

## Lecture 4

### **Instruction Set Architecture(2)**

# Recap

---

- Instruction set architecture
  - RISC vs. CISC
  - MIPS/ARM/x86
- Instructions:
  - Arithmetic instruction: add, sub, ...
  - Data transfer instruction: lw, sw, lh, sh, ...
  - Logical instruction: and, or, ...
  - Conditional branch beq, bne, ...
- Basic concepts:
  - Operands: register vs. memory vs. immediate
  - Numeric representation: signed, unsigned, sign extension
  - Instruction format: R-format vs. I-format

# Today's topic

---

- More control instructions
- Procedure call/return

# Control Instructions: if else

---

- Conditional branch: Jump to instruction L1 if register1 equals to register2: `beq register1, register2, L1`  
Similarly, `bne` and `slt` (set-on-less-than)

- Unconditional branch:

```
j    L1  
jr   $s0
```

Convert to assembly:

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

```
                bne    $s3, $s4, Else  
                add    $s0, $s1, $s2  
                j      Exit  
Else:          sub    $s0, $s1, $s2  
Exit:
```

# Loop

---

Convert to assembly:

```
while (save[i] == k)
    i += 1;
```

i and k are in \$s3 and \$s5 and  
base of array save[] is in \$s6

```
Loop: sll    $t1, $s3, 2
      add    $t1, $t1, $s6
      lw     $t0, 0($t1)
      bne    $t0, $s5, Exit
      addi   $s3, $s3, 1
      j      Loop
```

Exit:

# More Conditional Operations

---

- How to compile:
  - If ( $a < b$ ) ..., else, ...
- `slt rd, rs, rt`
  - if ( $rs < rt$ )  $rd = 1$ ; else  $rd = 0$ ;
- `slti rt, rs, constant`
  - if ( $rs < \text{constant}$ )  $rt = 1$ ; else  $rt = 0$ ;
- Use in combination with `beq`, `bne`  
`slt $t0, $s1, $s2 # if ($s1 < $s2)`  
`bne $t0, $zero, L # branch to L`

# Example

---

Convert to assembly:

Convert to assembly:

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

```
        slt    $t0, $s3, $s4
        beq    $t0, $zero, Else
        add    $s0, $s1, $s2
        j      Exit
Else:    sub    $s0, $s1, $s2
Exit:
```

i and j are in \$s3 and \$s4,  
f,g and h are in \$s0, \$s1 and \$s2

# Question


---

- C has many statements for decisions and loops, while MIPS has few. Which of the following do or do not explain this imbalance? Why?
  - A. More decision statements make code easier to read and understand.
  - B. Fewer decision statements simplify the task of the underlying layer that is responsible for execution.
  - C. More decision statements mean fewer lines of code, which generally reduces coding time.
  - D. More decision statements mean fewer lines of code, which generally results in the execution of fewer operations.



# Pseudo Instructions

---

- `blt $s0, $s1, Label`       `slt $s2, $s0, $s1`
    - If  $s0 < s1$ , jump to Label
  - `bgt $s0, $s1, Label`
    - If  $s0 > s1$ , jump to Label
  - `ble $s0, $s1, Label`
    - If  $s0 \leq s1$ , jump to Label
  - `beqz $s0, Label`
    - If  $s0 == 0$ , jump to Label
  - `li $t0, 5`
    - Load immediate,  $t0 = 5$
  - `Move $t0, $s0`
    - $t0 = s0$
- `bne $s2, $zero, Label`

**There is no such instructions in hardware,  
The assembler translates them into a  
combination of real instructions**

# Branch Instruction Design

---

- Why not `blt`, `bge`, etc?
- Hardware for `<`, `≥`, ... slower than `=`, `≠`
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!
- `beq` and `bne` are the common case
- This is a good design compromise

# Signed vs. Unsigned

---

- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltui`
- Example
  - `$s0 = 1111 1111 1111 1111 1111 1111 1111 1111`
  - `$s1 = 0000 0000 0000 0000 0000 0000 0000 0001`
  - `slt $t0, $s0, $s1 # signed`
    - $-1 < +1 \Rightarrow \$t0 = 1$
  - `sltu $t0, $s0, $s1 # unsigned`
    - $+4,294,967,295 > +1 \Rightarrow \$t0 = 0$

**The register contains bits without meaning.**

**Are the bits represents a signed number or unsigned one? See the instruction!**

# Procedures

---

- A procedure or function is one tool used by the programmers to structure programs
  - Benefit: easy to understand, reuse code
- We can think of a procedure like a spy
  - acquires resources → performs task → covers his tracks → returns back with desired result
- When the procedure is executed (when the caller calls the callee), there are six steps
  - parameters (arguments) are placed where the callee can see them
  - control is transferred to the callee
  - acquire storage resources for callee
  - execute the procedure
  - place result value where caller can access it
  - return control to caller

