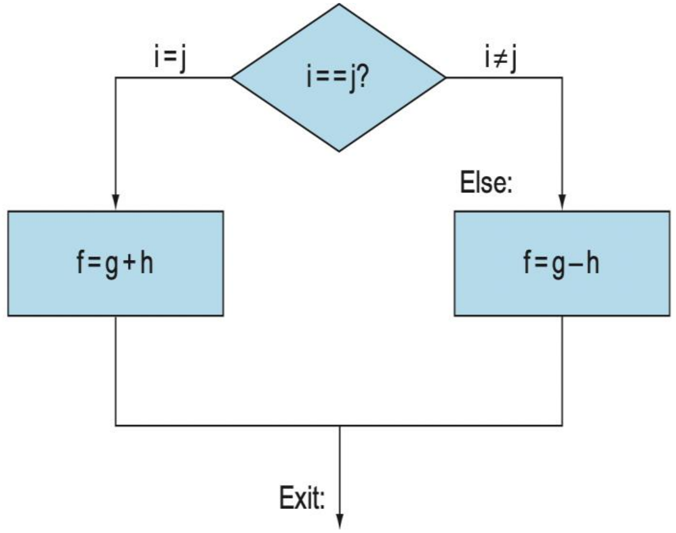
Computer Architecture
(ENE1004)

Lec – 5: Instructions: Language of the Computer (Chapter 2) - 4

Review: Decision-making Instructions: **if-else**

- **if (i == j) f = g + h; else f = g - h;**
  - f corresponds to $s0, g to $s1, h to $s2, i to $s3, j to $s4
  - is translated into

```
  bne  $s3,  $s4,  Else    # go to Else if i ≠ j
  add  $s0,  $s1,  $s2     # f = g + h (skipped if i ≠ j)
  j  Exit                  # go to Exit
Else:                      # here is the label Else
  sub  $s0,  $s1,  $s2     # f = g - h (skipped if i = j)
Exit:                      # here is the label Exit
```

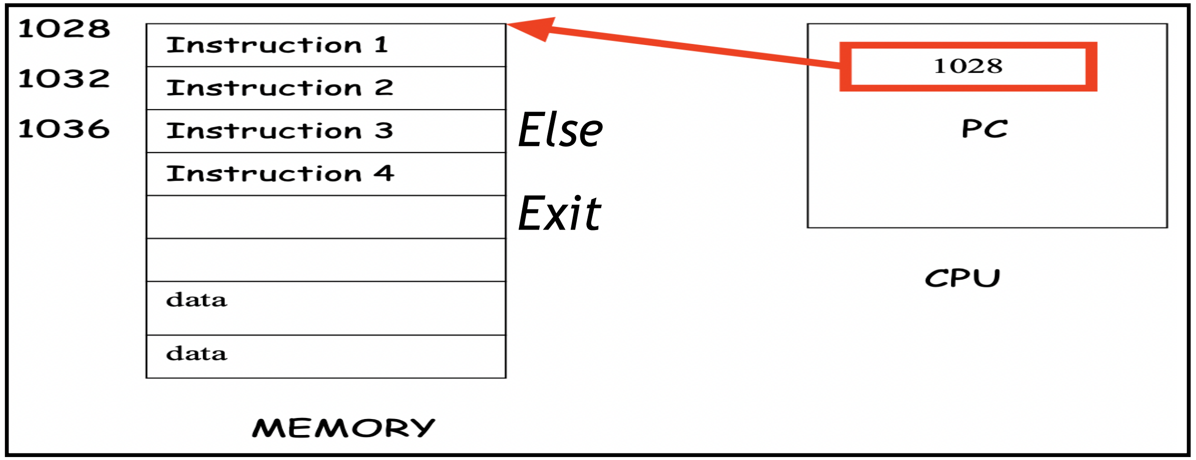


  - Without the unconditional branch (j Exit), both add and sub are executed
- An alternative: **beq $s3, $s4, Then # go to Then if i = j**
  - If you like this, where do you locate the case of where i ≠ j?
  - In general, the code will be more efficient if we test for the opposite condition to branch over the code that performs the subsequent *then* part of the if

Review: Overview

```
 bne  $s3, $s4, Else   # go to Else if i ≠ j
 add  $s0, $s1, $s2    # f = g + h (skipped if i ≠ j)
 j  Exit               # go to Exit
Else:                  # here is the label Else
 sub  $s0, $s1, $s2    # f = g - h (skipped if i = j)
Exit:                  # here is the label Exit
```

Compiled program (MIPS instructions)



