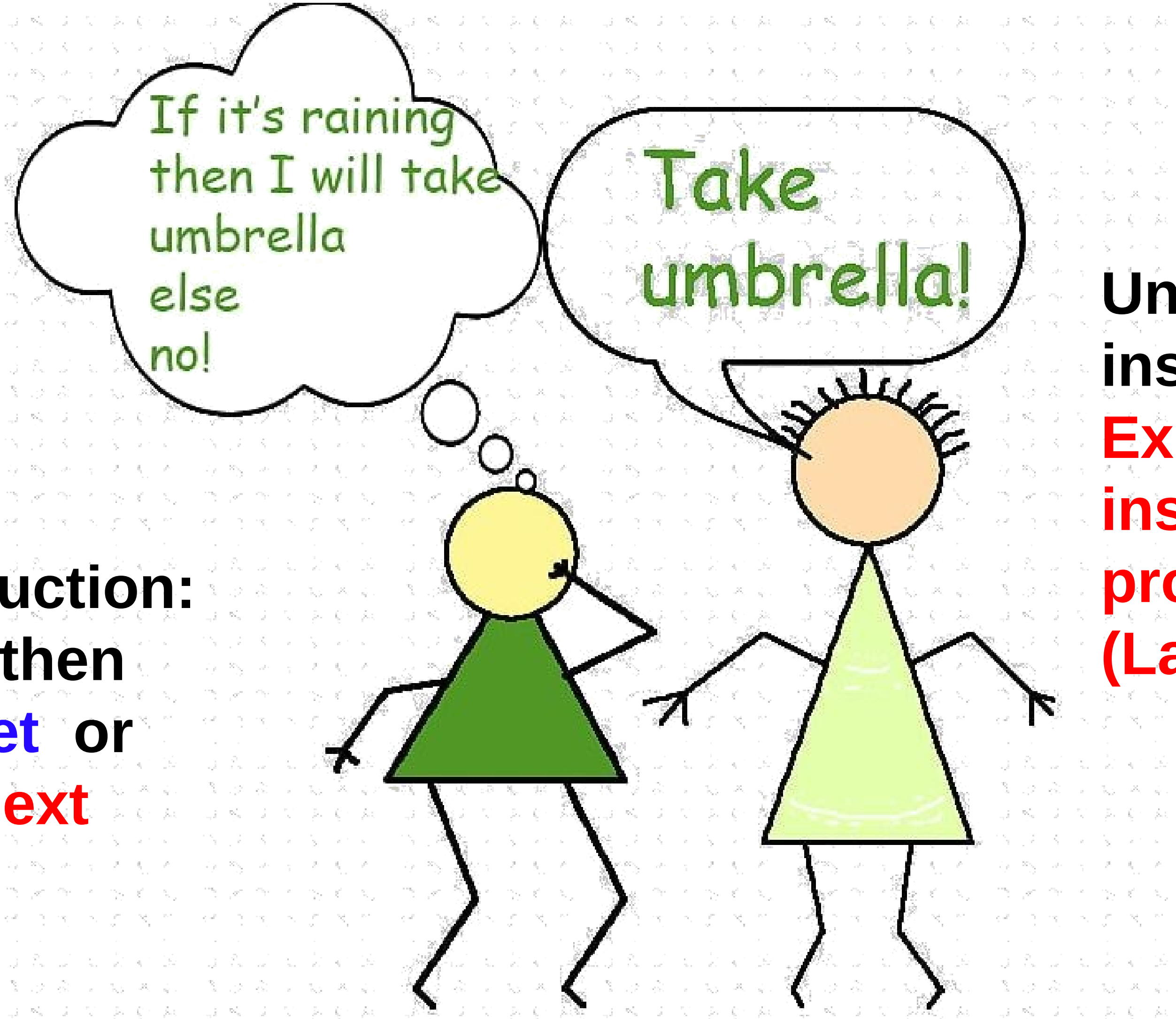
# Branching (flow control)

# Branching



**Conditional instruction:**
**Compare values then**
**Jump to the offset or**
**continue to the next instruction**

**Unconditional instruction**
**Execute the instruction at provided address (Label)**

- **GOTO**
- **Function calls**

Sections 2.7

# Branch instructions

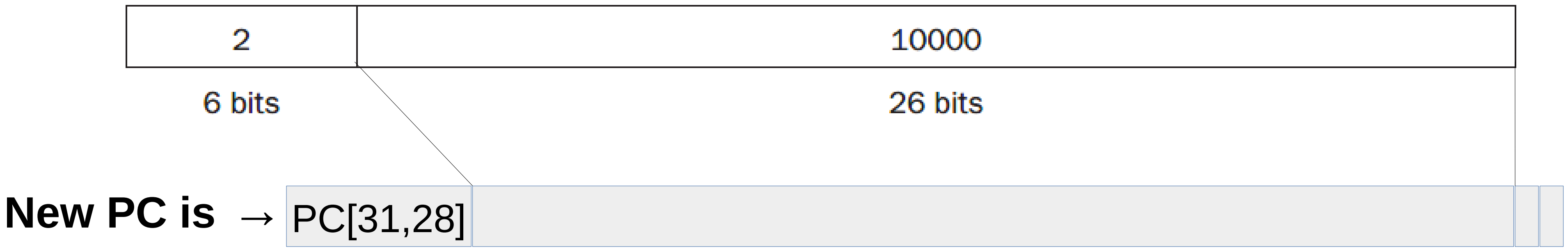
Branch to a labelled instruction: ***conditional*** **&** ***unconditional***

***Unconditional*** branch instruction

- ***j*** Label # jump to instruction at Label:

j instruction enables performing **FAR** jumps in the instruction sequence.
26 bits → ±32M instructions
→ ± 128 Mbytes

## J-format Instruction



| 2 | 10000 |
|---|---|
| 6 bits | 26 bits |

**New PC is** → PC[31,28]

**Do NOT use jump for function calls, which have a different instruction**

# Conditional Branch

- ***beq*** rs, rt, L1 # if (rs == rt) branch to instruction labeled L1;
- ***bne*** rs, rt, L1 # if (rs != rt) branch to instruction labeled L1;

**Some code**

```
Beq  rs,rt, L1
  Code executed if not branching
J common
L1: code executed when branching

Common: Rest of code
```

| op | rs | rt | ***Instruction*** offset |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

Target address = PC + offset × 4 ***[Relative address]***

## Other Instructions: bgtz, bltz, bgez, blez

## Why not to use BLT?

too complicated to implement → would stretch the clock cycle time or would take extra clock cycles per instruction
***Solution:*** use two basic instructions to execute its logic :)

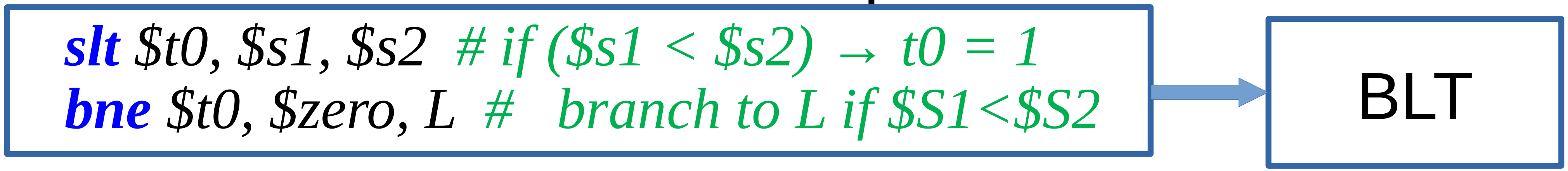
RISC

# More Conditional Operations

- Set result to 1 if a condition is true
  - Otherwise, set to 0

```
slt rd, rs, rt   # if (rs < rt) rd = 1; else rd = 0;
slti rt, rs, const #if (rs < constant) rt = 1; else rt = 0;
```