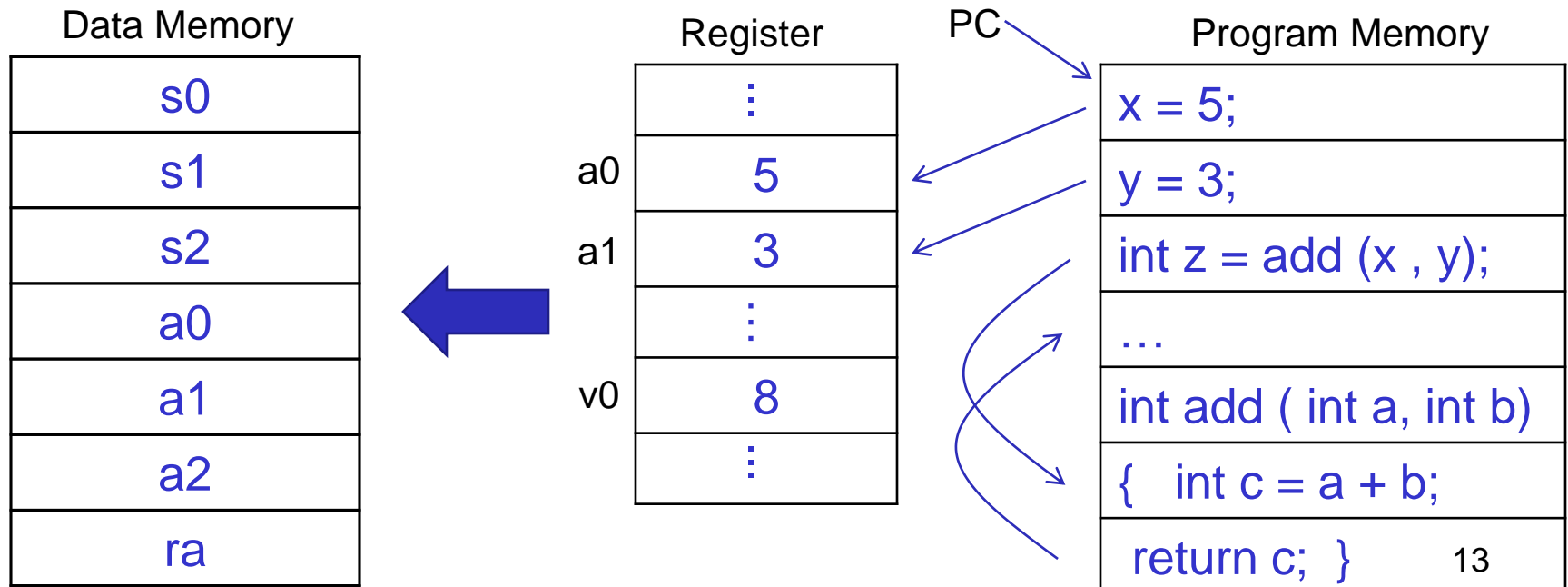
# Procedure Calling

Caller:

```
int x = 5;  
int y = 3;  
int z = add (x , y);  
x = x + 7;  
...
```

callee:

```
int add ( int a, int b)  
{  
    int c = a + b;  
    return c;  
}
```



# Registers Used during Procedure Calling

---

- The registers are used to hold data between the caller and the callee
  - \$a0 - \$a3: four **argument registers** to pass parameters
  - \$v0 - \$v1: **two value registers** to return the values
  - \$ra: one **return address register** to return to the point of origin in the caller

# Jump and Link

---

- *program counter* (PC)
  - A special register maintains the address of the instruction currently being executed
- The procedure call is executed by invoking the jump-and-link (jal) instruction – the current PC (actually, PC+4) is saved in the register \$ra and we jump to the procedure's address (the PC is accordingly set to this address)

jal    NewProcedureAddress

- Since jal may over-write a relevant value in \$ra, it must be saved somewhere (in memory?) before invoking the jal instruction
- How do we return control back to the caller after completing the callee procedure?

