INTRODUCTION TO COMPUTER ORGANIZATION

CCIT4026

Chapter 5: Instruction Set Design (Part 4)

Outline

- Procedure Call
- MIPS Addressing Mode

Procedure Call

Why do you need to write your programs in procedures ?

(1) ***Easier to understand how the program flows:***

- **For users :**

  You don't need to read a very long code and guess what they does.

- **For Programmers:**

  A complex problem can be broken down into several sub-problems (procedures), each sub-problem may be easier to tackle.

  This lets you understand what you are expected to do to solve the problem

(2) ***Easier to use:***

For users:

The users actually don't need to know how the functions work. They only need to understand how to use the functions (the input and the output required in this function

Procedure Call

(3) ***Easier to debug & build up your program systematically:***

For programmers:

After a procedure is written and fully tested, if you further build your program and encountered problems, the problems are usually caused by the new code added / modified.

Errors can be detected in an easier way.

(4) ***Functions can usually be tested individually:***

For both programmers and users:

You may build up another program and design test cases to check the correctness of the functions.

(5) ***Program code are reusable:***

For programmers :

Your program could be much shorter if the functions are well designed.

Example #1

Example:

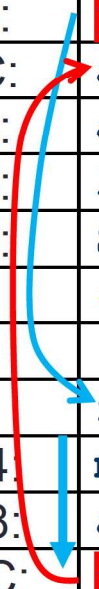
```
def foo(a):                    # this is the callee
    return a*a + a


print(foo(100))                # this is the "caller" function
```

To invoke a procedure in MIPS we will use the instruction,
"**`jal func_name`**".
When the function called (*callee*) is completed, we will use the instruction
"**`jr $ra`**"
to return to the caller's procedure.

If the function requires arguments passing, **by convention**, the caller will store arguments in registers **`$a0-$a3`**. If the function returns any value(s), the callee will store the evaluated result(s) into registers **`$v0`** & **`$v1`**.

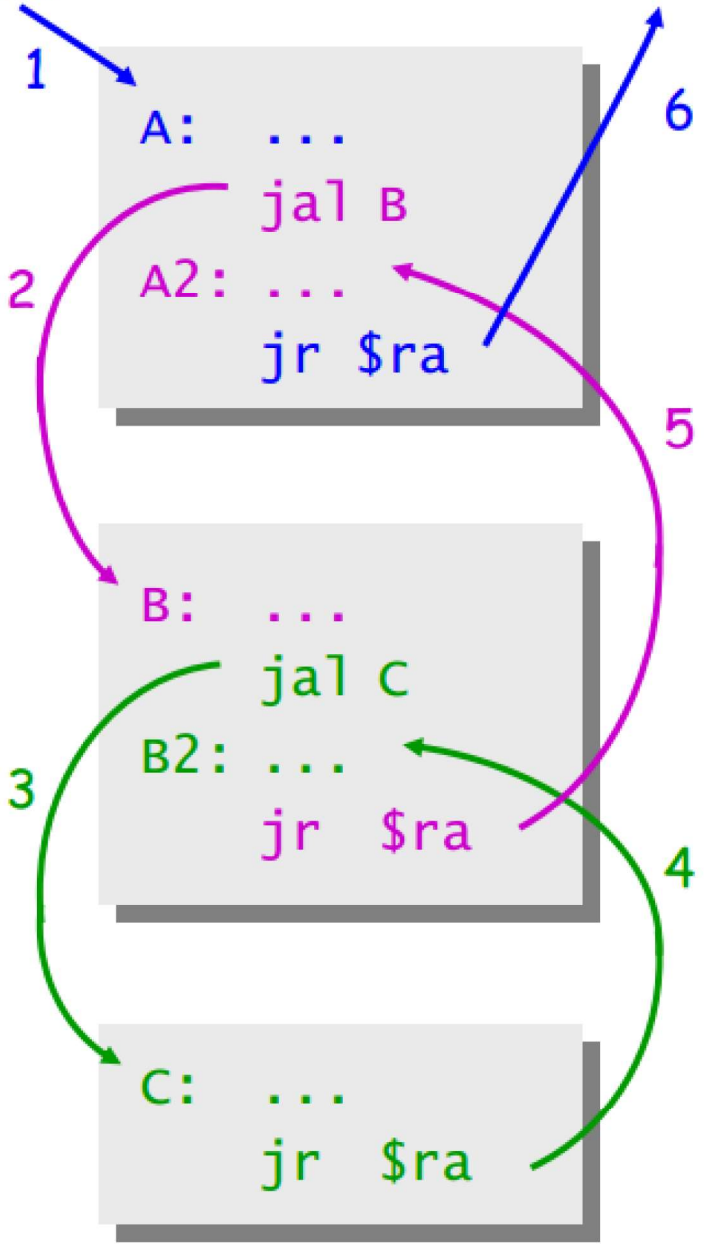
Example #1

Address (byte)	
	`main:`
0x1000:	`addi $s0, $zero, 100`
0x1004:	`add $a0, $zero, $s0` `# copy the value of $s0 into $a0`
0x1008:	`jal foo` `# call the procedure here`
0x100C:	`add $s1, $zero, $v0` `# use value in $v0 (copy to $s1)`
0x1010:	`add $a0, $zero, $s1` `# copy the value of $s1 to $a0`
0x1014:	`li $v0, 1` `# display the value of $s1`
0x1018:	`Syscall`
	`# something else to do here`
	`foo:`
0xAB24:	`mul $v0, $a0, $a0` `# evaluation of return value (a*a)`
0xAB28:	`add $v0, $v0, $a0` `# a * a + a, return in $v0`
0xAB2C:	`jr $ra` `# return to caller's procedure`



(Nested) Procedure Calls

1. Someone calls A
    2. A calls B
        3. B calls C
        4. C returns to B
    5. B returns to A
6. A returns



C must return to B before B can return to A. Is there any problem with a simple jal / jr pair of instructions???

