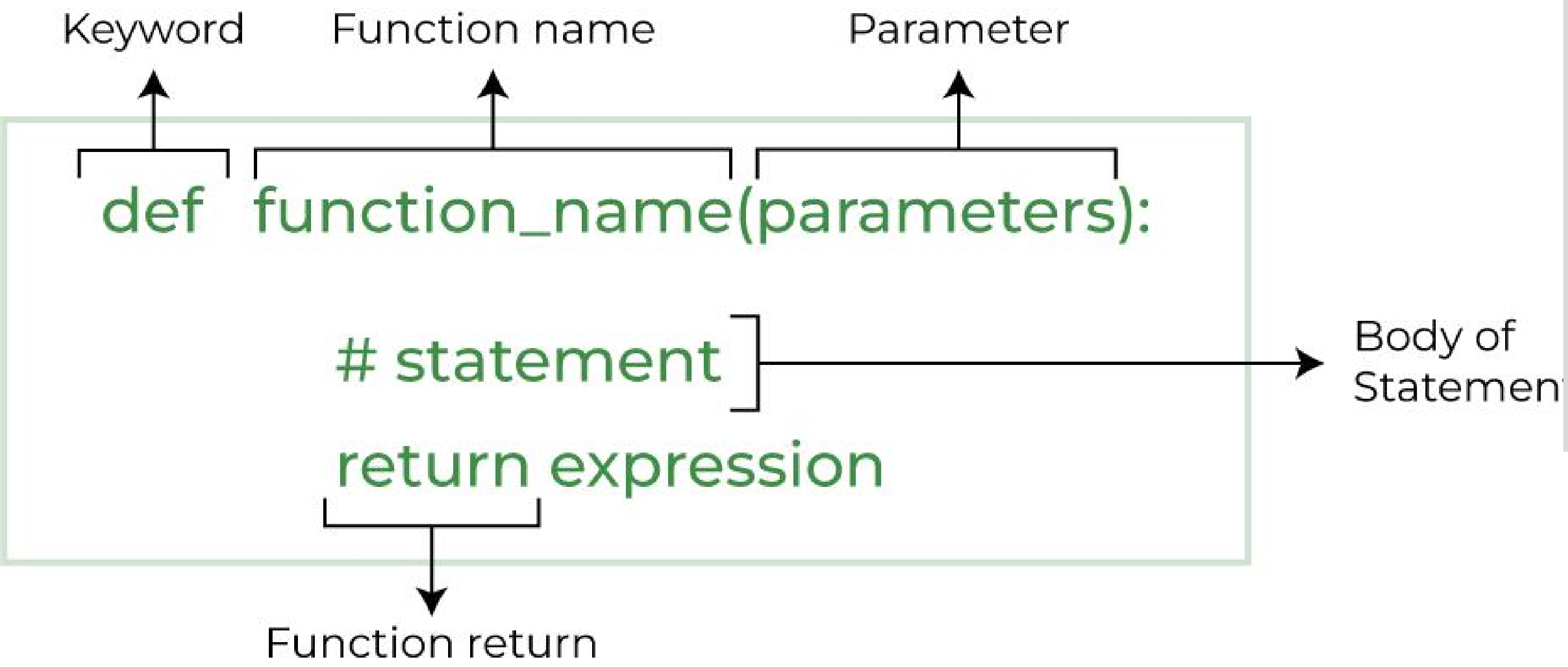
Procedures

Writing clean modular reusable code!!

Procedure (functions)



HLL Procedure

```
def sum_list(list1):
  sum = 0
      for i in list1:
         sum += i
      return sum
```

Section 2.8

```
# Main (calling) code here
myList =[5, 10, 15] #
function call
Sum = sum_list(myList)
```

Assembly Procedure Design

sum_list Plan

Arguments:
array address to the procedure $a0
array size to the procedure $a1

Return
the sum in $v0

procedure name is used as a label (remember that labels are just pointers to be mapped by the assembler

```
sum_list: li $v0, 0      # Initialize the sum to 0
loop: beqz $a1, done    # If size is 0, exit the loop
      lw $t0, 0($a0)    # Load the current integer into $t0
      add $v0, $v0, $t0 # Add the current integer to the sum
      addi $a0, $a0, 4  # Move to the next integer
      subi $a1, $a1, 1  # Decrement the size
      j loop            # Repeat the loop
done: jr $ra            # Return to the calling function
```

MIPS procedures end with jr instruction

How to call?