# Registers

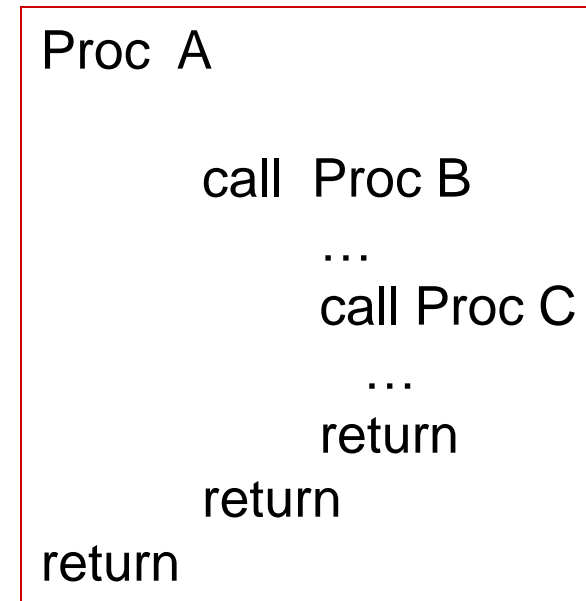
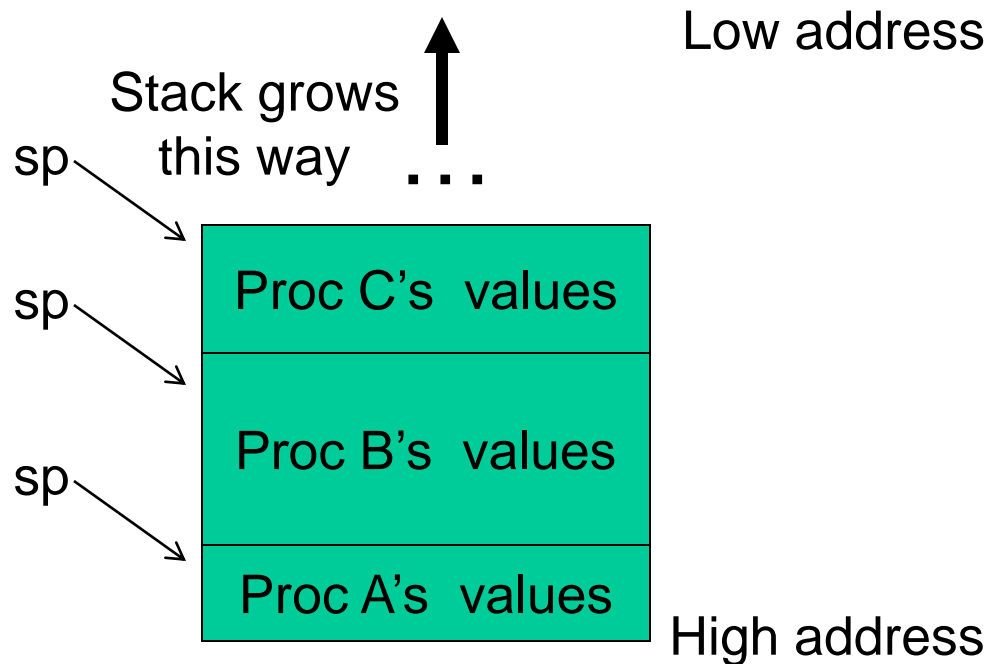
---

- The 32 MIPS registers are partitioned as follows:
  - Register 0 : \$zero      always stores the constant 0
  - Regs 2-3 : \$v0, \$v1      return values of a procedure
  - Regs 4-7 : \$a0-\$a3      input arguments to a procedure
  - Regs 8-15 : \$t0-\$t7      temporaries
  - Regs 16-23: \$s0-\$s7      variables
  - Regs 24-25: \$t8-\$t9      more temporaries
  - Reg 28 : \$gp      global pointer
  - Reg 29 : \$sp      stack pointer
  - Reg 30 : \$fp      frame pointer
  - Reg 31 : \$ra      return address



# The Stack

The registers for a procedure are volatile, it disappears every time we switch procedures. Therefore, a procedure's values in the registers are backed up in memory on a stack



# Storage Management on a Call/Return

---

- A new procedure must create space for all its variables on the stack
- Before executing the jal, the caller must save relevant values in \$s0-\$s7, \$a0-\$a3, \$ra, temps into its own stack space
- Arguments are copied into \$a0-\$a3; the jal is executed
- After the callee creates stack space, it updates the value of \$sp
- Once the callee finishes, it copies the return value into \$v0, frees up stack space, and \$sp is incremented
- On return, the caller may bring in its stack values, ra, temps into registers
- The responsibility for copies between stack and registers may fall upon either the caller or the callee

# Example 1- leaf procedure

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

The caller has saved:  
g → \$a0,  
h → \$a1,  
i → \$a2,  
j → \$a3,  
return address → \$ra

