Computer Architecture
(ENE1004)

Lec – 6: Instructions: Language of the Computer (Chapter 2) - 5

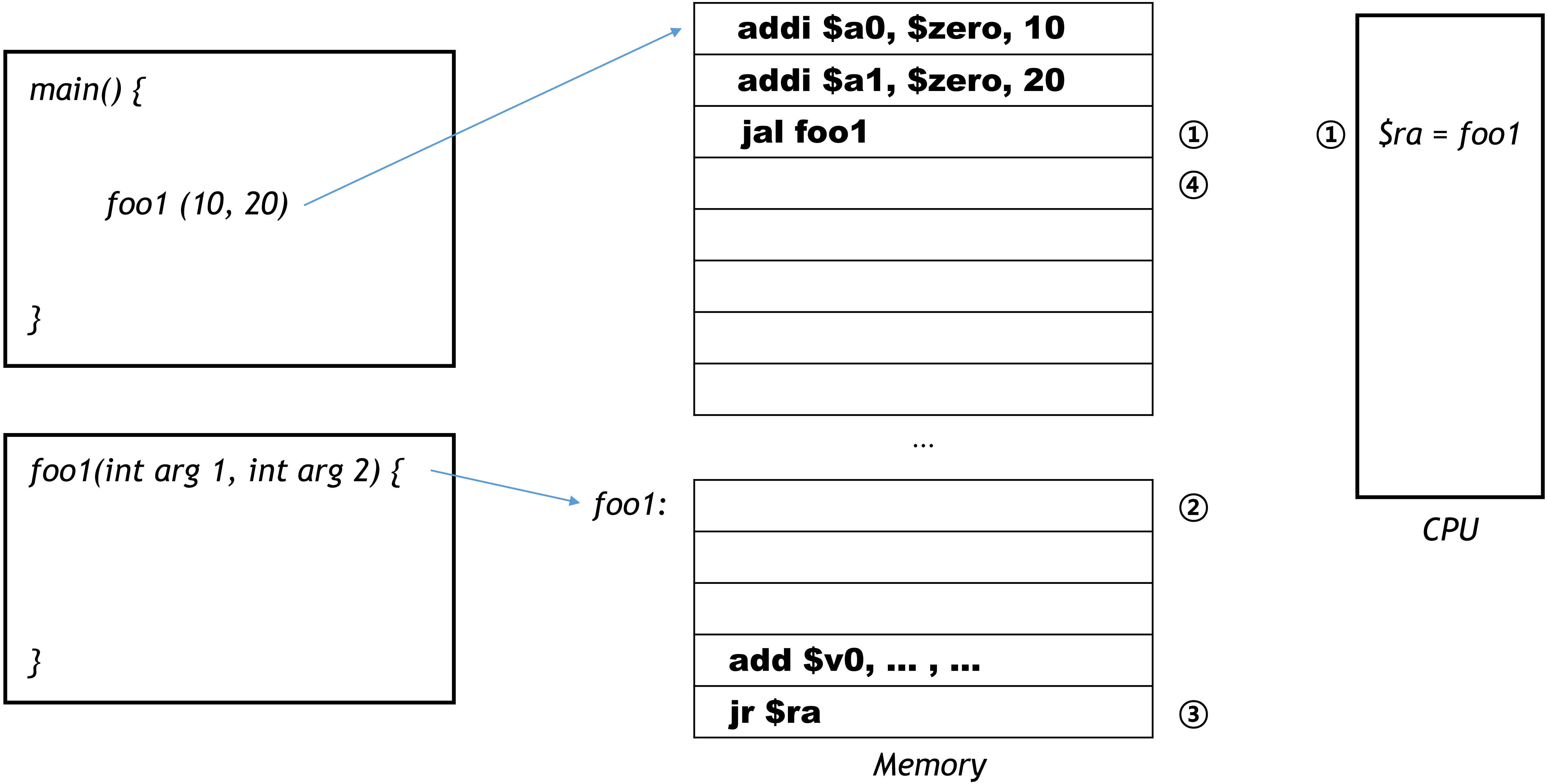
Supporting Procedures in Hardware

- Function (procedure) is one of the most widely used tool in programming
  - It makes programs easier to understand and allows code to be reused
- Caller & callee relationship
  - Caller: The program that calls a procedure
  - Callee: A procedure that executes instructions
  - A callee can be a caller if it calls another procedure
- There is an interface between a caller and a callee
  - A caller provides the parameter (argument) values to its callee
  - The callee returns the result value to its caller
- Program must follow the following six steps in the execution of a procedure
  - (1) Caller puts parameters in a place where the callee can access them
  - (2) Control is transferred to the callee
  - (3) Callee acquires the storage resources needed for the procedure
  - (4) Callee performs the desired task
  - (5) Callee puts the result value in a place where the caller can access it
  - (6) Control is returned to the point of the caller