MIPS Procedure Call and Return

Procedure call: jump and link (**jal**)

jal sum_list

- Instructs the processor to branch to the instruction at sum_list label with a return status

$PC ← address of the first instruction in the procedure [jump]

$ra ← stores address of the next instruction after ***jal*** [link]

Procedure return: jump register (**jr**)

***jr* $ra**

$PC ← $ra Unconditional jump to the next instruction in the calling code

*Execute the instruction after **jal***

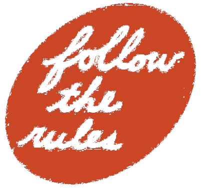
```
main: ……..
……..
……..
jal sum_List
Back: AND ………
………

exit: syscall

sum_List:  …..
……
……
jr    $ra
```

Why can't we just use
J sum_List
J back ← at the end of the procedure

MIPS Register Rules for Procedures



$a0 – $a3: ***arguments registers*** for passing parameters (reg's 4 – 7)

$v0, $v1: registers for **result values** (reg's 2 and 3)

Registers Rules

- $t0 – $t9: temporaries can be overwritten by callee >> must be saved by caller if needed!

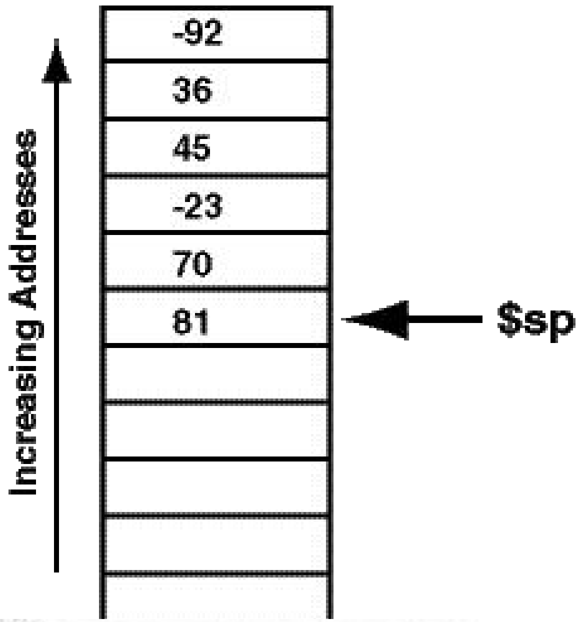
- $s0 – $s7: saved registers can **NOT** be overwritten by callee >> Must be saved/restored by callee

How to save and restore data?

Use the Stack

Stack

- A ***last-in-first-out (LIFO)*** queue for storing register content
  - Stack pointer ($SP) points to the most recent allocated address in stack
  - MIPS stack is managed ***manually***
  - The stack grows in a decreasing address direction




```
addi $sp, $sp, -4
sw   $s0, 0($sp)
```

Save s0 on stack before using it in the procedure

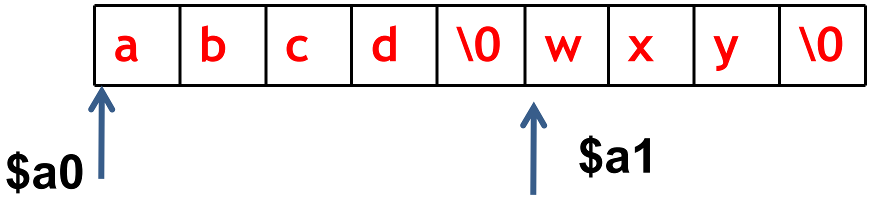
```
lw   $s0, 0($sp)
addi $sp, $sp, 4
```