Save t0,t1,s0  
Protect environment

Procedure body

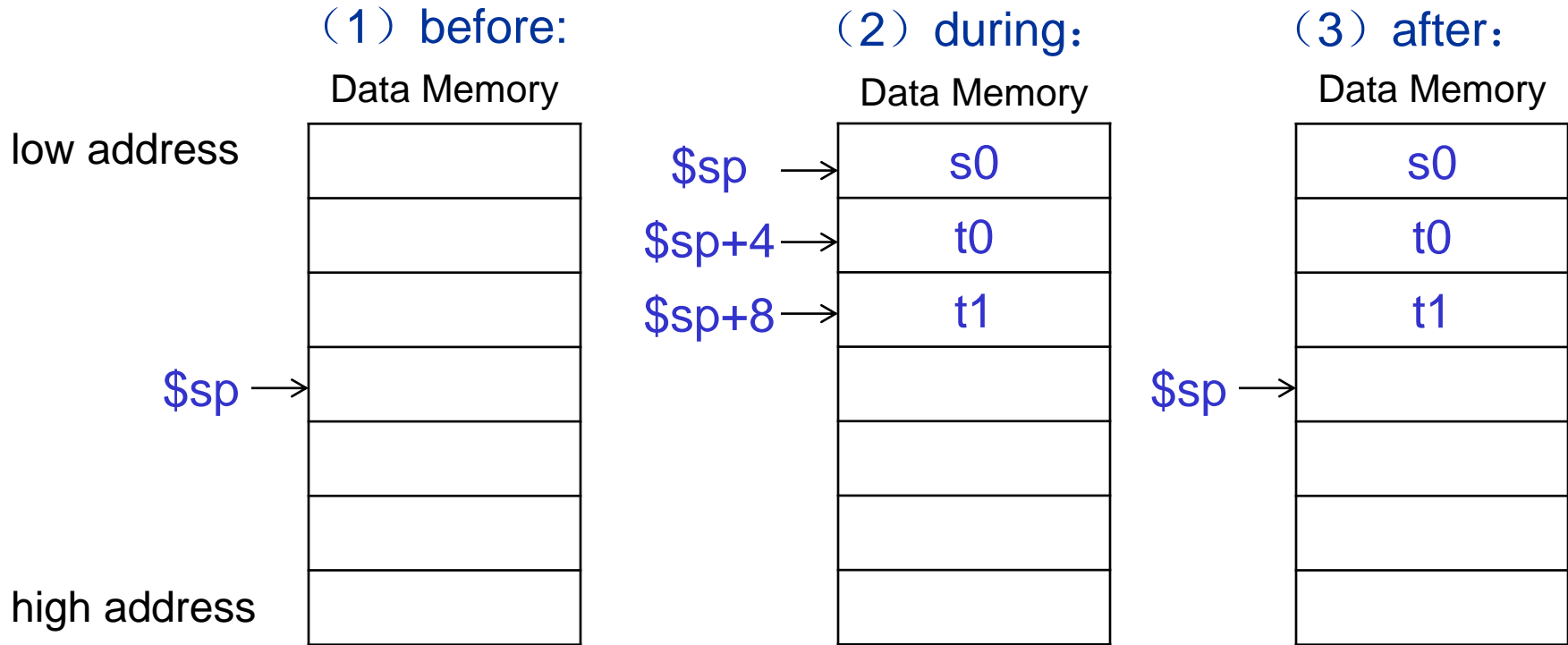
Restore t0 t1 s0

Return result

leaf\_example:

```
addi    $sp, $sp, -12
sw       $t1, 8($sp)
sw       $t0, 4($sp)
sw       $s0, 0($sp)
add      $t0, $a0, $a1
add      $t1, $a2, $a3
sub      $s0, $t0, $t1
add      $v0, $s0, $zero
lw       $s0, 0($sp)
lw       $t0, 4($sp)
lw       $t1, 8($sp)
addi     $sp, $sp, 12
jr       $ra
```

# Data in the stack in example 1



To avoid too many memory operations:

\$t0 - \$t9: temporary registers are not preserved by the callee

\$s0 - \$s7: saved registers must be preserved by the callee if used

## Example 2 – non-leaf procedure

---

- 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

# Example 2 – non-leaf procedure

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

## Notes:

The caller saves \$a0  
and \$ra in its stack  
space.

Temps are never saved.

Compare  $n < 1$

Return 1

Fact( $n-1$ )

Return  $n * \text{fact}(n-1)$

```
fact:
    addi    $sp, $sp, -8
    sw      $ra, 4($sp)
    sw      $a0, 0($sp)
    slti    $t0, $a0, 1
    beq     $t0, $zero, L1
    addi    $v0, $zero, 1
    addi    $sp, $sp, 8
    jr      $ra
L1:
    addi    $a0, $a0, -1
    jal     fact
    lw      $a0, 0($sp)
    lw      $ra, 4($sp)
    addi    $sp, $sp, 8
    mul     $v0, $a0, $v0
    jr      $ra
```

## Example 2 – non-leaf procedure



fact:

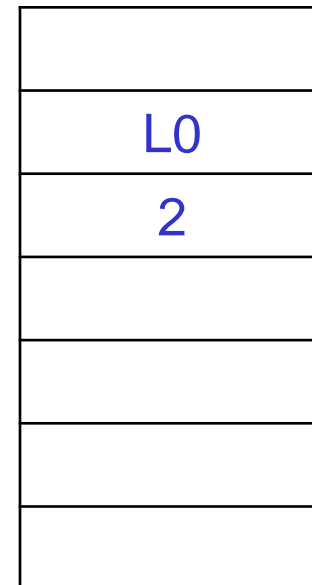
```
addi    $sp, $sp, -8
sw       $ra, 4($sp)
sw       $a0, 0($sp)
slti     $t0, $a0, 1
beq      $t0, $zero, L1
    addi  $v0, $zero, 1
    addi  $sp, $sp, 8
    jr    $ra
```

L1:

```
addi    $a0, $a0, -1
jal      fact
lw       $a0, 0($sp)
lw       $ra, 4($sp)
addi     $sp, $sp, 8
mul      $v0, $a0, $v0
jr       $ra
```

a0=2

\$sp →



high address

low address

## Example 2 – non-leaf procedure

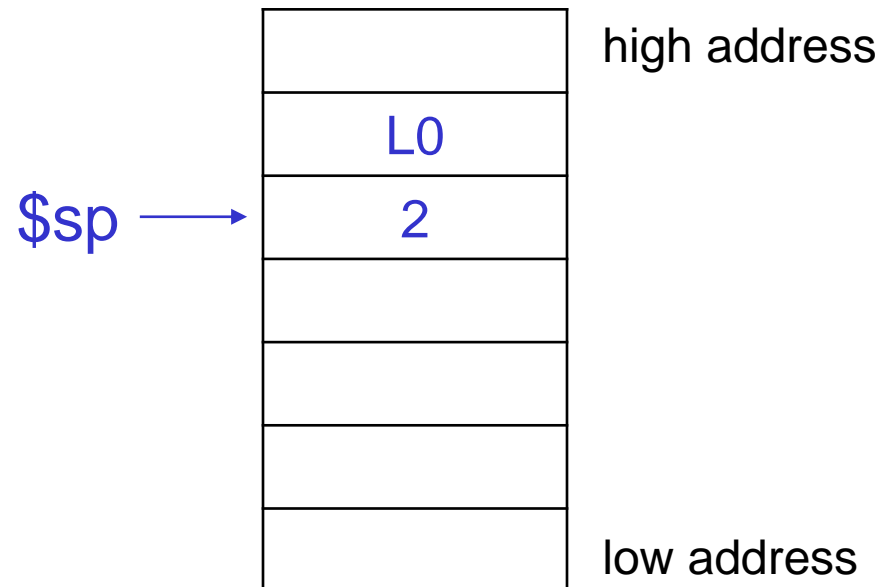
fact:

addi \$sp, \$sp, -8  
sw \$ra, 4(\$sp)  
sw \$a0, 0(\$sp)  
slti \$t0, \$a0, 1  
beq \$t0, \$zero, L1  
addi \$v0, \$zero, 1  
addi \$sp, \$sp, 8  
jr \$ra

L1:

addi \$a0, \$a0, -1  
jal fact  
lw \$a0, 0(\$sp)  
lw \$ra, 4(\$sp)  
addi \$sp, \$sp, 8  
mul \$v0, \$a0, \$v0  
jr \$ra

a0=2 t0=0





## Example 2 – non-leaf procedure

fact:

```
addi    $sp, $sp, -8
sw       $ra, 4($sp)
sw       $a0, 0($sp)
slti     $t0, $a0, 1
beq      $t0, $zero, L1
addi     $v0, $zero, 1
addi     $sp, $sp, 8
jr       $ra
```

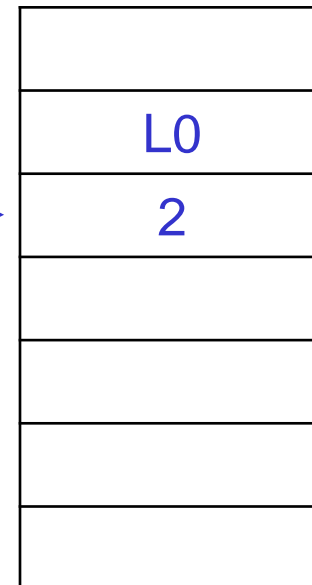
L1:

```
addi     $a0, $a0, -1
jal      fact
lw       $a0, 0($sp)
lw       $ra, 4($sp)
addi     $sp, $sp, 8
mul      $v0, $a0, $v0
jr       $ra
```

a0=1    t0=0



\$sp →



high address

low address

## Example 2 – non-leaf procedure

fact:

```
addi    $sp, $sp, -8
sw       $ra, 4($sp)
sw       $a0, 0($sp)
slti     $t0, $a0, 1
beq      $t0, $zero, L1
    addi  $v0, $zero, 1
    addi  $sp, $sp, 8
    jr    $ra
```