Procedure Call (variable)

- Unlike C / C++ / Python programs, the concept of global variables and local variables is not applicable for the registers in MIPS. All registers can be used and modified globally.

- There could be **conflicts** in the usage of registers between *caller* and *callee*.

- As the complexity of your program increases, more conflicts in using registers will be encountered and this problem should be handled systematically.

- **In MIPS, the conflicts should be resolved using stack.** For the example above,
  - **Before the procedure is called,**
    Push the register value(s) onto stack.
  - **After the procedure is completed,**
    Pop the stack to restore the value(s) back to the original register(s).

Example #2a

Modified in Callee

```
frodo: li   $a0, 3
       li   $a1, 1
       li   $s0, 4
       li   $s1, 1

       jal  gollum

       add  $v0, $a0, $a1
       add  $v1, $s0, $s1
       jr   $ra
```

```
gollum:

       li   $a0, 2
       li   $a2, 7
       li   $s0, 1
       li   $s2, 8
       ...

       jr   $ra
```

Procedure Call (caller and callee)

- **Caller**: saving and restore the following caller-saved registers:

  `$t0-$t9` `$a0-$a3` `$v0-$v1`

  (passing parameters) (return function value)

- **Callee**: saving and restore the following callee-saved registers:

  `$s0-$s7` `$ra`

  (return address)

- `$ra` is saved by a callee who is also a caller

Example #2b

```
frodo: li   $a0, 3
       li   $a1, 1
       li   $s0, 4
       li   $s1, 1

       # Save registers
       # $a0, $a1, $s0, $s1

       jal  gollum

       # Restore registers
       # $a0, $a1, $s0, $s1

       add  $v0, $a0, $a1
       add  $v1, $s0, $s1
       jr   $ra
```

```
gollum:
       # Save registers
       # $a0 $a2 $s0 $s2

       li   $a0, 2
       li   $a2, 7
       li   $s0, 1
       li   $s2, 8
       ...

       # save $a0 $a2 $ra
       jal gollumson
       # restore registers
       # $a0 $a2

       # Restore registers
       # $a0 $a2 $s0 $s2 $ra

       jr   $ra
```

Where are the registers saved?

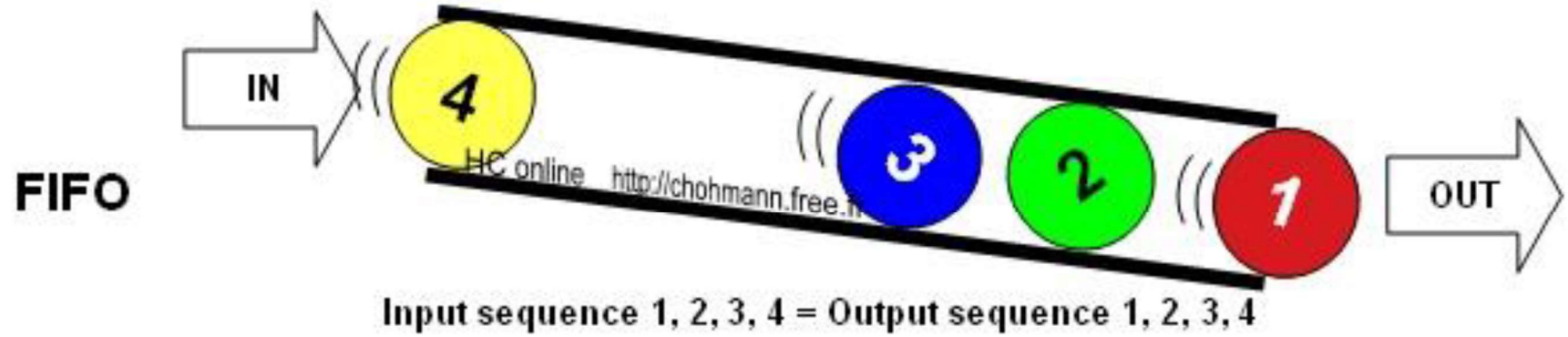
- Now we know who is responsible for saving which registers, but we still need to discuss where those registers are saved.

- It would be nice if each function call had its own private memory area.
  - **Prevent other function calls from overwriting our saved registers**
  - **Use this private memory for other purposes too, like storing local variables.**