Supporting Procedures in MIPS Instruction Set

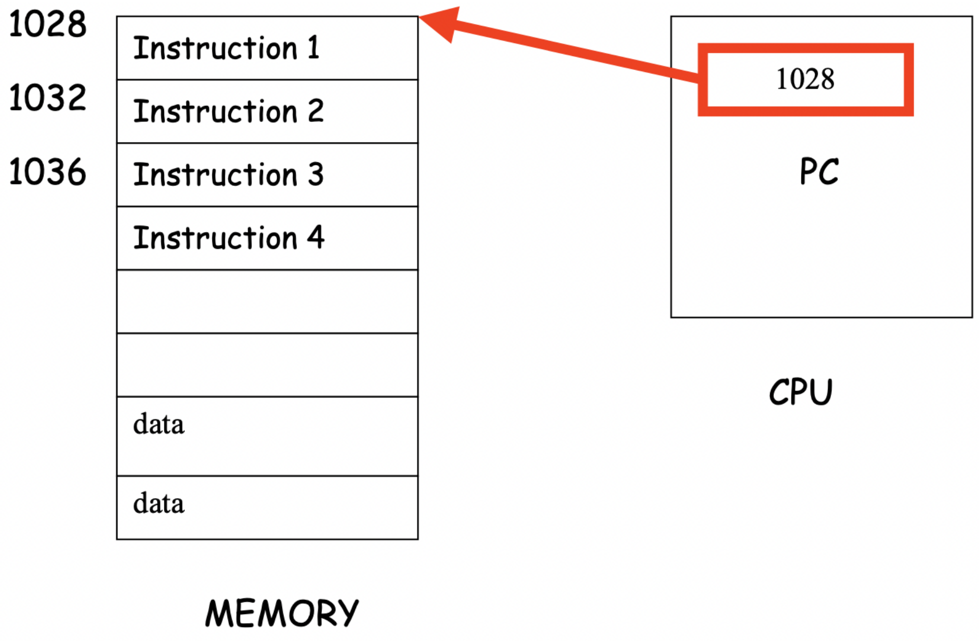
- Registers are used to support a procedure call and its return
  - **$a0—$a3**: four argument registers in which to pass parameters
  - **$v0—$v1**: two value registers in which to return values
  - **$ra**: one return address register to return to the point of origin
- Jump-and-link instruction (**jal**): **jal procedureaddress**
  - A caller uses this instruction to transfer control to the callee
  - (1) This jumps to an address (the beginning of the function)
  - (2) The return address (the subsequent address of the function call) is stored in **$ra** (register 31)
- Jump register (**jr**): **jr register**
  - This instruction indicates an unconditional jump to the address specified in a register
  - A callee uses this instruction to transfer control back to the caller — **jr $ra**
- Summary
  - Caller puts parameter values in **$a0—$a3** and uses **jal X** to jump to procedure X
  - Callee performs the calculations, places the results in **$v0—$v1**, and uses **jr $ra** to return to caller