L1:

```
addi    $a0, $a0, -1
jal      fact
```

L2: lw \$a0, 0(\$sp)

```
lw       $ra, 4($sp)
```

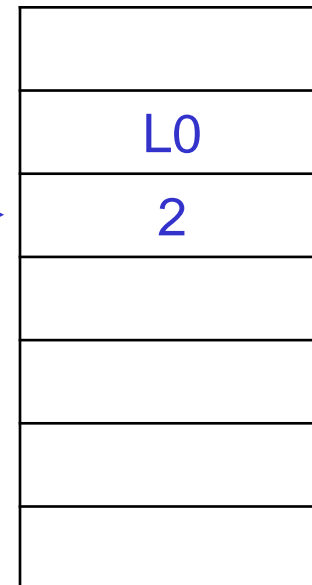
```
addi     $sp, $sp, 8
```

```
mul      $v0, $a0, $v0
```

```
jr       $ra
```

a0=1 t0=0 ra=L2

\$sp →



high address

low address

## Example 2 – non-leaf procedure

fact:

addi \$sp, \$sp, -8  
sw \$ra, 4(\$sp)  
sw \$a0, 0(\$sp)  
slti \$t0, \$a0, 1  
beq \$t0, \$zero, L1  
addi \$v0, \$zero, 1  
addi \$sp, \$sp, 8  
jr \$ra

L1:

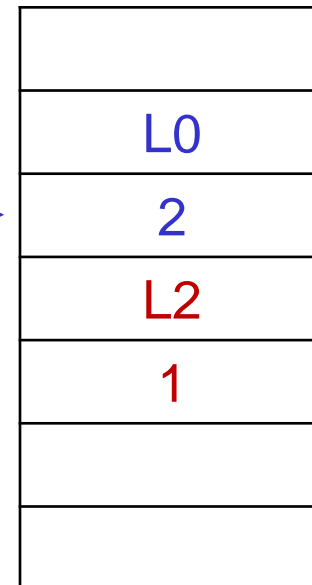
addi \$a0, \$a0, -1  
jal fact

L2: lw \$a0, 0(\$sp)

lw \$ra, 4(\$sp)  
addi \$sp, \$sp, 8  
mul \$v0, \$a0, \$v0  
jr \$ra

a0=1 t0=0 ra=L2

\$sp →



high address

low address

## Example 2 – non-leaf procedure

fact:

```
addi    $sp, $sp, -8
sw       $ra, 4($sp)
sw       $a0, 0($sp)
slti     $t0, $a0, 1
beq      $t0, $zero, L1
    addi  $v0, $zero, 1
    addi  $sp, $sp, 8
    jr    $ra
```

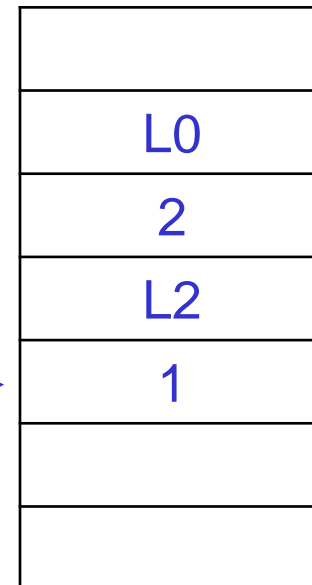
L1:

```
addi    $a0, $a0, -1
jal      fact
```

L2: lw \$a0, 0(\$sp)

```
lw       $ra, 4($sp)
addi     $sp, $sp, 8
mul      $v0, $a0, $v0
jr       $ra
```

a0=0   t0=0   ra=L2



## Example 2 – non-leaf procedure



fact:

```
addi    $sp, $sp, -8
sw       $ra, 4($sp)
sw       $a0, 0($sp)
slti     $t0, $a0, 1
beq      $t0, $zero, L1
    addi  $v0, $zero, 1
    addi  $sp, $sp, 8
    jr    $ra
```

L1:

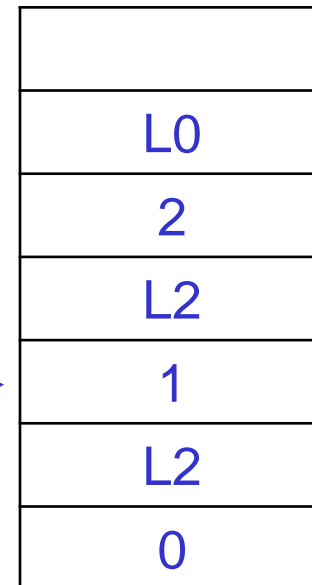
```
addi    $a0, $a0, -1
jal      fact
```