Restore s0 before exiting the procedure

String Copy Procedure

Write MIPS procedure that copies a string to another string

- Addresses of strings in $a0, $a1 - Null-terminated string
- i in $s0



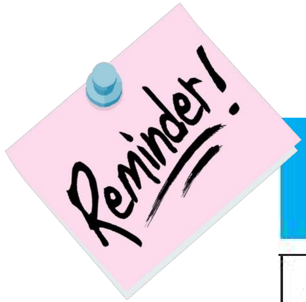
- **MIPS code:**

```
strcpy:
      addi $sp, $sp, -4              # adjust stack for 1 item
      sw   $s0, 0($sp)               # save $s0
      add  $s0, $zero, $zero         # i = 0
NEXT: add  $t1, $s0, $a1             # addr of y[i] in $t1
      lbu  $t2, 0($t1)               # $t2 = y[i]
      add  $t3, $s0, $a0             # addr of x[i] in $t3
      sb   $t2, 0($t3)               # x[i] = y[i]
      beq  $t2, $zero, ExitLoop      # exit loop if y[i] == 0
      addi $s0, $s0, 1               # i = i + 1
      j    NEXT                      # next iteration of loop
ExitLoop: lw  $s0, 0($sp)            # restore saved $s0
      addi $sp, $sp, 4               # pop 1 item from stack
      jr   $ra                       # and return
```

Leaf Procedure: A procedure that does NOT call another procedure (i.e., do the job and return to caller - do not use jal)

Non-Leaf Procedures

- Procedures that call other procedures (including recursive calls)
- Every non-leaf procedure **<u>should</u>**
  - **<u>save</u>** the return address register (every ***jal*** will change \$ra)
  - save any arguments and temporaries needed after the call
  - Restore saved from the stack after the call



Preserved	Not preserved
Saved registers: `$s0–$s7`	Temporary registers: `$t0–$t9`
Stack pointer register: `$sp`	Argument registers: `$a0–$a3`
Return address register: `$ra`	Return value registers: `$v0–$v1`
Stack above the stack pointer	Stack below the stack pointer

Non-Leaf Procedure Example

Write a procedure that calculate factorial n in a recursive fashion.

- Argument n in $a0
- Result in $v0

- MIPS code:

```
fact:
    addi $sp, $sp, -8      # adjust stack for 2 items
    sw   $ra, 4($sp)       # save return address
    sw   $a0, 0($sp)       # save argument
    slti $t0, $a0, 1       # test for n < 1
    beq  $t0, $zero, L1    # if so, result is 1
    addi $v0, $zero, 1     #  pop 2 items from stack
    addi $sp, $sp, 8       #and return
    jr   $ra
L1: addi $a0, $a0, -1      # else decrement n
    jal  fact              # recursive call
    lw   $a0, 0($sp)       # restore original n
    lw   $ra, 4($sp)       #   and return address
    addi $sp, $sp, 8       # pop 2 items from stack
    mul  $v0, $a0, $v0     # multiply to get result
    jr   $ra               # and return
```

What if I need to

- pass more than four arguments (\$a0 - \$a3) to the procedure?
- receive more values from the procedure?

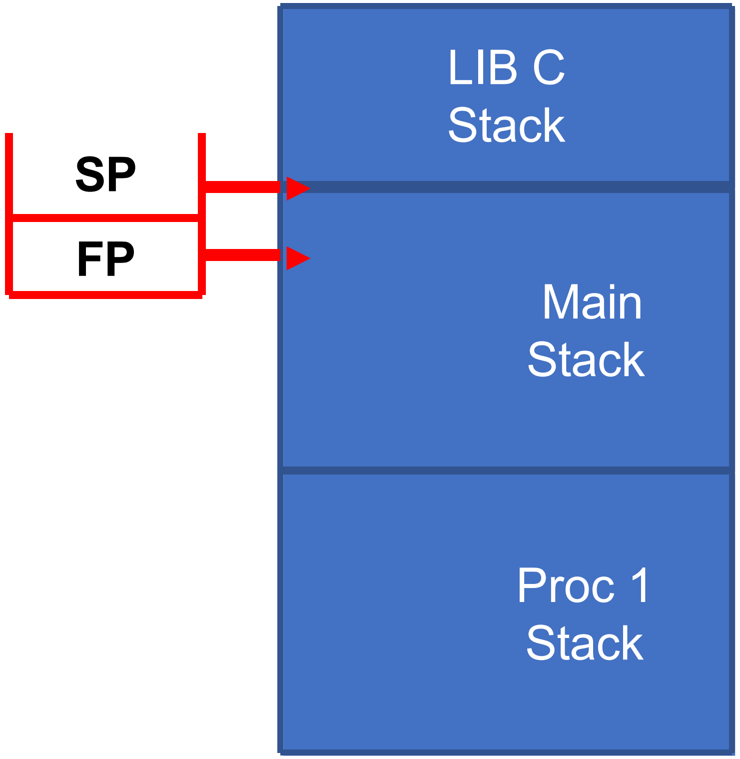
[Read ME]

OPTIONAL

What if I need to pass more than four arguments to the procedure? receive more values from the procedure?

