

Lecture 12

Function and Procedure Calls in MIPS

- Functions allow for more structured programs, code reuse, and makes code easier to understand and maintain.

Function Call Steps

- There are 6 steps to calling a function in MIPS.
 1. Place the parameters in argument registers – this is where the procedure can access them.
 2. Save the return address and transfer the control to the procedure.
 3. Allocate space for the function (local variables/some registers).
 4. Perform the desired task.
 5. Place the result in *value* registers – where the calling function can access them.
 6. Return control to the main program.

Function Call Bookkeeping

- The fastest, most convenient place to store data is in the registers.
- When working with function calls, the following must be considered:
 - Parameters (arguments) **\$a0, \$a1, \$a2, \$a3**
 - Four registers for arguments; if there are more than four, use the *stack*.
 - Return address **\$ra**
 - Return value **\$v0, \$v1**
 - Two registers to return values, unlike in high level languages.
 - Local variables **\$s0, \$s1, . . . , \$s7**

```
C  ... sum(a,b);... /* a,b:$s0,$s1 */
    }
    int sum(int x, int y) {
        return x+y;
    }
```

Caller Program

```
M  Pass up to four parameters in the argument registers: $a0,
I  $a1, $a2, $a3, and jump to procedure Sum.
P
S  1000 add  $a0,$s0,$zero  # x = a
    1004 add  $a1,$s1,$zero  # y = b
    1008 addi $ra,$zero,1016 # $ra=1016
    1012 j    sum           #jump to sum
    1016 ... <- Return address

    2000 sum: ...

    20xx jr  $ra
```

Instructions Support for Functions

- Use jump instructions to call and return from function calls.
 - To call a function: `jal function_label` → saves the PC+4 in register `$ra`, then jumps to the address of the function.
 - To return from a function: `jr $ra` → jump to the address in `$ra`.
- We have a single instruction to jump and save return address: *jump and link (jal)*.

Without using jal:

```
1008    addi $ra,$zero,1016 # $ra=1016
1012    j     sum           # go to sum
1016    ...
```

Using jal:

```
1008    jal  sum           # $ra=1012,go to sum
1012    ...
```

Saving the Register Contents

- The procedure can use the same registers that were used by *main* (caller) and which might later be needed by *main* after the procedure returns. So if the procedure does use such registers, it must first save these on the *stack* before it uses them.
- After the procedure completes, it must load these original values from the stack back to their respective registers. It then returns to *main* (caller) by using `jr`.
- To limit register spilling, MIPS uses the following convention:
 1. `$t0 - $t9`: 10 *temporary* registers, not preserved by the procedure (*'callee'*).
 2. `$s0 - $s7`: 8 *saved* registers that must be preserved and restored if used by the procedure.

Converting C code to a procedure

C Code:

```
f = (g + h) - (i + j);
```

Assume Register Allocations: **f: \$s0, g: \$s1, h: \$s2, i: \$s3, j: \$s4**

MIPS Instructions:

Operationally, the code would be:

```
add $s0,$s1,$s2# $s0 = g+h
add $t1,$s3,$s4# $t1 = i+j
sub $s0,$s0,$t1# f=(g+h) - (i+j)
```

If it's a procedure, **g, h, i, j** will be passed through **\$a0, \$a1, \$a2, \$a3**.

However, the registers **\$s0** and **\$t1** need to be saved in the stack.

```
addi $sp,$sp, -8      # Adjust stack for 2 items
sw   $t1, 4($sp)     # save register $t1 on stack
sw   $s0, 0($sp)     # save register $s0 on stack
add  $s0,$a0,$a1     # $s0 = g+h
add  $t1,$a2,$a3     # $t1 = i+j
sub  $s0,$s0,$t1     # f=(g+h) - (i+j)
add  $v0,$s0,$zero  # Return value of f
```

Before returning, restore the old values in the registers.

```
lw   $s0, 0($sp)
lw   $t1, 4($sp)
addi $sp, $sp, 8     # adjust stack pointer
jr   $ra             # Jump back to main
```

Note that MIPS convention demands that the procedure save the value that is in **\$s0** (a 'saved' register) on the *stack* before it uses the register, and then restore that value just before returning. It does not demand that the same be done for **\$t1** (a 'temporary' register), but we do it anyway in the code above.

The code in the caller of the procedure should not, in general, assume that the value it left in **\$t1** when it called the procedure will necessarily still be there when the procedure returns. It can, however, make that assumption about **\$s0**.

Nested Procedures

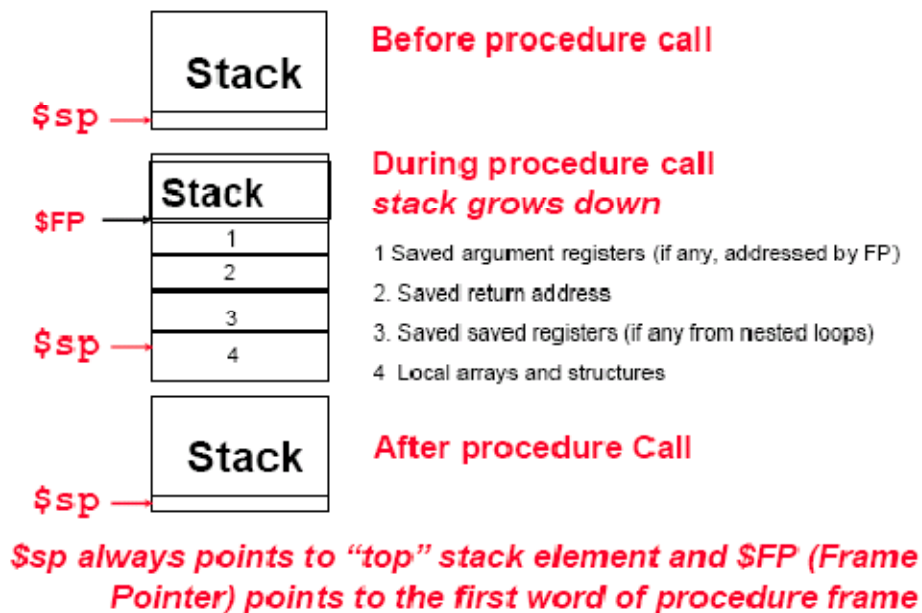
- When calling a procedure from another procedure,
 - Arguments need to be passed.
 - The return address needs to be stored.

Where are we going to put them? Eventually, we will run out of registers. Therefore, the stack is used for storing all these items temporarily.

- Begin by allocating space for the local variables on the stack

The Stack

- The stack segment (registers and local variables) is called a *procedure frame* or *activation record*.
- When we call a function, its activation record is stored on stack.
- When we return, this record is deleted from the stack.
- Some MIPS software use a *frame pointer*, **\$fp**, to point to the first word of activation record.
- All locations in an activation record are accessed using **\$fp**.
 - The **\$sp** may change
 - This makes accessing locations difficult if only **\$sp** is used.
 - **\$fp** provides a stable pointer whose value does not change during execution of a function.



Ex: nested procedure call

```
## vowel.asm - prints out number of vowels in
##           - the string str
##
##           a0 - points to the string
#####
#           text segment                               #
#####
```

```
        .text
        .globl main
main:           # execution starts here

        la    $a0,str
        jal   vcount    # call vcount

        move  $a0,$v0
        li    $v0,1
        syscall      # print answer

        la    $a0,endl
        li    $v0,4
        syscall      # print newline

        li    $v0,10
        syscall      # au revoir...
```

```
#-----
# vowelp - takes a single character as a
# parameter and returns 1 if the character
# is a lower case vowel otherwise return 0.
#     a0 - holds character
#     v0 - returns 0 or 1
#-----
```

```
vowelp: li    $v0,0
        beq  $a0,'a',yes
        beq  $a0,'e',yes
        beq  $a0,'i',yes
        beq  $a0,'o',yes
        beq  $a0,'u',yes
        jr   $ra
yes:    li    $v0,1
        jr   $ra
```

← Does not save any registers. Why?

- \$a0 is not changed, only read.
- \$v0 is not one of the 'saved' registers under MIPS convention. \$v0 & \$v1 are used to return function values.
- \$ra is not saved because function *vowelp* itself does not call another function. Also, \$ra is not a *saved* register: if the caller to *vowelp* needs to preserve its value, he has to take the responsibility of saving it before making the function call.

```

#-----
# vcount - use vowelp to count the vowels in a
# string.
#   a0 - holds string address
#   s0 - holds number of vowels
#   v0 - returns number of vowels
#-----

vcount:
    sub $sp,$sp,16      # save registers on stack
    sw $a0,0($sp)      # $a0 & $ra are saved because vcount calls vowelp
    sw $s0,4($sp)
    sw $s1,8($sp)
    sw $ra,12($sp)

    li $s0,0           # initialize count of vowels
    move $s1,$a0       # address of string

nextc: lb $a0,($s1)    # get each character
    beqz $a0,done      # zero(null char)marks end
    jal vowelp          # call vowelp ← Argument is in $a0
    add $s0,$s0,$v0    # add 0 or 1 to count ← Return value is in $v0
    add $s1,$s1,1      # move along string
    b nextc

done:  move $v0,$s0    # return result in $v0 ← Count is in $s0

    lw $a0,0($sp)      # restore registers
    lw $s0,4($sp)      ← Restore all the registers before leaving
    lw $s1,8($sp)      function
    lw $ra,12($sp)
    add $sp,$sp,16
    jr $ra             ← Actual return

#####
#                   data segment                   #
#####

.data
str:  .asciiz "long time ago in a galaxy far away"
endl: .asciiz "\n"

```

Recursion

- Check out the example in Appendix B
- Recursion in a high level language is implemented using the stack and activation records.
- The stack is absolutely necessary for every recursive call that has not been completed; its activation record is on the stack.
- As the recursion unwinds, these records get deleted from the stack.