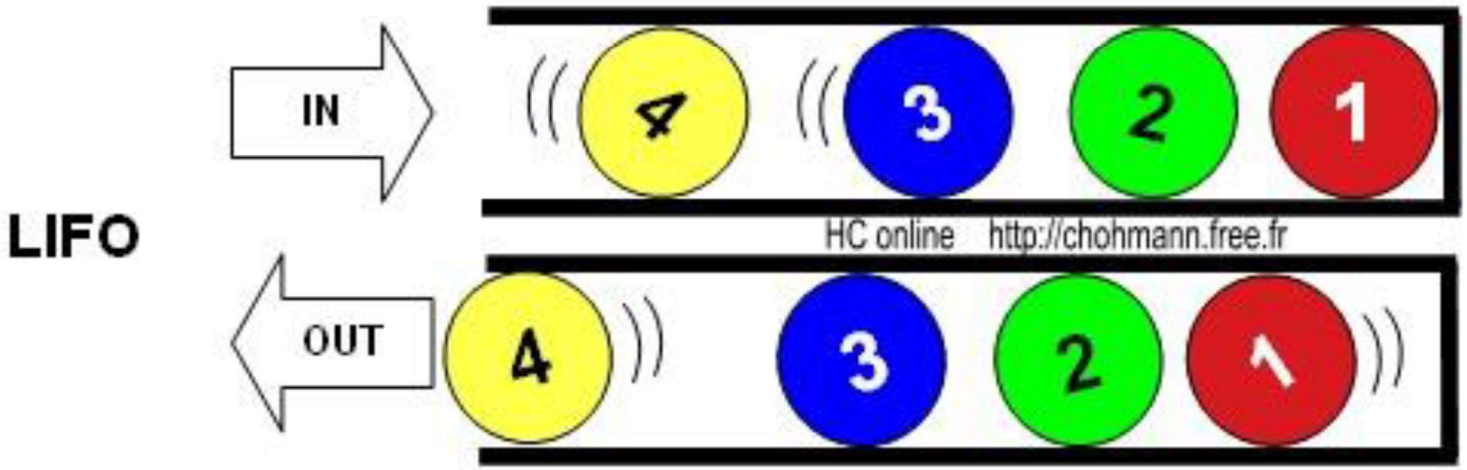
- Procedure calls and returns occur in a stack-like order (the most recently called function is the first one to return)

FIFO vs LIFO

Picture source:
http://chohmann.free.fr/SCM/fifo.htm



Input sequence 1, 2, 3, 4 = Output sequence 1, 2, 3, 4



Input sequence 1, 2, 3, 4 ≠ Output sequence 4, 3, 2, 1

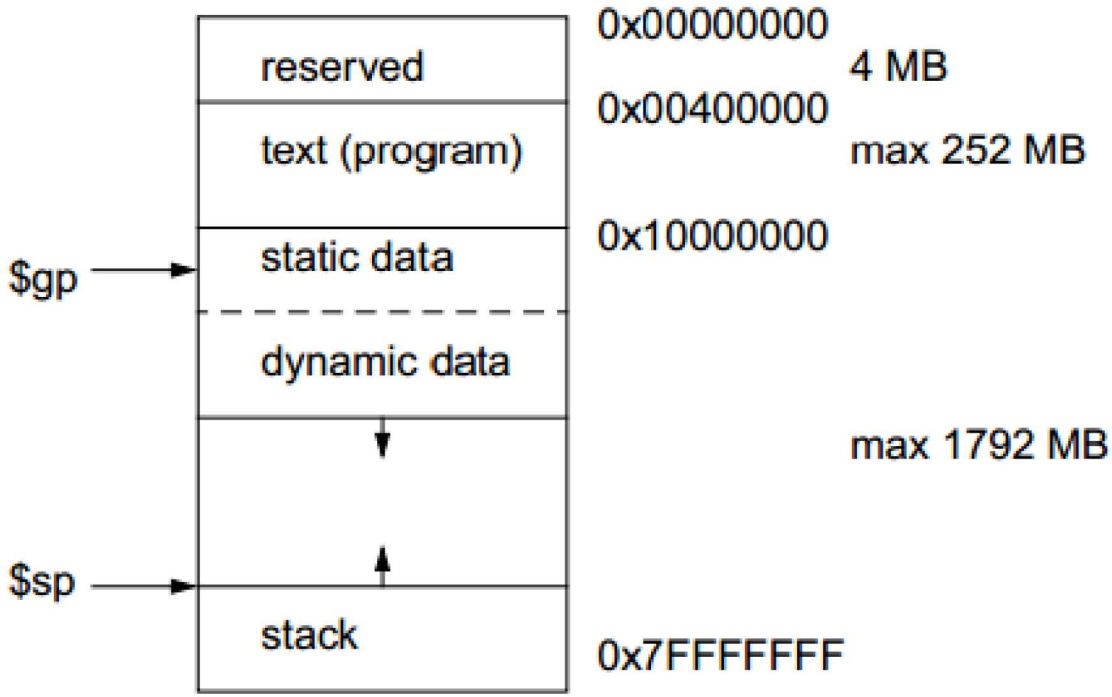
Procedure Call (Stack)

- Each executing program (process) has a stack.
- A stack is a dynamic data structure that is accessed in a LIFO.
- The register **`$sp`** is automatically loaded to point to the first empty slot on the top of the stack (high memory address).
- MIPS does not provide “push” and “pop” instructions. Instead, they must be done explicitly by the programmer (using *lw* and *sw instructions*).

Procedure Call (Stack)

By convention, the stack grows towards lower memory addresses.

- To allocate space on the stack, decrement **`$sp`**
- To free old stack space, increment **`$sp`**