Supporting Procedures in MIPS Instruction Set



Program Counter for Address of Instructions

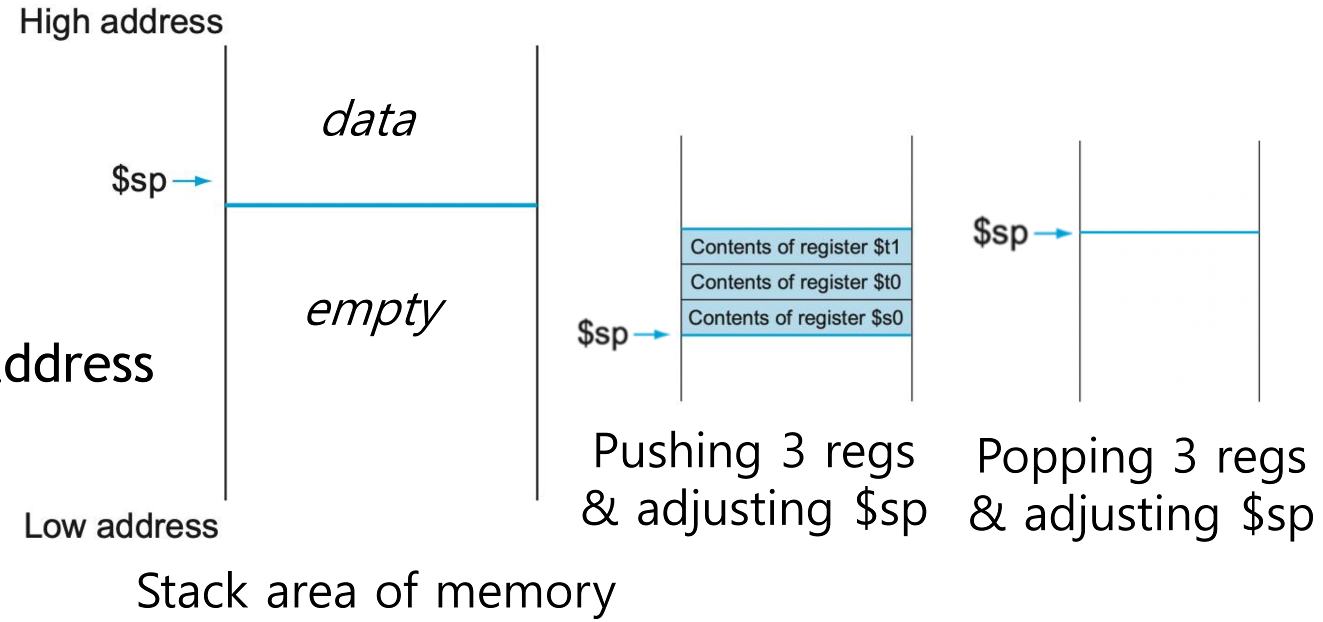
```
• Loop:
   sll   $t1,  $s3,  2        # temp reg $t1 = i *4
   add   $t1,  $t1,  $s6      # $t1 = address of save[i]
   lw    $t0,  0($t1)         # temp reg $t0 = save[i]
   bne   $t0,  $s5,  Exit     # go to Exit if save[i] ≠ k
   addi  $s3,  $s3,  1        # i = i + 1
   j  Loop                    # go to Loop
Exit:                         # here is the label Exit
```



- Instructions
  - Instructions are stored in memory
  - Note that the size of each instruction is 4 bytes (a word)
- A CPU has a register that holds the address of the current instruction being executed
  - Program counter (PC); in MIPS, PC is not part of the 32 registers
  - Basically, PC is incremented by 4 whenever an instruction is executed
  - Branch and jump instructions put the target address in PC
- The **jal** instruction actually saves PC+4 in **$ra** to link the following instruction to set up the procedure return

Using Stack for Procedure Call

- Question: Are **\$a0—\$a3** and **\$v0—\$v1** enough for a callee to work with?
  - What happens if callee uses **\$s** or **\$t**, which are being used by caller?
  - If so, once the procedure is returned, such registers (**\$s** or **\$t**) may be polluted
  - Registers must be restored to the values that they contained before the procedure was invoked
- Solution: Such register values are kept in an area of memory, called stack
  - Stack grows from higher to lower addresses
  - A last-in-first-out queue
    - Push: placing (storing) data onto the stack
    - Pop: removing (deleting) data from the stack
  - Stack pointer holds most recently allocated address
    - MIPS reserves **\$sp** (register 29) for stack pointer
    - **\$sp** is adjusted when pushing and popping
    - **\$sp** is decremented by 4 when pushing a register
    - **\$sp** is incremented by 4 when popping a register



