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

Ngo Lam Trung

[with materials from *Computer Organization and Design, 5th Edition*,
Patterson & Hennessy, © 2014, MK
and M.J. Irwin's presentation, PSU 2008]

Content

- Introduction
- MIPS Instruction Set Architecture
  - MIPS operands
  - MIPS instruction set
- Programming structures
  - Branching
  - Procedure call
- Practice
  - MIPS simulator
  - Writing program for MIPS

What is MIPS, and why MIPS?

- CPU designed by John Hennessy's team
  - Stanford's president 2000-2016
  - 2017 Turing award for RISC development
  - "god father" of Silicon Valley
- Very successful CPU in 80s-90s, the first that have 64 bit architecture
- Still very popular in embedded market: set top box, game console,…
- Simple instruction set, appropriate for education (the mini instruction set)

Computer language: hardware operation

- Want to command the computer?

  ➔ You need to speak its language!!!

- Example: MIPS assembly instruction

  add a, b, c #a ← b + c

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

$\underbrace{\text{add}}_{\text{operation}}$ $\underbrace{\text{a, b, c}}_{\text{operands}}$ $\underbrace{\text{\#a} \leftarrow \text{b + c}}_{\text{comments}}$

Hardware operation

- What does the following code do?

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

- Equivalent C code

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

➔ ***Why not making 4 or 5 inputs instructions?***

➔ ***DP1: Simplicity favors regularity!***

Operands

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

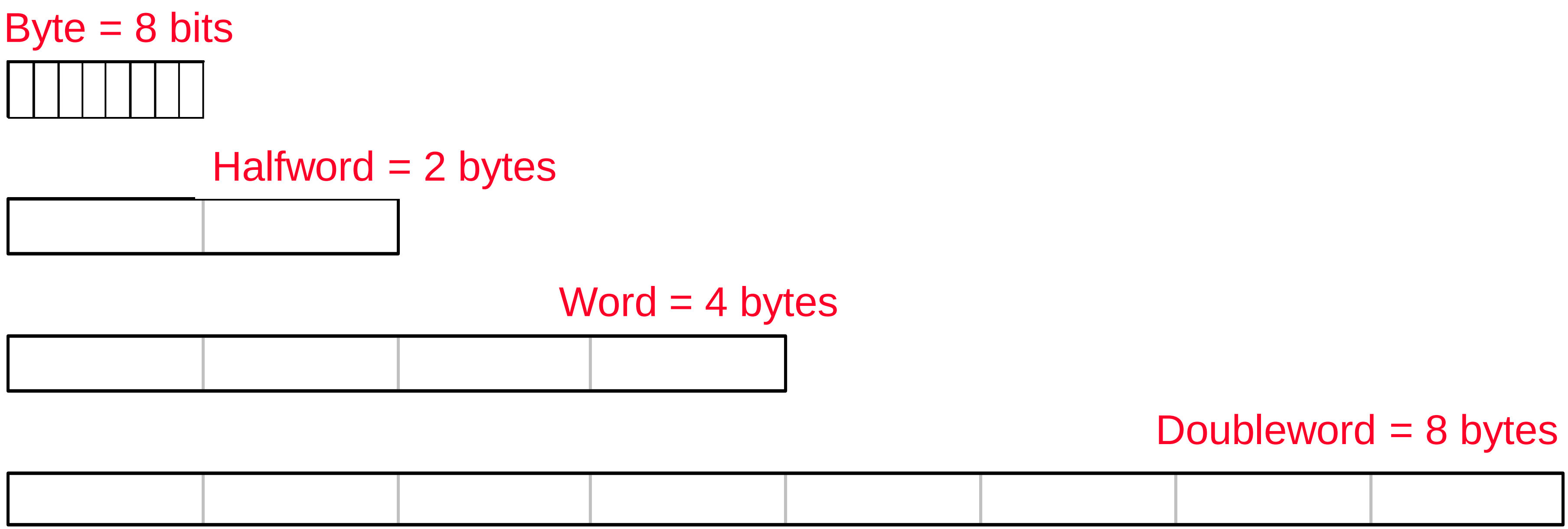
MIPS operands

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.

Register operand: MIPS Register File

- Special memory inside CPU, called register file
- 32 slots, each slot is called a register
- Each register holds 32 bits of data (a word)
- Each register has an unique address, and a name
- Register’s address is from 0 to 31, represented by 5 bits