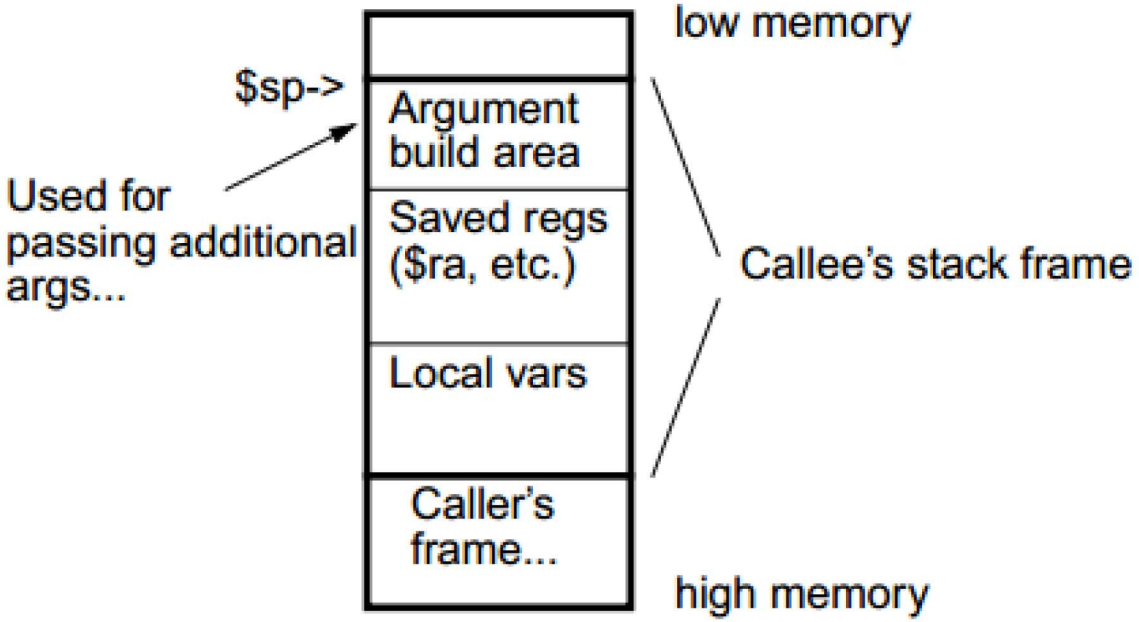


The “beauty” of learning assembly language programming – you are down to the “exact” details of how memories are segmented and used

Procedure Call (Stack)

A stack frame is a block of memory on the stack that is used for:

- passing arguments
- Saving registers
- Space for local variables



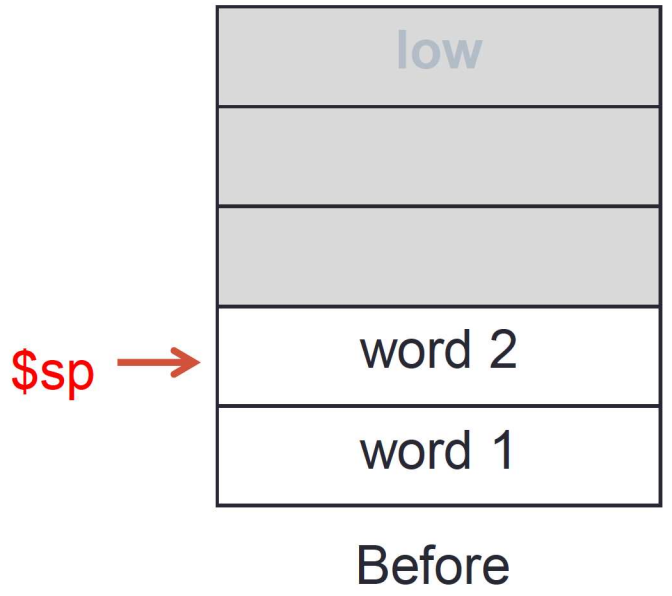
Procedure Call (push)

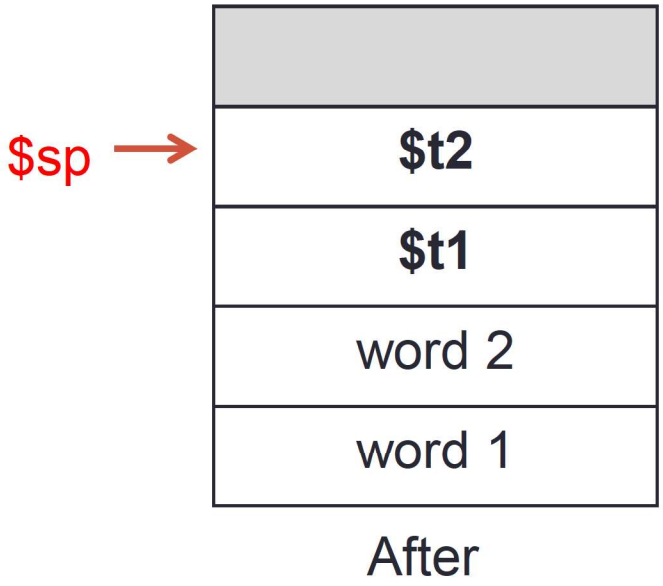
- To push elements onto the stack:
  - Move the stack pointer **$sp** up (to low address) to make room for new data
  - Store the element into the stack
- Example, push register **$t1** and **$t2** onto the stack:

```
addi $sp, $sp, -8
sw $t1, 4($sp)
sw $t2, 0($sp)
```

Equivalent to:

```
sw $t1, -4($sp)
sw $t2, -8($sp)
addi $sp, $sp, -8
```





Procedure Call (pop)

- You can access any element in the stack if you know where it is relative to **`$sp`**
  - To retrieve the value of **`$t1`** from stack:

```
lw $t1, 4($sp)
```