Stack Frame

- A stack frame is created to support procedure calls
  - Libc → main(...) → proc1(...) → proc2(...)
- ***Before the execution of every procedure, part of the stack is populated by procedure-specific information***
- *The exact contents and layout of the stack vary by processor architecture and function call convention*
- *Stack frames are manged using stack pointer (SP) and frame pointer (FP) registers*



OPTIONAL

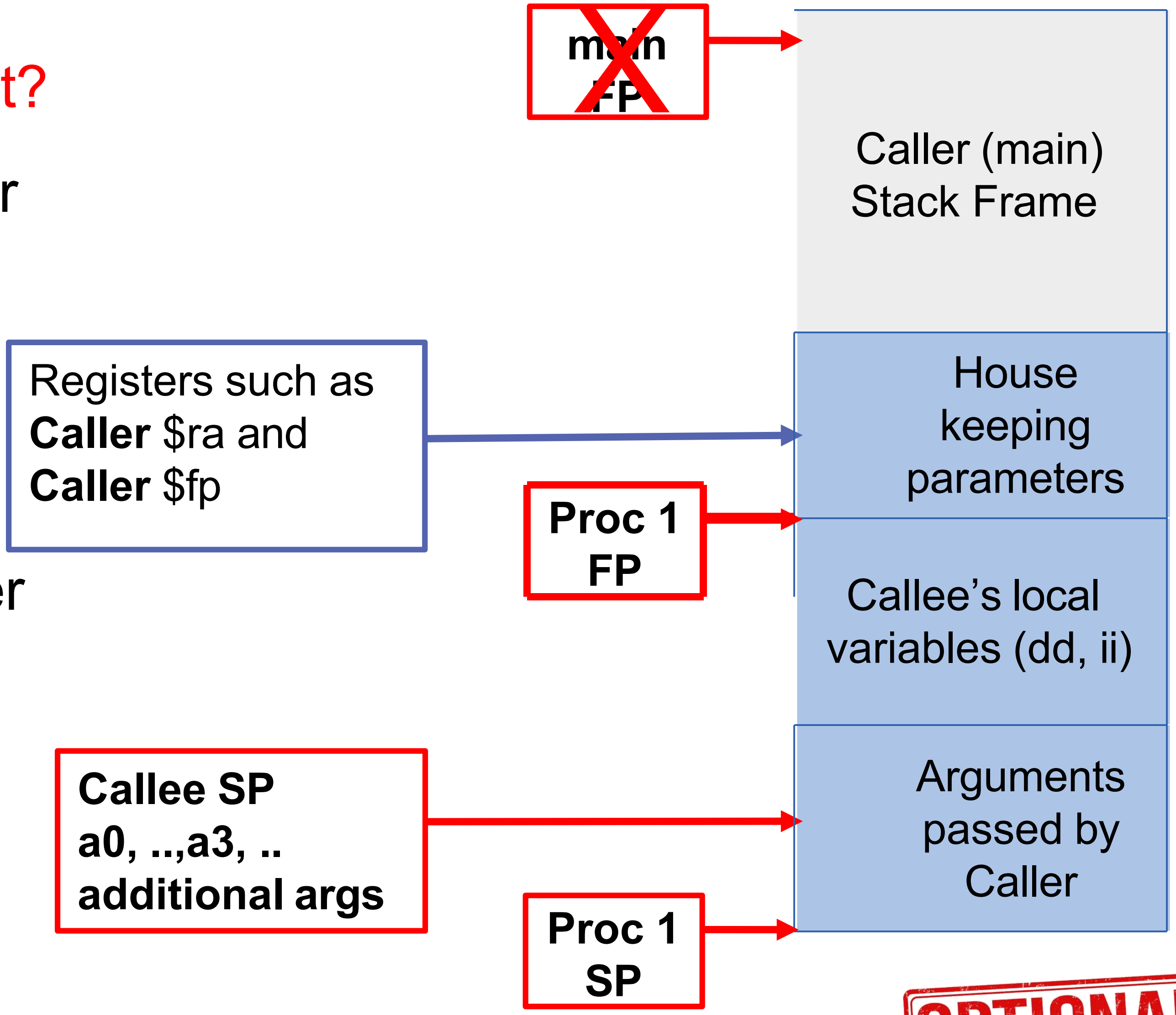
```
Int main(int )
{ update(a,b,….z);
}
double update(a,...z)
{ double dd;
int ii;
return a*dd* ...*z }
```