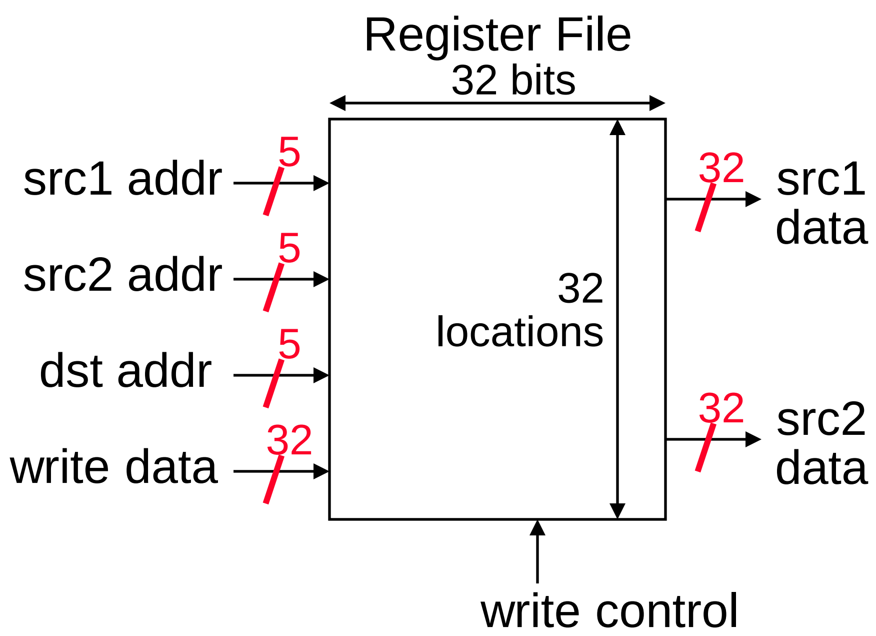
Data types in MIPS



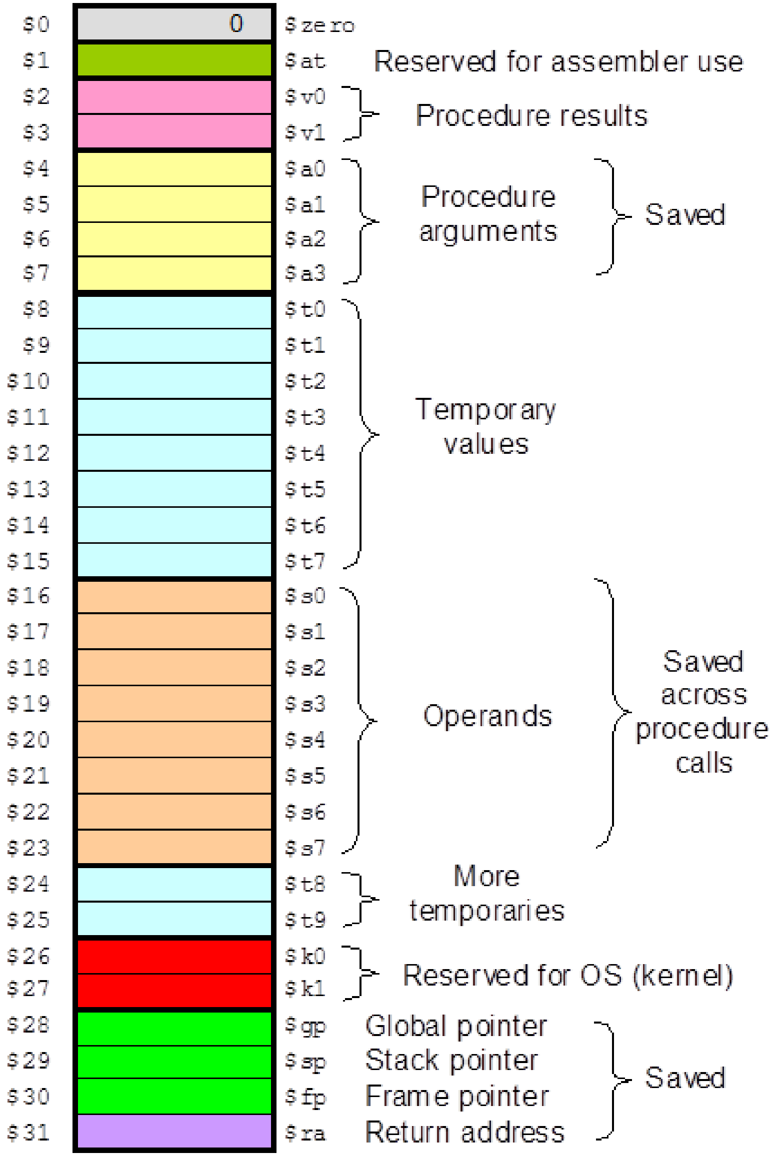
MIPS32 registers hold 32-bit (4-byte) words. Other common data sizes include byte, halfword, and doubleword.

Register operand: MIPS Register File

- Register file in MIPS CPU
  - Two read ports with two source address
  - One write port with one destination address
  - Located in CPU → fast, small size



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: MIPS Register File

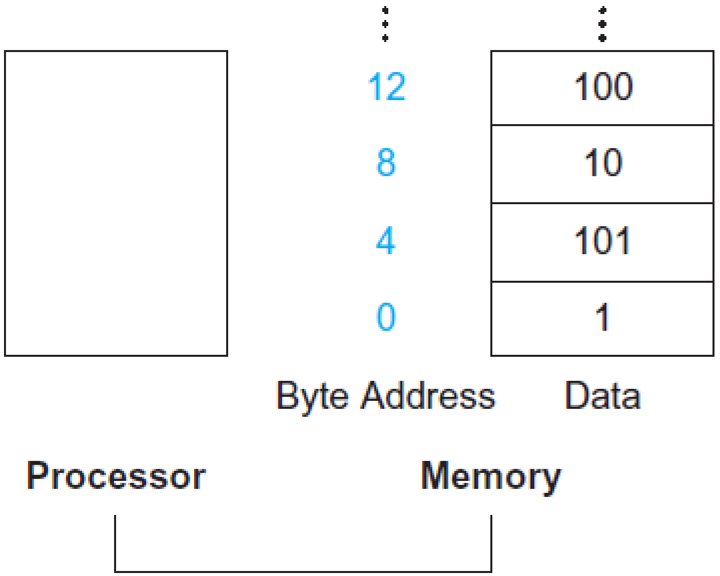
- Register file: “work place” right inside CPU.
- Larger register file should be better, more flexibility for CPU operation.
- Moore’s law: doubled number of transistor every 18 mo.
- Why only 32 registers, not more?

➔ ***DP2: Smaller is faster!***

Effective use of register file is critical!

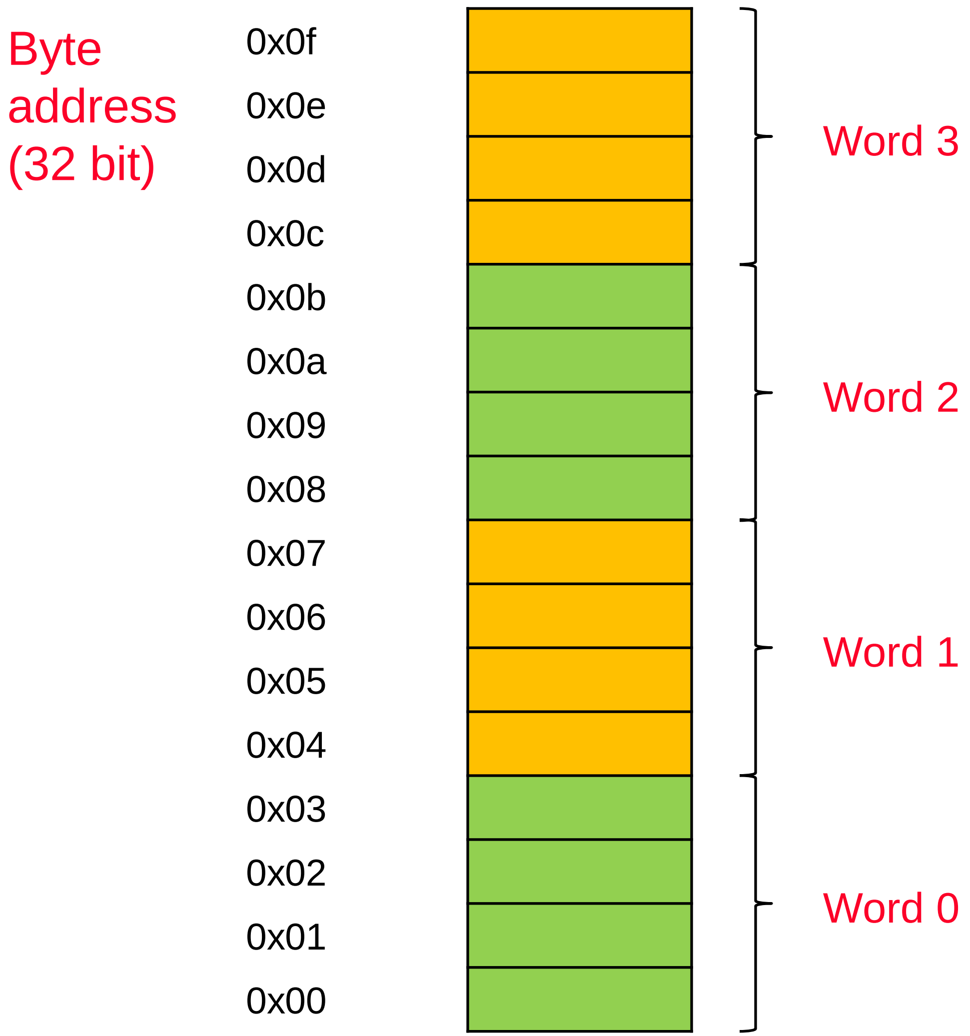
Memory operand

- Data stored in computer's main memory
  - Large size
  - Outsize CPU →Slower than register
- Operations with memory operand
  - Load values from memory to register
  - Store result from register to memory



