CO & ISA 2013-2014

Chapter 2: Instruction Set Architecture
(Language of the Computer)

Ngo Lam Trung

[with materials from *Computer Organization and Design, 4th Edition*,
Patterson & Hennessy, © 2008, MK
and M.J. Irwin's presentation, PSU 2008]

Content

Introduction

- Want to command the computer?

  ➔ You need to speak its language!!!

- Example: MIPS assembly instruction

  add a, b, c

- Operation performed
  - add b and c,
  - then store result into a

$$\underbrace{\textbf{add}}_{\text{operation}}\ \underbrace{\textbf{a, b, c}}_{\text{operands}}$$

Introduction

- What does the following code do?

```
add t0, g, h
add t1, i, j
sub f, t0, t1
```

- Equivalent C code

$$f = (g + h) - (i + j)$$

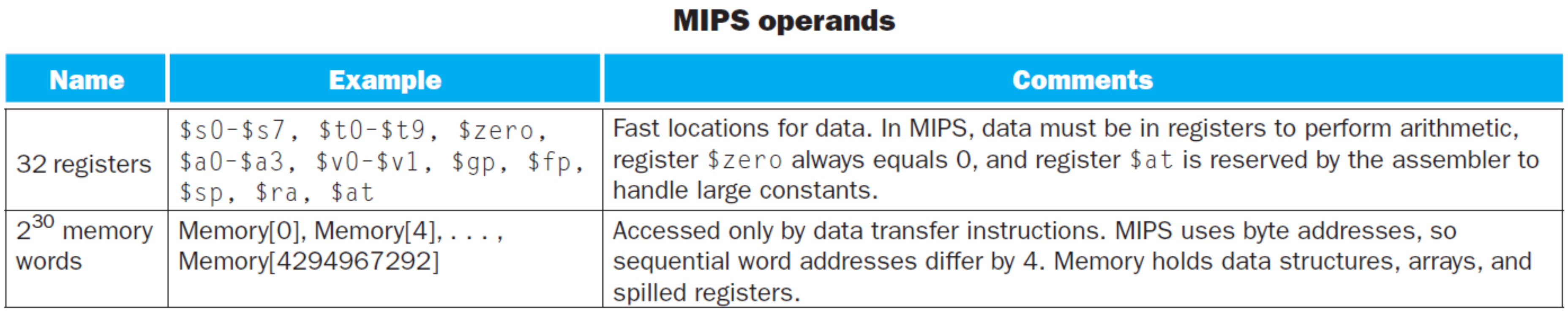
➔ In this chapter

- MIPS operands: register, memory, immediate
- MIPS instruction set

- What is MIPS, and why MIPS?
  - CPU designed by John L. Hennessy (Stanford Univ.'s president)
  - Simple instruction set, appropriate for education

Operands

- Object of operation
  - Source operand: contains input data
  - Destination operand: to store the result of operation
- MIPS operands
  - Registers
  - Memory locations
  - Constant/Immediate



MIPS operands

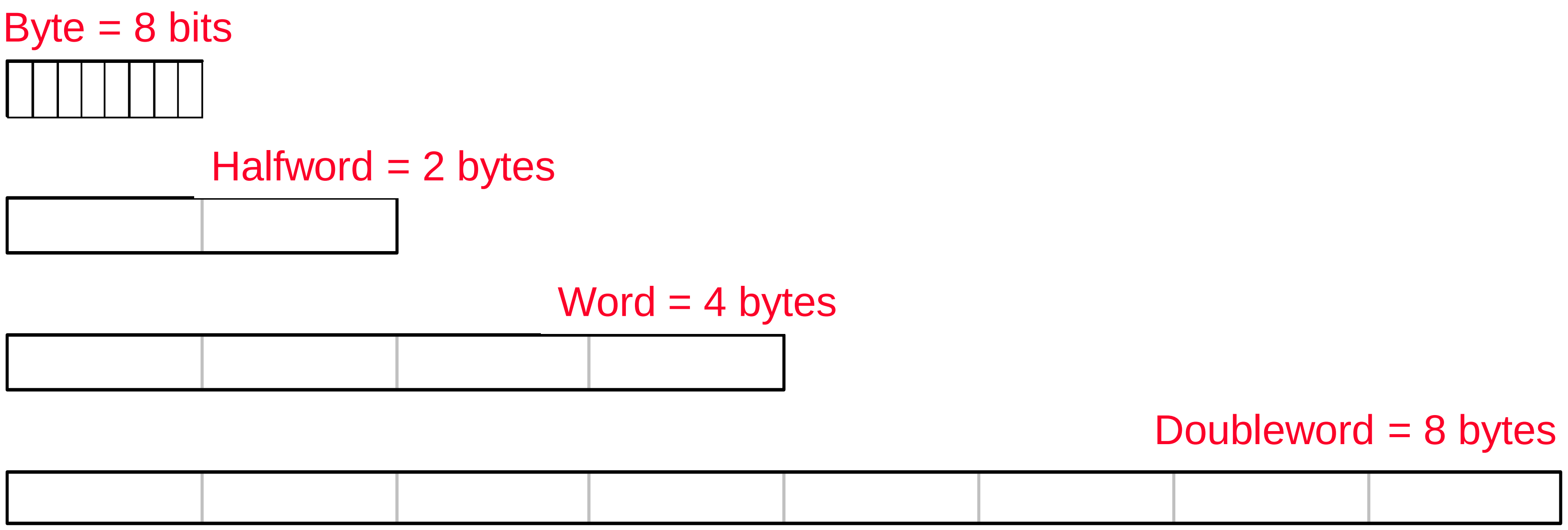
Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

Instruction set

- Arithmetic instructions: addition, subtraction
- Data transfer instructions: transfer data between registers, memory, and immediate
- Logical instructions: and, or, shift
- Conditional branch
- Unconditional branch
- **All MIPS instructions are 32 bits long**

Some basic things

- Data types



MIPS32 registers hold 32-bit (4-byte) words. Other common data sizes include byte, halfword, and doubleword.

Hexadecimal

- It's difficult to read/write/remember long binary number
- Represents in base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

0	0000	4	0100	8	1000	C	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

- Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

Some basic things: Unsigned Binary Integers

- Using n-bit binary number to represent non-negative integer

$$x = x_{n-1}x_{n-2}\ldots x_1x_0$$
$$= x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: 0 to $+2^n - 1$

- Example

  $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
  $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
  $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$

- Data range using 32 bits

  0 to $2^{32}-1 = 4{,}294{,}967{,}295$

Some basic things: 32 bit Unsigned Binary Integers

Hex	Binary	Decimal
0x00000000	0…0000	0
0x00000001	0…0001	1
0x00000002	0…0010	2
0x00000003	0…0011	3
0x00000004	0…0100	4
0x00000005	0…0101	5
0x00000006	0…0110	6
0x00000007	0…0111	7
0x00000008	0…1000	8
0x00000009	0…1001	9
	…	
0xFFFFFFFC	1…1100	2^{32}-4
0xFFFFFFFD	1…1101	2^{32}-3
0xFFFFFFFE	1…1110	2^{32}-2
0xFFFFFFFF	1…1111	2^{32}-1

Exercise

- Convert to 32 bit integers

  25 = 0000 0000 0000 0000 0000 0000 0001 1001

  125 = 0000 0000 0000 0000 0000 0000 0111 1101

  255 = 0000 0000 0000 0000 0000 0000 1111 1111

- Convert 32 bit integers to decimal value

  0000 0000 0000 0000 0000 0000 1100 1111 = 207

  0000 0000 0000 0000 0000 0001 0011 0011 = 307

Some basic things: Signed binary integers

- Using n-bit binary number to represent integer, including negative values

$$
\begin{aligned}
X &= x_{n-1}x_{n-2}\dots x_1x_0 \\
&= -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_1 2^1 + x_0 2^0
\end{aligned}
$$

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example

  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
  $= -1\times2^{31} + 1\times2^{30} + \dots + 1\times2^2 + 0\times2^1 + 0\times2^0$
  $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$

- Using 32 bits

  –2,147,483,648 to +2,147,483,647

Signed integer negation

- Given $x = xn\ _{-\ 1}x_{n\ -\ 2}\ _{\dots}\ x_1x_0$, how to calculate $-x$?
- Let $\bar{x} = 1's\ complement\ of\ x$

$$\bar{x} = 1111 \dots 11_2\ - x$$

(1 → 0, 0 → 1)

