Computer Architecture (ENE1004)

▼ Lec 5

Lec5: Instructions: Language of the Computer 4

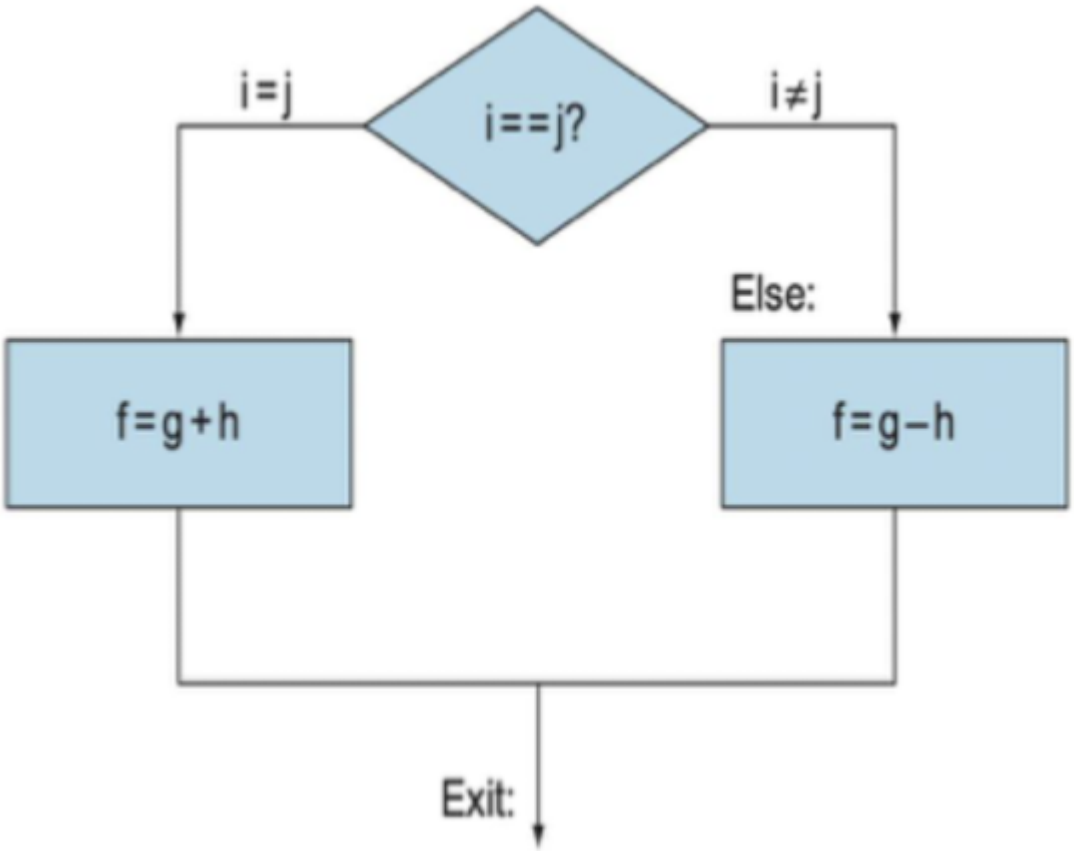
Review: Decision-making Instructions: if-else

```
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 (무조건 분기(j Exit)가 없으면 add와 sub가 모두 실행됩니다.)

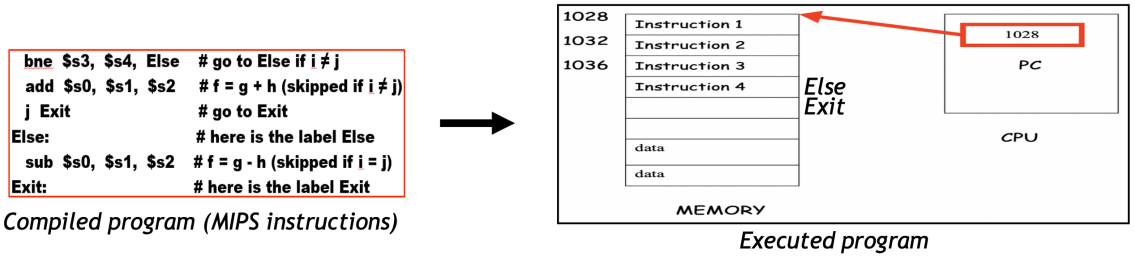


- An alternative:

```
beq $s3, $s4, Then # go to Then if i = j
```

  - If you prefer this, where do you locate the case of 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
  - Else로 하는 것이 더 편함

Big picture: Flow of program Execution



- When you execute your compiled program, its instructions are loaded to memory (컴파일된 프로그램은 실행하면 해당 명령어가 메모리에 로드된다.)