Byte addressable
Word aligned

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

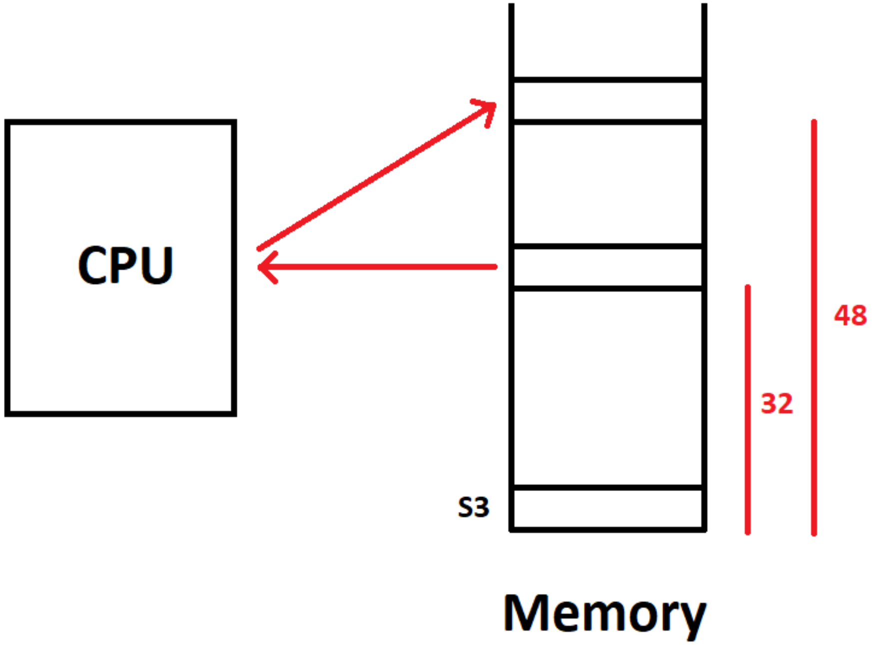
In binary:

Word address = word number + 00

Memory operand

- Sample instruction

```
lw $t0,32($s3)
#do sth
#
sw $t0,48($s3)
```



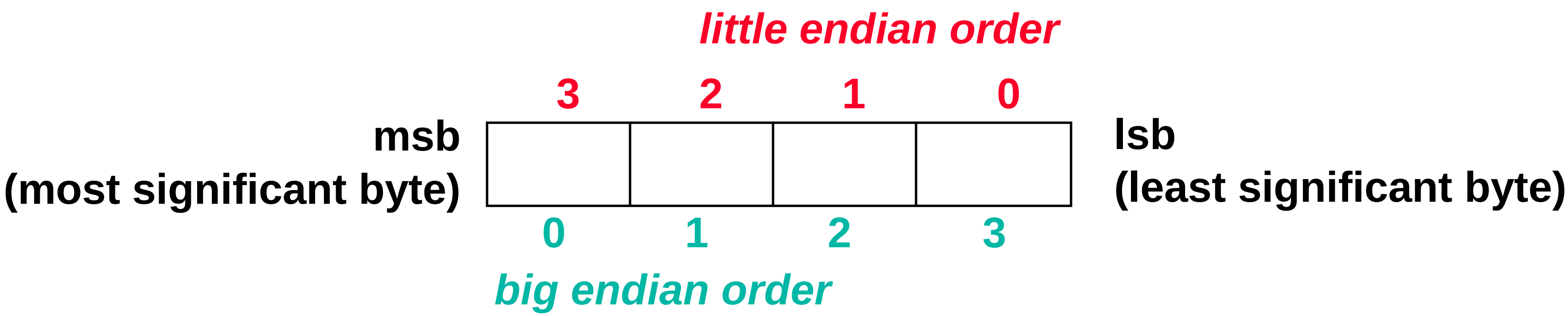
Byte Addresses

- 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

Immediate operand

- What is the mostly used constant?
- The special register: $zero
- Constant value of 0
- Why?

➔ ***DP3: Making common cases fast!***

What are stored inside operands?

- Data, of course!
- And data is represented as binary numbers
- Then how binary numbers are treated by MIPS?
  - As integers
  - Unsigned
  - Signed

Unsigned Binary Integers

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

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

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

- Example

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

- Data range using 32 bits

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

Eg: 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

Signed binary integers

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

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

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

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

- Using 32 bits

  –2,147,483,648 to +2,147,483,647

Signed integer negation

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

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

(1 → 0, 0 → 1)

Then

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

➔ $$\bar{x} + 1 = -x$$

- Example: find binary representation of -2

$+2 = 0000\ 0000 \ldots 0010_2$

$–2 = 1111\ 1111 \ldots 1101_2 + 1$
$= 1111\ 1111 \ldots 1110_2$

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 $x = xn_{\_1} x_{n\_2} {}_{...} x_1 x_0$
- Find corresponding m-bit representation (m > n) with the same numeric value

$$x = xm_{\_1} x_{m\_2} {}_{...} x_1 x_0$$

- → 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

Instruction set

- 3 instruction formats:
  - Register (R)
  - Immediate (I)
  - Branch (J)
- R-instruction: all operands are register
- I-instruction: one operand is immediate
- J-instruction: the unconditional branch
- **Note: All MIPS instructions are 32 bits long**

➔ ***Why not only one format?***

➔ ***DP4: Good design demands good compromises!***

5 instruction types

- Arithmetic: addition, subtraction
- Data transfer: transfer data between registers, memory, and immediate
- Logical: and, or, shift
- Conditional branch
- Unconditional branch

Overview of MIPS instruction set

Fig. 2.1

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

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 1110

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

MIPS Instruction set: Logical operations

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

```
sll  rd, rs, const   #rd ← rt << const

srl  rd, rs, const   #rd ← rt >> const

sra  rd, rs, const   #rd ← rt >> 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 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:

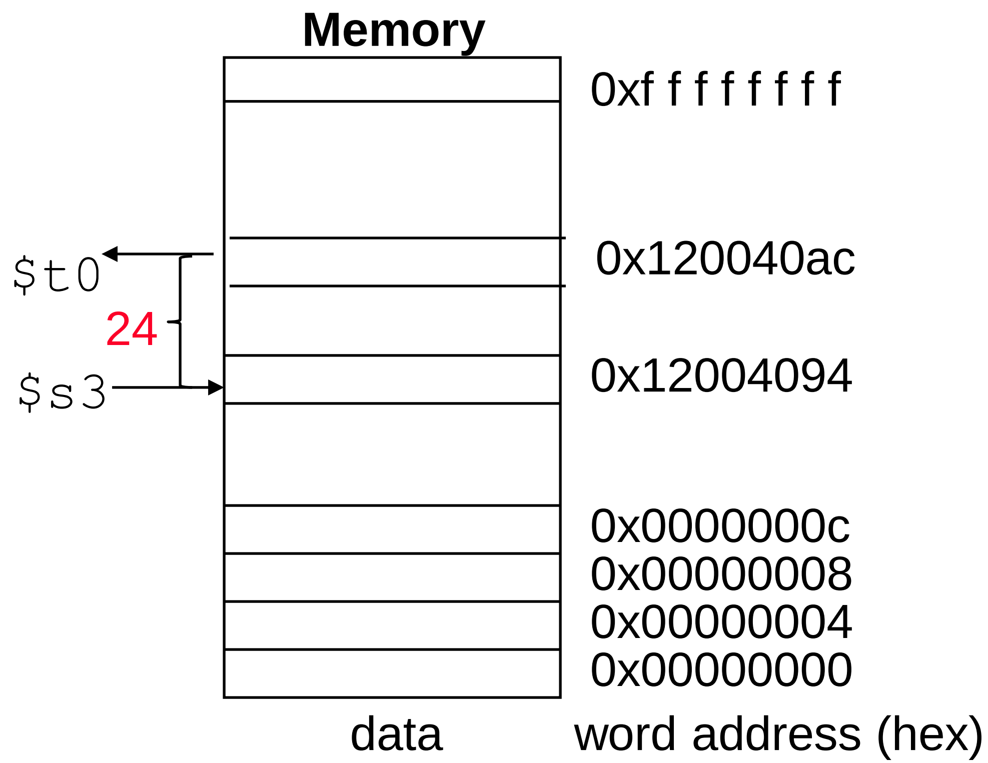
```
lw $t0, 24($s3)    #$t0← mem at 24+$s3
```

(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



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 :  ...
```

Example

```
start:
              addi   s0, zero, 2   #load value for s0
              addi   s1, zero, 2
              addi   s3, zero, 0
              beq    s0, s1, Exit
              add    s3, s2, s1
      Exit:   add    s2, s3, s1
.end start
```

What is final value of s2?

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: `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
```

  →pseudo-instructions
  - 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

- Write assembly code to do the following

```
if (i<5)
    X = 3;
else
    X = 10;
```

Solution

```
      slti $t0,$s1,5       # i<5? (inverse condition)
      beq  $t0,$zero,else  # if i>=5 goto else part
      addi $t1,$zero,3     # X = 3
      j    endif           # skip the else part
else: addi $t1,$zero,10    # X = 10
endif:...
```

Representation of MIPS instruction

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

3 Instruction Formats: all 32 bits wide

op	**rs**	**rt**	**rd**	**sa**	**funct**	R format
op	**rs**	**rt**	**immediate**			I format
op	**jump target**					J format

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
sub  $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

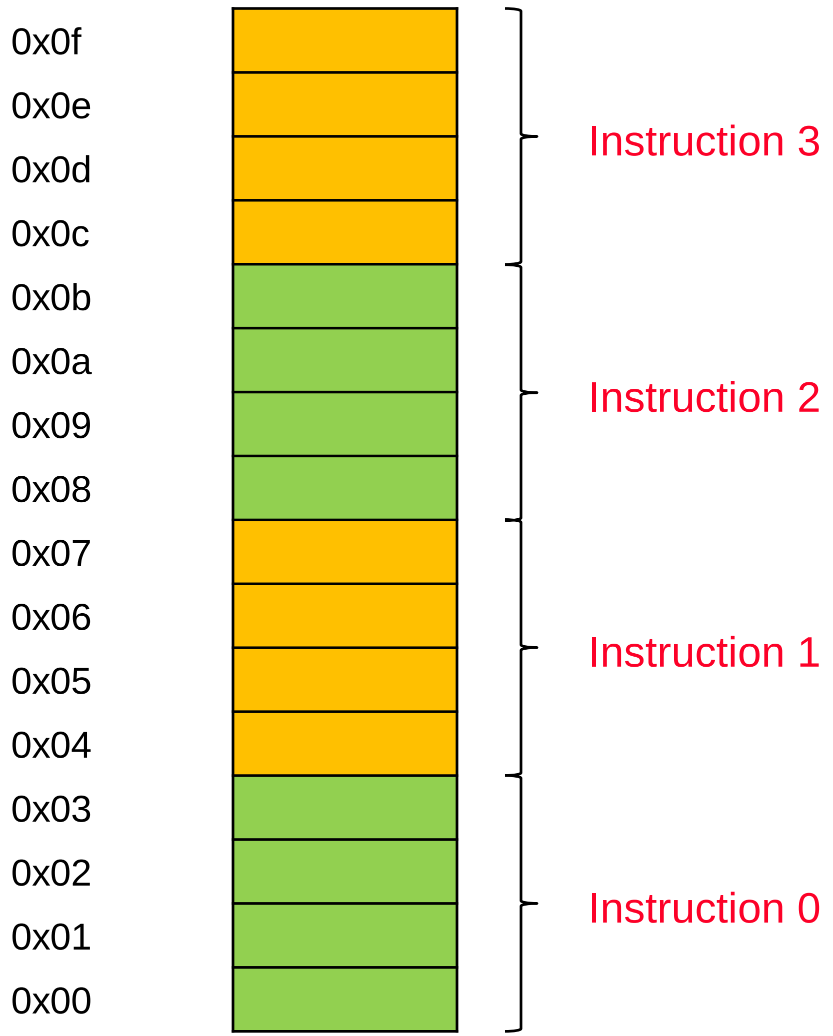
```
slti $t0, $s2, 15    #$t0 = 1 if $s2<15
```

- 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}

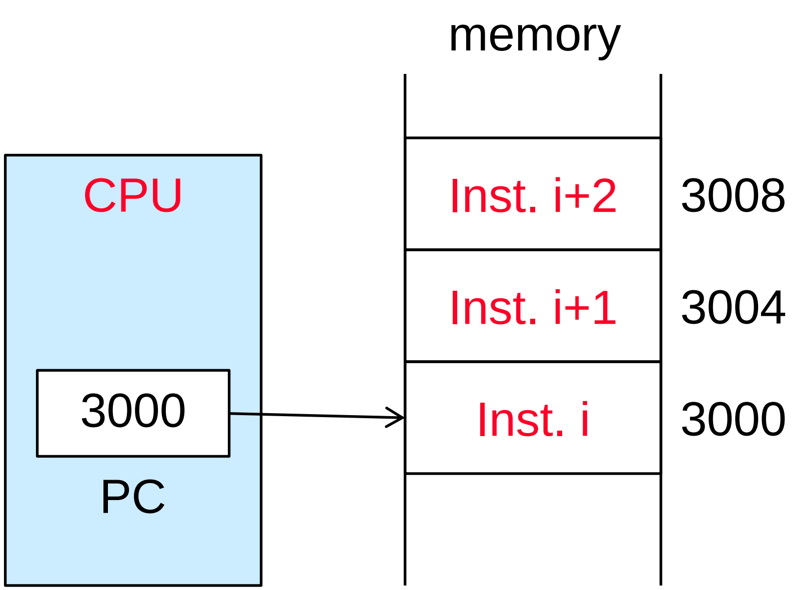
How MIPS executes program?



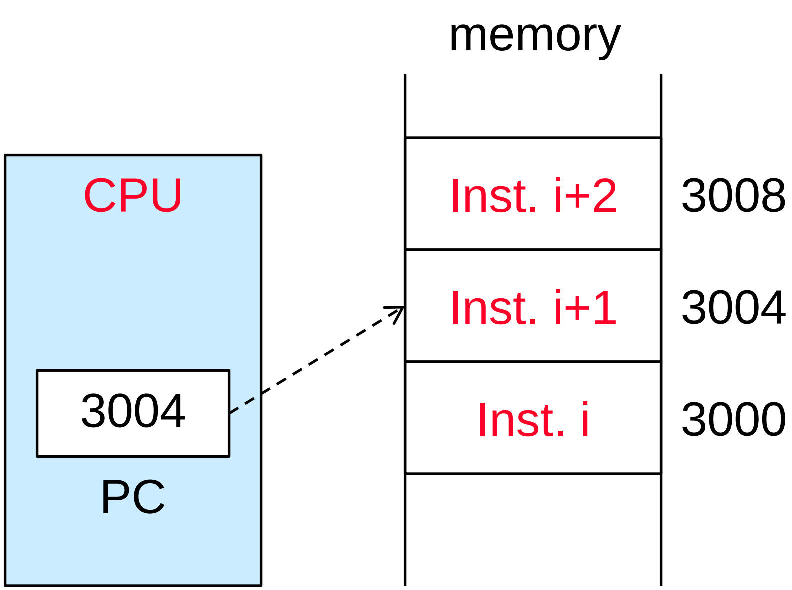
- Stored program in memory
- 1 instruction = 1 word
- Instruction cycle
  - Fetch
  - Decode
  - Execution

MIPS instruction cycle

- Program flow controlled by the PC register
  - **PC: Program Counter**
  - Specifies address of the next instruction to be fetched
  - Auto-increment after each instruction fetch
- MIPS instruction fetch cycle



Before fetching instruction i



After fetching instruction i

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:
```