- Use in combination with beq, bne



```
slt $t0, $s1, $s2  # if ($s1 < $s2) → t0 = 1
bne $t0, $zero, L  #   branch to L if $S1<$S2
```

BLT

**When you write blt in the editor, it will be translated to these two instructions**

# Signed vs. Unsigned

- Signed comparison: ***slt***, ***slti***
- Unsigned comparison: ***sltu***, ***sltui***
- ***Example***
  - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001

***slt*** $t0, $s0, $s1 # signed

- –1 < +1 → $t0 = 1

***sltu*** $t0, $s0, $s1 # unsigned

- +4,294,967,295 > +1 → $t0 = 0

# MIPS *Conditional* Branching Instructions

| Instruction | RTL | Notes |
|---|---|---|
| *beq $rs, $rt, imm* | $\mathit{if}(R[\$rs] = R[\$rt])\quad PC \leftarrow PC + 4 + \mathit{SignExt}_{18b}(\{imm, 00\})$ | *(I-format)* |
| *bne $rs, $rt, imm* | $\mathit{if}(R[\$rs] \mathrel{!=} R[\$rt])\quad PC \leftarrow PC + 4 + \mathit{SignExt}_{18b}(\{imm, 00\})$ | *(I-format)* |
| *blez $rs, imm* | $\mathit{if}(R[\$rs] <= 0)\quad PC \leftarrow PC + 4 + \mathit{SignExt}_{18b}(\{imm, 00\})$ | *Signed comparison (I-format)* |
| *bgtz $rs, imm* | $\mathit{if}(R[\$rs] > 0)\quad PC \leftarrow PC + 4 + \mathit{SignExt}_{18b}(\{imm, 00\})$ | *Signed comparison (I-format)* |
| *slt $rd, $rs, $rt* | $R[\$rd] \leftarrow R[\$rs] < R[\$rt]$ | *Signed comparison (R-format)* |
| *sltu $rd, $rs, $rt* | $R[\$rd] \leftarrow R[\$rs] < R[\$rt]$ | *Unsigned comparison (R-format)* |

# If Statements

- High-level code:

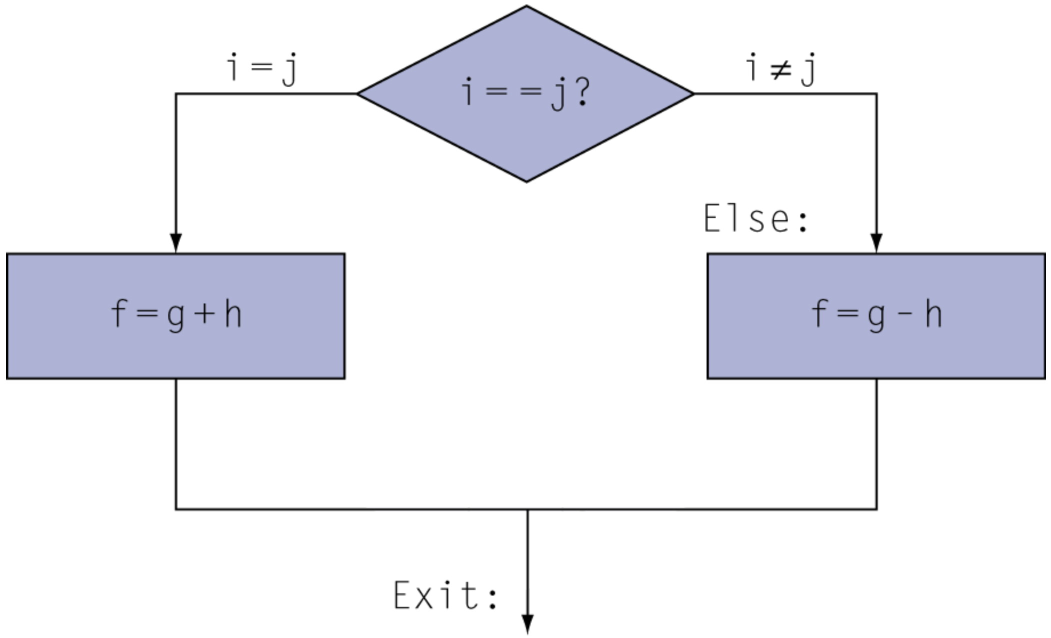
  if (i==j) f = g+h;
  else f = g-h;

  - f, g, … in $s0, $s1, …

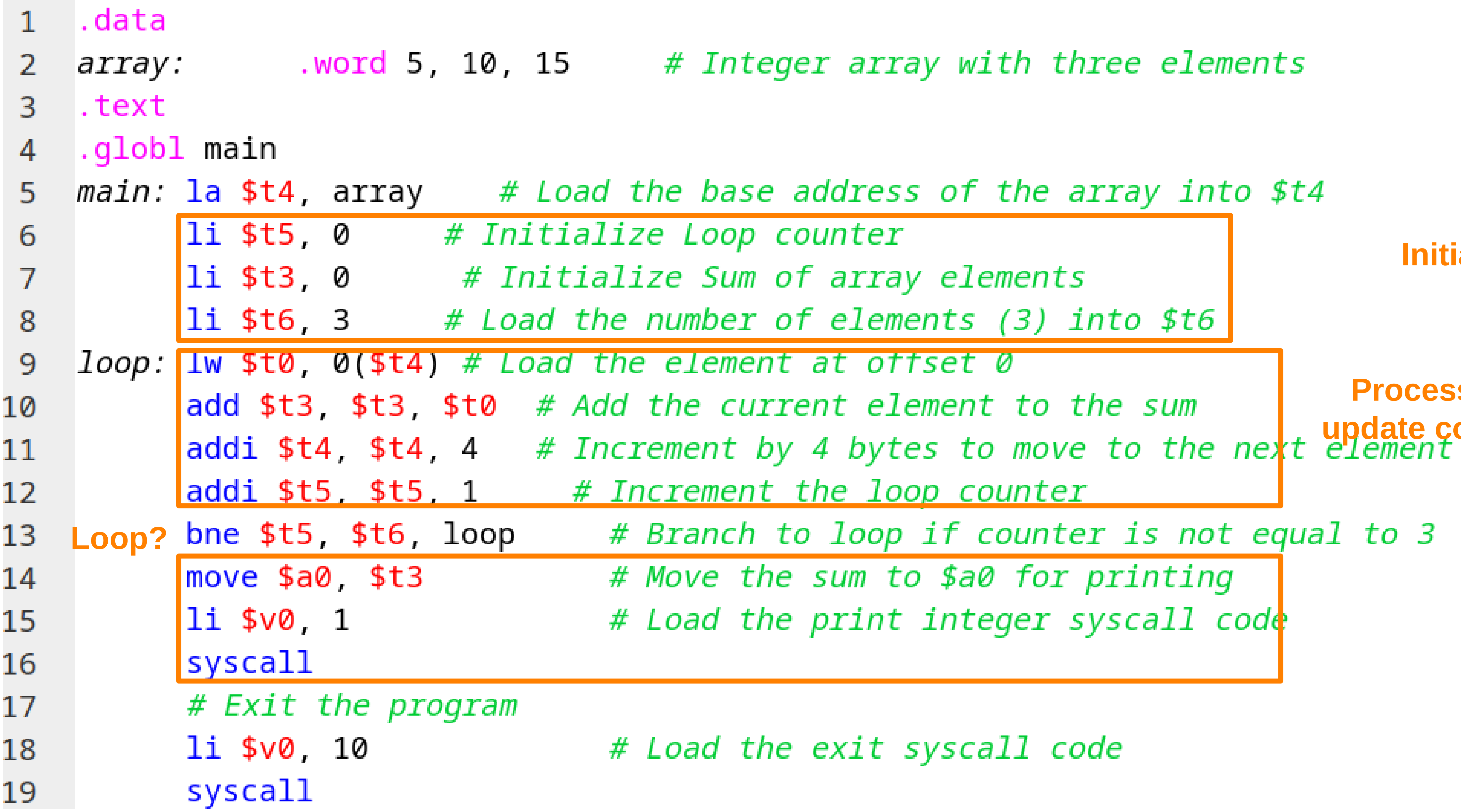
- Compiled MIPS code:

```
      bne $s3, $s4, Else
      add $s0, $s1, $s2
      j   Exit
Else: sub $s0, $s1, $s2
Exit:
```

Assembler calculates ***relative*** addresses of these labels



# Adding array elements using a loop



```
.data
array:      .word 5, 10, 15     # Integer array with three elements
.text
.globl main
main: la $t4, array     # Load the base address of the array into $t4
      li $t5, 0     # Initialize Loop counter
      li $t3, 0      # Initialize Sum of array elements
      li $t6, 3     # Load the number of elements (3) into $t6
loop: lw $t0, 0($t4) # Load the element at offset 0
      add $t3, $t3, $t0  # Add the current element to the sum
      addi $t4, $t4, 4   # Increment by 4 bytes to move to the next element
      addi $t5, $t5, 1     # Increment the loop counter
      bne $t5, $t6, loop     # Branch to loop if counter is not equal to 3
      move $a0, $t3          # Move the sum to $a0 for printing
      li $v0, 1              # Load the print integer syscall code
      syscall
      # Exit the program
      li $v0, 10             # Load the exit syscall code
      syscall
```

Initialize

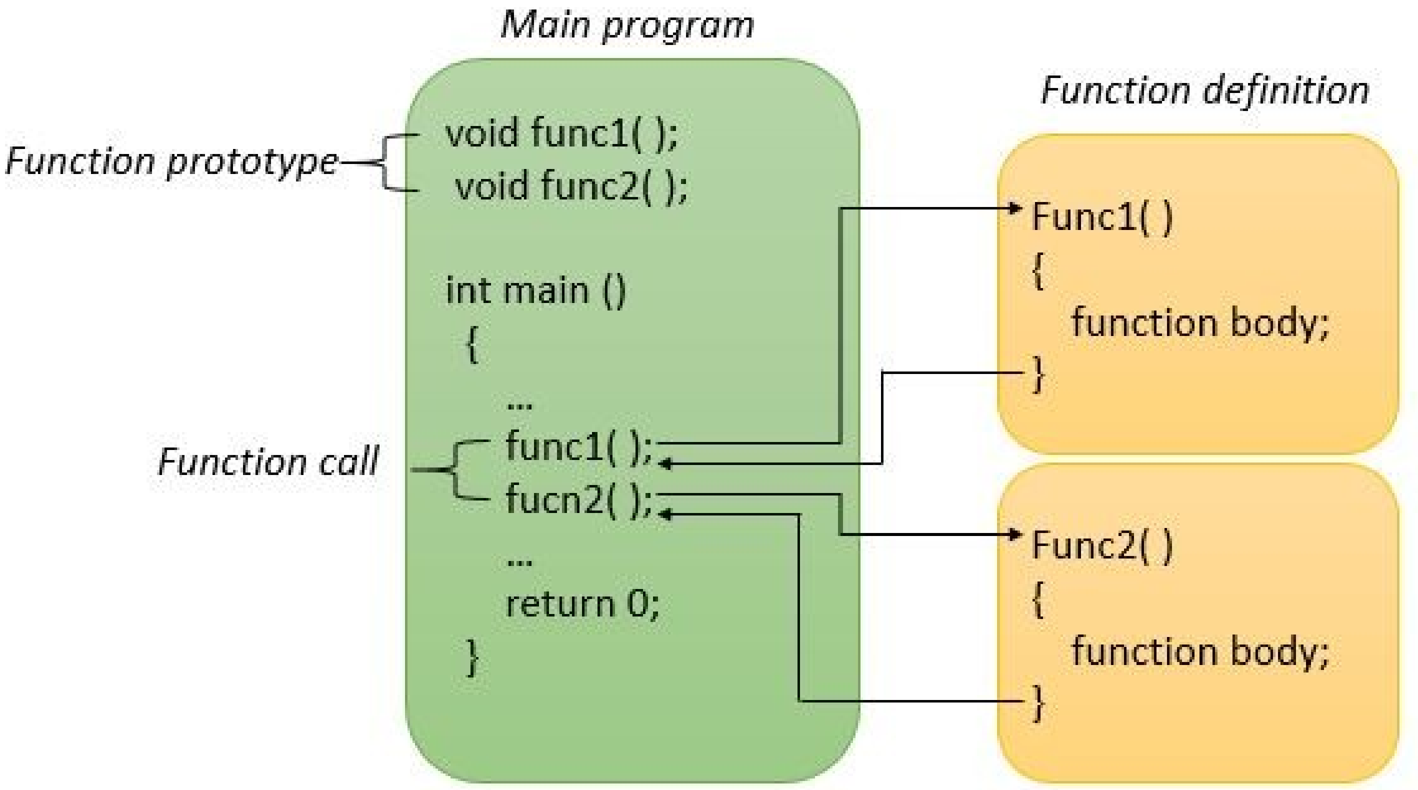
Process and update counter

Loop?

# Procedures

## Writing clean modular reusable code!!

# Procedure (functions) (aka def ….)



```
void func1( );
 void func2( );

int main ()
 {
   ...
   func1( );
   fucn2( );
   ...
   return 0;
 }
```

```
Func1( )
{
   function body;
}
```

```
Func2( )
{
   function body;
}
```

Procedure allows programmers to concentrate on a portion of a task at a time

**Takes**: argument(s)

**Returns**: result(s)

Sections 2.8

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

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

```c
int sum_list(int *list1, int size) {
  int sum = 0;
  for (int i = 0; i < size; i++) {
    sum += list1[i];
  }
  return sum;
}
```

# MIPS Procedure Instructions

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

*jal* sum_list **(J-Format)**

- Instructs the processor to execute the code at sum_list label

- **$PC** ← address of he first instruction in the procedure
- **$ra** ← address of the instruction after ***jal***

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

*jr* **$ra** #**R-Format**

- **$PC** ← **$ra**
  - Unconditional jump returns to the calling code

# Procedure Dynamics

```
main: ……..
……..
……..A1
jal sum_List   A2
AND ……..
     B3
……..

exit: syscall
```

```
sum_List:  …..
…….
…….B1
jr    $ra   B2
```

## How to call?

A1 Calling code puts **Arguments** in **$a0-$a3** registers

A2 *jal* ==> $ra ← $pc+4 & $pc ← ADDR[sum_List ]

## How to get the results?

B1 Procedure puts **return** value(s) in **$v0-$v1**

B2 *jr* $ra → moves $pc ← $ra

B3 after return, calling code reads the result form **$v0-$v1**

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

exit: syscall

sum_List:  …..
……
……
jr    $ra
```

BON
US

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

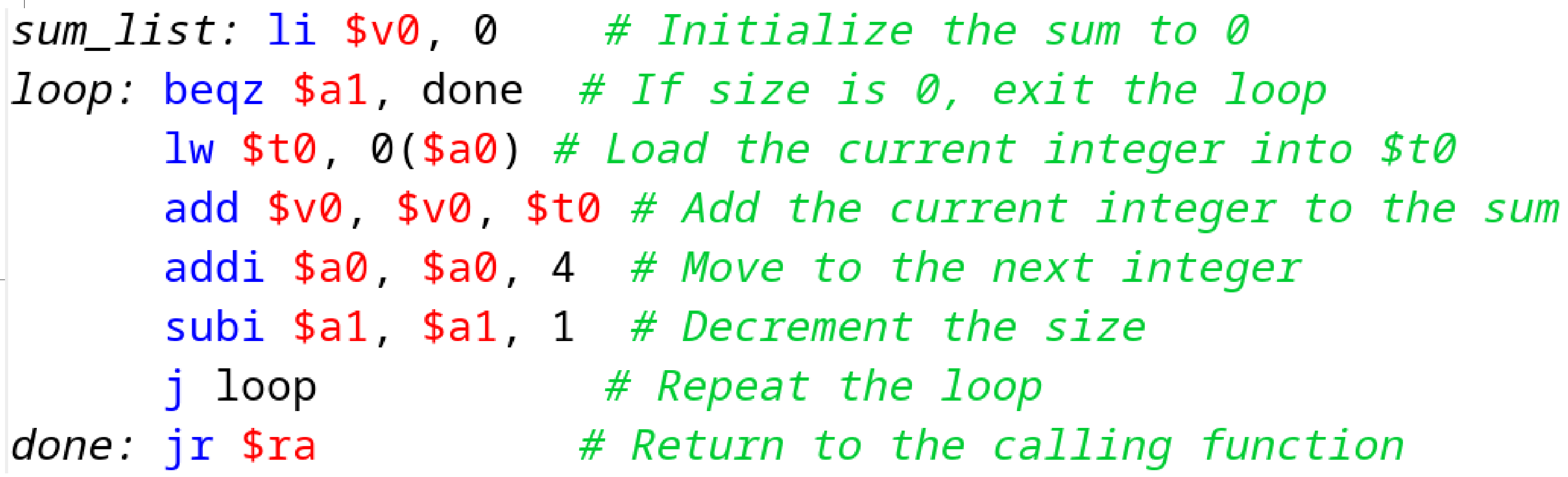
# Procedure Design

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

exit: syscall
```

```
sum_List:  …..
……
……
jr    $ra
```

Pass array address to the procedure **$a0**
Pass the array size to the procedure **$a1**
Returns the sum in **$v0**



```
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 Register Rules for Procedures

KNOW THE RULES !

$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

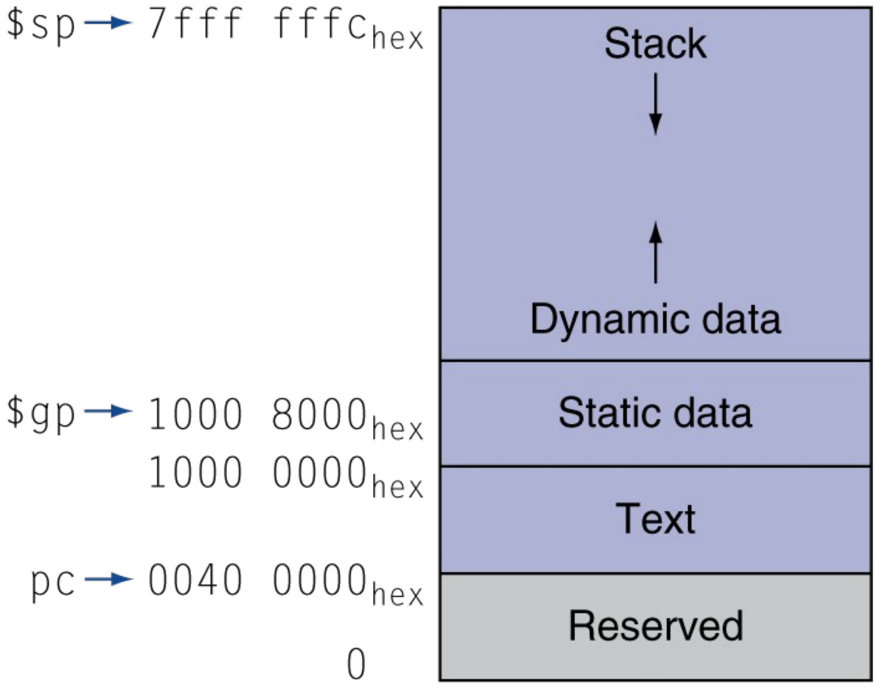
  - Can be overwritten by callee,
  - must be saved by caller if needed!
- $s0 – $s7: saved registers

  - Can **NOT** be overwritten by callee,
  - Must be saved/restored by callee

**How to save and restore data?**

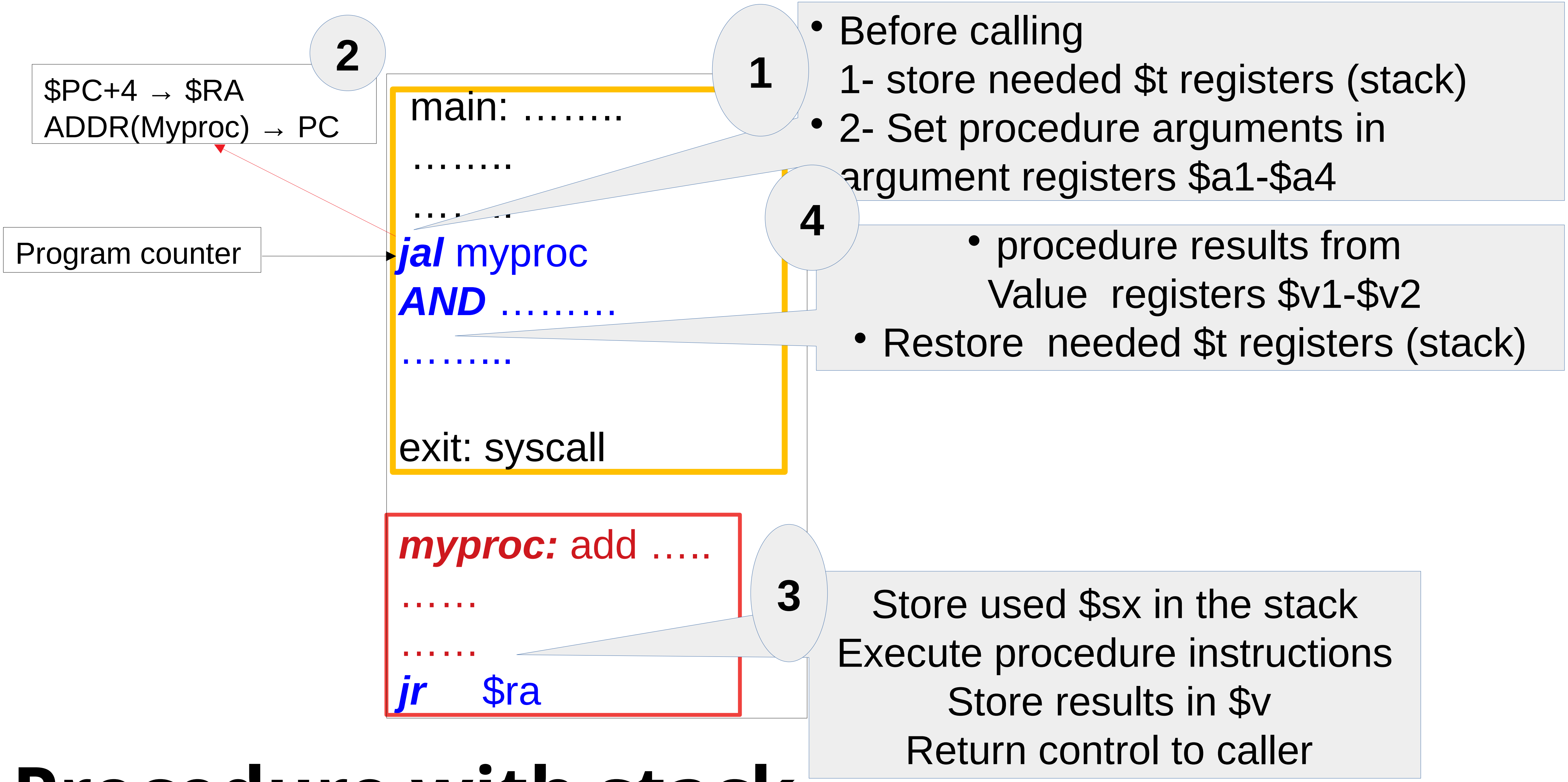
**Use the Stack**

# Memory Layout



- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp (global pointer register)
- Dynamic data: heap
  - E.g., malloc in C, new in Java

# Procedure with stack



2

$PC+4 → $RA
ADDR(Myproc) → PC

Program counter

```
main: ……..
……..
……..
jal myproc
AND ………
………

exit: syscall

myproc: add …..
……
……
jr     $ra
```

1

- Before calling
  1- store needed $t registers (stack)
- 2- Set procedure arguments in argument registers $a1-$a4

4

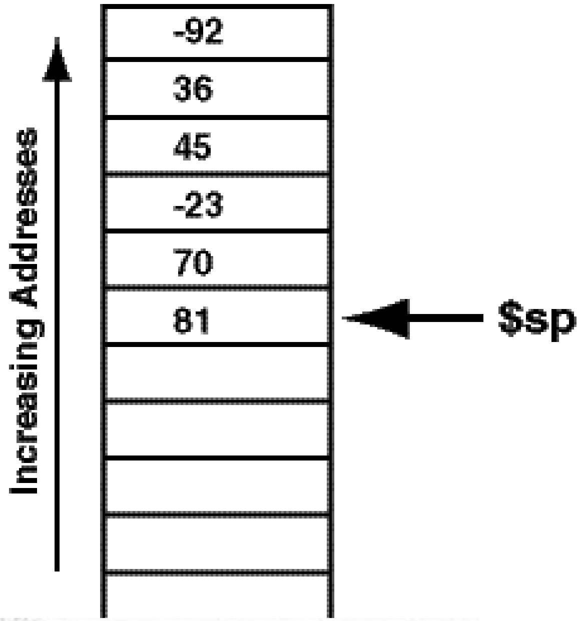
- procedure results from Value registers $v1-$v2
- Restore needed $t registers (stack)

3

Store used $sx in the stack
Execute procedure instructions
Store results in $v
Return control to caller

# 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***



Increasing Addresses

-92

36

45

-23

70

81

$sp

```
  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

Our computers are all down, so we're having to do everything manually ...

# Leaf Procedure and Stack Example

***Leaf procedure:*** A procedure that does not call another procedure (i.e., do the job and return to caller)

C code:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

- Arguments g, …, j in $a0, …, $a3
- Assume the procedure needs to use $s0
- Result in $v0

- MIPS code:

```
leaf_example:
  addi $sp, $sp, -4                  Save $s0
  sw   $s0, 0($sp)                   on stack
  add  $t0, $a0, $a1
  add  $t1, $a2, $a3                 Procedure
  sub  $s0, $t0, $t1                 body
  add  $v0, $s0, $zero               store Result in v0
  lw   $s0, 0($sp)
  addi $sp, $sp, 4                   Restore s0 from stack
  jr   $ra                           Return to caller
```

# Leaf Procedure and Stack Example

- Leaf procedure: A procedure that does not call another procedure

C code:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

- Arguments g, …, j in $a0, …, $a3
- Result in $v0

- MIPS code:

**necessary code**

```
 leaf_example:
add  $t0, $a0, $a1
add  $t1, $a2, $a3
sub  $v0, $t0, $t1
 jr   $ra
```

```
leaf_example:
  addi $sp, $sp, -4
  sw   $s0, 0($sp)
  add  $t0, $a0, $a1
  add  $t1, $a2, $a3
  sub  $s0, $t0, $t1
  add  $v0, $s0, $zero
  lw   $s0, 0($sp)
  addi $sp, $sp, 4
  jr   $ra
```

# String Copy Example (Leaf Procedure)

- C code:
  - Null-terminated string

```
void strcpy (char x[], char y[])
{ int i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

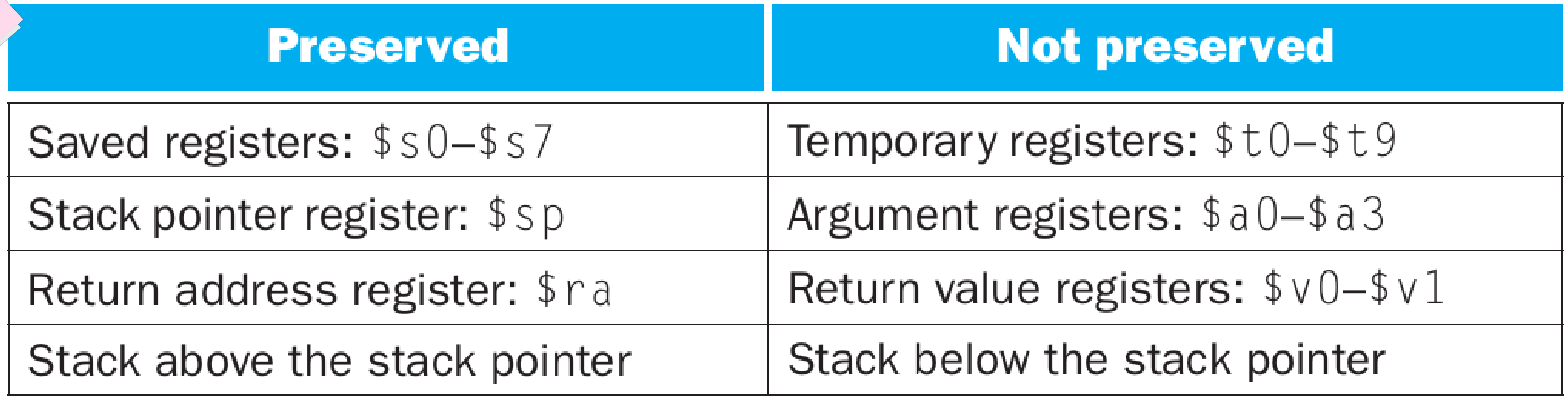
  - Addresses of x, y in $a0, $a1
  - i in $s0

- 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
```

# Non-Leaf Procedures

- Procedures that call other procedures (including recursive calls)
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore 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

- C code:

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

  - 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
    addi $v0, $zero, 1         # if so, result is 1
    addi $sp, $sp, 8     #  pop 2 items from stack
    jr   $ra                   #   and return
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**

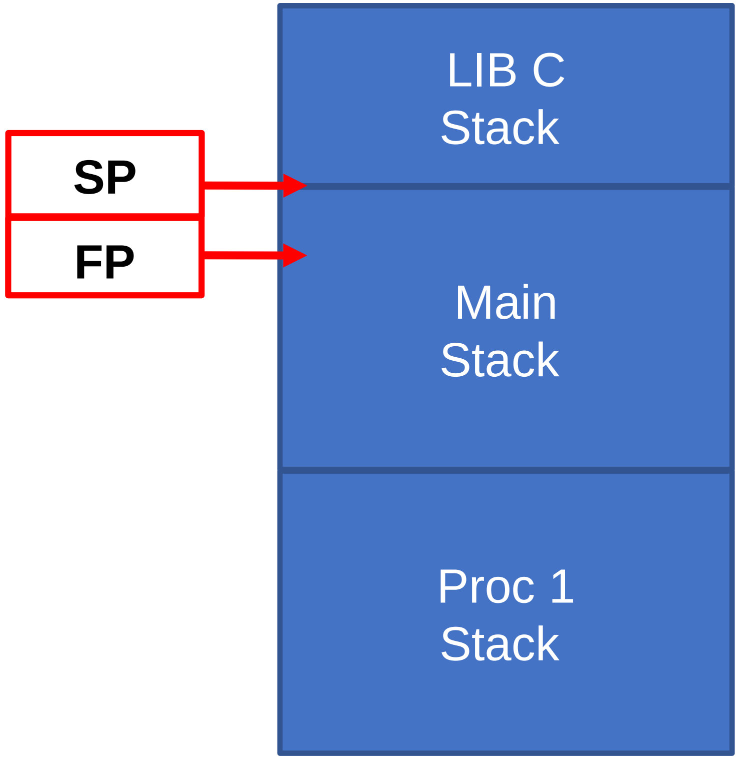
OPTIONAL

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

**Stack Frame**

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

- 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*

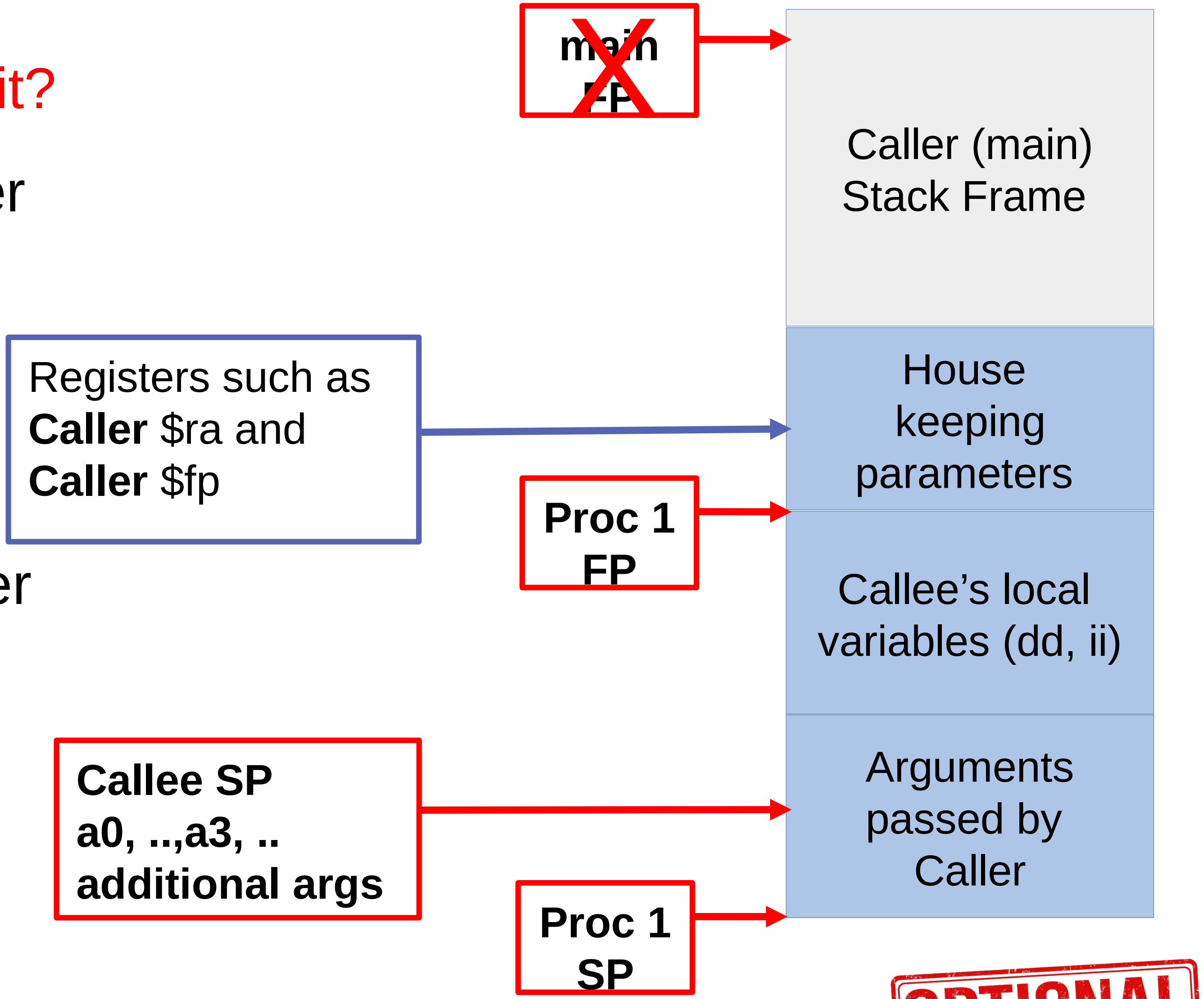


OPTIONAL

# 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!


~~main FP~~
Caller (main) Stack Frame
House keeping parameters
Proc 1 FP
Callee's local variables (dd, ii)
Arguments passed by Caller
Proc 1 SP

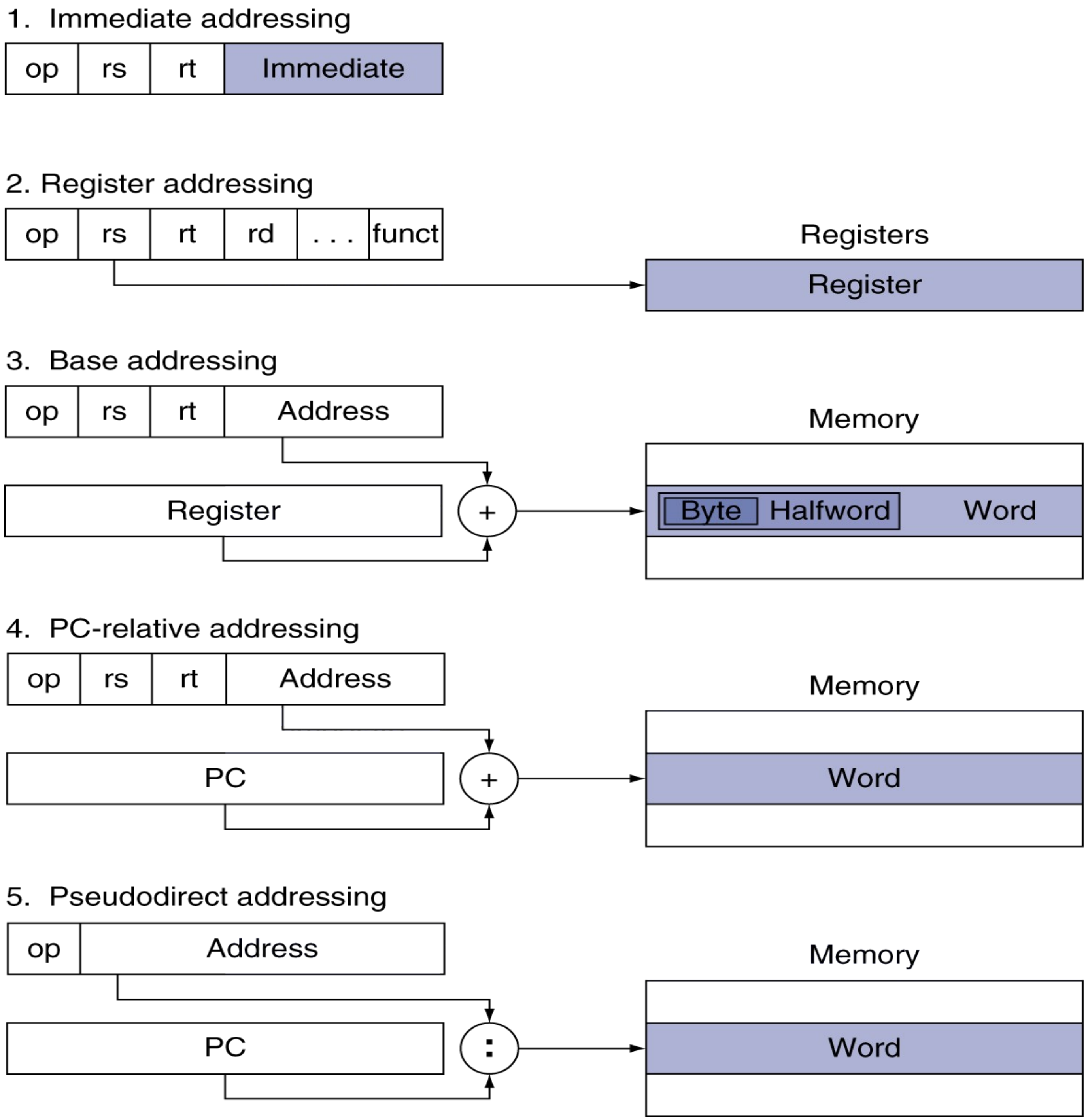
Registers such as **Caller** $ra and **Caller** $fp

**Callee SP a0, ..,a3, .. additional args**

OPTIONAL

**Read ME**

# 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 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***
  - Small constants are common (small immediate values)
  - Small loops are more common (small immediate values)
  - Immediate operand avoids a load instruction (addi, …)

# Other processors

OPTIONAL

# ARMv7 and MIPS Similarities

Sections 2.16-2.19
Optional

- ARMv7: the most popular embedded core
- Similar basic set of instructions to MIPS

| | ARMv7 | MIPS |
|---|---|---|
| Date announced | 1985 | 1985 |
| Instruction size | 32 bits | 32 bits |
| Address space | 32-bit flat | 32-bit flat |
| Data alignment | Aligned | Aligned |
| Data addressing modes | 9 | 3 |
| Registers | 15 × 32-bit | 31 × 32-bit |
| Input/output | Memory mapped | Memory mapped |

OPTIONAL

# Compare and Branch in ARMv7

- MIPS uses content of registers to evaluate conditional branches

- ARMv7 uses ***condition codes*** for result of an arithmetic/logical instruction
  - Negative, zero, carry, overflow
  - Compare instructions to set condition codes without keeping the result

- Each instruction can be conditional
  - Top 4 bits of instruction word: condition value
  - Can avoid branches over single instructions

OPTIONAL

# ARMv8 Instructions

- In moving to 64-bit, ARM did a complete overhaul
- ARM v8 resembles MIPS
  - Changes from v7:
    - No conditional execution field
    - Immediate field is 12-bit constant
    - PC is no longer a GPR
    - GPR set expanded to 32
    - Addressing modes work for all word sizes
    - Divide instruction
    - Branch if equal/branch if not equal instructions

OPTIONAL

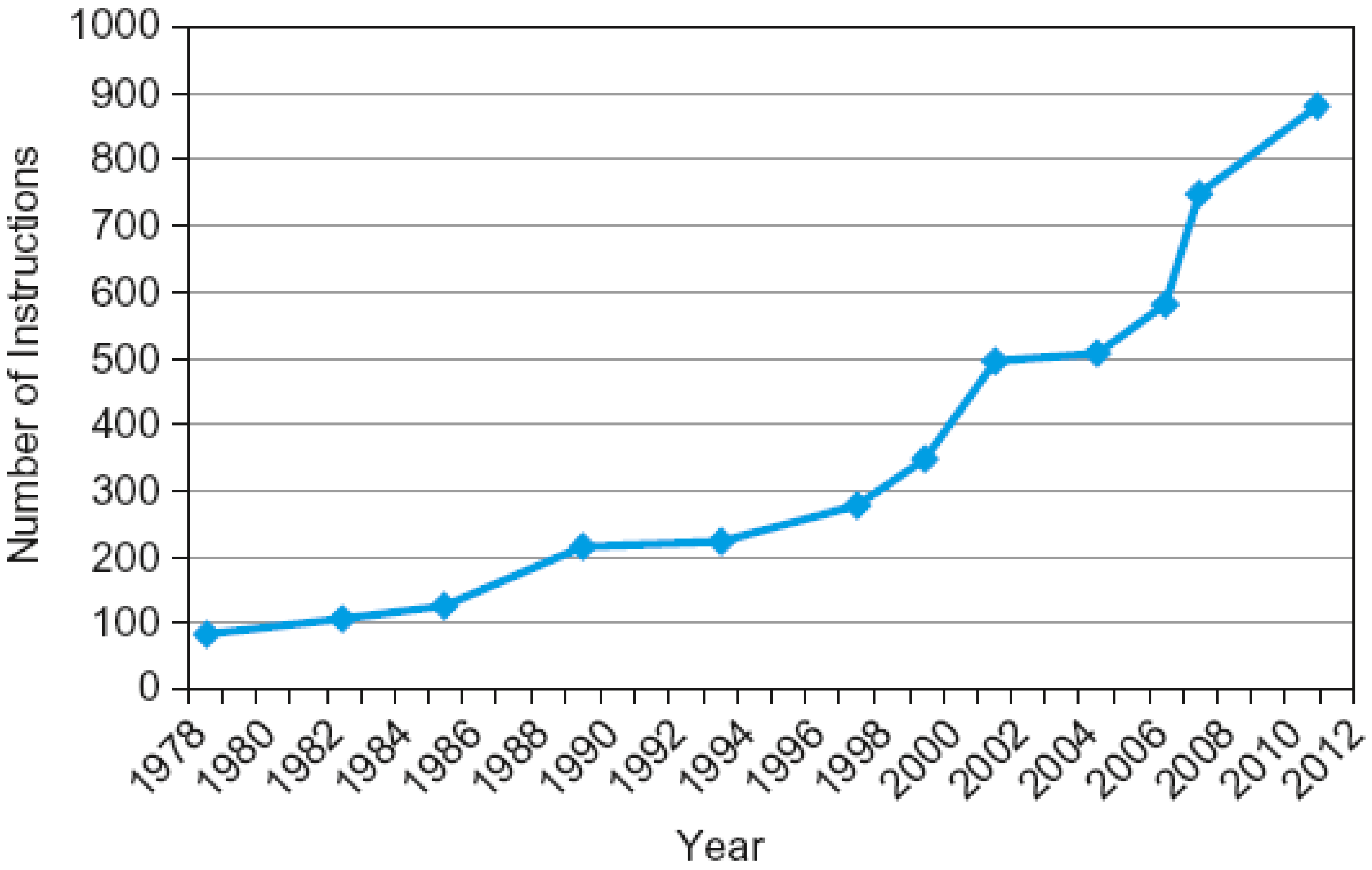
# The Intel x86 ISA Evolution

- Evolution with backward compatibility
  - 8080 (1974): 8-bit microprocessor
    - Accumulator, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080
    - Complex instruction set (CISC)
  - 8087 (1980): floating-point coprocessor
    - Adds FP instructions and register stack
  - 80286 (1982): 24-bit addresses
    - Segmented memory mapping and protection
  - 80386 (1985): 32-bit extension (now IA-32)
    - Additional addressing modes and operations
    - Paged memory mapping as well as segments

OPTIONAL

# Further Evolution

- **Backward compatibility** ⏶ instruction set doesn't change
  - But they do have more instructions



x86 instruction set

OPTIONAL

# Basic x86 Addressing Modes

- Two operands per instruction (destination register implied!)

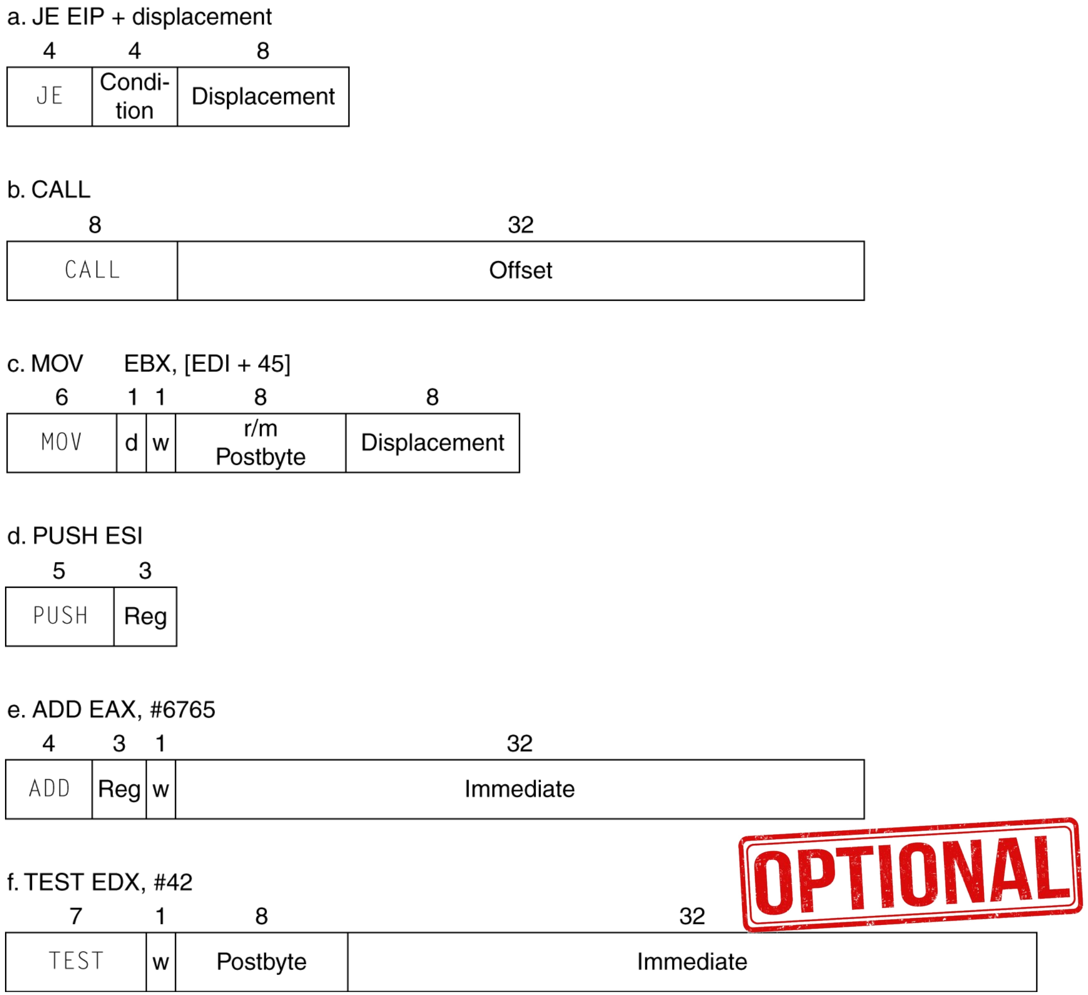
| Source/dest operand | Second source operand |
|---|---|
| Register | Register |
| Register | Immediate |
| Register | Memory |
| Memory | Register |
| Memory | Immediate |

- Memory addressing modes
  - Address in register
  - Address = $R_{base}$ + displacement
  - Two further addressing modes
    - Register restriction

OPTIONAL

# X86 Instruction Encoding

- Variable length encoding
  - Postfix bytes specify addressing mode
  - Prefix bytes modify operation
    - Operand length, repetition, locking, …



a. JE EIP + displacement

| 4 | 4 | 8 |
|---|---|---|
| JE | Condi-tion | Displacement |

b. CALL

| 8 | 32 |
|---|---|
| CALL | Offset |

c. MOV EBX, [EDI + 45]

| 6 | 1 | 1 | 8 | 8 |
|---|---|---|---|---|
| MOV | d | w | r/m<br>Postbyte | Displacement |

d. PUSH ESI

| 5 | 3 |
|---|---|
| PUSH | Reg |

e. ADD EAX, #6765

| 4 | 3 | 1 | 32 |
|---|---|---|---|
| ADD | Reg | w | Immediate |

f. TEST EDX, #42

| 7 | 1 | 8 | 32 |
|---|---|---|---|
| TEST | w | Postbyte | Immediate |

OPTIONAL

# Comparison to MIPS and ARMv7

- X86 is more difficult to build than computers using MIPS and ARMv7
  - Complex instructions

- Market size advantage over MIPS and ARMv7
  - Frequently used component of x86 are not too difficult to implement
  - Intel and AMD have expertise in this area

- In PostPCEra
  - X86 has not been competitive in personal mobile device regardless of implementation expertise

OPTIONAL

# Fallacies

- Powerful instruction ⏶ higher performance
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions
- Use assembly code for high performance
  - But modern compilers are better at dealing with modern processors
  - More lines of code ⏶ more errors and less productivity

OPTIONAL

# Pitfalls

- Sequential words are not at sequential addresses
  - Increment by 4, not by 1!
  - MIPS uses 32-bit word that is equivalent to 4 bytes

OPTIONAL