    - So, each instruction can be identified using its memory address (따라서 각 명령어는 메모리 주소를 사용하여 식별할 수 있다.)
    - Each label in your program indicates the memory addres of the instruction right after it (프로그램의 각 레이블은 바로 다음 명령어의 메모리 주소를 나타낸다.)
- By default, CPU executes instructions from the first to the last sequentially (기본적으로 CPU는 명령을 처음부터 마지막까지 순차적으로 실행한다.)
- However, when CPU executes a branch instruction (beq, bne, j) (단, CPU가 분기 명령 (beq, bne, j)를 실행하는 경우)
    - If the condition is satisfied, CPU jumps to the target label, execute the following instruction, and continue the following instructions sequentially (조건이 만족되면 CPU는 대상 레이블로 점프하고 다음 명령을 실행한다. 계속해서 다음 명령을 순차적으로 실행한다.)
    - If not satisfied, CPU does not jump and executes the next instruction (만족하지 않으면 CPU는 점프하지 않고 다음 명령어를 실행한다.)
- 계산은 CPU에서 하는데 왜 명령어가 CPU가 아닌 메모리에 로드되는 것인가?
    - 컴파일된 프로그램이 실행될 때 프로그램의 명령어들은 메모리에 로드된다. 이것은 컴퓨터 시스템의 작동 방식 중 하나로, CPU가 명령어를 실행하기 위해서는 그 명령어가 메모리에 있어야만 한다. 이 명령어들은 컴파일러에 의해 생성되며, 이러한 명령어들은 메모리에 저장되어야만 CPU가 실행할 수 있다.
    - 여기서 중요한 점은 명령어가 메모리에 있다는 것이 CPU가 직접적으로 메모리에서 명령어를 가져와 실행한다는 것을 의미하는 것은 아니다. 대신, CPU는 명령어를 메모리에서 가져와서 자신의 내부 구조에 있는 명령어 실행 유닛에서 실행한다.
    - 따라서 명령어가 메모리에 로드되는 것은 CPU가 명령어를 실행하기 위해 필요한 프로그램의 일부가 메모리에 존재해야 한다는 컴퓨터 시스템의 동작 방식에 따른 것이다.

Supporting Procedures in Hardware (하드웨어 자원 프로세서)

- Function (procedure) is one of the most widely used tool in programming (함수(프로시저)는 프로그래밍에서 가장 널리 사용되는 도구 중 하나이다.)
    - It makes programs easier to understand and allows code to be reused (프로그램을 더 쉽게 이해하고 코드를 재사용 할 수 있다.)
- Caller & callee relationship (호출자와 수신자의 관계)
    - Caller: The program that calls a procedure (호출자: 프로시저를 호출하는 프로그램)
    - Callee: A procedure that includes instructions (수신자: 명령어가 포함된 프로시저)
    - A callee can be a caller if it calls another procedure (수신자는 다른 프로시저를 호출하는 경우 호출자가 될 수 있다. (ex: 재귀))
- There is an interface between a caller and a callee
    - A caller provides the parameter (argument) values to its callee (호출자는 수신자에게 매개변수(인수) 값을 제공한다.)
    - The callee returns the result value to its caller (수신자는 결과 값을 호출자에게 반환한다.)
- Program must follow the following six steps in the execution(실행) 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 parameter values by accessing the place
    - (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
    - (1) 호출자는 수신자가 액세스 할 수 있는 위치에 매개변수를 배치한다.
    - (2) 제어권이 수신자에게 이전된다.
    - (3) 수신자는 장소에 접근하여 매개변수 값을 획득한다.
    - (4) 수신자가 원하는 작업을 수행한다.

    - (5) 수신자는 호출자가 접근할 수 있는 위치에 결과 값을 넣는다.
    - (6) 발신자에게 제어권이 반환된다.
- 프로시저(Procedure)
    - 프로시저(Procedure)는 프로그래밍에서 특정 작업을 수행하기 위해 구성된 코드 블록이나 함수이다. 프로시저는 일련의 작업을 수행하고, 필요한 경우에는 입력을 받아들이고 결과를 반환할 수 있다. 프로시저는 코드의 재사용성을 증가시키고 프로그램을 모듈화하여 유지보수 및 디버깅을 쉽게 만드는 데 도움이 된다.