Using Stack for Procedure Call: Example

```
int leaf_example (int g, int h, int i, int j)
{
     int f;

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

is translated into

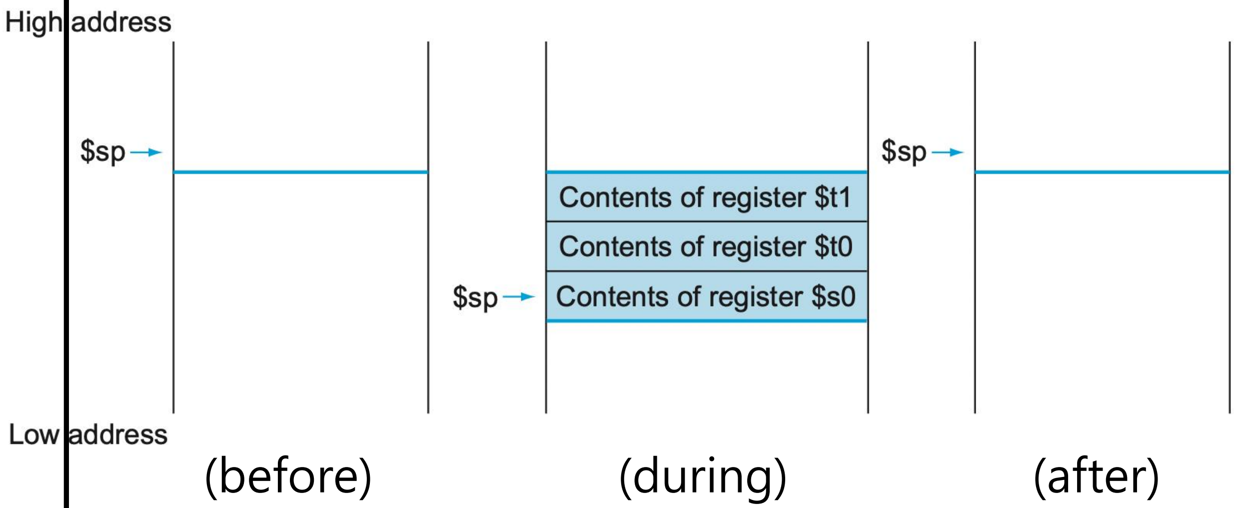
```
leaf_example:

add  $t0,  $a0,  $a1        # register $t0 contains g + h
add  $t1,  $a2,  $a3        # register $t1 contains i + j
sub  $s0,  $t0,  $t1        # f = $t0 - $t1

add  $v0,  $s0,  $zero      # returns f

jr $ra                      # jump back to caller
```

- Assumption
  - g, h, i, and j correspond to **$a0, $a1, $ a2, and $a3**
  - f corresponds to **$s0**
- Caller sets argument registers
  - E.g., **add $a0, $t0, $zero**
  - E.g., **addi $a1, $zero, 6**
- Caller invokes **jal leaf_example**
  - **$ra** ← PC + 4
  - PC ← leaf_example

Using Stack for Procedure Call: Example

```
leaf_example:

addi  $sp,  $sp,  -12      # adjust stack to make room for 3 items
sw  $t1,  8($sp)           # save $t1 for use afterwards
sw  $t0,  4($sp)           # save $t0 for use afterwards
sw  $s0,  0($sp)           # save $s0 for use afterwards

add  $t0,  $a0,  $a1       # register $t0 contains g + h
add  $t1,  $a2,  $a3       # register $t1 contains i + j
sub  $s0,  $t0,  $t1       # f = $t0 - $t1

add  $v0,  $s0,  $zero     # returns f

lw  $s0,  0($sp)           # restore $s0 for caller
lw  $t0,  4($sp)           # restore $t0 for caller
lw  $t1,  8($sp)           # restore $t1 for caller
addi  $sp,  $sp,  12       # adjust stack to delete 3 items

jr $ra                     # jump back to caller
```

- What if **$t0, $t1, $s0** are holding data needed by caller afterwards?
  - After returning, program malfunctions
- The three register data can be protected by keeping them in stack
  - Pushing the values before using them
  - Popping them when returning
- **$sp** must be adjusted correspondingly



Saved vs Temporary Registers

- Then, do we need to save and restore all the registers whenever calling a function?
  - In the previous example, we assumed that the old values of temporary registers must be saved an d restored
  - Actually, we do not have to save and restore registers whose values are never used
- To avoid such unnecessary saving/restoring, MIPS separates registers into two groups
- Temporary registers ($t0—$t9)
  - These registers are not preserved by the callee on a procedure call
- Saved registers ($s0—$s7)
  - These registers must be preserved on a procedure call
  - If used, the callee saves and restores them