Exercise

- How branch instruction is executed?

```
        slti $t0,$s1,5
        bne  $t0,$zero,0xFFFE
        addi $t1,$zero,3
        j    endif
        addi $t1,$zero,10
endif:...
```

How can CPU jump from here to the “else” label?

Example

- Write assembly code correspond to the following C code

```
for (i = 0; i < n; i++)
    sum = sum + A[i];
```

```
loop:
    add   s1,s1,1       #i=i+step
    add   t1,s1,s1      #t1=2*s1
    add   t1,t1,t1      #t1=4*s1
    add   t1,t1,s2      #t1 <- address of A[i]
    lw    t0,0(t1)      #load value of A[i] in t0
    add   s5,s5,t0      #sum = sum+A[i]
    bne   s1,s3,loop    #if i != n, goto loop
```

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: …                      #
```

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

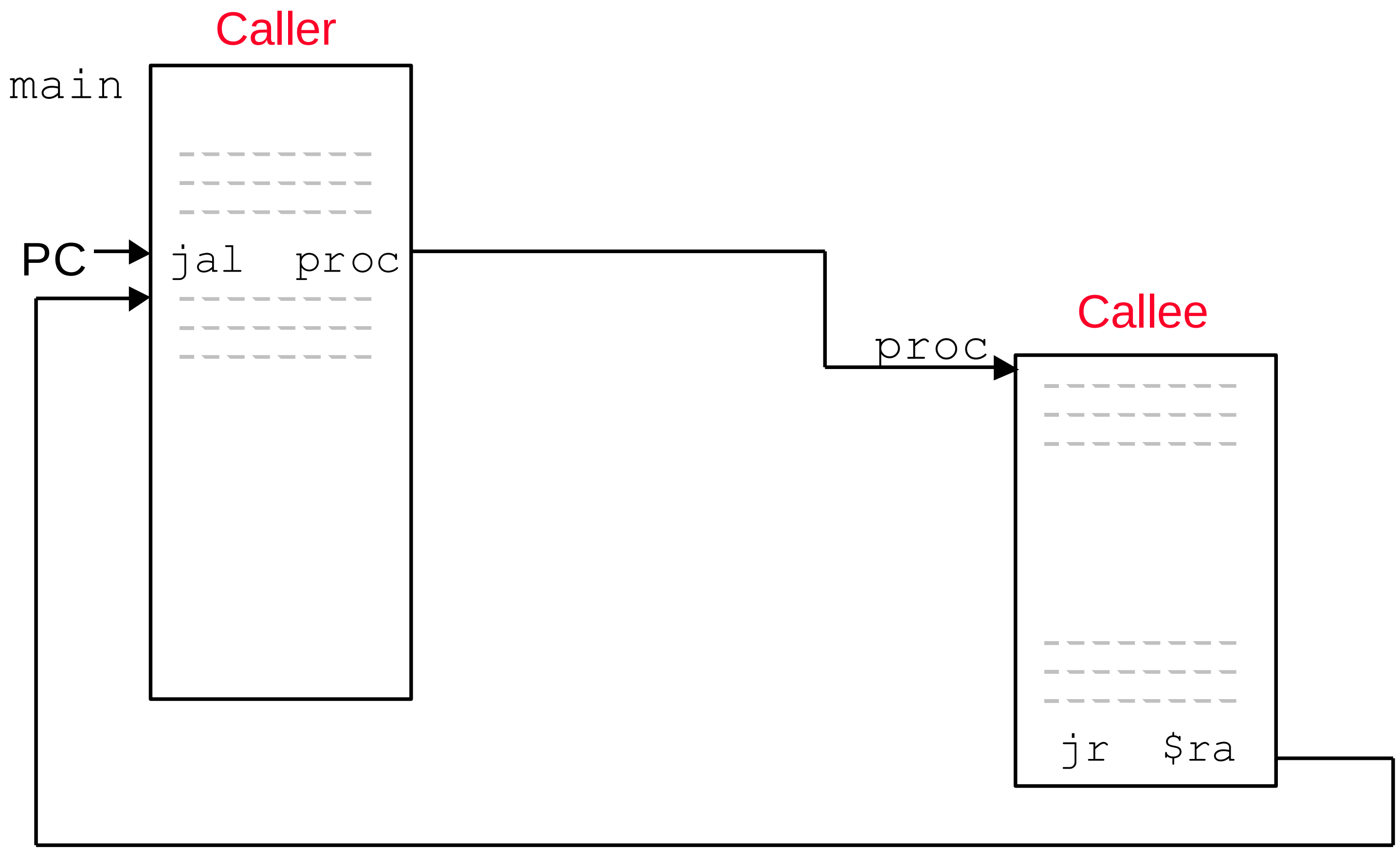
- Then can do procedure return with