    - 프로시저를 호출하면 프로그램의 실행 흐름이 해당 프로시저로 전환되며, 프로시저 내에서 정의된 작업이 수행된다. 이러한 프로시저 호출은 프로그램에서 Caller와 Callee 간의 관계를 형성하게 된다. Caller는 프로시저를 호출하는 프로그램 또는 코드이며, Callee는 호출되어 실행되는 프로시저이다.
    - 프로시저는 매개변수(인수)를 통해 정보를 받아들이고, 필요한 작업을 수행한 후 결과를 반환할 수 있다. 이때 Caller는 매개변수를 제공하고, Callee는 결과 값을 Caller에게 반환한다.
    - 프로시저는 프로그램의 구성요소 중 하나로, 다양한 작업을 수행하는데 사용된다. 함수, 서브루틴, 메소드 등 다양한 용어로 불릴 수 있지만, 모두 프로시저의 일종이다.

Supporting Procedures in MIPS Instruction Set

- Registers are used to support a procedure call and its return (레지스터는 프로시저 호출과 그 반환을 지원하는데 사용된다.)
    - $a0—$a3: four argument registers in which to pass parameters (매개변수를 전달할 4가지 인수 레지스터)
    - $v0—$v1: two value registers in which to return values (값을 반환하는 두 개의 값 레지스터)
    - $ra: one return address register to return to the point of origin (출발지로 돌아가기 위한 반환 주소 레지스터 하나)
- Jump-and-link instruction(점프와 링크 명령어) - jal

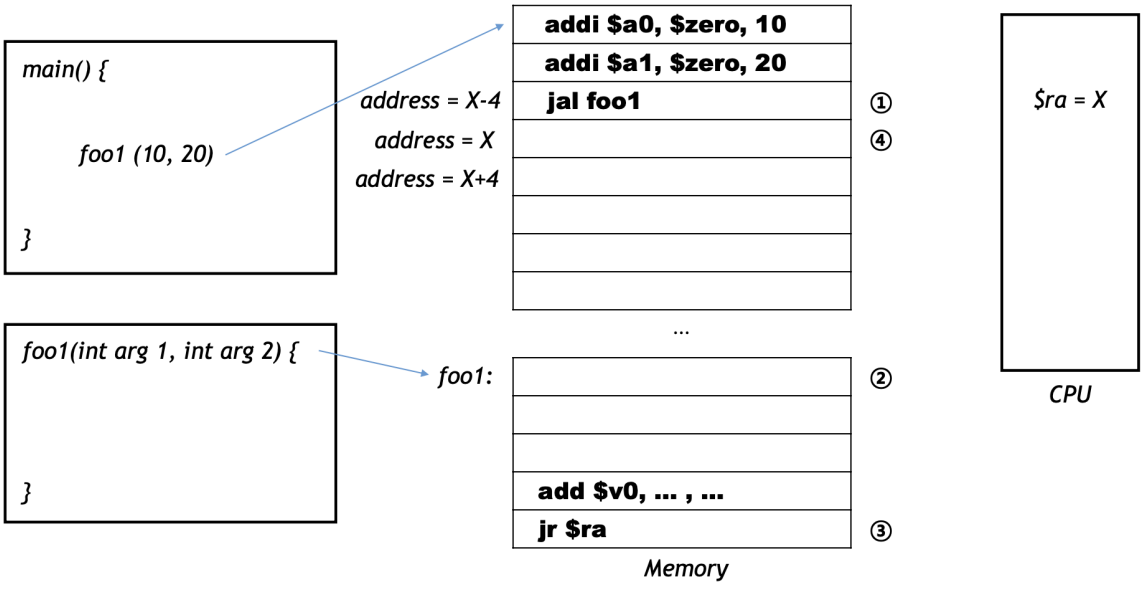
```
jal procedure_address
```

    - A caller uses this instruction to transfer control to the callee (호출자는 이 명령을 사용하여 수신자에게 제어권을 이전한다.)

    - (1) This jumps to an address (the beginning of the function) : 이것은 주소(함수의 시작 부분)로 이동한다. 이때, 함수도 레지스터에 있다.