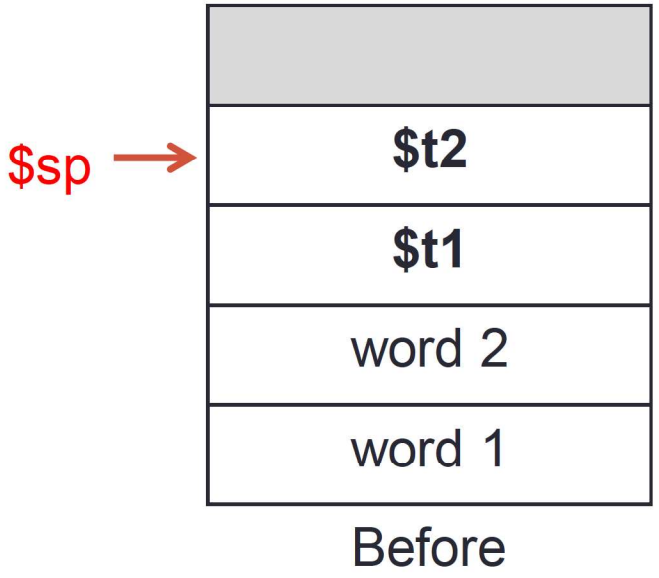
  - To retrieve the value of **`$t2`** from stack:

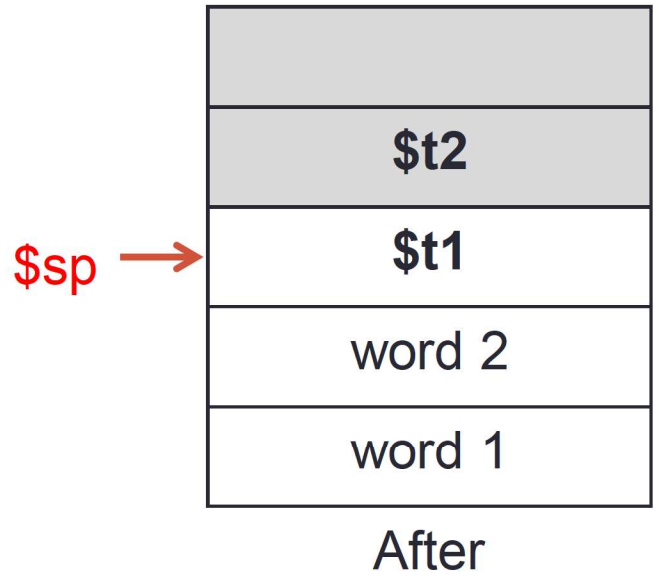
```
lw $t2, 0($sp)
```

- Pop or erase elements by adjusting the stack point downwards.

```
addi $sp, $sp, 4
```

The popped data $t2 is still present in memory, but data past the stack pointer is considered invalid.





Example #3

```
caller:
# .. assume $t0,$t1,$s0,$s1 are in use
# (~~ Step 1 ~~ push caller saved
# register [$t0 - $t1] )
addi $sp, $sp, -8 # memory needed = 2 x
                  # 4 byte = 8 bytes
sw $t0, 4($sp)
sw $t1, 0($sp)
# (~~ Step 2 ~~ invoke the procedure )
jal callee

# (~~ Step 6 ~~ Restored the register
# [$t0 - $t1] )
lw $t1, 0($sp)
lw $t0, 4($sp)
addi $sp, $sp, 8

# Terminate the program
li $v0, 10
syscall
```

```
# (~~ Step 3 ~~ push caller saved
register [$s0 -$s1])
callee:
addi $sp, $sp, -8 # memory needed =
                  # 2 x 4 byte = 8 bytes
# push callee saved register on stack
sw $s0, 4($sp)
sw $s1, 0($sp)

# Actual implementation of the procedure
# assume $s0, $s1 are in use only

# (~~ Step 4 ~~ Restore callee saved
# registers from stack )
lw $s0, 4($sp)
lw $s1, 0($sp)
addi $sp, $sp, 8

# (~~ Step 5 ~~ Return to caller )
jr $ra
```

Procedure Call (summary)

(1) The following steps are necessary to invoke a call:

(a) Parameter passing. **By convention**, the first four parameters of a procedure call are normally passed to registers `$4` to `$7` (`$a0` to `$a3`). The remaining arguments are pushed onto the stack. [as described before]

(b) Save the *Caller-Saved Registers*. This includes `$t0-$t9`, if they contain live values at the call site. That is, if you do not want some values of `$t0-$t9` to be modified after the procedure is called, you need to push them into stack.

(c) Use the `jal` (Jump and Link) instruction which jumps to the specified memory location and store the address of the next instruction into register `$31` (`$ra`), which is the return address of the procedure call.

Procedure Call (summary)

(2) Within a called routine, the following steps are necessary:

(a) Establish the stack frame by subtracting the frame size from the stack pointer (`$sp`).

(b) Save the *Callee-Saved Registers* into the frame. Any of the registers `$s0-$s7` should be saved if they are used in the procedure.

Register `$ra` ***should be saved if the procedure will call another procedure (The reason behind will be described later).***

(3) Finally, to return from a call:

(a) If the call is a function, place the returned value into `$v0`.

(b) Restore any *Callee-Saved Registers* (`$s0-$s7`) that were saved at the beginning of the procedure.

(c) Pop the stack frame by adding the frame size to `$sp`.

(d) Return by using `jr $ra` to jump to the return address stored in `$ra`

Procedure Call (summary)

- Registers for procedure calling:
  - `$a0`-`$a3`: four **argument registers** for passing parameters
  - `$v0`-`$v1`: two **value registers** for returning values
  - `$ra`: one **return address register** for returning to the point of origin
- **Program counter** (PC) or **instruction address register**:
  - Register that holds address of the current instruction being executed

Procedure Call (summary)

- **`jal`** ('jump and link'):
  - **`jal ProcedureAddress`**
  - Two things happen at the same time
    1. First, it save the address of the following instruction (i.e., PC + 4 as return address) to register **`$ra`**
    2. Then, jump to address specified by **`ProcedureAddress`**

- **`jr`** ('jump register')
  - **`jr register`**
  - An unconditional jump to the address specified in a register
  - Can be used to return from a procedure
    - How?