https://manybutfinite.com/post/journey-to-the-stack/

Procedure 1 Stack frame

- **Who** creates the stack frame and fill it?
  - Compiler or assembly programmer
- **How** is the stack frame populated?
  - Using assembly instructions
  - sw, sh, sb for storing data
  - add, sub for adjusting stack pointer

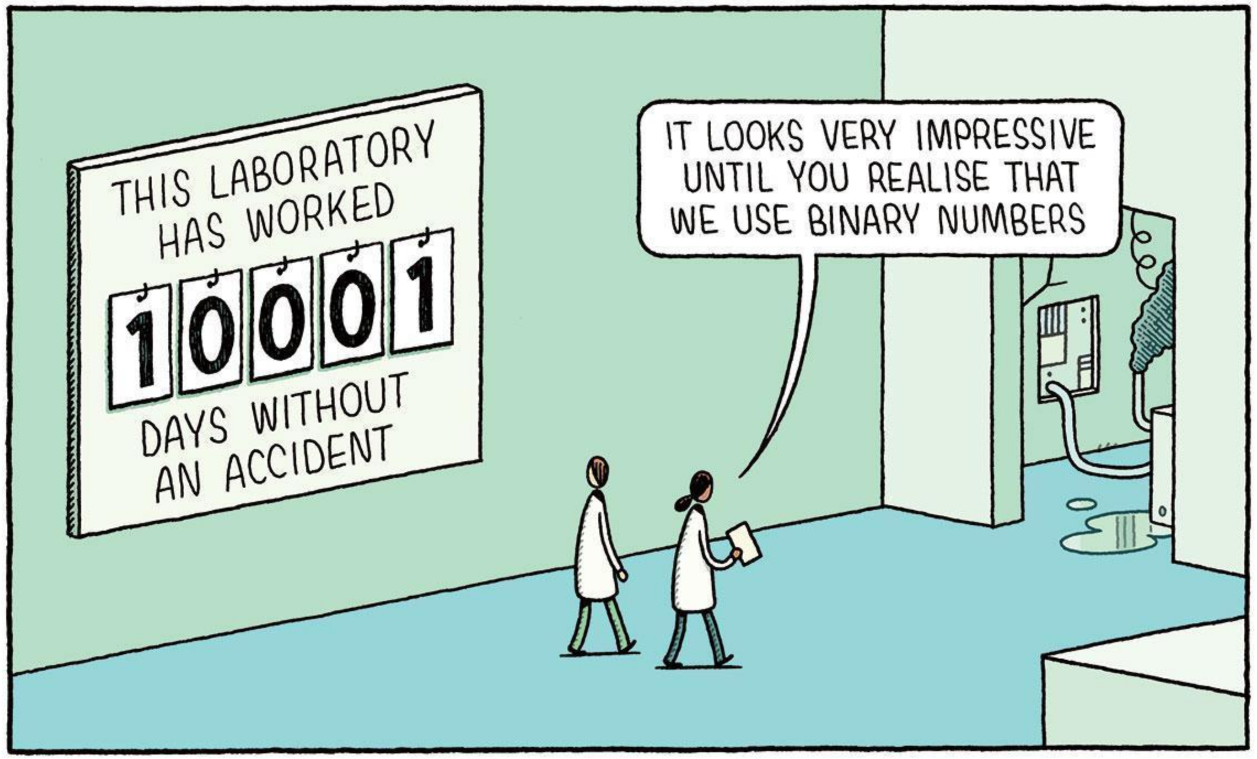
Remember: Each call comes with some overhead for creating a stack frame. That is the cost we have to pay for modularity!