    - (2) The return address (the subsequent address of the function call) is stored in $ra (register 31) : 반환 주소(함수 호출의 후속 주소)는 $ra(레지스터 31)에 저장된다.
- Jump register - jr

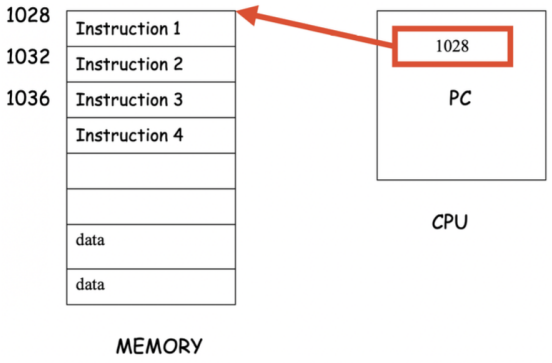
```
jr register
```

    - This instruction indicates an unconditional jump to the address specified in a register (이 명령은 레지스터에 지정된 주소로 무조건 점프하는 것을 나타낸다.)
    - A callee uses this instruction to transfer control back to the caller — jr $ra (수신자는 이 명령을 사용하여 호출자에게 제어권을 다시 전송한다.)
- Summary
    - Caller puts parameter values in $a0—$a3 and uses jal X to jump to procedure X (호출자는 $a0-$a3 레지스터에 매개변수로 값을 넣고 jal x를 사용하여 프로시저 x로 이동한다. 이때, 프로시저 x도 레지스터에 있다.)
    - Callee performs its task, places the results in $v0—$v1, and uses jr $ra to return to caller (수신자가 작업을 수행하고 결과를 $v0-$v1에 배치한 후 jr $ra를 사용하여 호출자에게 반환된다.)



Program Counter for Address of Instructions (명령어 주소 프로그램 카운터)

```
• 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
```

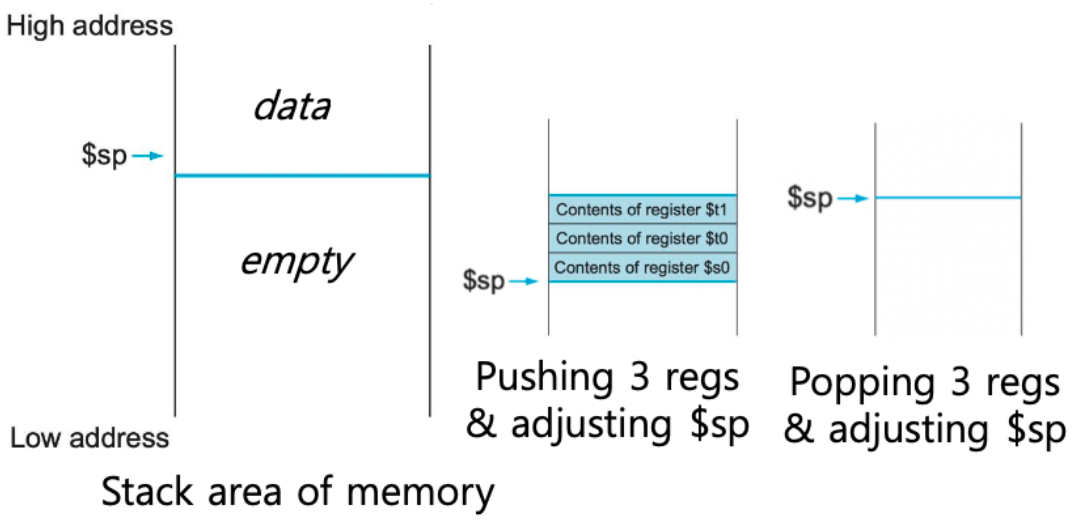


- Instructions (명령어)
  - Instructions are stored in memory : 명령어들은 메모리에 저장된다.
  - Note that the size of each instruction is 4 bytes (a word) : 각 명령어의 크기는 4바이트 (1 word)이다.
- A CPU has a register that holds the address of the current instruction being executed (CPU는 현재 실행중인 명령어의 주소를 저장하는 레지스터가 있다.)
  - Program counter (PC); in MIPS, PC is not part of the 32 registers (PC는 32개의 레지스터에 포함되지 않는다.)
  - Basically, PC is incremented by 4 whenever an instruction is executed (기본적으로 PC는 명령어가 실행될 때 마다 4씩 증가한다.)