      `jr $ra` (jumps to the address stored in register **`$ra`**)

Example #4 (Python)

Python code example

```
def f1():
    print("This is function F1")
    f2()
    print("Try return to main procedure")

def f2():
    print("This is function F2")

# The main program begins here
# caller function
f1()
print("All functions completed!")
```

Example #4 (Assembly)

```
1  #------- DATA SEGMENT ----------------
2  .data
3  msg1: .asciiz "This is function F1\n"
4  msg2: .asciiz "This is function F2\n"
5  msg3: .asciiz "Try return to main procedure\n"
6  msg4: .asciiz "All functions completed!\n"
7  #------- TEXT SEGMENT ----------------
8  .text
9  .globl __start
10 __start:
11 # Main function starts here
12             jal f1                          # call function f1 - (1)
13             la $a0, msg4                    # point to msg4
14             syscall                         # to print it out
15             # Terminate the program
16             li $v0, 10
17             syscall
```

Example #4 (Assembly)

```
18  ### def f1(): ###
19  f1:         la $a0, msg1            # print msg1
20              li $v0, 4
21              syscall
22              jal f2                  # call function f2 - (2) - to print msg2
23              la $a0, msg3            # print msg 3
24              syscall
25              jr $ra                  # Expected to return to main function - (4)
26  ### End of function f1 ###
27
28  ### void f2( ) ###
29  f2:         la $a0, msg2            # f2 to print msg2
30              syscall
31              jr $ra                  # Return to function f1 - (3)
32  ### end of function f2 ###
```

Try to run this program?

What is wrong with the code?
- **`$ra`** is modified by **`jal f2`**

What can you do to correct it?
- Save previous value of **`$ra`**

Example #4 (Assembly)

```
18  ### def f1(): ###
19  f1:         la $a0, msg1            # print msg1
20              li $v0, 4
21              syscall
22              addi $sp,$sp,-4
23              sw $ra,0($sp)
24              jal f2                  # call function f2 - (2) - to print msg2
25              lw $ra,0($sp)
26              addi $sp,$sp,4
27              la $a0, msg3            # print msg 3
28              syscall
29              jr $ra                  # Expected to return to main function - (4)
30  ### End of function f1 ###
31
32  ### void f2( ) ###
33  f2:         la $a0, msg2            # f2 to print msg2
34              syscall
35              jr $ra                  # Return to function f1 - (3)
36  ### end of function f2 ###
```

MIPS Addressing Modes

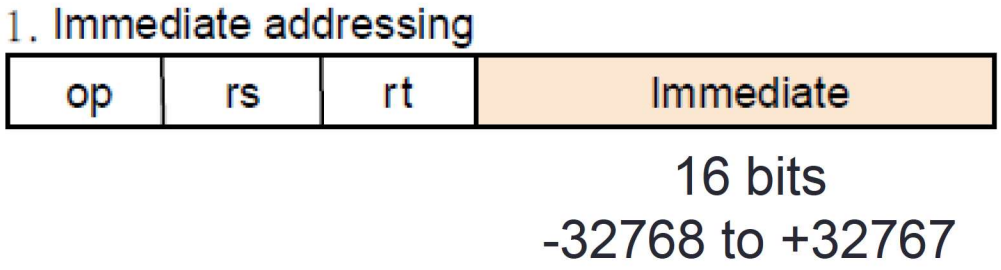
(1) Immediate addressing
(2) Register addressing
(3) Base addressing
(4) PC-relative addressing
(5) Pseudodirect addressing

MIPS Addressing Modes

Addressing modes are the ways of specifying an operand or a memory address

(1) Immediate addressing

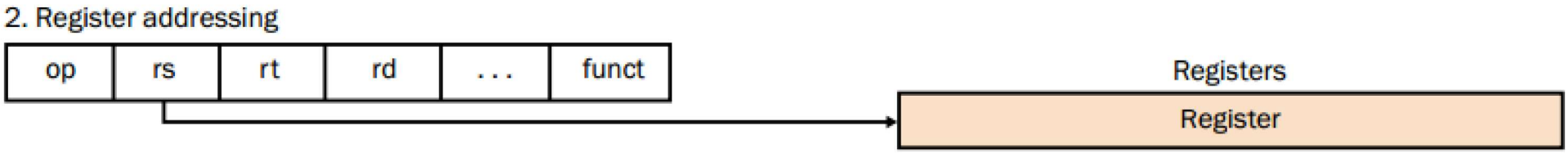
- Operand is a constant within the instruction
- Advantage: No need to have extra memory access to fetch the operand
- use when data is in the instruction itself



e.g. `addi $t0, $t1, 250`

MIPS Addressing Modes

- (2) Register addressing
  - The simplest addressing mode, operands are in registers
  - Avoid delays associated with memory access
  - use in R-type instructions