Then

$$\bar{x} + x = 1111 \dots 112 = -1$$

→ $\bar{x} + 1 = -x$

- Example: find binary representation of -2

  +2 = 0000 0000 … 0010_2

  –2 = 1111 1111 … 1101_2 + 1
  = 1111 1111 … 1110_2

Signed binary negation

complement all the bits

0101 1011

and add a 1

and add a 1

0110 1010

complement all the bits

	2’sc binary	decimal
-2^3 =	1000	-8
$-(2^3 - 1)$ =	1001	-7
	1010	-6
	1011	-5
	1100	-4
	1101	-3
	1110	-2
	1111	-1
	0000	0
	0001	1
	0010	2
	0011	3
	0100	4
	0101	5
	0110	6
$2^3 - 1$ =	0111	7

Exercise

- Find 16 bit signed integer representation of

16 = 0000 0000 0001 0000

-16 = 1111 1111 1111 0000

100 = 0000 0000 0110 0100

-100 = 1111 1111 1001 1100

Sign extension

- Given n-bit integer $x = xn_{-1}x_{n-2} \dots x_1x_0$
- Find corresponding m-bit representation (m > n) with the same numeric value

$$x = xm_{-1}x_{m-2} \dots x_1x_0$$

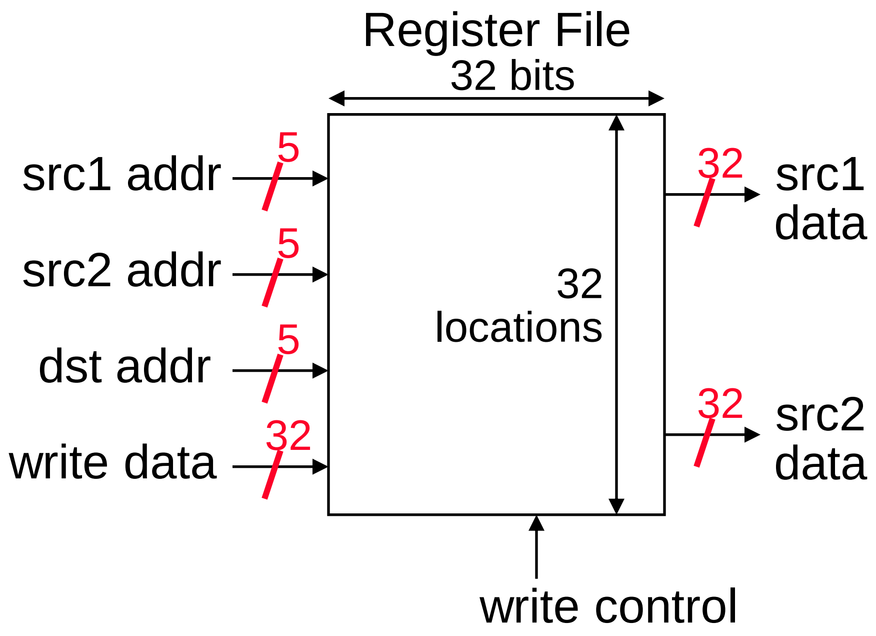
- → Replicate the sign bit to the left
- Examples: 8-bit to 16-bit

  +2: 0000 0010 => 0000 0000 0000 0010

  –2: 1111 1110 => 1111 1111 1111 1110

Register operand: MIPS Register File

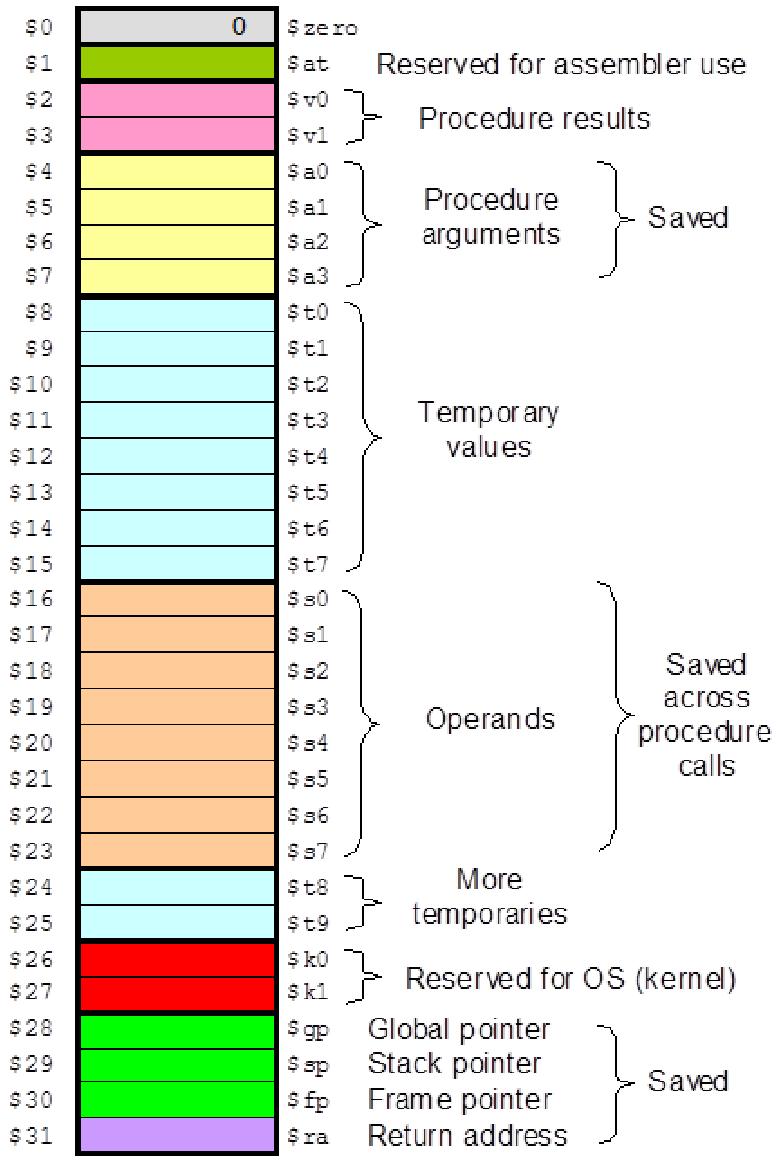
- Special memory inside CPU
- Holds thirty-two 32-bit registers
  - Two read ports with two source address
  - One write port with one destination address
  - Located in CPU → fast, small size



MIPS Register Convention

Name	Register Number	Usage	Preserve on call?
$zero	0	constant 0 (hardware)	n.a.
$at	1	reserved for assembler	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	yes
$t0 - $t7	8-15	**temporaries**	no
$s0 - $s7	16-23	**saved values**	yes
$t8 - $t9	24-25	**temporaries**	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return addr (hardware)	yes

MIPS Register Convention



- MIPS: load/store machine.
- Typical operation
  - Load data from memory to register
  - **Data processing in CPU**
  - Store data from register to memory

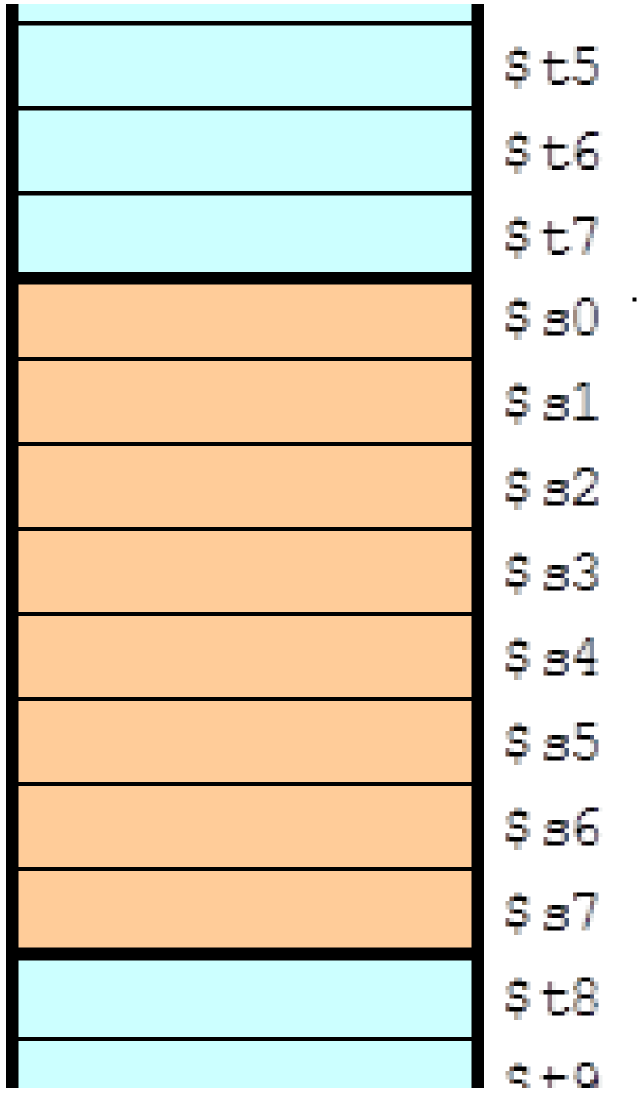
Register operand

- Example

  add $s1, $s2, $s3

  sub $s1, $s1, $s4

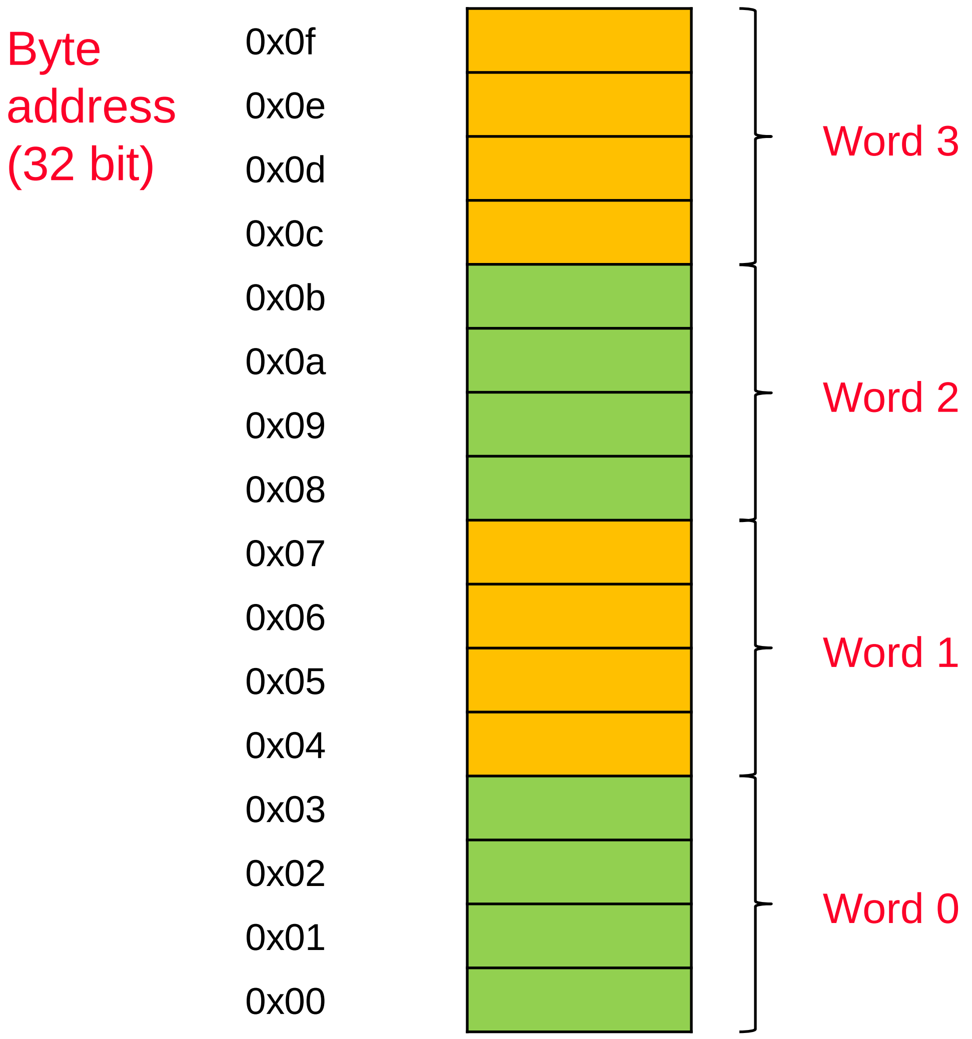
➔ $s1 = ($s2 + $s3) - $s4



Memory operand

- Data stored in computer's main memory
  - Large size
  - Outsize CPU →Slower than register
- Data processing
  - Load values from memory to register
  - Store result from register to memory
- MIPS is big endian

MIPS memory organization



- Byte addressable
- Word data access via byte address
- **Only accessible via load/store instructions**

Alignment
In decimal:
Word address = 4 * word number

In binary:
Word address
= word number + 00

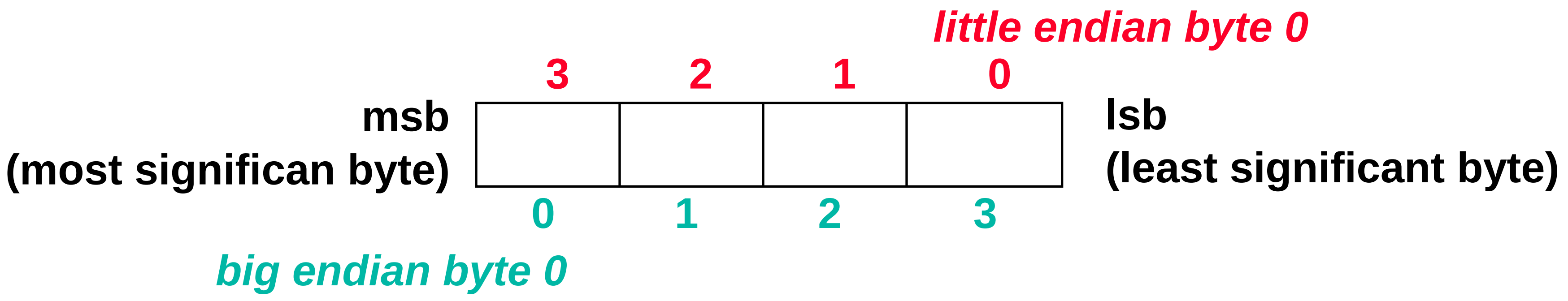
Byte Addresses

- Big Endian: leftmost byte is word address

  IBM 360/370, Motorola 68k, **MIPS**, Sparc, HP PA

- Little Endian: rightmost byte is word address

  Intel 80x86, DEC Vax, DEC Alpha (Windows NT)



Example

- Consider a word in MIPS memory consists of 4 byte with hexa value as below
- What is the word’s value?

address	value
X+3	68
X+2	1B
X+1	5D
X	FA

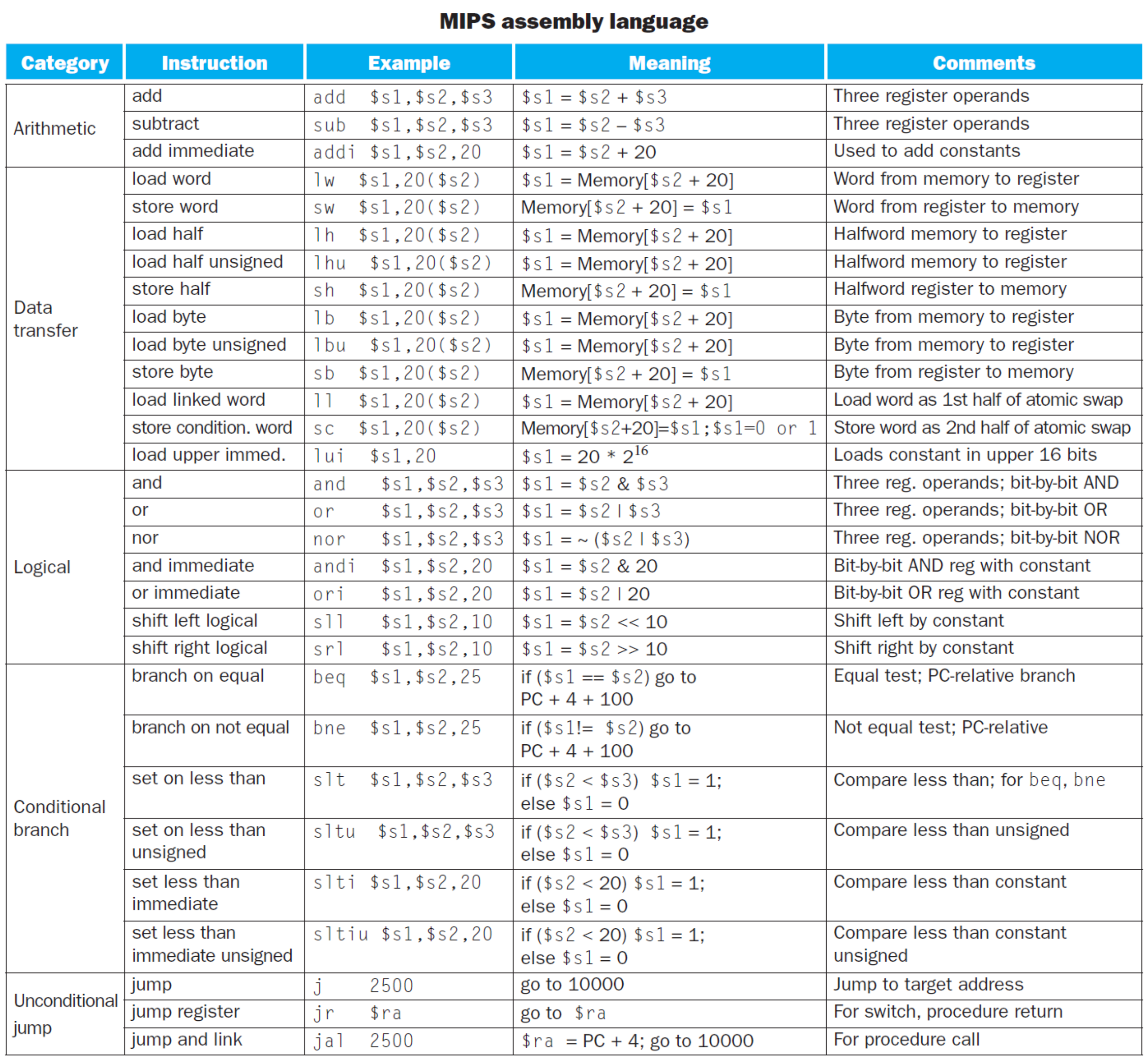
- MIPS is big-endian: address of MSB is X

➔ word’s value: FA5D1B68

Immediate operand

- Immediate value specified by the constant number
- Does not need to be stored in register file or memory
  - Value encoded right in instruction → very fast
  - Fixed value specified when developing the program
  - Cannot change value at run time

Overview of MIPS instruction set



MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	`$s1` = `$s2` + `$s3`	Three register operands
	subtract	`sub $s1,$s2,$s3`	`$s1` = `$s2` – `$s3`	Three register operands
	add immediate	`addi $s1,$s2,20`	`$s1` = `$s2` + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[`$s2` + 20] = `$s1`	Word from register to memory
	load half	`lh $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[`$s2` + 20] = `$s1`	Halfword register to memory
	load byte	`lb $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[`$s2` + 20] = `$s1`	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[`$s2+20`]=`$s1`;`$s1=0 or 1`	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	`$s1` = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	`$s1` = `$s2` & `$s3`	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	`$s1` = `$s2` | `$s3`	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	`$s1` = ~ (`$s2` | `$s3`)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	`$s1` = `$s2` & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	`$s1` = `$s2` | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	`$s1` = `$s2` << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	`$s1` = `$s2` >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if (`$s1` == `$s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if (`$s1`!= `$s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if (`$s2` < `$s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for `beq`, `bne`
	set on less than unsigned	`sltu $s1,$s2,$s3`	if (`$s2` < `$s3`) `$s1` = 1; else `$s1` = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if (`$s2` < 20) `$s1` = 1; else `$s1` = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if (`$s2` < 20) `$s1` = 1; else `$s1` = 0	Compare less than constant unsigned
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

Fig. 2.1

MIPS Instruction set: Arithmetic operations

- MIPS arithmetic statement

```
add  rd, rs, rt      #rd ← rs + rt
sub  rd, rs, rt      #rd ← rs – rt
addi rd, rs, const   #rd ← rs + const
```

rs	5-bits	register file address of the first source operand
rt	5-bits	register file address of the second source operand
rd	5-bits	register file address of the result’s destination

Example

- Currently \$s1 = 6
- What is value of \$s1 after executing the following instruction

```
addi    $s2, $s1, 3
addi    $s1, $s1, -2
sub     $s1, $s2, $s1
```

MIPS Instruction set: Logical operations

- Basic logic operations

```
and  rd, rs, rt      #rd ← rs & rt
andi rd, rs, const   #rd ← rs & const
or   rd, rs, rt      #rd ← rs | rt
ori  rd, rs, const   #rd ← rs | const
nor  rd, rs, rt      #rd ← ~(rs | rt)
```

- Example $s1 = 8 = 0000 1000, $s2 = 14 = 0000 1100

```
and  $s3, $s1, $s2
or   $s4, $s1, $s2
```

MIPS Instruction set: Logical operations

- Logical shift and arithmetic shift: move all the bits in a word left or right

```
sll  rd, rs, const   #rd ← rs << const
srl  rd, rs, const   #rd ← rs >> const
sra  rd, rs, const   #rd ← rs >> const
                     (keep sign bit)
```

MIPS Instruction set: Memory Access Instructions

- MIPS has two basic data transfer instructions for accessing memory

```
lw $t0, 4($s3)   #load word from memory

sw $t0, 8($s3)   #store word to memory
```

- The data is loaded into (lw) or stored from (sw) a register in the register file

- The memory address – a 32 bit address – is formed by adding the contents of the base address register to the offset value

- Offset can be negative, and must be multiple of 4

MIPS Instruction set: Load Instruction

- Load/Store Instruction Format:

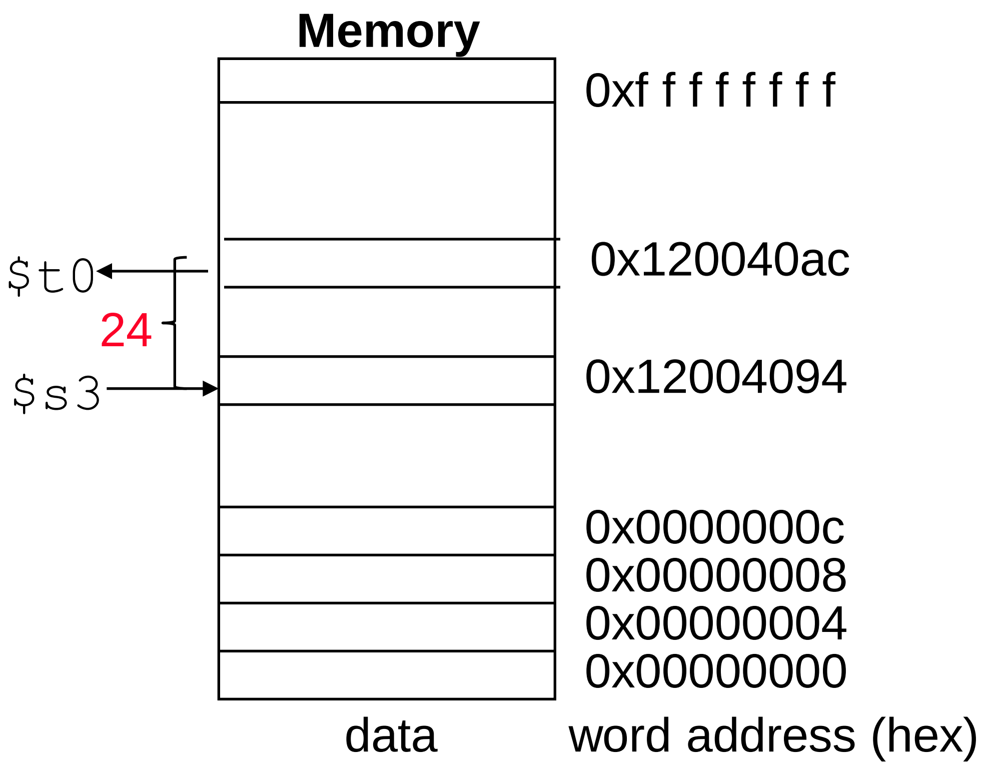
```
lw $t0, 24($s3)    #$t0← mem at 24+$s3
```

(move a word from memory to $t0)

24_{10} + $s3 =

```
  . 0001 1000 (24)
+ . 1001 0100 (94)
  . 1010 1100 (ac)
= 0x1200 40ac
```

0x..94 = ..1001 0100



Loading and Storing Bytes

- MIPS provides special instructions to move bytes

```
lb    $t0, 1($s3)   #load byte from memory

sb    $t0, 6($s3)   #store byte to  memory
```

0x28	19	8	16 bit offset

- What 8 bits get loaded and stored?
  - load byte places the byte from memory in the rightmost 8 bits of the destination register
    - what happens to the other bits in the register?
  - store byte takes the byte from the rightmost 8 bits of a register and writes it to a byte in memory
    - what happens to the other bits in the memory word?

Loading 16/32 bit constants

- Use two instructions to load a 32 bit constant into a register
- a new "load upper immediate" instruction

```
lui $t0, 1010101010101010
```

- Then must get the lower order bits right, use

```
ori $t0, $t0, 1010101010101010
```

1010101010101010	0000000000000000
0000000000000000	1010101010101010

1010101010101010	1010101010101010

MIPS Control Flow Instructions

- MIPS conditional branch instructions:

```
bne $s0, $s1, Exit #go to Exit if $s0≠$s1
beq $s0, $s1, Exit #go to Exit if $s0=$s1
```

- Ex:

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

              bne $s0, $s1, Exit
              add $s3, $s0, $s1
Exit :        ...
```

- How is the branch destination address specified?

Example in MipsIT

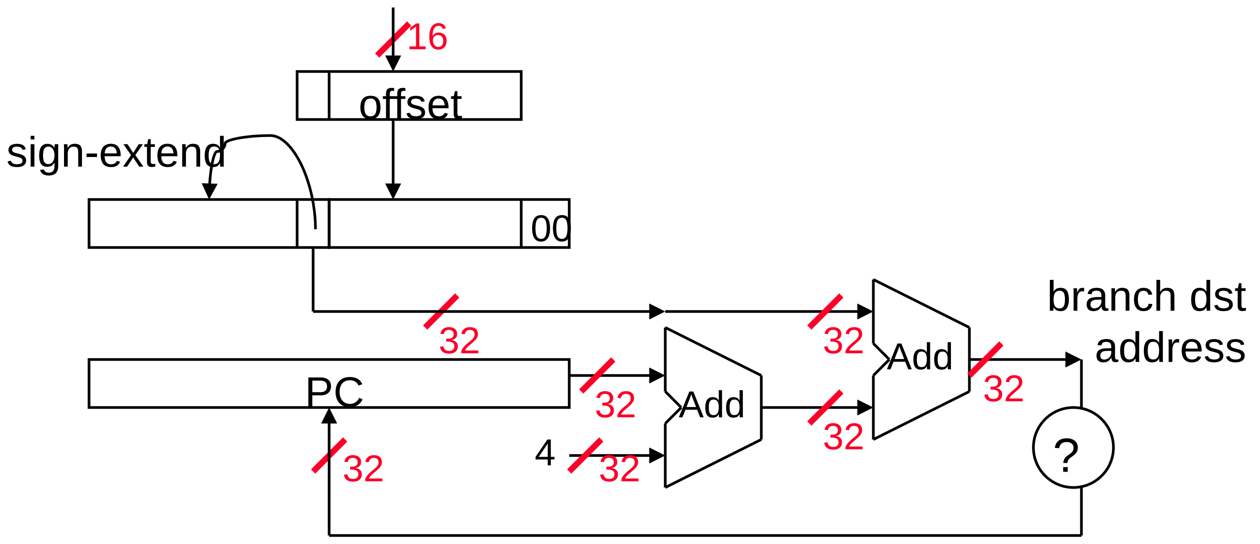
```
#include <iregdef.h>
.text
.set reorder
.globl start
.ent start
start:
                li      s0, 1     #load value for s0
                li      s1, 2
                li      s3, 0
                beq     s0, s1, Exit
                add     s3, s2, s1
        Exit:   add     s2, s3, s1
.end start
```

What is final value of s2?

Specifying Branch Destinations

- Use a register (like in lw and sw) added to the 16-bit offset
  - which register? Instruction Address Register (the PC)
    - its use is automatically implied by instruction
    - PC gets updated (PC+4) during the fetch cycle so that it holds the address of the next instruction
  - limits the branch distance to -2^{15} to $+2^{15}-1$ (word) instructions from the (instruction after the) branch instruction, but most branches are local anyway

from the low order 16 bits of the branch instruction



In Support of Branch Instructions

- How to use `beq`, `bne`, to support other kinds of branches (e.g., branch-if-less-than)?
- Set flag based on condition: eg `slt`
- Set on less than instruction:

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

- Alternate versions of `slt`

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

sltu $t0, $s0, $s1    # if $s0 < $s1 then $t0=1 ...

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

- How about set on bigger than?

More Branch Instructions

- Combine `slt`, `beq`, `bne`, and the register `$zero` to create other conditions
  - less than `blt $s1, $s2, Label`

```
slt   $at, $s1, $s2         #$at set to 1 if
bne   $at, $zero, Label     #$s1 < $s2
```

  - less than or equal to `ble $s1, $s2, Label`
  - greater than `bgt $s1, $s2, Label`
  - great than or equal to `bge $s1, $s2, Label`

- Such branches are included in the instruction set as pseudo instructions - recognized (and expanded) by the assembler
  - Its why the assembler needs a reserved register (`$at`)

Unconditional branch

- MIPS also has an unconditional branch instruction or jump instruction:

```
j   label          #go to label
```

Example

Show a sequence of instructions corresponding to:

```
if (i<=j) {x = x+1; z = 1;}
else {y = y-1; z = 2*z}
```

Solution

```
      slt  $t0,$s2,$s1     # j<i? (inverse condition)
      bne  $t0,$zero,else  # if j<i goto else part
      addi $t1,$t1,1       # begin then part: x = x+1
      addi $t3,$zero,1     # z = 1
      j    endif           # skip the else part
else: addi $t2,$t2,-1      # begin else part: y = y-1
      add  $t3,$t3,$t3     # z = z+z
endif:...
```

Example

The simple while loop: `while (A[i]==k) i=i+1;`

Assuming that: `i, A, k` are stored in `$s1,$s2,$s3`

Solution

```
loop: add   $t1,$s1,$s1        # t1 = 4*i
      add   $t1,$t1,$t1        #
      add   $t1,$t1,$s2        # t1 = A + 4*I,
                                 address of A[i]
      lw    $t0,0($t1)         # load data in A[i]
                                 into t0
      bne   $t0,$s3,endwhl     #
      addi $s1,$s1,1           #
      j     loop               #
endwhl: …                      #
```

Finding the Maximum Value in a List of Integers

List `A` is stored in memory beginning at the address given in `$s1`.
List length is given in `$s2`.
Find the largest integer in the list and copy it into `$t0`.

Solution

Scan the list, holding the largest element identified thus far in `$t0`.

```
       lw    $t0,0($s1)        # initialize maximum to A[0]
       addi  $t1,$zero,0       # initialize index i to 0
loop:  add   $t1,$t1,1         # increment index i by 1
       beq   $t1,$s2,done      # if all elements examined, quit
       add   $t2,$t1,$t1       # compute 2i in $t2
       add   $t2,$t2,$t2       # compute 4i in $t2
       add   $t2,$t2,$s1       # form address of A[i] in $t2
       lw    $t3,0($t2)        # load value of A[i] into $t3
       slt   $t4,$t0,$t3       # maximum < A[i]?
       beq   $t4,$zero,loop    # if not, repeat with no change
       addi  $t0,$t3,0         # if so, A[i] is the new maximum
       j     loop              # change completed; now repeat
done:  ...                     # continuation of the program
```

Example

The simple switch

```
switch(test) {
   case 0:
         a=a+1; break;
   case 1:
         a=a-1; break;
   case 2:
         b=2*b; break;
   default:
}
```

Assuming that: `test,a,b` are stored in `$s1,$s2,$s3`

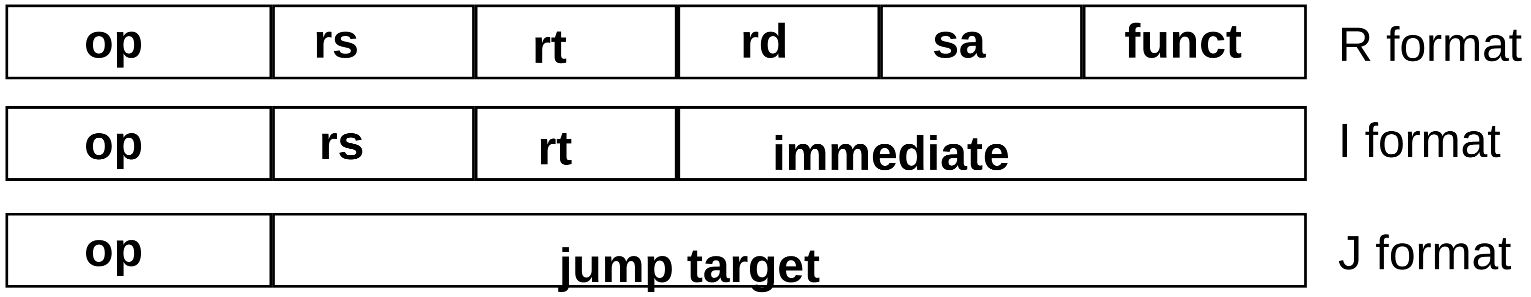
Solution

```
         beq    s1,t0,case_0
         beq    s1,t1,case_1
         beq    s1,t2,case_2
         b      default
case_0:
         addi   s2,s2,1       #a=a+1
         b      continue
case_1:
         sub    s2,s2,t1      #a=a-1
         b      continue
case_2:
         add    s3,s3,s3      #b=2*b
         b      continue
default:
continue:
```

Representation of MIPS instruction

- All MIPS instructions are 32 bits wide
- Instructions are 32 bits binary number

3 Instruction Formats: all 32 bits wide



Reference: MIPS Instruction Reference (MIPS_IR.pdf)

R-format instruction

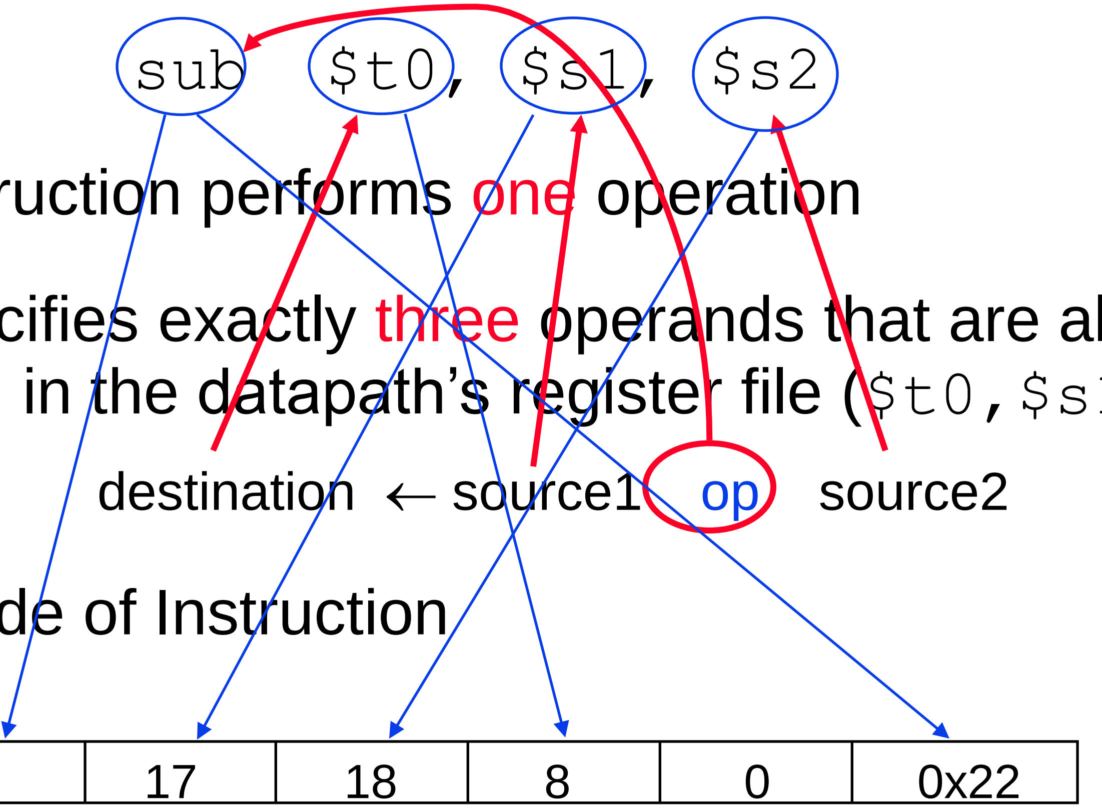
- All fields are encoded by mnemonic names

op	rs	rt	rd	shamt	funct

op	6-bits	opcode that specifies the operation
rs	5-bits	register file address of the first source operand
rt	5-bits	register file address of the second source operand
rd	5-bits	register file address of the result's destination
shamt	5-bits	shift amount (for shift instructions)
funct	6-bits	function code augmenting the opcode

Example of R-format instruction

```
add  $t0, $s1, $s2
```



- Each instruction performs one operation
- Each specifies exactly three operands that are all contained in the datapath's register file (`$t0,$s1,$s2`)

destination ← source1 op source2

- Binary code of Instruction

0	17	18	8	0	0x22

Example

- Find machine codes of the following instructions

```
lw     $t0,0($s1)    # initialize maximum to A[0]
addi   $t1,$zero,0 # initialize index i to 0
add    $t1,$t1,1    # increment index i by 1
```

Example of I-format instruction

```
slti $t0, $s2, 15    #$t0 = 1 if $s2<15
```

- Machine format (I format):

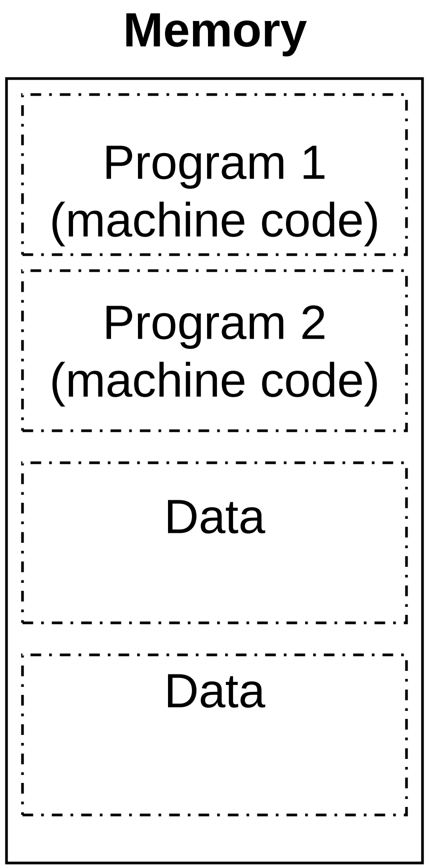
0x0A	18	8	0x0F

- The constant is kept inside the instruction itself!
  - Immediate format limits values to the range $+2^{15}–1$ to -2^{15}

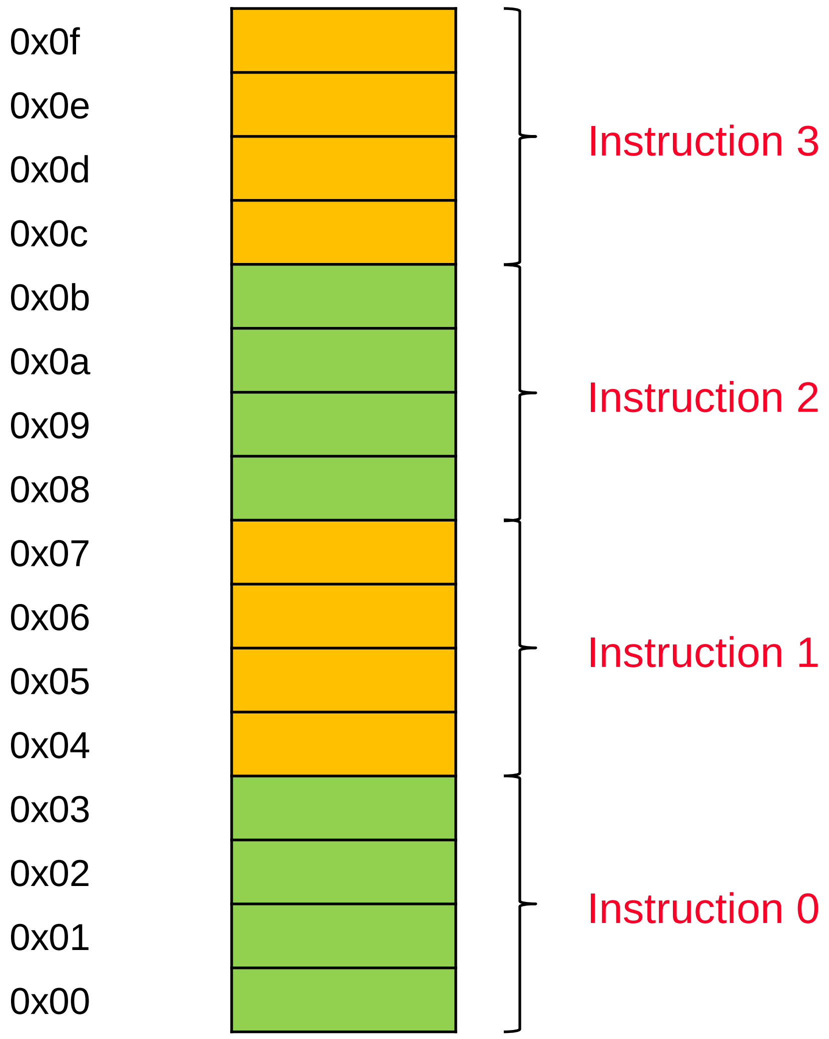
Two Key Principles of Machine Design

1. Instructions are represented as numbers
2. Programs are stored in alterable memory (that can be read or written to)

- Stored-program concept
  - Programs can be shipped as files of binary numbers
  - Computers can inherit ready-made software provided they are compatible with an existing ISA



Stored program concept



- Stored program in memory
- 1 instruction = 1 word
- Instruction cycle:
  - Fetch
  - Decode
  - Execution
- Program flow
  - Next instruction = ?
  - **PC: Program Counter**
  - Auto-increment after each instruction fetch

Instructions for Accessing Procedures

- MIPS procedure call instruction:

```
jal   ProcedureAddress       #jump and link
```

- Saves PC+4 in register $ra to have a link to the next instruction for the procedure return
- Machine format (J format):

0x03	26 bit address

- Then can do procedure return with a

```
jr    $ra                    #return
```

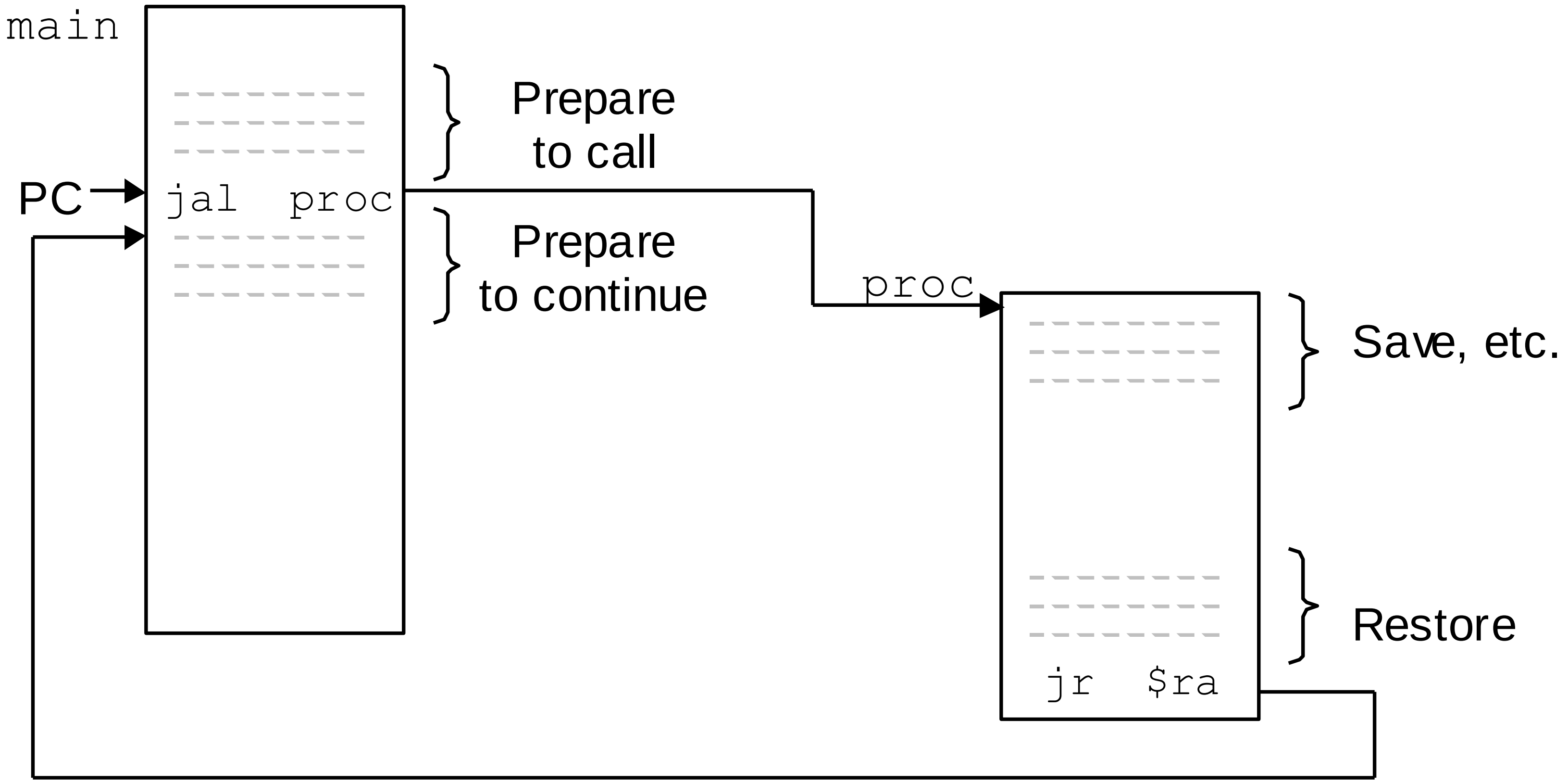
- Instruction format (R format):

0	31				0x08

Six Steps in the Execution of a Procedure

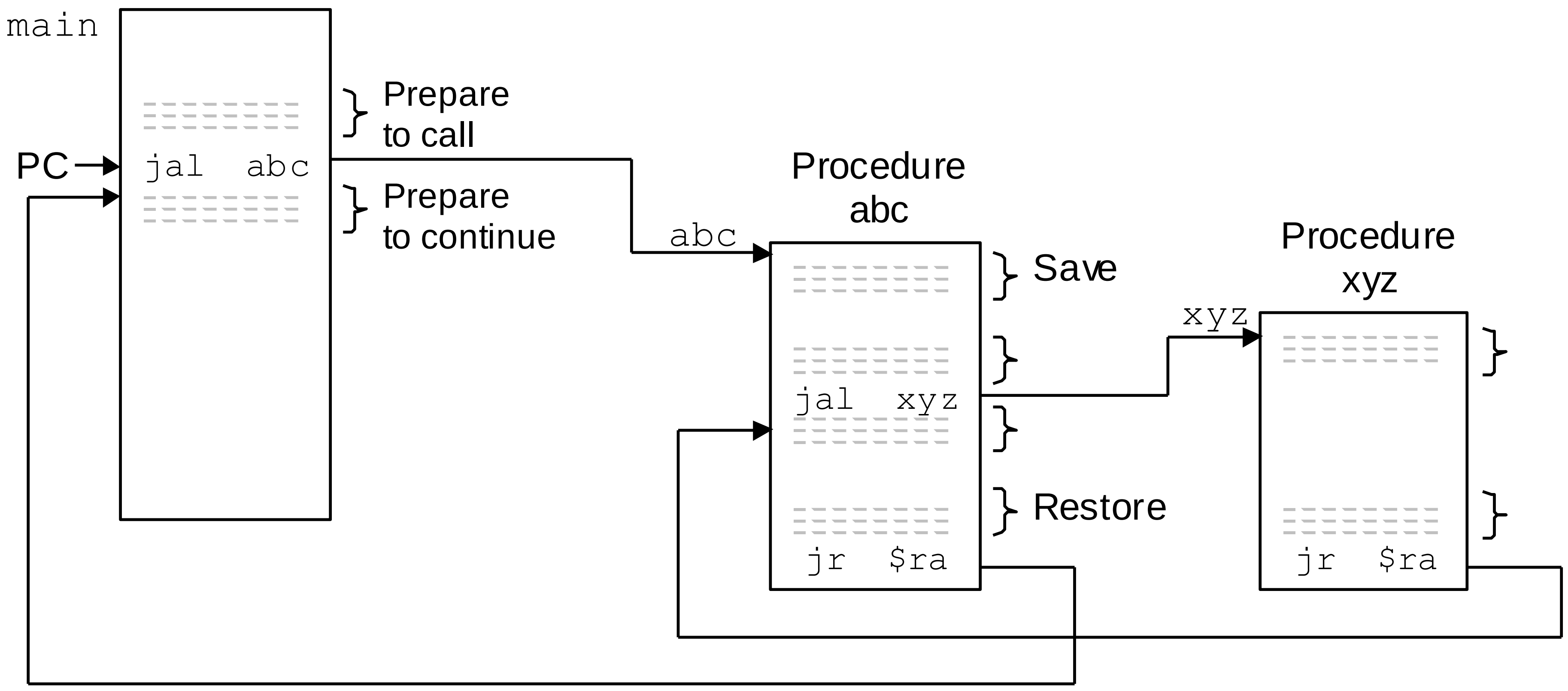
1. Main routine (caller) places parameters in a place where the procedure (callee) can access them
   - `$a0` - `$a3`: four argument registers
2. Caller transfers control to the callee (jal)
3. Callee acquires the storage resources needed
4. Callee performs the desired task
5. Callee places the result value in a place where the caller can access it
   - `$v0` - `$v1`: two value registers for result values
6. Callee returns control to the caller (jr)
   - `$ra`: one return address register to return to the point of origin

Illustrating a Procedure Call



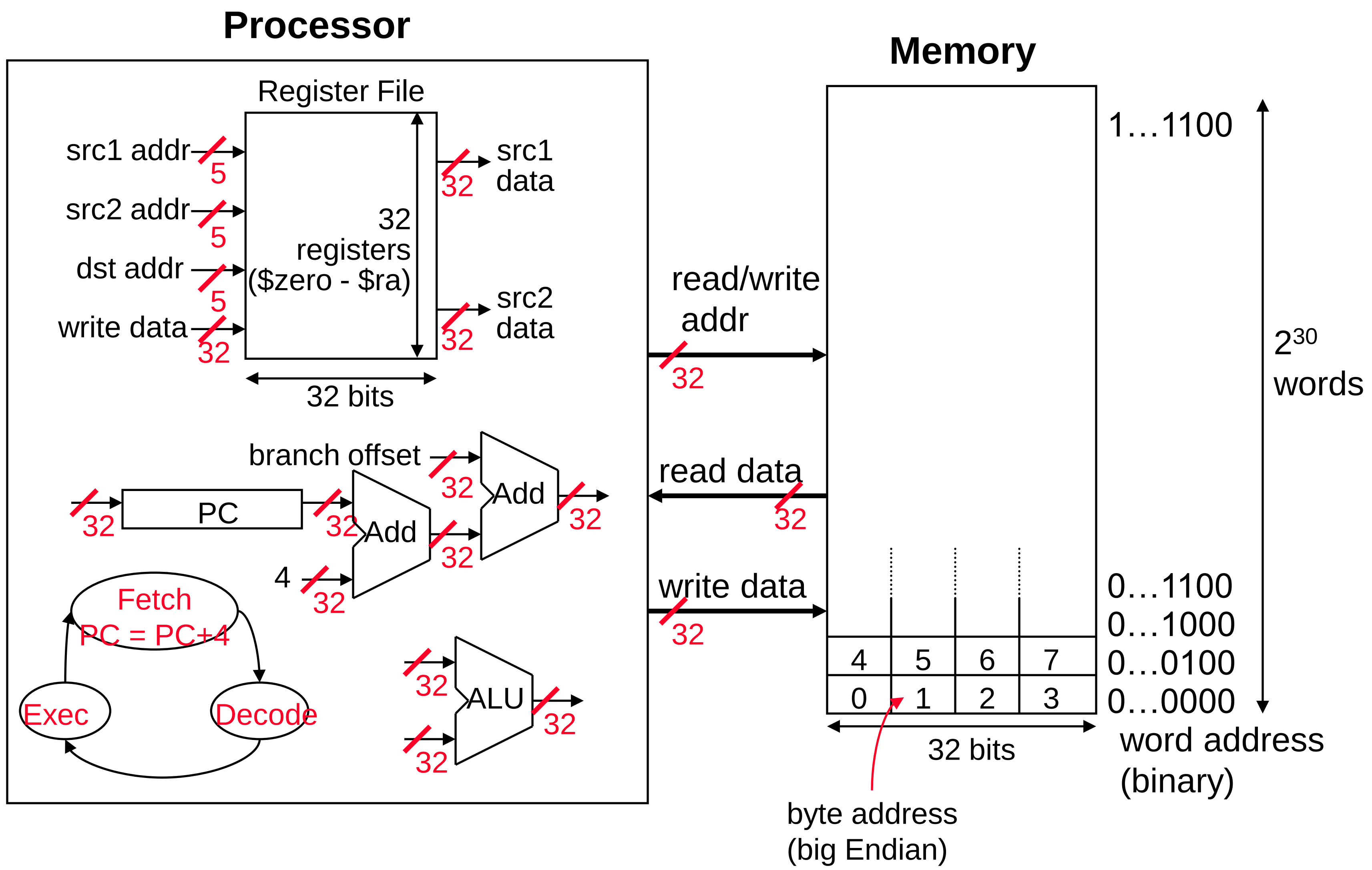
Relationship between the main program and a procedure.

Nested Procedure Calls

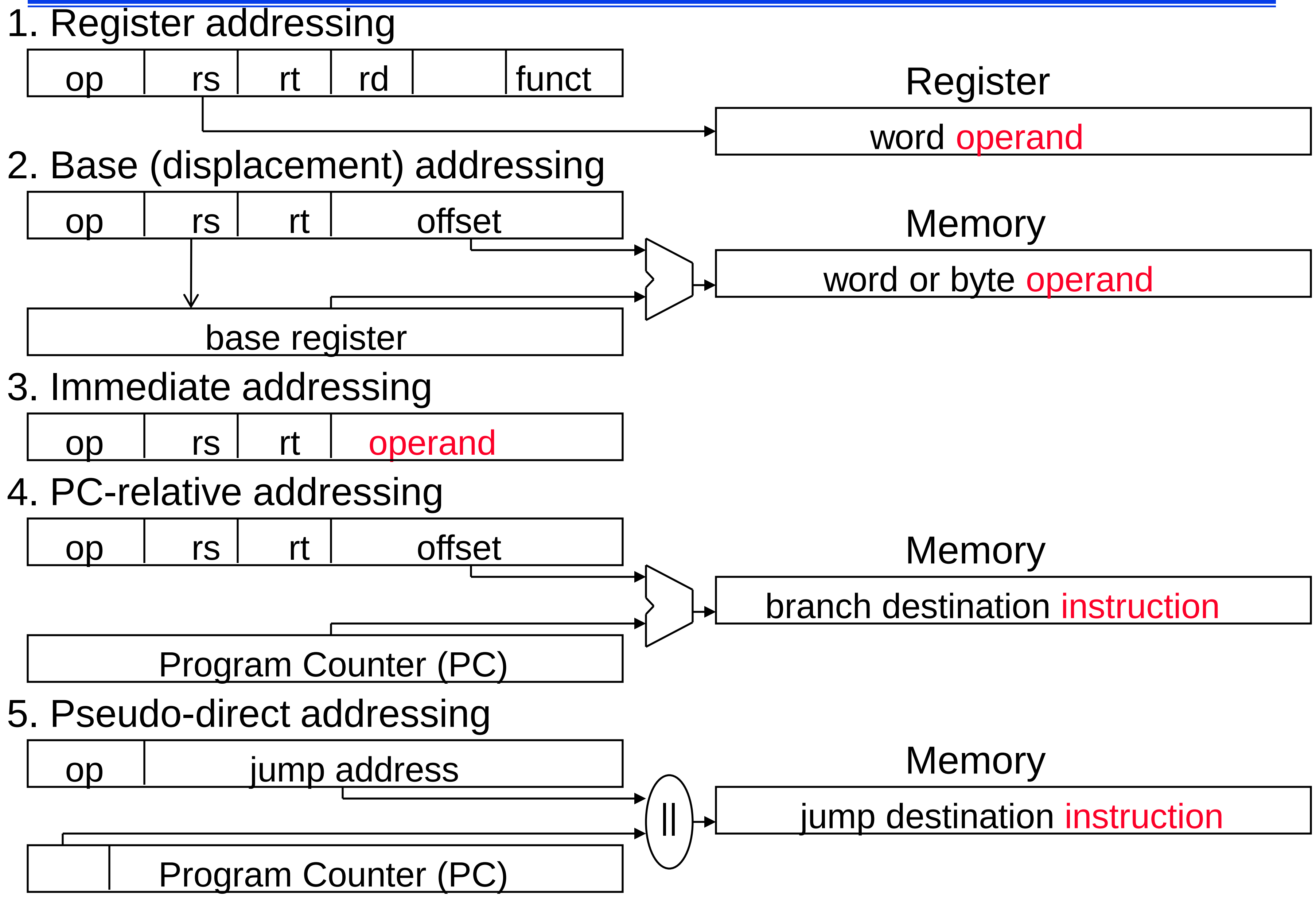


Example of nested procedure calls.

MIPS Organization So Far



Addressing Modes Illustrated



Summary

- Provided one problem to be solved by computer
  - Can it be implemented?
  - Can it be programmed?
  - Which CPU is suitable?
- Metric of performance
  - How many bytes does the program occupy in memory?
  - How many instructions are executed?
  - How many clocks are required per instruction?
  - How much time is required to execute the program?

➔ Largely depend on Instruction Set Architecture (ISA)