Executed program

- When you execute your compiled program, its instructions are loaded to memory
  - So, each instruction can be identified using its memory address
  - Each label in your program indicates the memory address of the instruction right after it
- By default, CPU executes instructions from the first to the last sequentially
- However, when CPU executes a branch instruction (**beq**, **bne**, **j**)
  - If the condition is satisfied, CPU jumps to the target label, execute the following instruction, and continue to execute the following instructions sequentially
  - If not satisfied, CPU does not jump and execute the next instruction

MIPS Decision-making Instructions: **while**

- Decisions are also important for iterating a computation, which are found in loops
- **while (save[i] == k) i += 1;**
  - i corresponds to $s3, k to $s5, the base of the array save is in $s6
  - is translated into

```
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
```

- You may be able to optimize this sequence

MIPS Decision-making Instructions: **for**

- **for (i = 0; i < 4; i++) { //do something }**
  - Do you have any idea for handling “i < 4”?
  - It would be useful to see if a variable is less than another variable
- Two useful instructions, which can tell if a variable is less than another variable
  - Set less than: **slt reg1, reg2, reg3 # reg1 = 1 if reg2 < reg3**
    - It compares two registers (reg2 and reg3), and sets a register (reg1) to 1 if reg2 < reg3
    - Otherwise (if reg2 >= reg3), it sets reg1 to 0
  - Set less than immediate: **slti reg1, reg2, const #reg1=1 if reg2<const**
    - Immediate version of slt
    - It compares reg2 and constant, and sets a register (reg1) to 1 if reg2 < constant
- **for (i = 0; i < 4; i++) { //do something }**
  - Do you have any idea for initializing a variable “i = 0”?
  - MIPS includes a register, named “$zero”, which always holds a value of 0
  - **add $s0, $zero, $zero # $s0 will hold a value of 0**

register	assembly name	Comment
r0	$zero	Always 0

MIPS Decision-making Instructions: **for**

- **for (i = 0; i < 4; i++) { //do something }**
  - i corresponds to $t0
  - is translated into

```
   add  $t0,  $zero,  $zero  # i is initialized to 0, $t0 = 0
Loop:

   # do something

   addi  $t0,  $t0,  1           # i ++
   slti   $t1,  $t0,  4           # $t1 = 1 if i < 4
   bne   $t1,  $zero,  Loop   # go to Loop if i < 4
```

MIPS Decision-making Instructions: **switch-case**

- A straightforward way to implement switch is using a chain of if-then-else statements
  - Would you try to write a switch statement?
- An alternative is to take advantage of jump (address) table
  - We will discuss this later