  - Branch and jump instructions put the target address in PC (분기 및 점프 명령어는 대상 주소를 PC에 넣는다.)
- The jal instruction actually saves PC+4 in $ra to link the following instruction to set up
  the procedure return (jal 명령은 실제로 PC+4를 $ra에 저장하여 다음 명령과 연결하여 프로시저 리턴을 설명한다.)

Using Stack for Procedure Call

- Question: Are $a0-$a3 and $v0-$v1 enough for a callee to work with? (수신자가 작업하기에 $a0-$a3 및 $v0-$v1이 충분한가요?)
  - What happen if callee uses $s or $t, which are being used by caller? (호출자가 사용 중인 $s 또는 $t를 수신자가 사용하면 어떻게 되나요?)

    - If so, once the prodedure is returned, such registers ($s or $t) may be polluted (그렇다면 프로시저가 반환되면 해당 레지스터($s 또는 $t)가 오염될 수 있습니다.)
    - Registers must be restored to the values that they contained before the procedure was invoked (레지스터는 프로시저가 호출되기 전에 포함되었던 값으로 복원되어야 합니다.)
- Solution: Such register values are kept in an area of memory, called stack (이러한 레지스터 값은 스택이라고 하는 메모리 영역에 보관됩니다.)
    - 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(조정된다) when pushing and popping
        - $sp is decremented(감소된다) by 4 when pushing a register
        - $sp is incremented(증가된다) by 4 when popping a registers



- 스택을 사용한 프로시저 호출: 컴퓨터 과학에서 스택은 데이터를 임시 저장하는 방식 중 하나로, 마지막에 데이터가 가장 먼저 나가는 특성(Last in First Out, LIFO)을 가지고 있다. 이 방식은 함수 또는 프로시저 호출 시 사용되는 레지스터의 값을 보호하고 복원하는 데 유용하다.

- 스택과 레지스터의 필요성: 함수(또는 프로시저)를 호출할 때, 수신자(callee, 호출된 함수)가 작업을 수행하기 위해 사용하는 레지스터는 주로 $a0-$a3(인자 전달용)과 $v0-$v1(결과 반환용)이다. 하지만, callee가 더 많은 레지스터를 사용해야 할 경우나 $s 또는 $t와 같이 호출자(callee)가 이미 사용중인 레지스터를 사용해야 할 경우가 있다.
- 레지스터의 오염 문제: callee가 caller가 사용중인 $s 또는 $t 레지스터를 사용하게 되면, 이 레지스터들의 원래 값이 변경되어, callee로부터 제어가 돌아왔을 때 caller가 기대하는 값이 아닐 수 있다. 이를 "레지스터 오염"이라고 한다.
- 해결책 - 스택의 사용: 이 문제의 해결책은 callee가 사용할 레지스터의 원래 값(caller에서 이미 사용 중인 레지스터의 값)을 스택에 임시 저장하는 것이다. callee는 작업을 시작하기 전에 필요한 레지스터의 값을 스택에 푸시(push)하여 저장하고, 작업이 끝난 후에는 이 값을 다시 팝(pop)하여 레지스터에 복원한다. 이를 통해 callee가 레지스터 값을 변경해도, 원래의 값을 보존하고 복원할 수 있다.
- 스택 포인터와 스택의 동작: 스택은 주로 메모리의 높은 주소에서 낮은 주소로 성장한다. MIPS 아키텍처에서는 $sp(스택 포인터, 레지스터 29)를 이용해 현재 스택의 가장 상단을 가르킨다. 데이터를 스택에 푸시할 때는 $sp를 감소시키고(주소가 낮아짐), 팝할때는 $sp를 증가시킨다(주소가 높아짐).
- 이러한 방식은 함수 호출 시 레지스터 값의 안전한 보존과 복원을 가능하게 하여, 프로그램의 정확성과 신뢰성을 높인다.