```
addi  $sp,  $sp,  -12 -4
sw  $t1,  8($sp)
sw  $t0,  4($sp)
sw  $s0,  0($sp)
```

```
lw  $s0,  0($sp)
lw  $t0,  4($sp)
lw  $t1,  8($sp)
addi  $sp,  $sp,  12  4
```

Nested Procedures

- All procedures are not leaf procedures, which do not call others
  - main() calls func_A(), which calls func_B(); here, func_A() is a nested procedure
  - Recursive procedures are also nested
- A problematic example in nested procedures
  - main() calls procedure A with an argument of 3 (①**addi $a0, $zero, 3;** ②**jal A**)
  - Procedure A calls procedure B with an argument of 7 (③**addi $a0, $zero, 7;** ④**jal B**)
  - You may find two conflicts;
    - At ③, procedure B updates **$a0** with 7; what if procedure A continues to expect that **$a0** holds 3?
    - At ④, procedure B updates **$ra** with its return address; procedure A loses its return address
- One solution is to push all the registers that must be preserved onto the stack
  - Caller pushes arg registers (**$a0-$a3**) or temp registers (**$t0-$t9**) that are needed after the call
  - Callee pushes return address register (**$ra**) and saved registers (**$s0-$s7**) used by the callee
  - Note that stack pointer (**$sp**) should be adjusted correspondingly

Nested Procedures: Example

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

is translated into

- **$a0** and **$ra** can be used in the subsequent call, which is kept onto the stack
- **slti** & **beq** for if-then-else statement
- If n < 1, this leaf procedure returns to the caller; here, **$a0** and **$ra** still hold the original values; so, you don't have to get those values from the stack

```
fact:
addi  $sp, $sp, -8        # adjust stack for 2 items
sw  $ra, 4($sp)           # save the return address
sw  $a0, 0($sp)           # save the argument n

slti  $t0, $a0, 1         # test for n < 1
beq  $t0, $zero, L1       # if n >= 1, go to L1

addi  $v0, $zero, 1       # return 1
addi  $sp, $sp, 8         # pop 2 items off stack
jr $ra                    # return to caller
```

```
L1:
addi  $a0, $a0, -1    # n >= 1: arg gets (n-1)
jal  fact             # call fact with (n-1)

lw  $a0, 0($sp)       # retrun from jal; restore arg n
lw  $ra, 4($sp)       # restore return address
addi  $sp, $sp, 8     # adjust $sp to pop 2 items

mul   $v0, $a0, $v0   # return n * fact (n-1)
jr $ra                # return to caller
```

Nested Procedures: Example

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

is translated into

- If n >= 1, fact(n-1) is called
- The return address of fact() is here

$a0 and **$ra** are restored, and **$sp** is readjusted

- The current routine returns to the caller

with an argument of n * fact (n-1)

```
fact:

addi  $sp, $sp, -8       # adjust stack for 2 items
sw  $ra, 4($sp)          # save the return address
sw  $a0, 0($sp)          # save the argument n

slti  $t0, $a0, 1        # test for n < 1
beq  $t0, $zero, L1      # if n >= 1, go to L1

addi  $v0, $zero, 1      # return 1
addi  $sp, $sp, 8        # pop 2 items off stack
jr $ra                   # return to caller
```

```
L1:
addi  $a0, $a0, -1       # n >= 1: arg gets (n-1)
jal  fact                # call fact with (n-1)

lw  $a0, 0($sp)          # return from jal; restore arg n
lw  $ra, 4($sp)          # restore return address
addi  $sp, $sp, 8        # adjust $sp to pop 2 items

mul   $v0, $a0, $v0      # return n * fact (n-1)
jr $ra                   # return to caller
```