L2: lw \$a0, 0(\$sp)

```
lw       $ra, 4($sp)
addi     $sp, $sp, 8
mul      $v0, $a0, $v0
jr       $ra
```

a0=0    t0=1    ra=L2

\$sp →



high address

low address

## Example 2 – non-leaf procedure

fact:

addi \$sp, \$sp, -8  
sw \$ra, 4(\$sp)  
sw \$a0, 0(\$sp)  
slti \$t0, \$a0, 1  
beq \$t0, \$zero, L1  
addi \$v0, \$zero, 1  
addi \$sp, \$sp, 8  
jr \$ra

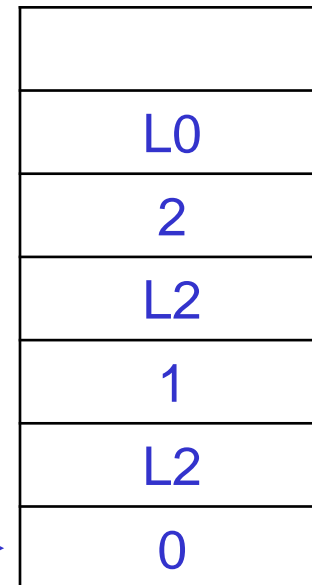
L1:

addi \$a0, \$a0, -1  
jal fact

L2: lw \$a0, 0(\$sp)

lw \$ra, 4(\$sp)  
addi \$sp, \$sp, 8  
mul \$v0, \$a0, \$v0  
jr \$ra

a0=0 t0=1 ra=L2 v0=1



high address

\$sp →

low address

## Example 2 – non-leaf procedure

fact:

```
addi    $sp, $sp, -8
sw       $ra, 4($sp)
sw       $a0, 0($sp)
slti     $t0, $a0, 1
beq      $t0, $zero, L1
    addi  $v0, $zero, 1
    addi  $sp, $sp, 8
    jr    $ra
```

L1:

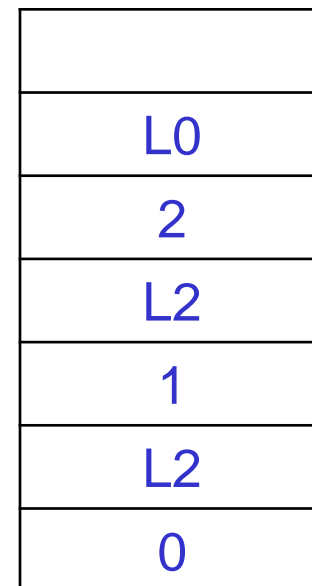
```
addi    $a0, $a0, -1
jal      fact
```

L2: lw \$a0, 0(\$sp)

```
lw       $ra, 4($sp)
addi     $sp, $sp, 8
mul      $v0, $a0, $v0
jr       $ra
```

a0=1 t0=1 ra=L2 v0=1

\$sp →



high address

low address

## Example 2 – non-leaf procedure

fact:

```
    addi    $sp, $sp, -8
    sw      $ra, 4($sp)
    sw      $a0, 0($sp)
    slti    $t0, $a0, 1
    beq     $t0, $zero, L1
    addi    $v0, $zero, 1
    addi    $sp, $sp, 8
    jr      $ra
```

L1:

```
    addi    $a0, $a0, -1
    jal     fact
```

L2: lw \$a0, 0(\$sp)

```
    lw      $ra, 4($sp)
```

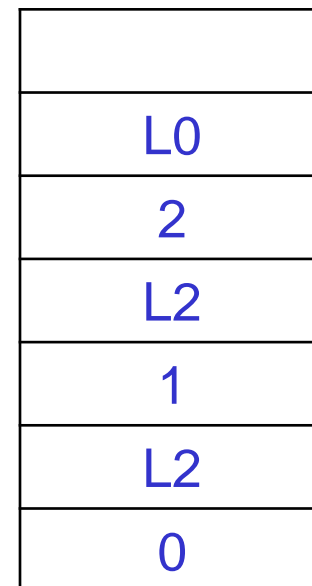
```
    addi    $sp, $sp, 8
```

```
    mul     $v0, $a0, $v0
```

```
    jr      $ra
```

a0=2 t0=1 ra=L0 v0=2

\$sp →



high address

low address



# Saving Conventions

---

- Caller saved: Temp registers \$t0-\$t9 (the callee won't bother saving these, so save them if you care), \$ra (it's about to get over-written), \$a0-\$a3 (so you can put in new arguments)
- Callee saved: \$s0-\$s7 (these typically contain “valuable” data)