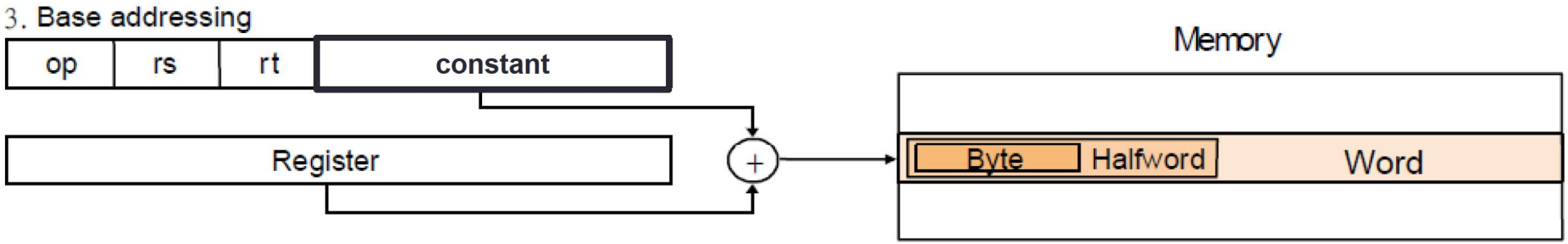


e.g. `add $t0, $t1, $t2`

`jr $ra`

MIPS Addressing Modes

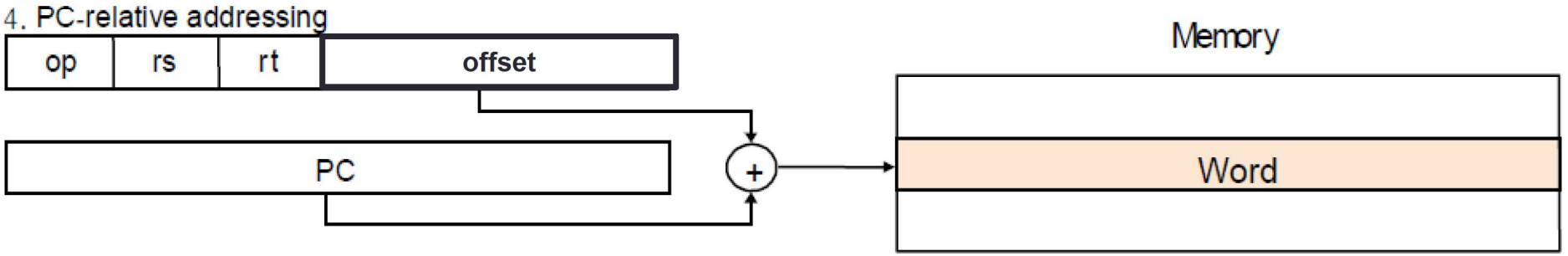
- (3) Base addressing
  - Operand is the address of sum of a register and a constant in the instruction
  - use in load and store instructions



e.g. `lw $t0, 10($t1)`

MIPS Addressing Modes

- (4) PC-relative addressing
  - Instruction address is the sum of the PC and a 16-bit constant
  - The offset is the number of instructions to jump
  - Hence, if we take the branch:
    PC = (PC+4) + offset*4
  - use in branch instructions



e.g. `beq $t0, $t5, L1`

MIPS Addressing Modes

- (4) PC-relative addressing (Example)

```
Loop:  beq    $9,$0,Label
       addu   $8,$8,$10
       addiu  $9,$9,-1
       j      Loop
Label: addi   $5, $5, 1
```

Given the address of the first instruction is 0x00400000,

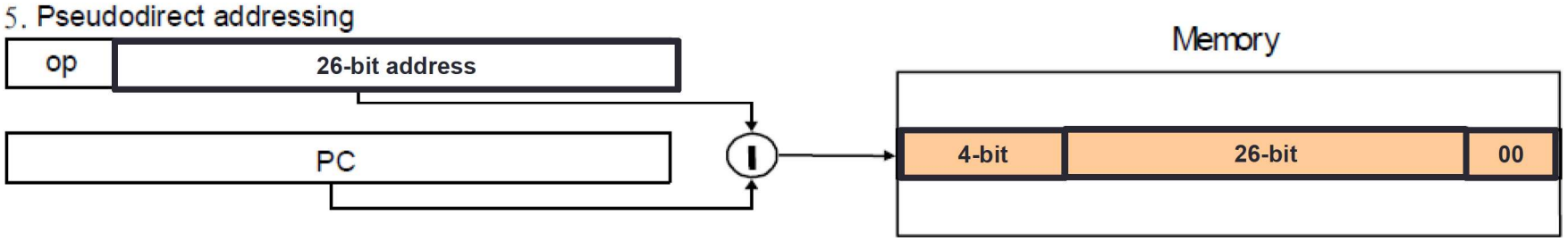
i) What is the machine code of `beq $9, $0, Label`?

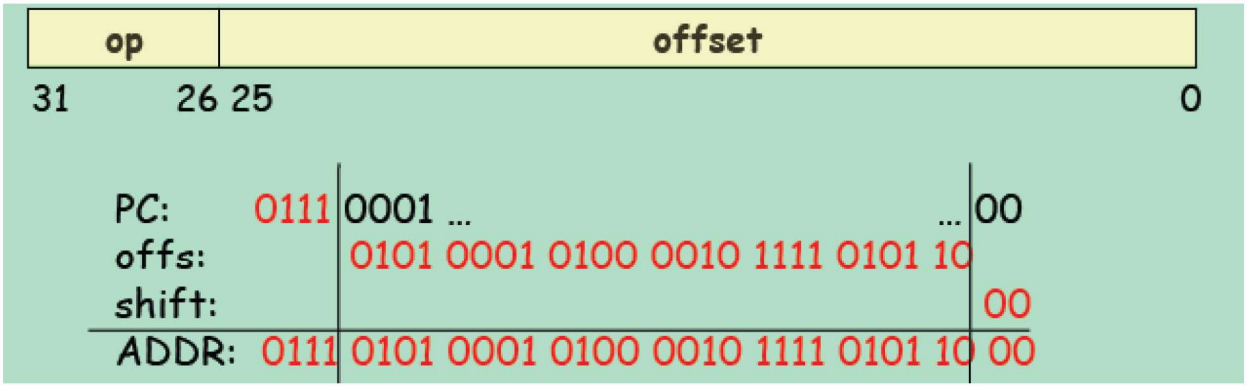
000100	01001	00000	0000000000000011

ii) What is the value of the updated `PC` if `$9` is equal to `$0`?