OPTIONAL

Read ME

MIPS Instructions Encoding (Assembly → Binary)



TOM GAULD for NEW SCIENTIST

MIPS instructions have a fixed size of 32 bits

The layout of the 32 bits is defined as the Instruction format

Basic MIPS instructions have three key instruction formats:

1) R- format
(ALU, …)

Op code	rs	rt	rd	shamt	funct
31:26	25:21	20:16	15:11	10:6	5:0

2) I-Format
(lw, sw, addi)

Op code	rs	rt	address
31:26	25:21	20:16	15:0

3) J-Format
(j)

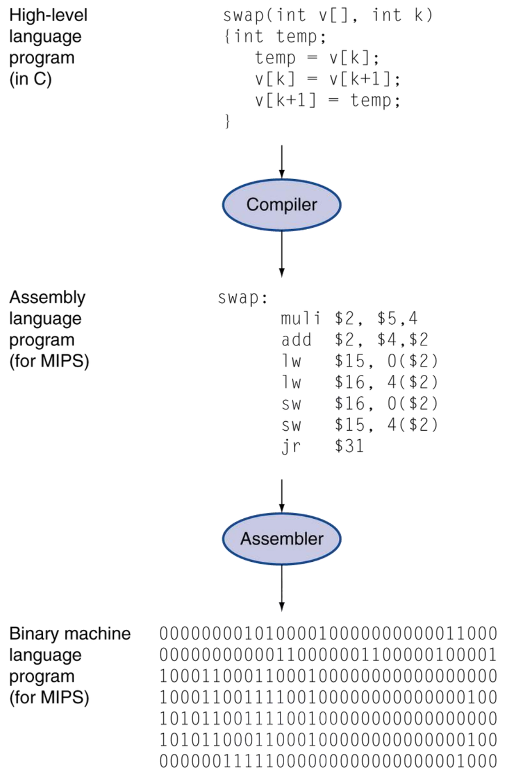
Op code	
31:26	25:0

MIPS R-format Instructions

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Fields for **Register instruction format**
  - op: operation code (opcode)
  - rs: first source register number
  - rt: second source register number
  - rd: destination register number
  - shamt: shift amount (00000 for now)
  - funct: function code (extends opcode)
- ALU instructions, others later

Remember that MIPS has 32 registers (32 = 2^5) → 5 bits are needed to identify every register in the register file



R-format Example

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

***add** $t0, $s1, $s2*

special	$s1	$s2	$t0	0	add	
0	17	18	8	0	32	← Decimal
000000	10001	10010	01000	00000	100000	← Binary

$$00000010001100100100000000100000_2 = 02324020_{16}$$

MIPS I-format Instructions

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- Immediate arithmetic, load/store, branch instructions
  - rs: [source] register
  - rt: [target] register
  - Constant: -2^{15} to $+2^{15}-1$
- Example: *lw $t0, 1200($t1)*

35	9	8	1200

Signed Extension (revisited)

- Signed extension is also used for extending
  - immediate values (e.g., addi, ...)
  - OFFSET (16 bits) of lw, sw, …

Immediate value is limited by 16 bits.
How does MIPS deal with large numbers?

Supporting Large Constants

Sections 2.10

- Most constants are small
  - 16-bit immediate is sufficient (***make common case fast***)

- For the occasional 32-bit constant (***large constant***) [2 steps]