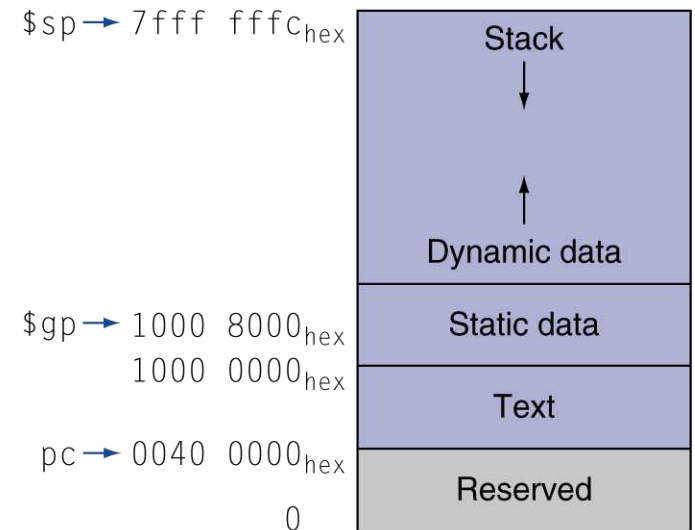
# Saving Conventions

---

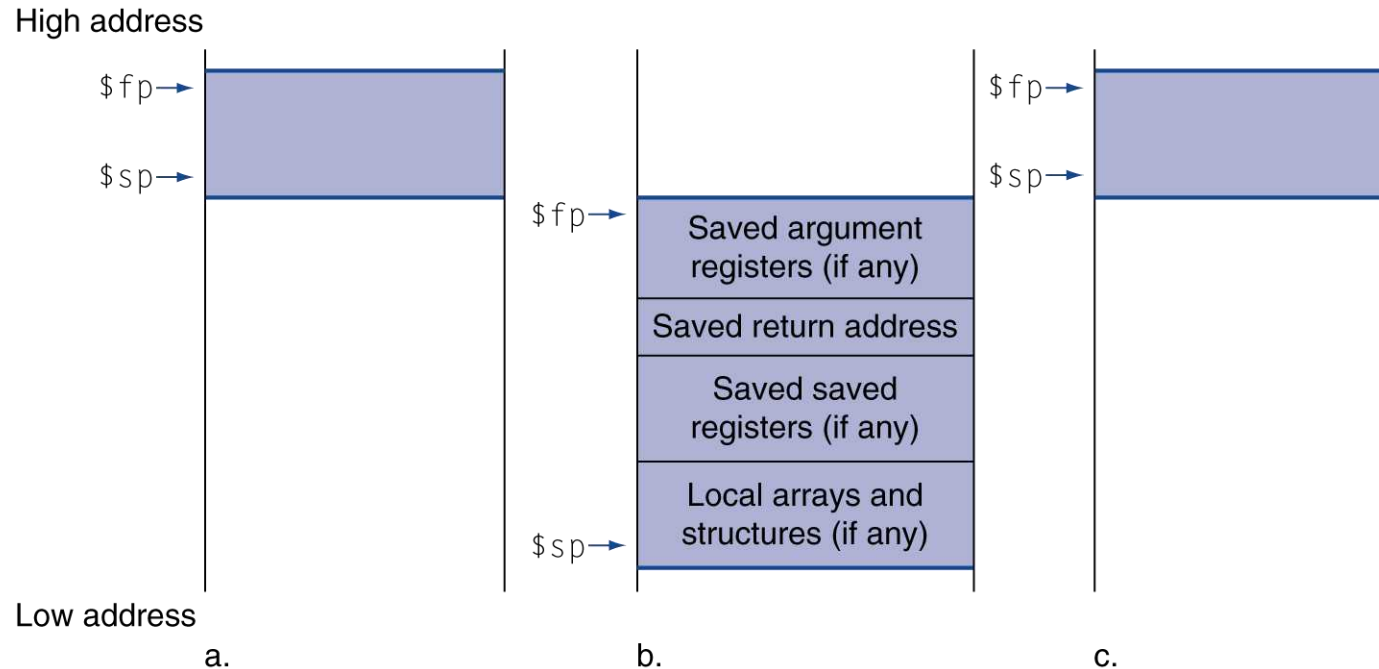
- Caller saved: Temp registers \$t0-\$t9 (the callee won't bother saving these, so save them if you care), \$ra (it's about to get over-written), \$a0-\$a3 (so you can put in new arguments)
- Callee saved: \$s0-\$s7 (these typically contain “valuable” data)

# Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - \$gp initialized to address allowing  $\pm$  offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



# 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