MIPS Addressing Modes

- (5) Pseudo-direct addressing
  - jump address is formed by shifting the 26-bit address within the instruction by 2 bits (i.e. concatenate 2 zero bits to the lower order end to yield 28 bits) and concatenated with the upper 4 bits of the PC to the higher order (left end) to give 32-bit jump address.
  - use in J-type instructions





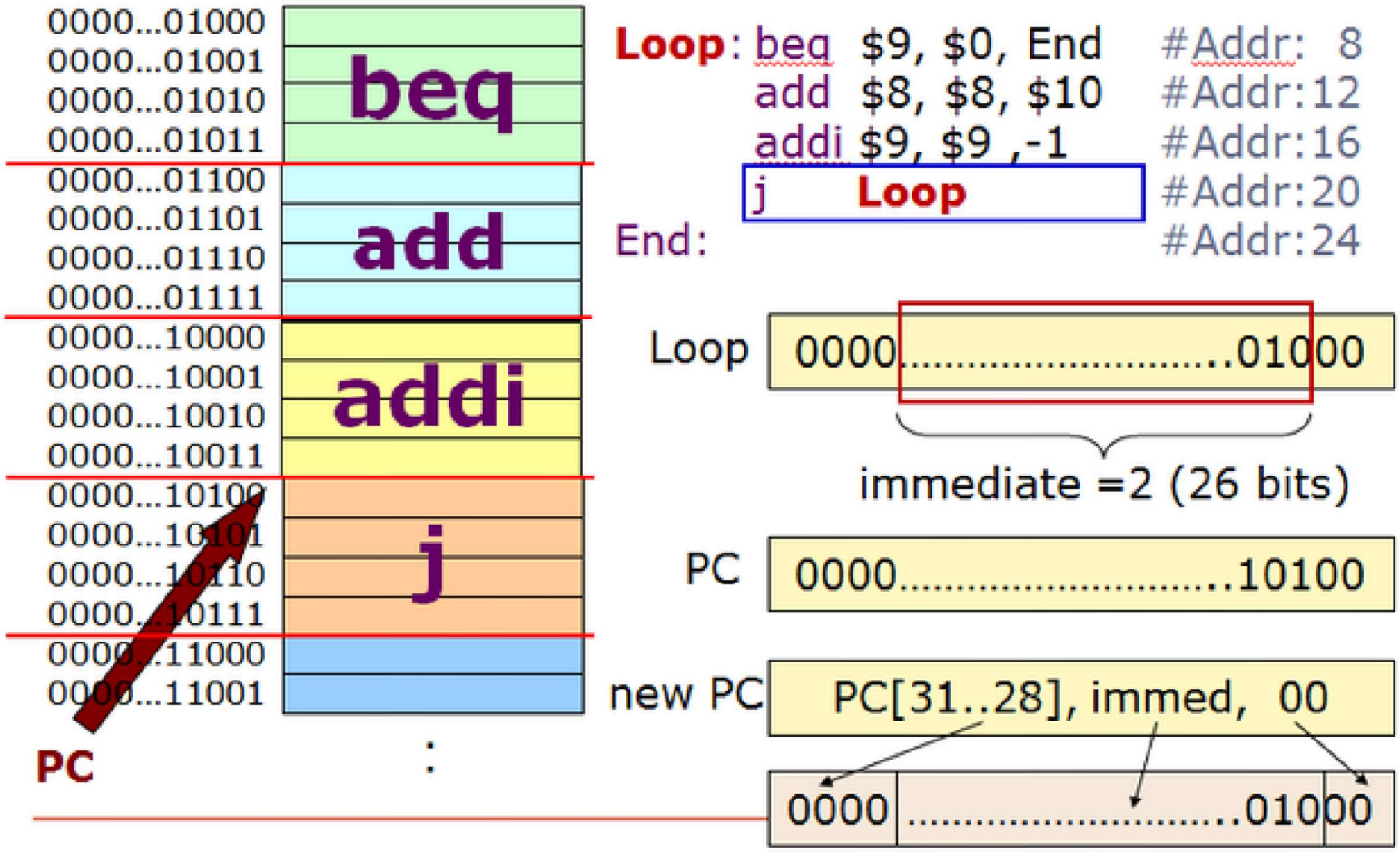
MIPS Addressing Modes

- (5) Pseudo-direct addressing (Example)

Given the instruction `j L1`, and the address of `L1` and current `PC` are 8_{10} and 20_{10} respectively. What is the value of the 26-bit jump address in the instruction?

MIPS Addressing Modes

- (5) Pseudo-direct addressing (Example)



Addressing Mode vs. Instruction Type

- Addressing mode is how an address (memory or register) is determined
- Instruction type is how the instruction is put together
- **`addi`**, **`beq`**, and **`lw`** are all I-types instructions, but
  - **`addi`** uses immediate addressing mode (and register)
  - **`beq`** uses pc-relative addressing (and register)
  - **`lw`** uses base addressing (and register)