $0
Loop:       blez    $a1, Return
            addi    $a1, $a1, -1
            lw      $t0, 0($a0)
            addi    $a0, $a0, 4
            blez    $t0, negative
            add     $v0, $v0, $t0
            j       Loop
negative:   add     $v1, $v1, $t0
            j       Loop
Return:jr   $ra
```

6.7 EXERCISE:

Write a *recursive* function to solve *Towers of Hanoi* puzzle. A stack of N disks of decreasing size (from bottom to top) is placed on one of three posts. The task is to move the disks one at a time from the first post to the second. To do this, any disk can be moved from any post to any other post, subject to the rule that you can never place a larger disk over a smaller disk. The (spare) third post is provided to make the solution possible.

```
# Function name: Hanoi(N, a, b)
# It will display the sequence of steps to move N disks from a to b
# Register usage in function:
#       a0 = N
#       a1 = the source post number (for example a=1, b=2, c=3)
#       a2 = the destination post number
# Algorithmic description
#       if (a0 == 1)
#               print ("move a disk from", a1, " to ", a2)
#               return
#       else
#               a3 = 6 – a1 – a2 (Why ?)
#               Hanoi(a0 – 1, a1, a3)
#               print ("move a disk from", a1, " to ", a2)
#               Hanoi(a0 – 1, a3, a2)
# How to use:
#       addiu   $a0, $zero, 4
#       addiu   $a1, $zero, 1
#       addiu   $a2, $zero, 2
#       jal     Hanoi
```