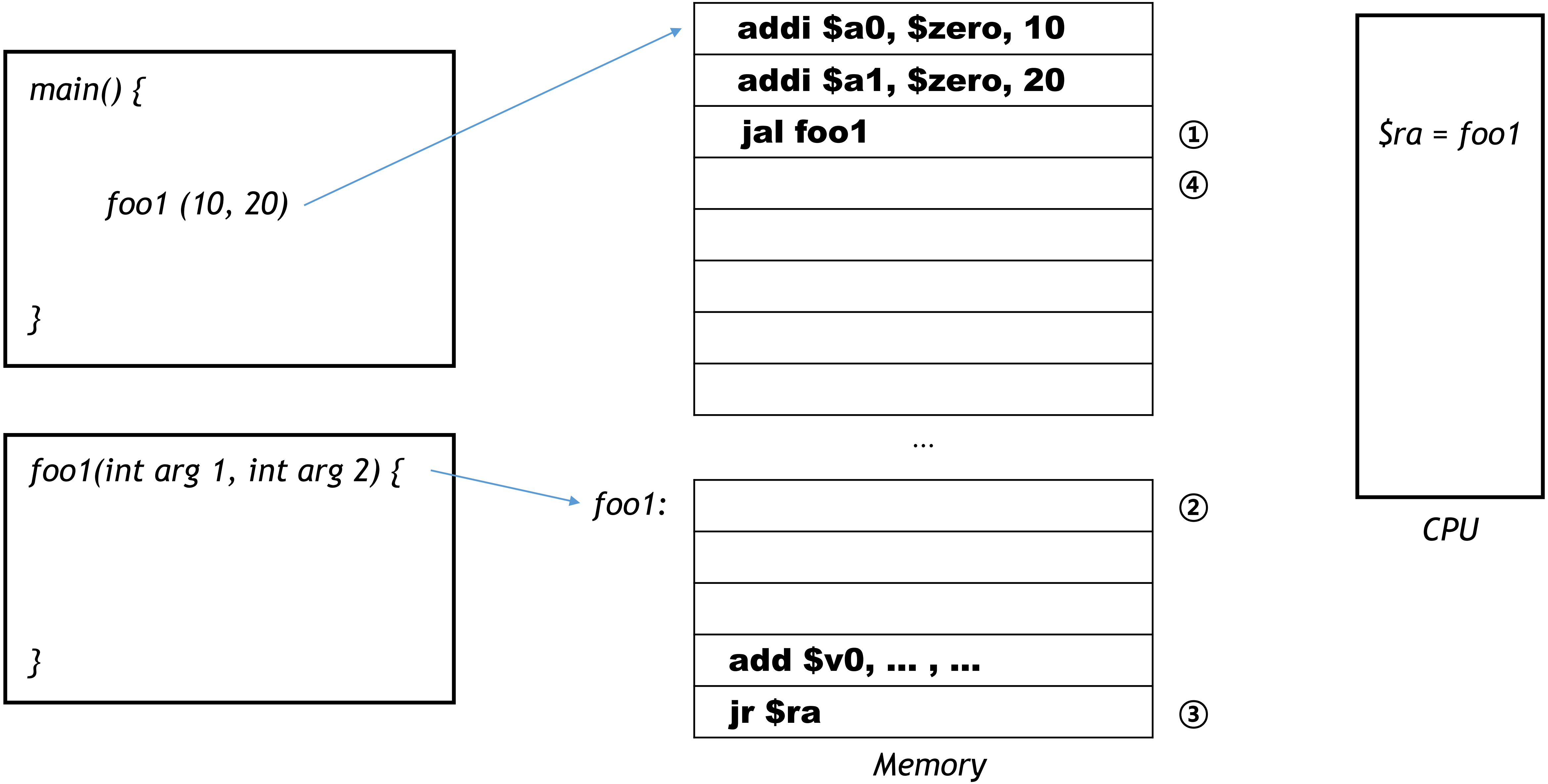
Supporting Procedures in Hardware

- 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

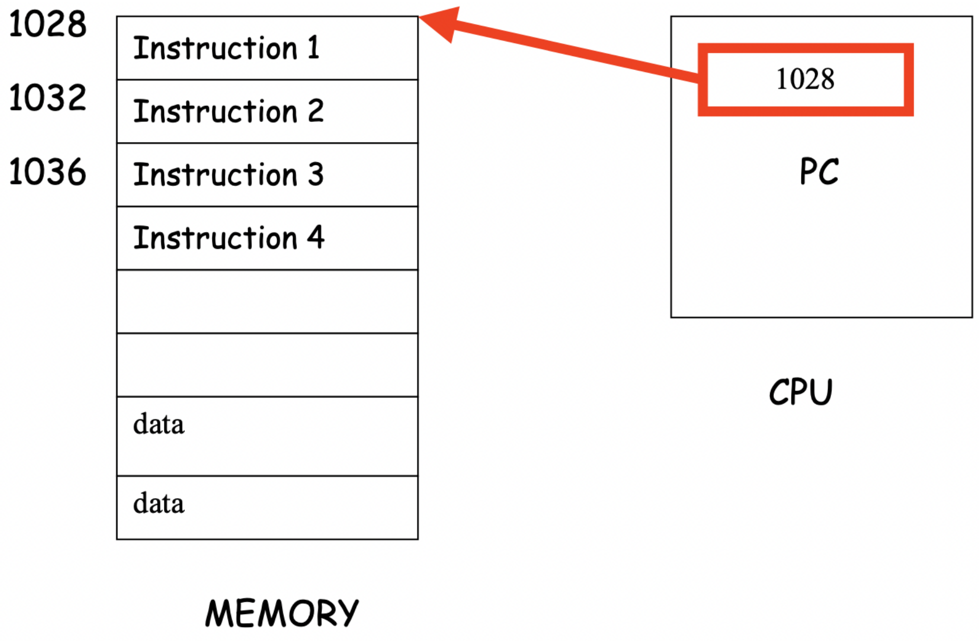
- Registers are used to support a procedure call and its return
  - **$a0—$a3**: four argument registers in which to pass parameters
  - **$v0—$v1**: two value registers in which to return values
  - **$ra**: one return address register to return to the point of origin
- Jump-and-link instruction (**jal**): **jal procedureaddress**
  - A caller uses this instruction to transfer control to the callee
  - (1) This jumps to an address (the beginning of the function)
  - (2) The return address (the subsequent address of the function call) is stored in **$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