*1 **lui** $at, constant*

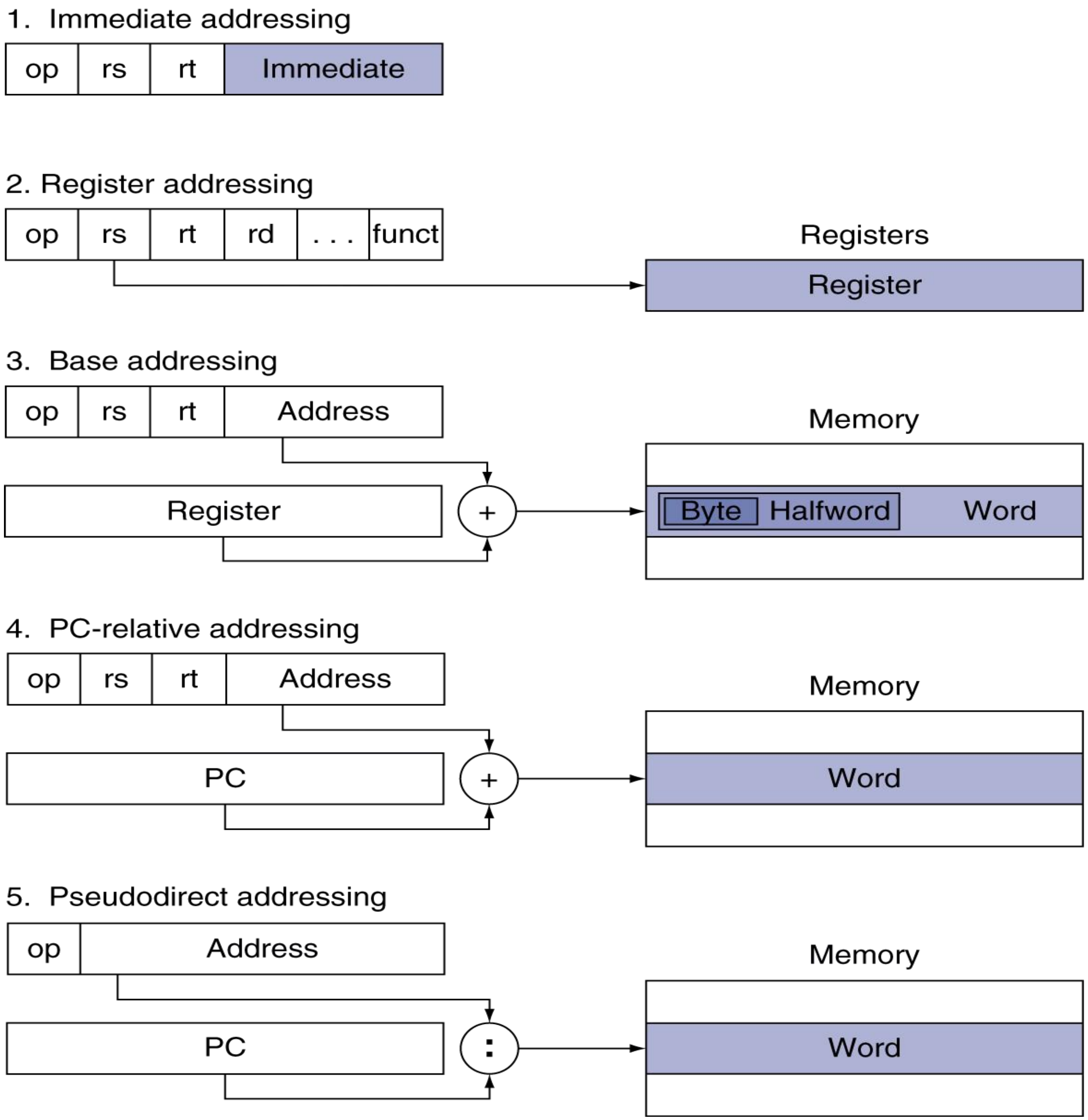
- Copies Most significant 16-bit constant to left 16 bits of ***$at***
- Clears right 16 bits of $at to 0
- $at (register #1): **assembler temporary**

2- ori lower half

Example: *li $t0 , 0x 007D 0900*

lui $at, 0x7d	0000 0000 0111 1101	0000 0000 0000 0000
ori $t0, $at, 0x900	0000 0000 0111 1101	0000 1001 0000 0000

MIPS Addressing Mode Summary



Addressing modes refers to the way in which the operand of an instruction is specified.

Give me an example instruction for every address mode

Remarks on MIPS ISA Design

THE BIG PICTURE

- The design of instruction set requires a ***delicate balance*** among
  - the number of instructions needed to execute a program,
  - the number of clock cycles needed by an instruction, and
  - the speed of the clock
- MIPS achieves this balance by following some design principals

1) **Make the common case fast**
2) Simplicity favours regularity
3) Smaller is faster

Tell me examples for these principals in MIPS design

MIPS ISA Design Principals

- ***Design Principle 1: Smaller is faster***
  - Desire to maintain fast execution time
  - **Number of registers**. *More registers mandates longer identifier*
  - **Instruction size.** *one word instructions enables fetching the instruction in one step*
- ***Design Principle 2: Simplicity favors regularity***
  - Regularity makes implementation simpler → higher performance at lower cost
  - Instruction format layout is similar → simplifies the HW implementation
- ***Design Principle 3: Make the common case fast (design for common case)***
  - Small constants are common (small immediate values)
  - Small loops are more common (small immediate values)
  - Immediate operand avoids a load instruction (addi, ...)