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

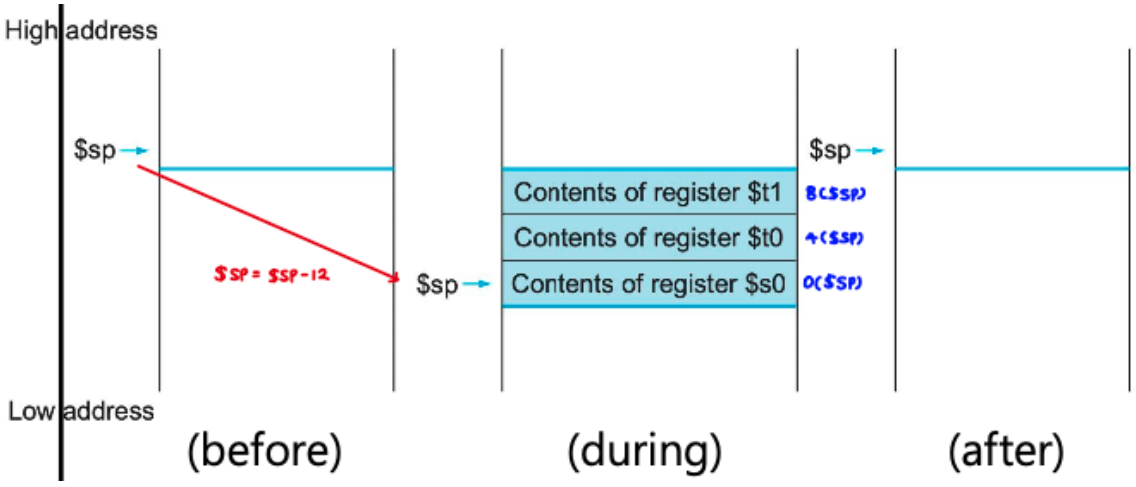
- Assumption
  - g, h, i and j correspond to $a0, $a1, $a2, $a3
  - f corresponds to $s0

- Caller invokes jal leaf_example
  - $ra = PC + 4
  - PC = leaf_example

```
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
```

Using Stack for Procedure Call: Example2

- What if $t0, $t1, $s0 are holding data needed by caller afterwards? (나중에 호출자가 필요로 하는 데이터를 $t0, $t1, $s0이 보유하고 있다면 어떻게 해야 할까?)
  - After returning, program malfunctions (return 후 프로그램 오작동)
- The three register data can be protected by keeping them in stack
  - Pushing the values before using them
  - Popping them when returning
- $sp must be adjusted correspondingly ($sp를 적절히 조정해야 한다.)



```
leaf_example:
addi $sp, $sp, -12 # adjust stack to make room for 3 ite
ms
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
```

- addi $sp, $sp, -12 : 스택 포인터 $sp를 조정하여 스택에 3개의 항목을 저장할 공간을 만든다. 스택은 메모리 주소가 낮은 쪽으로 성장하므로, $sp를 3 * 4 = 12 만큼 감소시킨다.
- sw $t1, 8($sp): $t1의 레지스터의 값을 스택 포인터 $sp가 가리키는 주소에 8바이트 떨어진 위치에 저장한다.
- sw $t0, 4($sp): $t0의 레지스터의 값을 스택 포인터 $sp가 가리키는 주소에 4바이트 떨어진 위치에 저장한다.
- sw $s0, 0($sp): $s0의 레지스터의 값을 스택 포인터 $sp가 가리키는 주소에 0바이트 떨어진 위치에 저장한다.
- 스택은 LIFO라는 특성을 가지고 있는데 lw와 sw의 순서는 상관이 없는가?
    - 상관없다. 하지만 스택의 특성때문에 데이터를 저장할 때는 어떤 순서로 저장하더라도, 데이터를 사용할 때는 가장 최신에 저장된 데이터부터 역순으로 꺼내어 사용된다.

Saved vs Temporary Registers

- Then, do we need to save and restore all the registers whenever calling function? (그렇다면 함수를 호출할 때마다 모든 레지스터를 저장하고 복원해야 하는가?)
    - In the previous example, we assumed that the old values of temporary registers must be saved and restored (이전 예제에서는 임시 레지스터의 이전 값을 저장하고 복원해야 한다고 가정했다.)
    - Actually, we do not have to save and restore registers whose values are never used (실제로는 값이 전혀 사용되지 않는 레지스터를 저장하고 복원할 필요가 없다.)
- To avoid such unnecessary saving/restoring, MIPS separates registers into two groups (이러한 불필요한 저장/복원을 방지하기 위해 MIPS는 레지스터를 두 그룹으로 분리한다.)
- Temporary registers ($t0 - $t9)
    - These registers are not preserved by callee on a procedure call (보존되지 않는다)
- Saved registers ($s0—$s7)
    - These registers must be preserved on a procedure call (보존되어야만 한다)
    - 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
```