```
jr    $ra                    #return
```

- Instruction format (R format):

0	31				0x08

Illustrating a Procedure Call



Instructions for accessing procedures

How to pass arguments and get return value?

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)

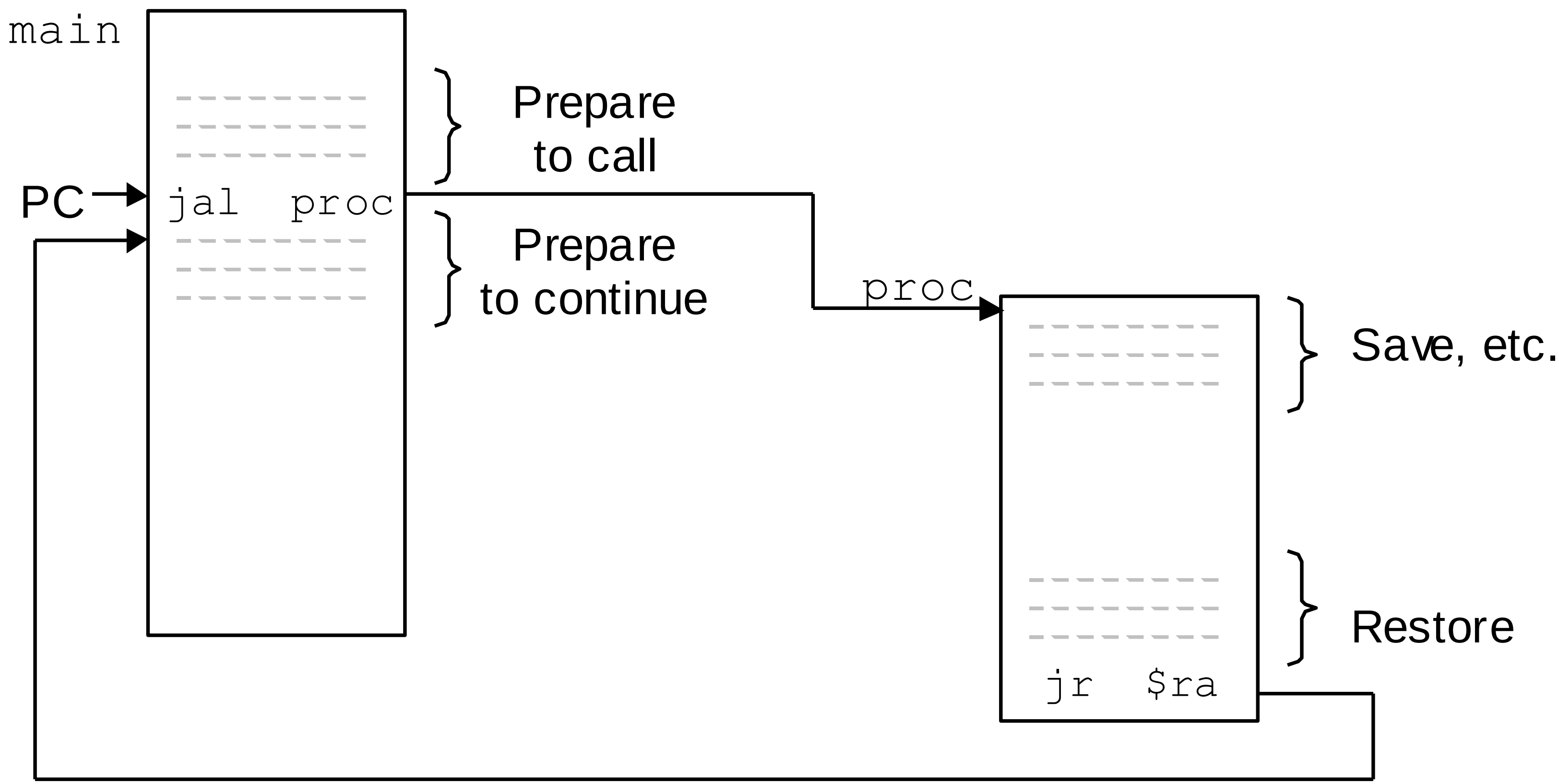
3. Callee acquires the storage resources needed

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

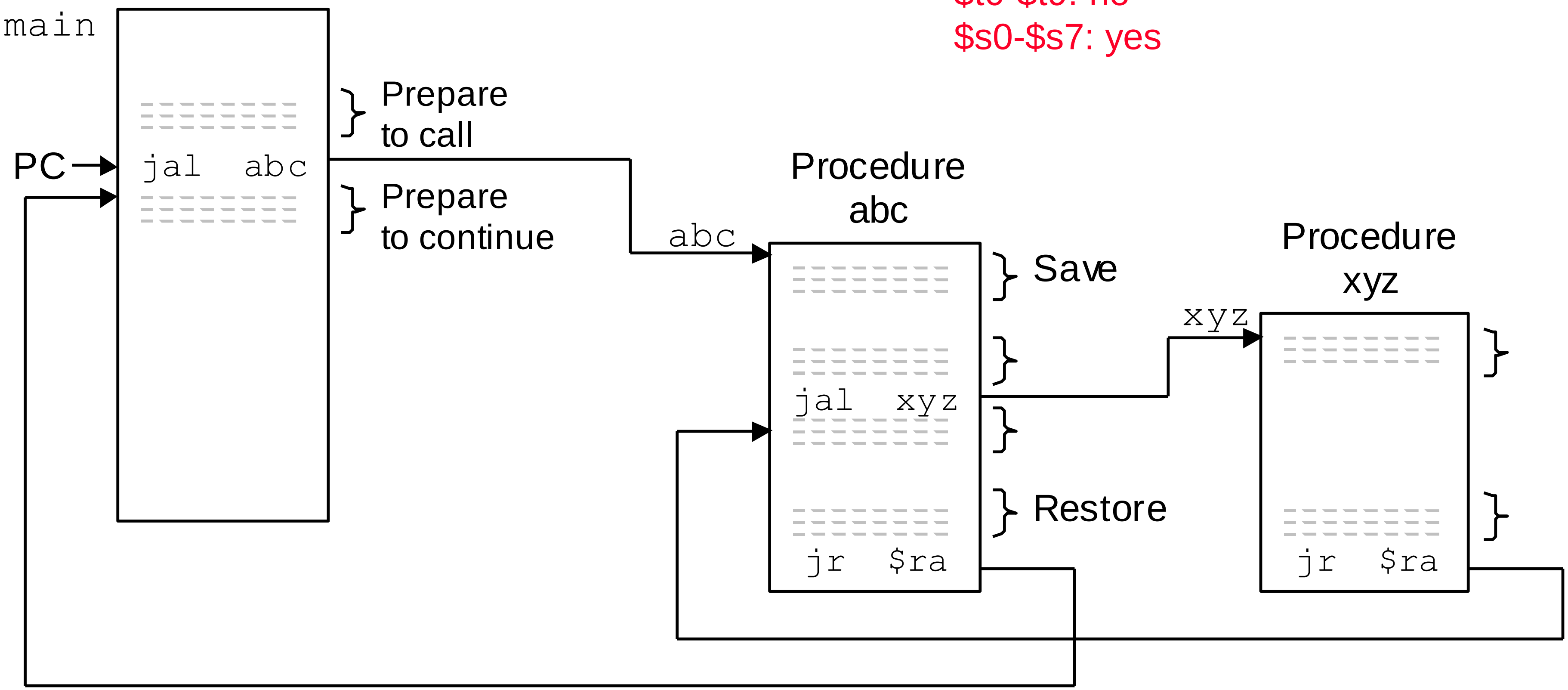


Relationship between the main program and a procedure.

How can main and proc share the same registers?

Nested Procedure Calls

Registers preserved on call
$t0-$t9: no
$s0-$s7: yes



Example of nested procedure calls.

Leaf procedures

- Procedures that do not call another proc.
- C code:

```
int leaf_example (int g, h, i, j)
{

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

  - g, h, i, j stored in $a0, $a1, $a2, $a3
  - f in $s0 (need to be saved)
  - $t0 and $t1 used for temporary data, also need to be saved
  - Result in $v0

Sample code

```
leaf_example:
        addi  $sp, $sp, -12     # room for 3 items
        sw    $t1, 8($sp)       # save $t1
        sw    $t0, 4($sp)       # save $t0
        sw    $s0, 0($sp)       # save $s0
        add   $t0, $a0, $a1     # $t0 = g+h
        add   $t1, $a2, $a3     # $t1 = i+j
        sub   $s0, $t0, $t1     # $s0 = (g+h)-(i+j)
        add   $v0, $s0, $zero   # return value in $v0
        lw    $s0, 0($sp)       # restore $s0
        lw    $t0, 4($sp)       # restore $t0
        lw    $t1, 8($sp)       # restore $t1
        addi  $sp, $sp, 12      # shrink stack
        jr    $ra               # return to caller
```

Stack usage

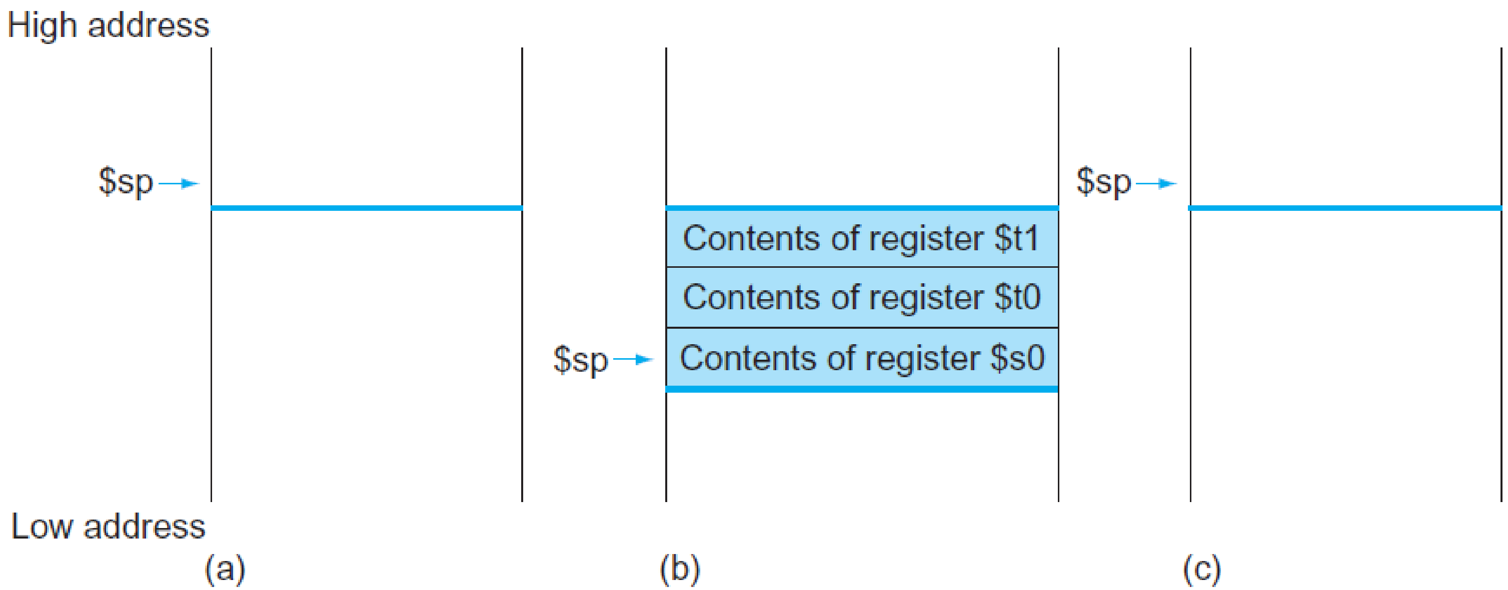


FIGURE 2.10 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.

Non-leaf procedure

- C code (recursive function):

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

  - n in $a0
  - Result in $v0

Sample code

```
fact:
      addi  $sp, $sp, -8        #2 items in stack
      sw    $ra, 4($sp)         #save return address
      sw    $a0, 0($sp)         #and current n
      slti  $t0, $a0, 1         #check base case
      beq   $t0, $zero, L1      #
      addi  $v0, $zero, 1       #value 1 for base case
      addi  $sp, $sp, 8         #then shrink stack
      jr    $ra                 #and return
L1:   addi  $a0, $a0, -1        #otherwise reduce n
      jal   fact                #then call fact again
      lw    $a0, 0($sp)         #restore n
      lw    $ra, 4($sp)         #and return address
      addi  $sp, $sp, 8         #shrink stack
      mul   $v0, $a0, $v0       #value for normal case
      jr    $ra                 #and return
```

Working with 32 bit immediates and addresses

- Operations that needs 32-bit literals
  - Loading 32-bit integers to registers
  - Loading variable addresses to registers
- I-format instructions only support 16-bit literals → combine two instructions
- Example: load the value 0x3D0900 into $s0

```
lui $s0, 0x003D           #$s0 ← 0x003D0000
ori #s0, $s0, 0x0900   #$s0 ← 0x003D0900
```

- Pseudo-instructions:
  - `la`
  - `li`

Working with 32 bit immediates and addresses

- Branching requires 32 bit addresses
- Conditional branch

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

  - PC-relative addressing:
    - Branch address = (PC + 4) + (imm * 4)
    - imm: offset from "following instruction" to branch address, in *words*

- Jump instructions

op	address
6 bits	26 bits

  - Pseudo-direct addressing
    - Branch address = $PC_{31\text{-}28}$: (imm * 4)

How to go further than these boundaries?