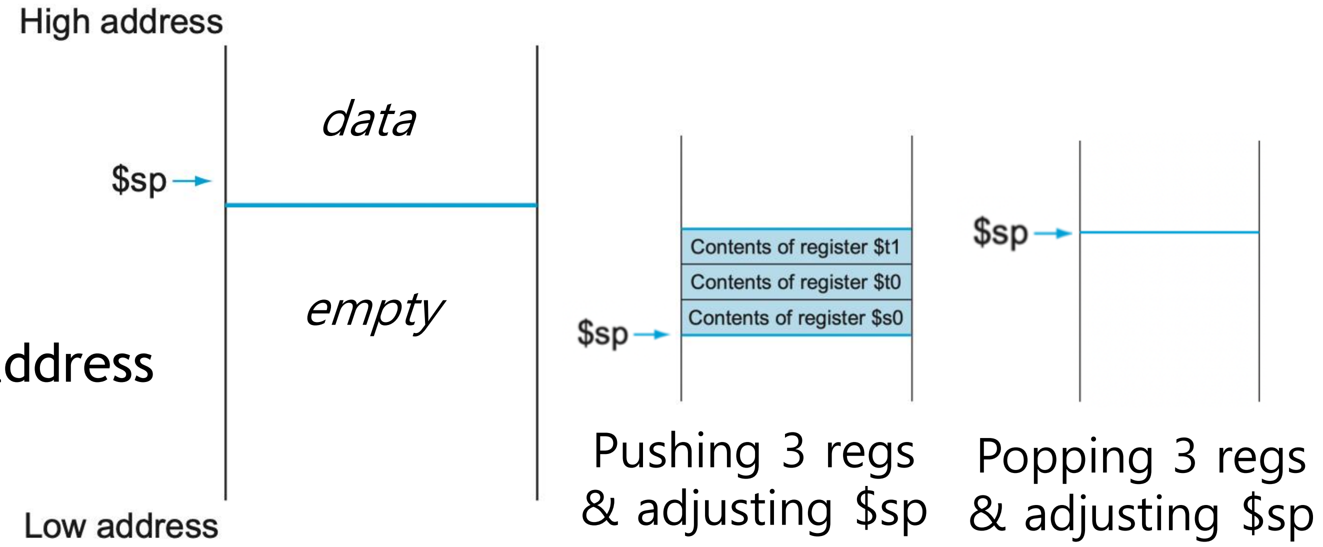
```
• Loop:
  sll  $t1, $s3, 2        # temp reg $t1 = i *4
  add  $t1, $t1, $s6      # $t1 = address of save[i]
  lw   $t0, 0($t1)        # temp reg $t0 = save[i]
  bne  $t0, $s5, Exit     # go to Exit if save[i] ≠ k
  addi $s3, $s3, 1        # i = i + 1
  j  Loop                 # go to Loop
Exit: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



Stack area of memory

Pushing 3 regs & adjusting $sp

Popping 3 regs & adjusting $sp

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

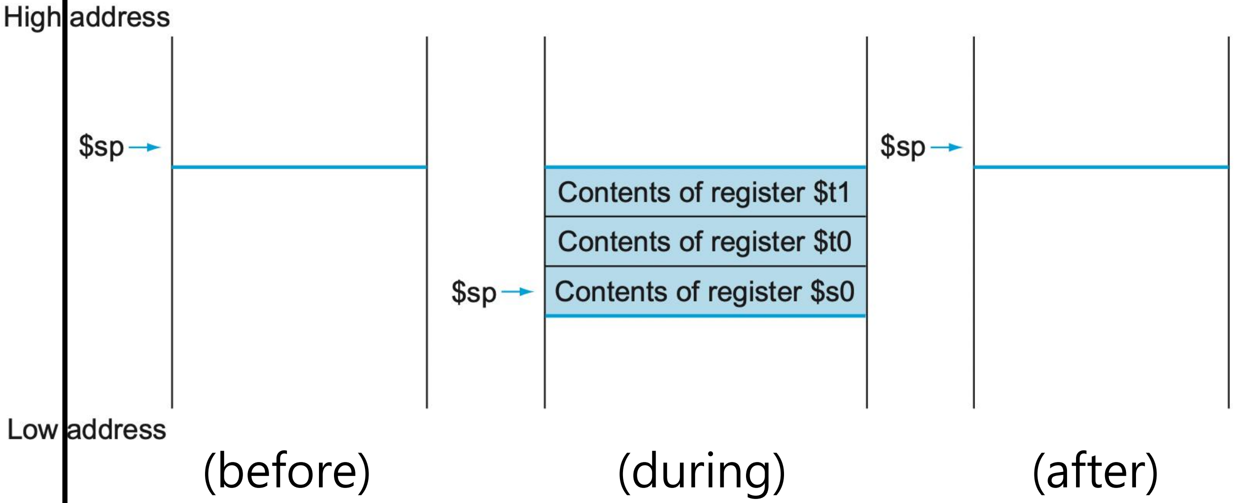
```
leaf_example:

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 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 $s1 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, $1, $s0** are holding data needed by caller afterwards?
  - After returning, program malfunctions
- The three register data can be protected by keeping them in stack
  - Pushing the values before using them
  - Popping them when returning
- **$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
```