# INSTRUCTIONS: LANGUAGE OF THE COMPUTER

Prof. Sebastian Eslava Ph.D.

# Introduction: Supporting Procedures

- What is a procedure (function, method, subroutine)?
  - Used for structured programming
    - Allow code reuse
- What needs to be done to implement procedures?
  - Changing the program's flow of control
    - When the procedure is called
    - Resume execution after the procedure call
  - Allocating memory space for local variables
  - Passing arguments and returning values

# Procedure Calling

- 6 required steps for supporting procedures:
  1. Place parameters in registers/stack
  2. Transfer control to the procedure
  3. Acquire storage resources for procedure
  4. Perform procedure's operations/tasks
  5. Place result in register/stack for caller
  6. Return to place of call

- Before calling a function:

  the calling function (known as the *caller*) needs to save values of registers that the function may use and override
  → *caller-saved registers*
- The called function also needs to save values of some registers
  → *callee-saved registers.*

# Register Usage

- $a0 – $a3: arguments (reg's 4 – 7)
- $v0, $v1: result values (reg's 2 and 3)
- $t0 – $t9: temporaries
  - Can be overwritten by callee
- $s0 – $s7: saved
  - Must be saved/restored by callee
- $gp: global pointer for static data (reg 28)
- $sp: stack pointer (reg 29)
- $fp: frame pointer (reg 30)
- $ra: return address (reg 31)

# Procedure Call Instructions

- For procedure call: “jump and link” instruction
  - 2 versions

```
jal target_label  # jal to label
jalr $dest              # jal to reg. dest
```

  - Address of following instruction (i.e. PC + 4) put in the dedicated register $ra before control is transferred
  - Jumps to target address specified by label or register

# Procedure Return Instruction

- For procedure return: jump register

```
jr $ra  #goto addr. specified in $ra
```

  - Copy $ra back into PC → transfer control back to the caller.
  - Can also be used for computed jumps
    - e.g., for case/switch statements

# Simple Procedure Example

- **In MIPS, arguments and results are passed in registers.**
- **Example:**

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

- Up to 4 arguments can be passed by placing them in registers $a0-$a3 before calling *jal*

  → Arguments g, h, i and j in $a0, $a1, $a2 and $a3, respectively
- Up to 2 values can be returned by placing them in $v0 and $v1 before calling *jr*

  → Result stored in $v0
- Local variable *f* will use the saved register $s0 (hence, need to save $s0 on stack)

# Leaf Procedure Example

- MIPS code:

```
leaf_example:
    addi $sp, $sp, -4
    sw   $s0, 0($sp)
    add  $t0, $a0, $a1
    add  $t1, $a2, $a3
    sub  $s0, $t0, $t1
    add  $v0, $s0, $zero
    lw   $s0, 0($sp)
    addi $sp, $sp, 4
    jr   $ra
```

Save $s0 on stack

Procedure body

Result

Restore $s0

Return

# Procedure

- A procedure
  - is called using jal, passing arguments in $a0-$a3
  - Places results in $v0-$v1 and returns using jr $ra
- Example: Procedure Swap

```
swap: sll $t1, $a1, 2    # $t1 = k * 4
      add $t1, $a0, $t1  # $t1 = v+(k*4)
                         #   (address of v[k])
      lw $t0, 0($t1)     # $t0 (temp) = v[k]
      lw $t2, 4($t1)     # $t2 = v[k+1]
      sw $t2, 0($t1)     # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)     # v[k+1] = $t0 (temp)
      jr $ra             # return to calling routine
```

# Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

# Non-Leaf Procedure Example

- C code:

```
int fact (int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```

  - Argument n in $a0
  - Result in $v0

# Non-Leaf Procedure Example

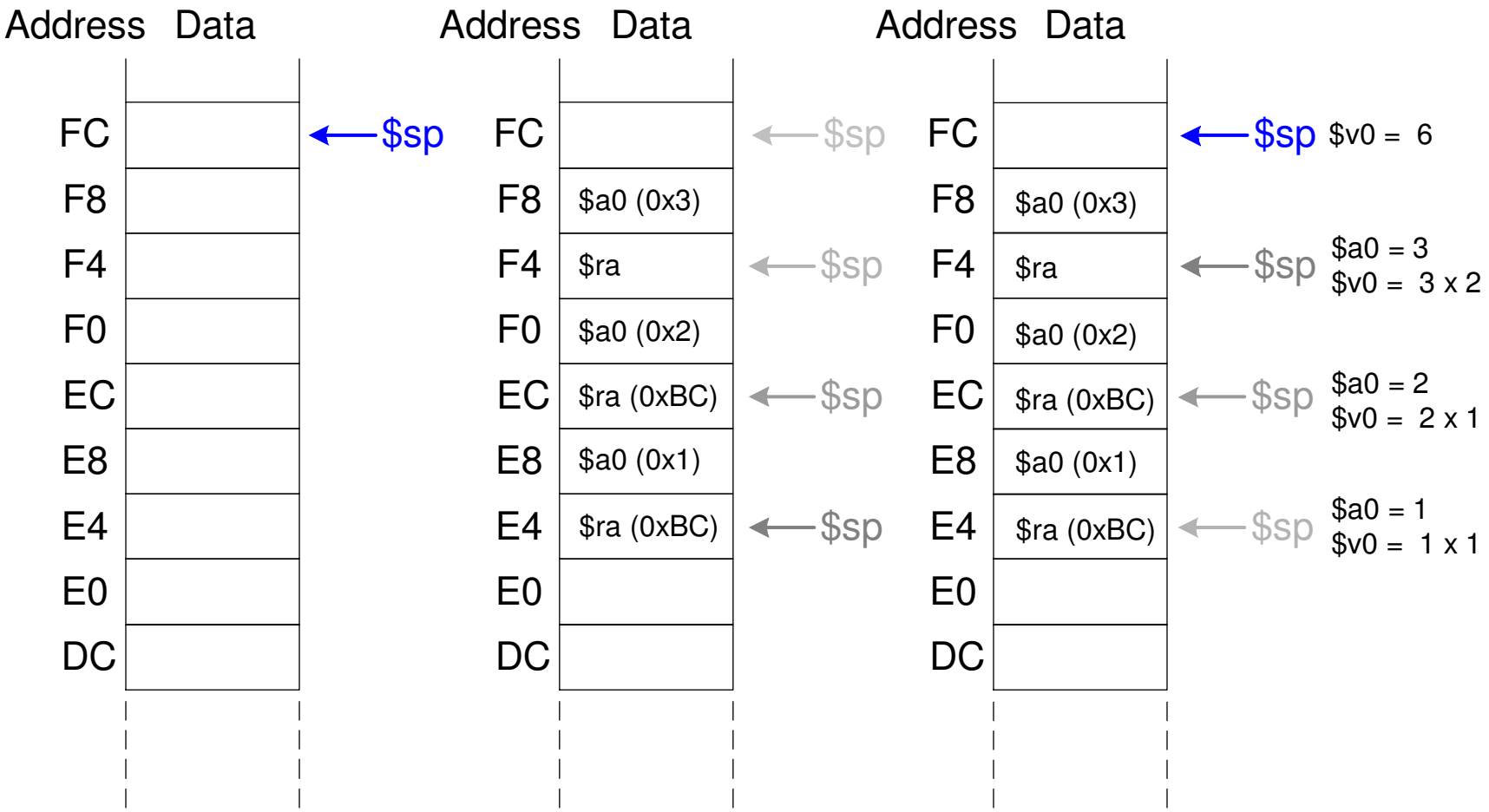
- MIPS code:

```
fact:
    addi $sp, $sp, -8      # adjust stack for 2 items
    sw   $ra, 4($sp)       # save return address
    sw   $a0, 0($sp)       # save argument
    slti $t0, $a0, 1       # test for n < 1
    beq  $t0, $zero, L1
    addi $v0, $zero, 1     # if so, result is 1
    addi $sp, $sp, 8       #   pop 2 items from stack
    jr   $ra               #   and return
L1: addi $a0, $a0, -1      # else decrement n
    jal  fact              # recursive call
    lw   $a0, 0($sp)       # restore original n
    lw   $ra, 4($sp)       #   and return address
    addi $sp, $sp, 8       # pop 2 items from stack
    mul  $v0, $a0, $v0     # multiply to get result
    jr   $ra               # and return
```

# Procedure example

- n = 3
  - Describe the stack behavior
  - Trace the fact execution
  - Trace the saved registers

# Stack during Recursive Call



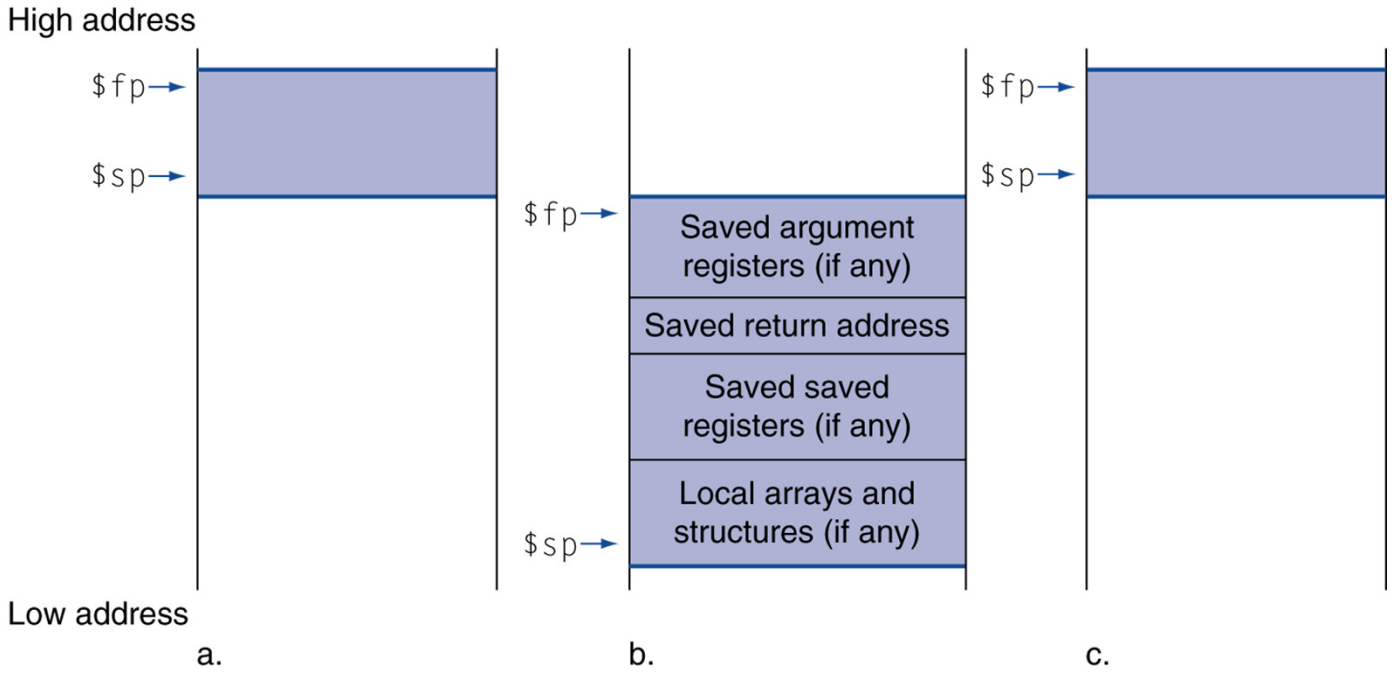
# Procedure Call Summary

- Caller
  - Put arguments in `$a0–$a3`
  - Save any registers that are needed (`$ra`)
  - `jal callee`
  - Restore registers
  - Look for result in `$v0`
- Callee
  - Save registers that might be disturbed (`$s0–$s7`)
  - Perform procedure
  - Put result in `$v0`
  - Restore registers
  - `jr $ra`

# Registers preserved

| Name | Register number | Usage | Preserved on call? |
|---|---|---|---|
| $zero | 0 | The constant value 0 | n.a. |
| $v0–$v1 | 2–3 | Values for results and expression evaluation | no |
| $a0–$a3 | 4–7 | Arguments | no |
| $t0–$t7 | 8–15 | Temporaries | no |
| $s0–$s7 | 16–23 | Saved | yes |
| $t8–$t9 | 24–25 | More temporaries | no |
| $gp | 28 | Global pointer | yes |
| $sp | 29 | Stack pointer | yes |
| $fp | 30 | Frame pointer | yes |
| $ra | 31 | Return address | yes |

# Local Data on the Stack




- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage