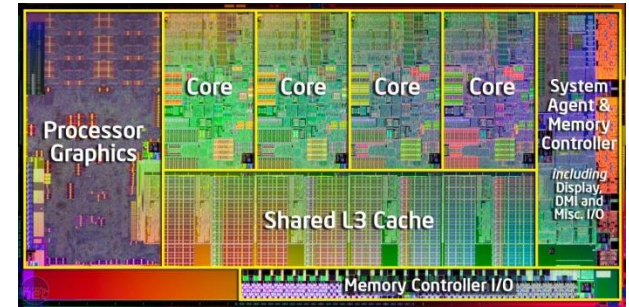




고려대학교
KOREA UNIVERSITY



COSE222, COMP212 Computer Architecture

Lecture 4. MIPS Instructions #3

Branch Instructions

Prof. Taeweon Suh
Computer Science & Engineering
Korea University

Why Branch?

- A computer performs different tasks depending on condition
 - Example: In high-level language, `if/else`, `case`, `while` and `for loops` statements all conditionally execute code

“if” statement

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

“while” statement

```
// determines the power
// of x such that 2x = 128
int pow = 1;
int x    = 0;

while (pow != 128) {
    pow = pow * 2;
    x = x + 1;
}
```

“for” statement

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
    sum = sum + i;
}
```

Why Branch?

- An advantage of a computer over a calculator is its ability to make decisions
 - A computer performs different tasks depending on conditions
 - In high-level language, `if/else`, `case`, `while` and `for` loops statements all conditionally execute code
- To sequentially execute instructions, the **pc** (program counter) increments by 4 after each instruction in MIPS since the size of each instruction is 4-byte
- **branch** instructions modify the **pc** to skip over sections of code or to go back to repeat the previous code
 - There are 2 kinds of branch instructions
 - **Conditional branch** instructions perform a test and branch only if the test is true
 - **Unconditional branch** instructions always branch

Branch Instructions in MIPS

- **Conditional branch** instructions
 - beq (**b**ranch if **e**qual)
 - bne (**b**ranch if **n**ot **e**qual)
- **Unconditional branch** instructions
 - j (**j**ump)
 - jal (**j**ump **a**nd **l**ink)
 - jr (**j**ump **r**egister)

beq, bne

- I format instruction

beq (bne) rs, rt, label

- Examples:

bne \$s0, \$s1, skip // go to "skip" if \$s0≠\$s1

beq \$s0, \$s1, skip // go to "skip" if \$s0==\$s1

...

skip: add \$t0, \$t1, \$t2

opcode	rs	rt	immediate
4	16	17	?

High-level code

```
if (i==j) h = i + j;
```

compile

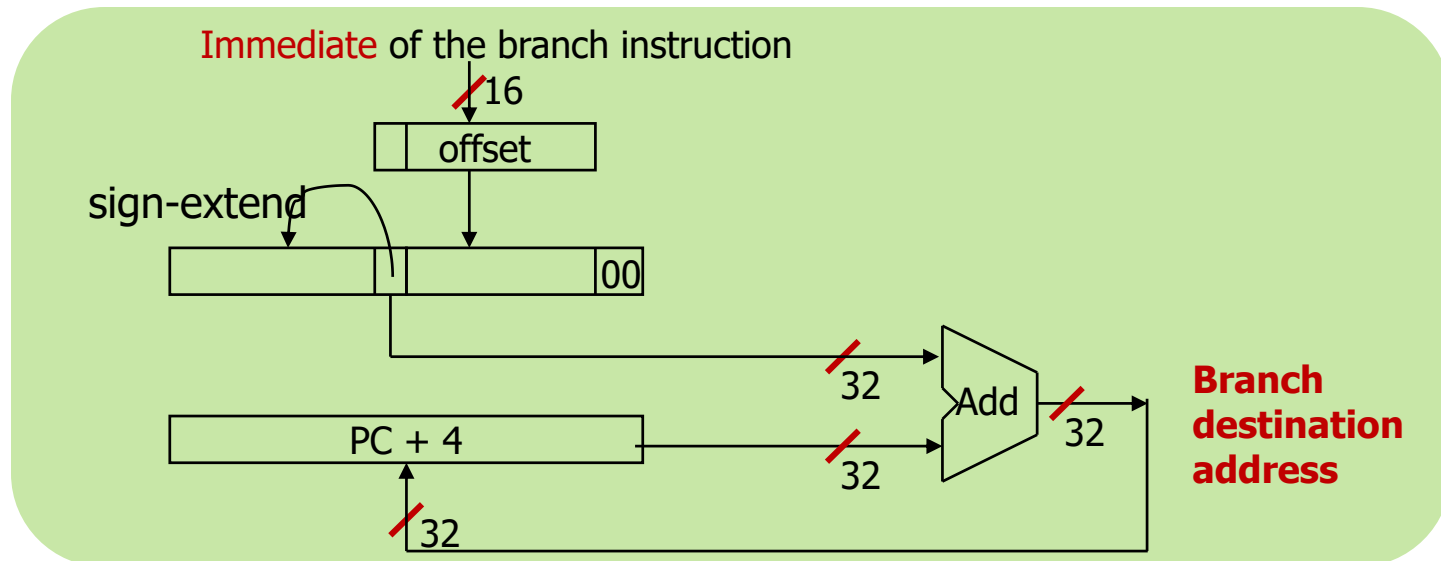
MIPS assembly code

```
// $s0 = i, $s1 = j  
bne $s0, $s1, skip  
add $s3, $s0, $s1  
skip: ...
```

- How is the branch destination address specified?

Branch Destination Address

- `beq` and `bne` instructions are I-type, which has the 16-bit immediate
 - Branch instructions use the immediate field as offset
 - Offset is relative to the **PC**
- Branch destination calculation
 - **PC** gets updated to **PC+4** during the **fetch** cycle so that it holds the address of the next instruction – **Will cover this in chapter 4**
 - It limits the branch distance to a range of **$-2^{15} \sim (+2^{15} - 1)$ instructions** from the instruction **after** the branch instruction
 - As a result, **destination = (PC + 4) + (imm << 2)**



bne Example

High-level code

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

f = f - i;
```

compile



MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j

    bne $s3, $s4, L1
    add $s0, $s1, $s2

L1:  sub $s0, $s0, $s3
```

Notice that the assembly tests for the opposite case ($i \neq j$), as opposed to the test in the high-level code ($i == j$).

In Support of Branch

- There are 4 instructions (`slt`, `sltu`, `slti`, `sltiu`) that help you set the conditions
`slt`, `slti` for signed numbers
`sltu`, `sltiu` for unsigned numbers

- Instruction format

```

slt   rd, rs, rt    // Set on less than (R format)
sltu  rd, rs, rt    // Set on less than unsigned (R format)
slti  rt, rs, imm   // Set on less than immediate (I format)
sltiu rt, rs, imm   // Set on less than unsigned immediate (I format)
    
```

- Examples:

```

slt $t0, $s0, $s1   # if $s0 < $s1    then
                    # $t0 = 1          else
                    # $t0 = 0
    
```

```

sltiu $t0, $s0, 25  # if $s0 < 25 then $t0=1
    
```

opcode	rs	rt	immediate
11	16	8	25

Name	Register Number
\$zero	0
\$at	1
\$v0 - \$v1	2-3
\$a0 - \$a3	4-7
\$t0 - \$t7	8-15
\$s0 - \$s7	16-23
\$t8 - \$t9	24-25
\$gp	28
\$sp	29
\$fp	30
\$ra	31

Branch Pseudo Instructions

- `blt`, `ble`, `bgt` and `bge` are **pseudo instructions** for **signed** number comparison
 - The assembler uses a reserved register (`$at`) when expanding the pseudo instructions
 - MIPS compilers use `slt`, `slti`, `beq`, `bne` and the fixed value of 0 (always available by reading the register `$zero`) to create all relative conditions (**equal**, **not equal**, **less than**, **less than or equal**, **greater than**, **greater than or equal**)

less than	<code>blt \$s1, \$s2, Label</code>
<code>slt \$at, \$s1, \$s2</code>	# \$at set to 1 if \$s1 < \$s2
<code>bne \$at, \$zero, Label</code>	
less than or equal to	<code>ble \$s1, \$s2, Label</code>
greater than	<code>bgt \$s1, \$s2, Label</code>
great than or equal to	<code>bge \$s1, \$s2, Label</code>

- `bltu`, `bleu`, `bgtu` and `bgeu` are **pseudo instructions** for **unsigned** number comparison

Bounds Check Shortcut

- Treating signed numbers as if they were unsigned gives a low cost way of checking if $0 \leq x < y$ (index out of bounds for arrays)
 - The key is that negative integers in two's complement look like large numbers in unsigned notation.
 - Thus, an unsigned comparison of $x < y$ also checks **if x is negative** as well as **if x is less than y**

```
int    my_array[100] ;
```

```
// $t2 = 100
```

```
// $s1 has a index to the array and changes dynamically while executing the program
```

```
// $s1 and $t2 contain signed numbers, but the following code treats them as  
unsigned numbers
```

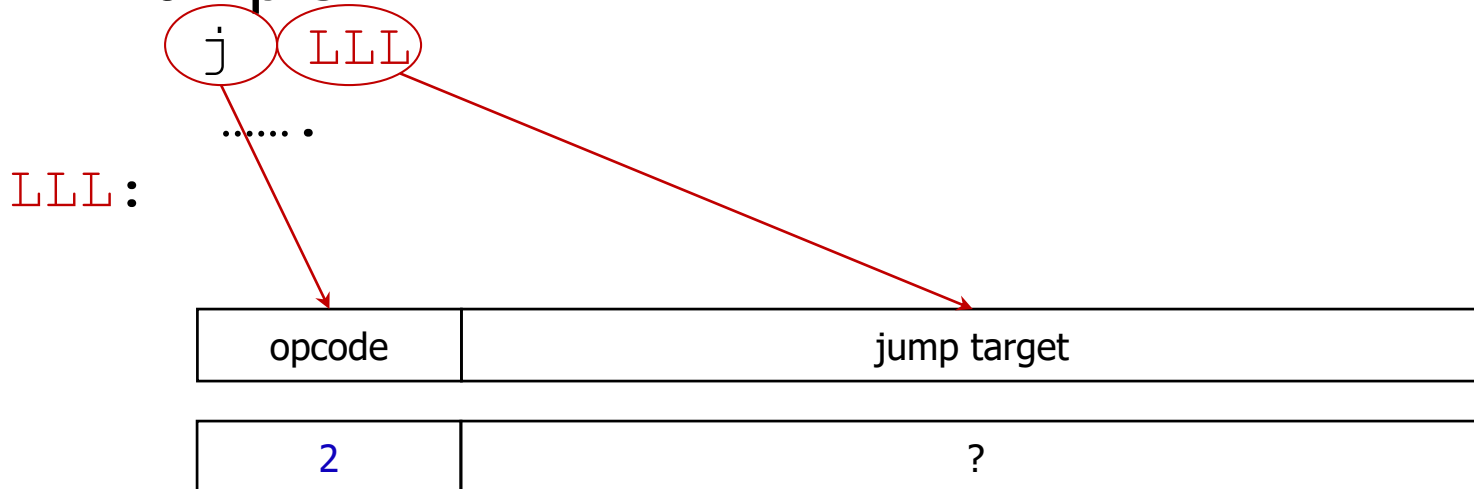
```
sltu $t0, $s1, $t2    # $t0 = 0 if $s1 >= 100 (=$t2) or $s1 < 0  
beq $t0, $zero, IOOB  # go to IOOB if $t0 = 0
```

j, jr, jal

- Unconditional branch instructions

- `j target // jump (J-format)`
- `jal target // jump and link (J-format)`
- `jr rs // jump register (R-format)`

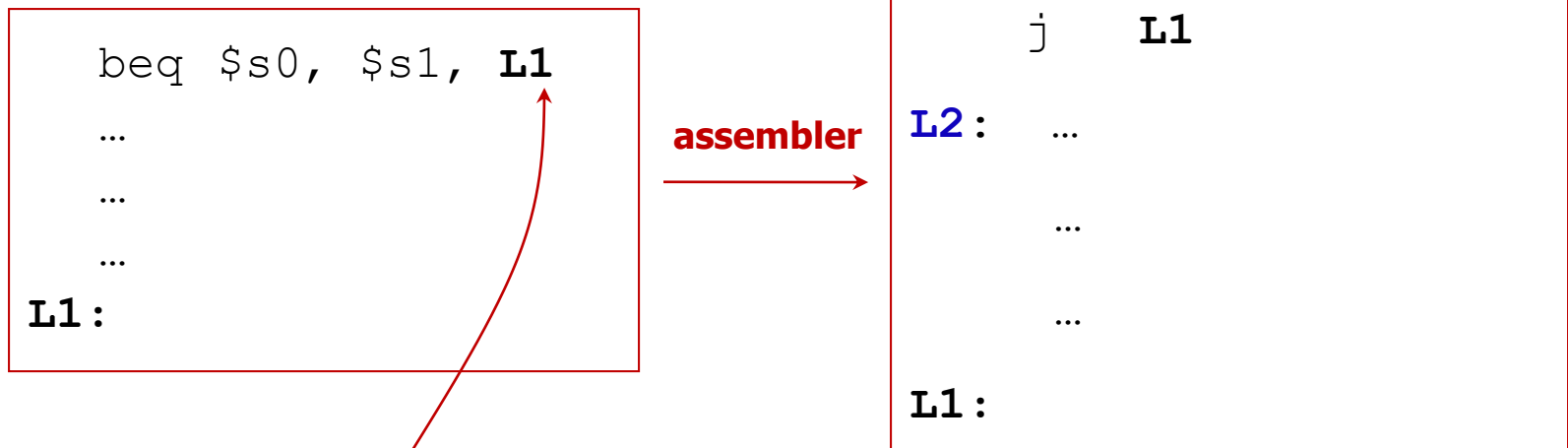
- Example



destination = {(PC+4)[31:28] , jump target, 2'b00}

Branching Far Away

- What if the branch destination is further away than can be captured in the 16-bit immediate field of `beq`?
- The assembler comes to the rescue; It inserts an unconditional jump to the branch target and inverts the condition



`L1` is too far to be accommodated in 16-bit immediate field of `beq`

While in C

High-level code

```
// determines the power
// of x such that 2x = 128

int pow = 1;
int x   = 0;

while (pow != 128) {
    pow = pow * 2;
    x = x + 1;
}
```

compile

MIPS assembly code

```
# $s0 = pow, $s1 = x

        addi $s0, $0, 1
        add  $s1, $0, $0
        addi $t0, $0, 128
while:   beq  $s0, $t0, done
        sll  $s0, $s0, 1
        addi $s1, $s1, 1
        j    while
done:
```

Notice that the assembly tests for the opposite case (`pow == 128`) than the test in the high-level code (`pow != 128`).

for in C

High-level code

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
    sum = sum + i;
}
```

compile

MIPS assembly code

```
# $s0 = i, $s1 = sum
    addi $s1, $0, 0
    add  $s0, $0, $0
    addi $t0, $0, 10
for:  beq  $s0, $t0, done
      add  $s1, $s1, $s0
      addi $s0, $s0, 1
      j    for
done:
```

Notice that the assembly tests for the opposite case ($i == 10$) than the test in the high-level code ($i \neq 10$).

Comparisons in C

High-level code

```
// add the powers of 2 from 1
// to 100
int sum = 0;
int i;

for (i=1; i < 101; i = i*2) {
    sum = sum + i;
}
```

compile

MIPS assembly code

```
# $s0 = i, $s1 = sum
    addi $s1, $0, 0
    addi $s0, $0, 1
    addi $t0, $0, 101
loop: slt  $t1, $s0, $t0
      beq  $t1, $0, done
      add  $s1, $s1, $s0
      sll  $s0, $s0, 1
      j    loop
done:
```

$\$t1 = 1$ if $i < 101$

Procedure (Function)

- Programmers use **procedure (or function)** to structure programs
 - To make the program modular and easy to understand
 - To allow code to be reused
 - Procedures allow the programmer to focus on just one portion of the task at a time
- **Parameters (arguments)** act as an interface between the procedure and the rest of the program
- Procedure calls
 - **Caller:** calling procedure (`main` in the example)
 - **Callee:** called procedure (`sum` in the example)

High-level code example

```
void main()  
{  
    int y;  
    y = sum(42, 7);  
    ...  
}  
  
int sum(int a, int b)  
{  
    return (a + b);  
}
```

jal

- Procedure call instruction (J format)

```
jal ProcedureAddress    # jump and link
                        # $ra <- pc + 4
                        # pc <- jump target
```

- jal saves PC+4 in the register \$ra to return from the procedure

3	26-bit address
---	----------------

High-level code

```
int main() {
    simple();
    a = b + c;
}

void simple() {
    return;
}
```

void means that simple doesn't return a value.

PC
compile

PC+4

MIPS assembly code

```
0x00400200 main: jal simple
0x00400204      add $s0, $s1, $s2
...
0x00401020 simple: jr $ra
```

jal: jumps to simple and saves PC+4 in the return address register (\$ra). In this case, \$ra = 0x00400204 after jal executes

jr

- Return instruction (R format)

jr \$ra #return (pc <- \$ra)

0	31				8
---	----	--	--	--	---

High-level code

```
int main() {  
    simple();  
    a = b + c;  
}  
  
void simple() {  
    return;  
}
```

compile

MIPS assembly code

```
0x00400200 main: jal    simple  
0x00400204          add    $s0, $s1, $s2  
...  
  
0x00401020 simple: jr    $ra
```

\$ra contains 0x00400204

jr \$ra: jumps to address in \$ra (in this case 0x00400204)

Procedure Call Conventions

- Procedure calling conventions
 - Caller
 - Passes arguments to a callee
 - Jumps to the callee
 - Callee
 - Performs the procedure
 - Returns the result to the caller
 - Returns to the point of call
- MIPS conventions
 - `jal` calls a procedure
 - Arguments are passed via `$a0`, `$a1`, `$a2`, `$a3`
 - `jr` returns from the procedure
 - Return results are stored in `$v0` and `$v1`

Arguments and Return Values

High-level code

```
int main()
{
    int y;
    ...
    // 4 arguments
    y = diffofsums(2, 3, 4, 5);
    ...
}

int diffofsums(int f, int g, int h, int i)
{
    int result;
    result = (f + g) - (h + i);
    return result;    // return value
}
```

MIPS assembly code

```
# $s0 = y

main:
    ...
    addi $a0, $0, 2    # argument 0 = 2
    addi $a1, $0, 3    # argument 1 = 3
    addi $a2, $0, 4    # argument 2 = 4
    addi $a3, $0, 5    # argument 3 = 5
    jal  diffofsums    # call procedure
    add  $s0, $v0, $0   # y = returned value
    ...

# $s0 = result
diffofsums:
    add $t0, $a0, $a1 # $t0 = f + g
    add $t1, $a2, $a3 # $t1 = h + i
    sub $s0, $t0, $t1 # result = (f + g) - (h + i)
    add $v0, $s0, $0  # put return value in $v0
    jr  $ra           # return to caller
```

Register Corruption

High-level code

```
int main()
{
    int a, b, c;
    int y;

    a = 1;
    b = 2;

    // 4 arguments
    y = diffofsums(2, 3, 4, 5);

    c = a + b;
    printf("y = %d, c = %d", y, c)
}

int diffofsums(int f, int g, int h, int i)
{
    int result;
    result = (f + g) - (h + i);
    return result;    // return value
}
```

- We need a place to temporarily store registers

MIPS assembly code

```
# $s0 = y

main:
    ...
    addi $t0, $0, 1    # a = 1
    addi $t1, $0, 2    # b = 2

    addi $a0, $0, 2    # argument 0 = 2
    addi $a1, $0, 3    # argument 1 = 3
    addi $a2, $0, 4    # argument 2 = 4
    addi $a3, $0, 5    # argument 3 = 5
    jal  diffofsums    # call procedure
    add  $s0, $v0, $0   # y = returned value

    add  $s1, $t0, $t1 # a = b + c
    ...

# $s0 = result
diffofsums:
    add $t0, $a0, $a1 # $t0 = f + g
    add $t1, $a2, $a3 # $t1 = h + i
    sub $s0, $t0, $t1 # result = (f + g) - (h + i)
    add $v0, $s0, $0  # put return value in $v0
    jr  $ra           # return to caller
```

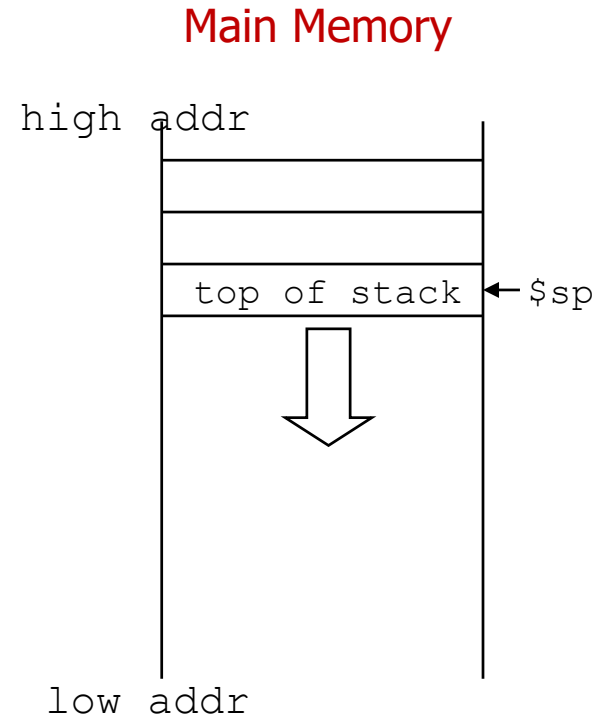
The Stack

- CPU has only a limited number of registers (32 in MIPS), so it typically can not accommodate all the variables you use in the code
 - So, programmers (or compiler) use the stack for backing up the registers and restoring those when needed
- Stack is a memory area used to **temporarily** save and restore data
 - Like a stack of dishes, **stack** is a data structure for spilling (saving) registers to memory and filling (restoring) registers from memory



The Stack - Spilling Registers

- **Stack** is organized as a **last-in-first-out (LIFO)** queue
- One of the general-purpose registers, $\$sp$ ($\$29$), is used to point to **the top of the stack**
 - The stack “grows” from high address to low address in MIPS
 - **Push**: add data onto the stack
 - $\$sp = \$sp - 4$
 - Store data on stack at new $\$sp$
 - **Pop**: remove data from the stack
 - Restore data from stack at $\$sp$
 - $\$sp = \$sp + 4$



Example (Problem)

- Called procedures (callees) must not have any unintended side effects to the caller
- `diffofsums` uses (overwrites) 3 registers (`$t0`, `$t1`, `$s0`)

MIPS assembly code

```
# $s0 = y

main:
    ...
    addi $a0, $0, 2    # argument 0 = 2
    addi $a1, $0, 3    # argument 1 = 3
    addi $a2, $0, 4    # argument 2 = 4
    addi $a3, $0, 5    # argument 3 = 5
    jal  diffofsums   # call procedure
    add  $s0, $v0, $0   # y = returned value
    ...

# $s0 = result
diffofsums:
    add $t0, $a0, $a1 # $t0 = f + g
    add $t1, $a2, $a3 # $t1 = h + i
    sub $s0, $t0, $t1 # result = (f + g) - (h + i)
    add $v0, $s0, $0   # put return value in $v0
    jr  $ra            # return to caller
```

Example (Solution with Stack)

```
# $s0 = result
diffofsums:
```

```
addi $sp, $sp, -12 # make space on stack
                        # to store 3 registers
sw    $s0, 8($sp)   # save $s0 on stack
sw    $t0, 4($sp)   # save $t0 on stack
sw    $t1, 0($sp)   # save $t1 on stack
```

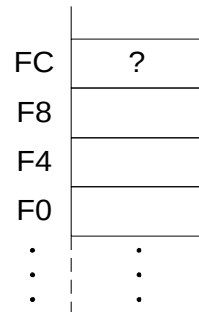
```
add $t0, $a0, $a1 # $t0 = f + g
add $t1, $a2, $a3 # $t1 = h + i
sub $s0, $t0, $t1 # result = (f + g) - (h + i)
add $v0, $s0, $0  # put return value in $v0
```

```
lw    $t1, 0($sp)   # restore $t1 from stack
lw    $t0, 4($sp)   # restore $t0 from stack
lw    $s0, 8($sp)   # restore $s0 from stack
addi $sp, $sp, 12   # deallocate stack space
jr    $ra           # return to caller
```

"Push" (back up) the registers to be used in the callee to the stack

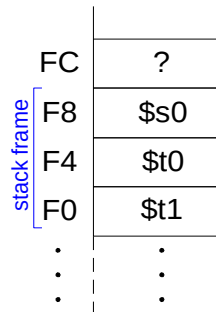
"Pop" (restore) the registers from the stack prior to returning to the caller

Address Data



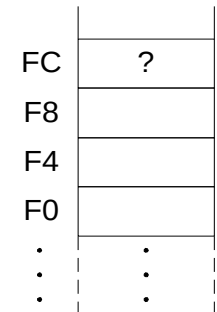
(a)

Address Data



(b)

Address Data



(c)

Nested Procedure Calls

- Procedures that do not call others are called **leaf** procedures
- Life would be simple if all procedures were leaf procedures, but they aren't
 - The main program calls `procedure 1 (proc1)` with an argument of 3 (by placing the value 3 into register `$a0` and then using `jal proc1`)
 - `Proc1` calls `procedure 2 (proc2)` via `jal proc2` with an argument 7 (also placed in `$a0`)
 - There is a conflict over the use of register `$a0` and `$ra`
 - Use stack to preserve registers

```
proc1:
    addi $sp, $sp, -4    # make space on stack
    sw   $ra, 0($sp)     # save $ra on stack
    jal  proc2
    ...
    lw   $ra, 0($sp)     # restore $s0 from stack
    addi $sp, $sp, 4     # deallocate stack space
    jr   $ra             # return to caller
```

Recursive Procedure Call

- Recursive procedures invoke **clones** of themselves

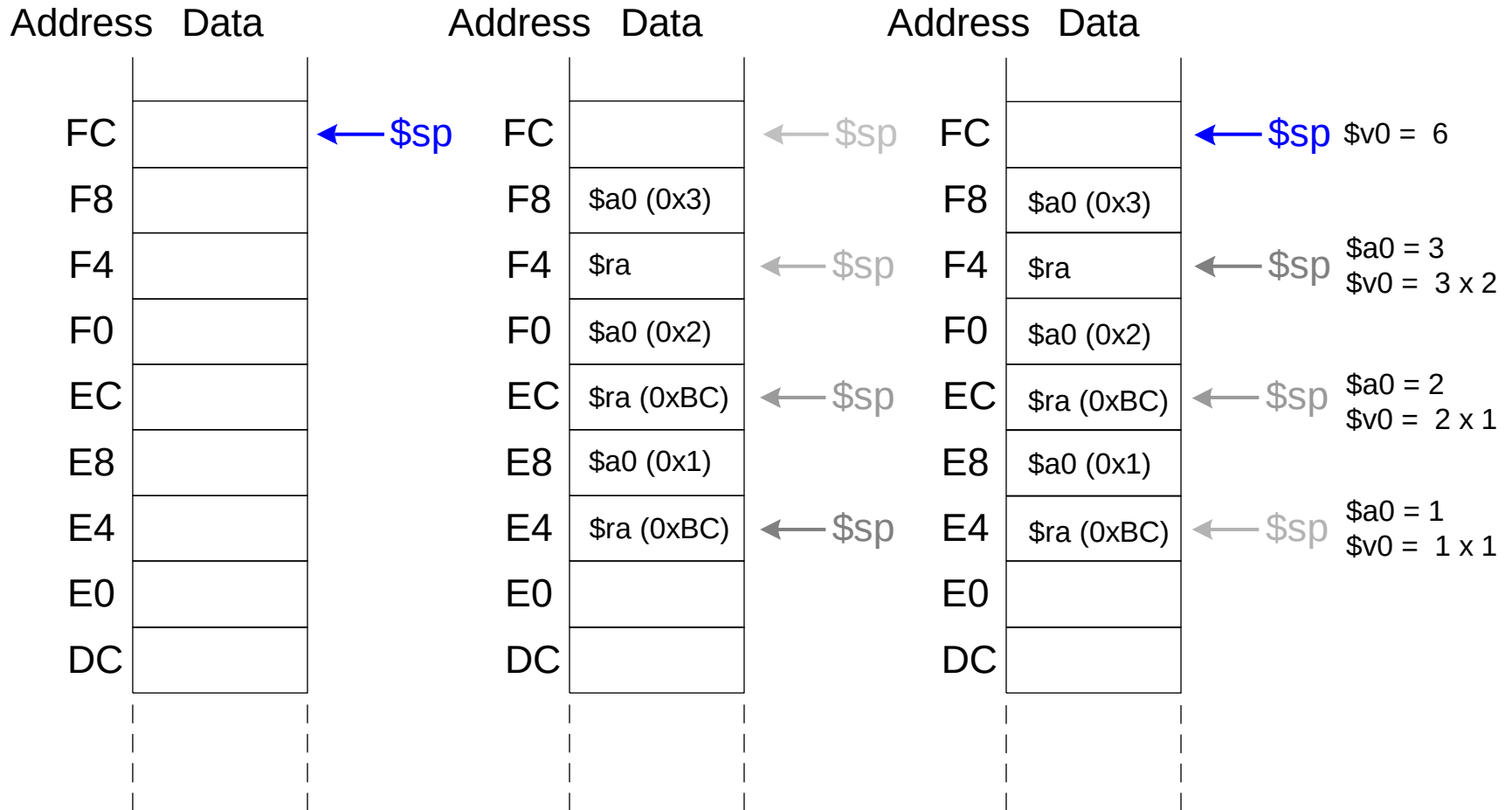
High-level code

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

MIPS assembly code

```
0x90 factorial: addi $sp, $sp, -8 # make room  
0x94            sw  $a0, 4($sp)  # store $a0  
0x98            sw  $ra, 0($sp)  # store $ra  
0x9C            addi $t0, $0, 2  
0xA0            slt  $t0, $a0, $t0 # a <= 1 ?  
0xA4            beq  $t0, $0, else # no: go to else  
0xA8            addi $v0, $0, 1   # yes: return 1  
0xAC            addi $sp, $sp, 8  # restore $sp  
0xB0            jr   $ra         # return  
0xB4            else: addi $a0, $a0, -1 # n = n - 1  
0xB8            jal  factorial    # recursive call  
0xBC            lw   $ra, 0($sp)  # restore $ra  
0xC0            lw   $a0, 4($sp)  # restore $a0  
0xC4            addi $sp, $sp, 8  # restore $sp  
0xC8            mul  $v0, $a0, $v0 # n * factorial(n-1)  
0xCC            jr   $ra         # return
```

Stack during Recursive Call (3!)



Backup Slides

Stack Example

```
int main()
{
    int a, b, c; // local variable:
                // allocated in stack
    int myarray[5]; // local variable:
                  // allocated in stack

    a = 2;
    b = 3;

    *(myarray+1) = a;
    *(myarray+3) = b;

    c = myarray[1] + myarray[3];

    return c;
}
```

compile

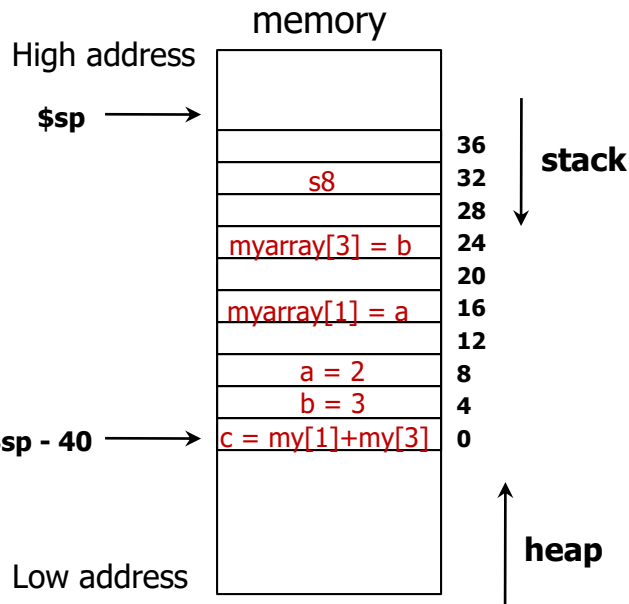
```
int main()
{
    400168:    27bdfdd8    addiu    sp,sp,-40
    40016c:    afbe0020    sw      s8,32(sp)
    400170:    03a0f021    move    s8,sp
    int a, b, c; // local variable: allocated in stack
    int myarray[5]; // local variable: allocated in stack

    a = 2;
    400174:    24020002    li      v0,2
    400178:    afc20008    sw      v0,8(s8)
    b = 3;
    40017c:    24020003    li      v0,3
    400180:    afc20004    sw      v0,4(s8)

    *(myarray+1) = a;
    400184:    27c2000c    addiu    v0,s8,12
    400188:    24430004    addiu    v1,v0,4
    40018c:    8fc20008    lw      v0,8(s8)
    400190:    00000000    nop
    400194:    ac620000    sw      v0,0(v1)
    *(myarray+3) = b;
    400198:    27c2000c    addiu    v0,s8,12
    40019c:    2443000c    addiu    v1,v0,12
    4001a0:    8fc20004    lw      v0,4(s8)
    4001a4:    00000000    nop
    4001a8:    ac620000    sw      v0,0(v1)

    c = myarray[1] + myarray[3];
    4001ac:    8fc30010    lw      v1,16(s8)
    4001b0:    8fc20018    lw      v0,24(s8)
    4001b4:    00000000    nop
    4001b8:    00621021    addu    v0,v1,v0
    4001bc:    afc20000    sw      v0,0(s8)

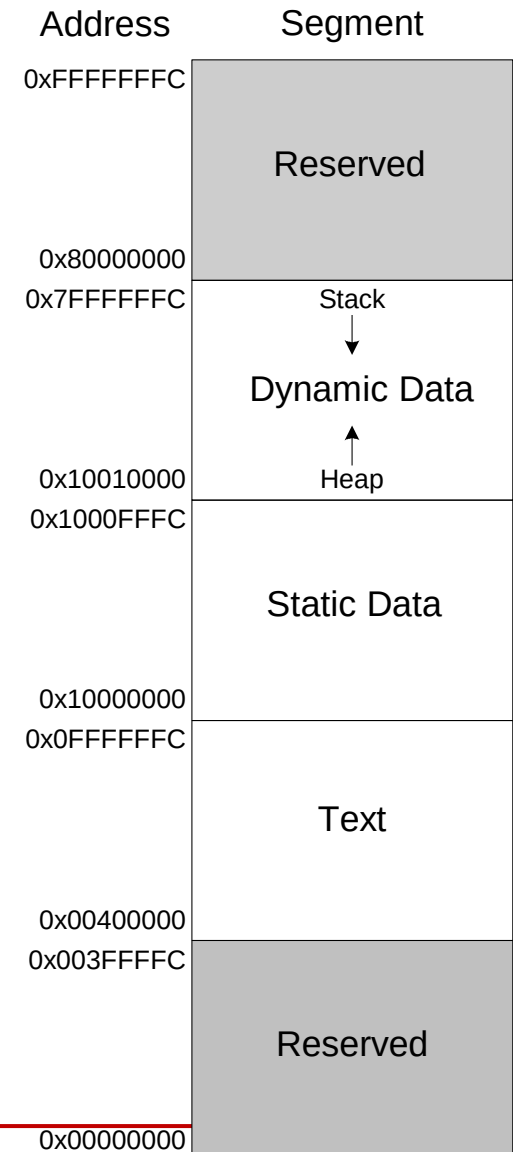
    return c;
    4001c0:    8fc20000    lw      v0,0(s8)
}
    4001c4:    03c0e821    move    sp,s8
    4001c8:    8fbe0020    lw      s8,32(sp)
    4001cc:    27bd0028    addiu    sp,sp,40
    4001d0:    03e00008    jr      ra
    4001d4:    00000000    nop
```



$\$s8 = \sp $\$sp = \$sp - 40$

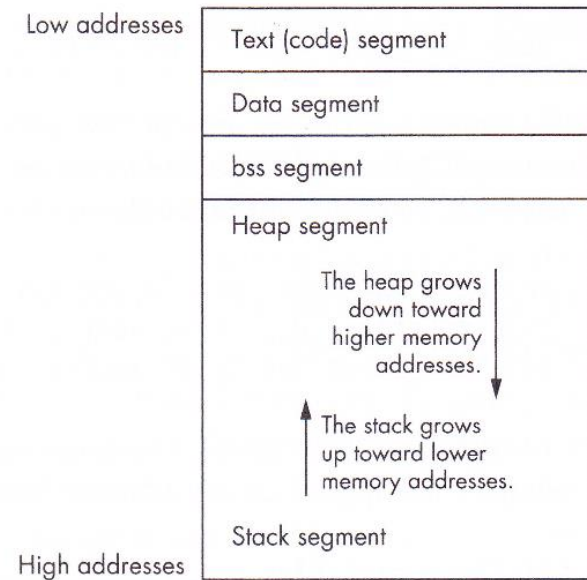
The MIPS Memory Map

- Addresses shown are only a software convention (not part of the MIPS architecture)
- **Text segment:** Instructions are located here
 - The size is almost 256MB
- **Static and global data segment** for constants and other static variables
 - In contrast to local variables, global variables can be seen by all procedures in a program
 - Global variables are declared outside the main in C
 - The size of the global data segment is 64KB
- **Dynamic data segment** holds **stack** and **heap**
 - Data in this segment are dynamically allocated and deallocated throughout the execution of the program
 - **Stack** is used
 - To save and restore registers used by procedures
 - To hold local variables
 - **Heap** stores data that is allocated by the program during runtime
 - Allocate space on the heap with `malloc()` and free it with `free()` in C
- **Reserved segments** are used by the operating system



Linear Space Segmentation

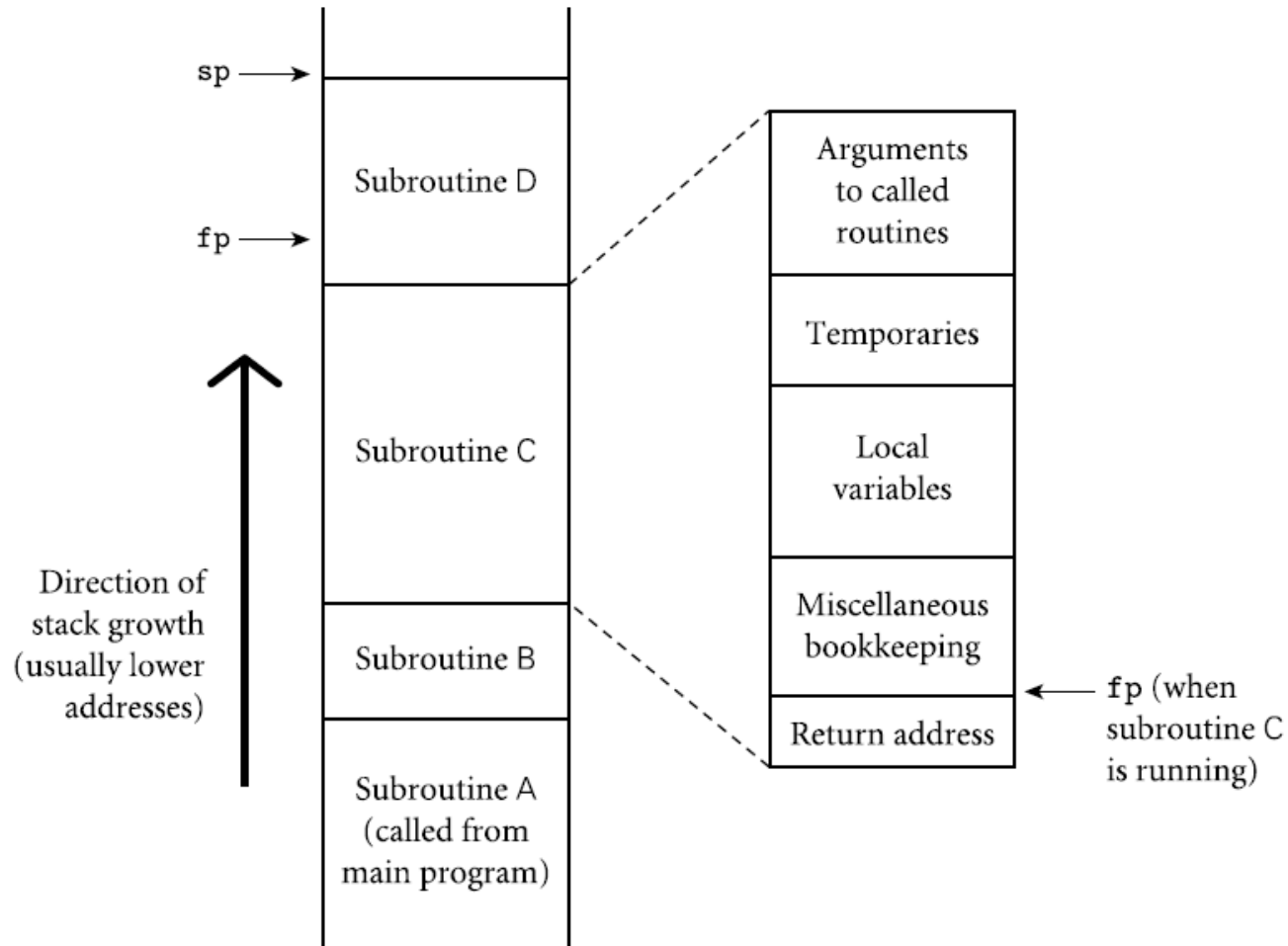
- A compiled program's memory is divided into 5 segments:
 - **Text segment** (code segment) where program (assembled machine instructions) is located
 - **Data and bss segments**
 - Data segment is filled with the initialized data and static variables
 - bss (Block Started by Symbol) is filled with the uninitialized data and static variables
 - **Heap segment** for dynamic allocation and deallocation of memory using `malloc()` and `free()`
 - **Stack segment** for scratchpad to store local variables and context during context switch



Stack Frame

- Frame Pointer (FP) or Stack Base Pointer(BP) is for referencing local variable in the current stack frame
- Each routine is given a new stack frame when it is called, and each stack frame contains
 - Parameters to the function
 - Local variables
 - Return address

Frame Pointer



Code that needs **to access a local variable** within the current frame, or an argument near the top of the calling frame, can do so **by adding a predetermined offset to the value in the frame pointer**.

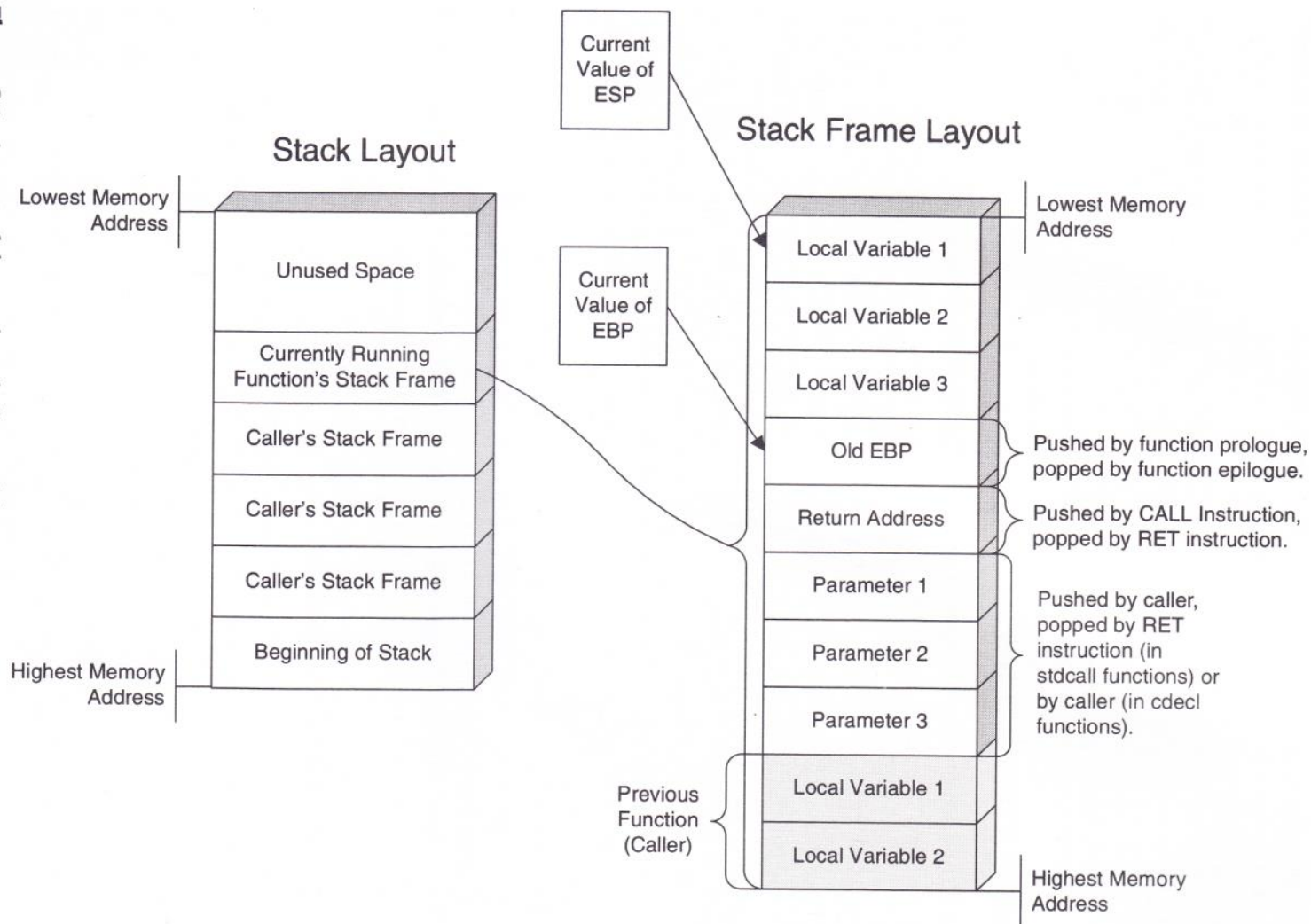
SP & FP

- The data stored in the stack frame may sometimes be accessed directly via the **stack pointer** register (SP, which indicates the current top of the stack).
- However, as the stack pointer is variable during the activation of the routine, memory locations within the stack frame are more typically accessed via a separate register.
- This register is often termed the **frame pointer** or **stack base pointer** (BP) and is set up at procedure entry to point to a *fixed* location in the frame structure (such as the return address).

-Wiki

Stack Layout with x86

Figure C.1 Layout of the stack and of a stack frame.



Preserved and NonPreserved Registers

- In the previous example, if the calling procedure does not use the temporary registers (\$t0, \$t1), the effort to save and restore them is wasted
- To avoid this waste, MIPS divides registers into **preserved** and **non-preserved** categories
 - The preserved registers include \$s0 ~ \$s7 (saved)
 - The non-preserved registers include \$t0 ~ \$t9 (temporary)
 - So, a procedure must save and restore any of the preserved registers it wishes to use, but it can change the non-preserved registers freely
- The callee must save and restore any preserved registers it wishes to use
- The callee may change any of the non-preserved registers
 - But, if the caller is holding active data in a non-preserved register, the caller needs to save and restore it

Preserved (Callee-saved)	Non-preserved (Caller-saved)
\$s0 - \$s7	\$t0 - \$t9
\$ra	\$a0 - \$a3
\$sp	\$v0 - \$v1
stack above \$sp	stack below \$sp

Storing Saved Registers on the Stack

```
# $s0 = result
diffsums:
    addi $sp, $sp, -4    # make space on stack to
                        # store one register
    sw  $s0, 0($sp)      # save $s0 on stack
                        # no need to save $t0 or $t1

    add $t0, $a0, $a1    # $t0 = f + g
    add $t1, $a2, $a3    # $t1 = h + i
    sub $s0, $t0, $t1    # result = (f + g) - (h + i)
    add $v0, $s0, $0     # put return value in $v0
    lw  $s0, 0($sp)      # restore $s0 from stack
    addi $sp, $sp, 4     # deallocate stack space
    jr  $ra              # return to caller
```