Accessing characters and string

- Accessing characters

  lb $s0, 0($s1) #load byte with sign-extension

  lbu $s0, 0($s1) #load byte with zero-extension

  sb $s0, 0($s1) #store LSB to memory

- String is accessed as array of characters
- Example: string copy

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

Accessing characters and string

```
#x and y are in $a0 and $a1, i in $s0
strcpy:
    addi $sp,$sp,-4          # adjust stack for 1 more item
    sw $s0, 0($sp)           # save $s0
    add $s0,$zero,$zero      # i = 0 + 0
L1: add $t1,$s0,$a1          # address of y[i] in $t1
    lbu $t2, 0($t1)          # $t2 = y[i]
    add $t3,$s0,$a0          # address of x[i] in $t3
    sb $t2, 0($t3)           # x[i] = y[i]
    beq $t2,$zero,L2         # if y[i] == 0, go to L2
    addi $s0, $s0,1          # i = i + 1
    j L1                     # go to L1
L2: lw $s0, 0($sp)           # y[i] == 0: end of string.
                             # Restore old $s0
    addi $sp,$sp,4           # pop 1 word off stack
    jr $ra                   # return
```

Example: Interchange sort function

```
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1)
    {
        for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j-=1)
        {
            swap(v,j);
        }
    }
}
void swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

Sorting function

Procedure body

```
swap: sll    $t1, $a1, 2           # reg $t1 = k * 4
      add    $t1, $a0, $t1         # reg $t1 = v + (k * 4)
                                   # reg $t1 has the address of v[k]
      lw     $t0, 0($t1)           # reg $t0 (temp) = v[k]
      lw     $t2, 4($t1)           # reg $t2 = v[k + 1]
                                   # refers to next element of v
      sw     $t2, 0($t1)           # v[k] = reg $t2
      sw     $t0, 4($t1)           # v[k+1] = reg $t0 (temp)
```

Procedure return

```
      jr     $ra                   # return to calling routine
```

Saving registers

```
sort:    addi   $sp,$sp, -20        # make room on stack for 5 registers
         sw     $ra, 16($sp)# save $ra on stack
         sw     $s3,12($sp)         # save $s3 on stack
         sw     $s2, 8($sp)# save $s2 on stack
         sw     $s1, 4($sp)# save $s1 on stack
         sw     $s0, 0($sp)# save $s0 on stack
```

Procedure body

Move parameters	`move $s2, $a0 # copy parameter $a0 into $s2 (save $a0)` `move $s3, $a1 # copy parameter $a1 into $s3 (save $a1)`

Outer loop	`move $s0, $zero# i = 0`
	`for1tst:slt $t0, $s0,$s3 #reg$t0=0if$s0Š$s3(iŠn)`
	`beq $t0, $zero, exit1# go to exit1 if $s0 Š $s3 (i Š n)`

Inner loop	`addi $s1, $s0, -1# j = i - 1`
	`for2tst:slti $t0, $s1,0 #reg$t0=1if$s1<0(j<0)`
	`bne $t0, $zero, exit2# go to exit2 if $s1 < 0 (j < 0)`
	`sll $t1, $s1, 2# reg $t1 = j * 4`
	`add $t2, $s2, $t1# reg $t2 = v + (j * 4)`
	`lw $t3, 0($t2)# reg $t3 = v[j]`
	`lw $t4, 4($t2)# reg $t4 = v[j + 1]`
	`slt $t0, $t4, $t3 # reg $t0 = 0 if $t4 Š $t3`
	`beq $t0, $zero, exit2# go to exit2 if $t4 Š $t3`

Pass parameters and call	`move $a0, $s2 # 1st parameter of swap is v (old $a0)`
	`move $a1, $s1 # 2nd parameter of swap is j`
	`jal swap # swap code shown in Figure 2.25`

Inner loop	`addi $s1, $s1, -1# j -= 1` `j for2tst # jump to test of inner loop`
Outer loop	`exit2: addi $s0, $s0, 1 # i += 1` `j for1tst # jump to test of outer loop`

Restoring registers

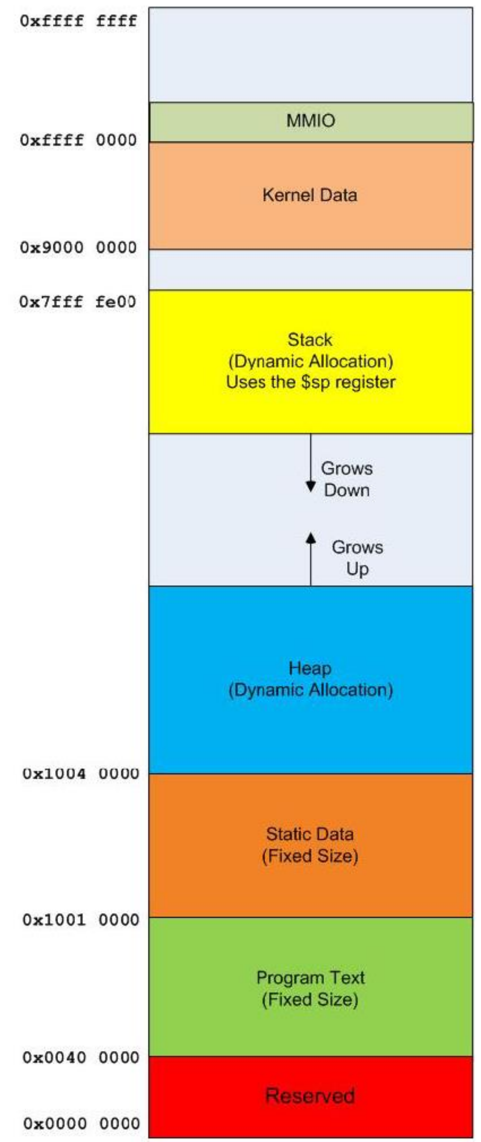
```
exit1:   lw     $s0, 0($sp)         # restore $s0 from stack
         lw     $s1, 4($sp)# restore $s1 from stack
         lw     $s2, 8($sp)# restore $s2 from stack
         lw     $s3,12($sp)         # restore $s3 from stack
         lw     $ra,16($sp)         # restore $ra from stack
         addi   $sp,$sp, 20         # restore stack pointer
```

Procedure return

```
         jr     $ra                 # return to calling routine
```

MIPS assembly programming: memory configuration

- Program text: stores machine code of program, declared with *.text*
- Static data: data segment, declared with *.data*
- Heap: for dynamic allocation
- Stack: for local variable and dynamic allocation via push/pop
- Kernel: for OS's use
- MIMO: memory mapped IO for accessing input/output devices



A sample program

```
int f, g, y;  // global
  variables

int main(void)
{
  f = 2;
  g = 3;
  y = sum(f, g);
  return y;
}

int sum(int a, int b) {
  return (a + b);
}
```

A sample program

```
.data
f: .word 0
g: .word 0
y: .word 0
.text
main:
   addi $sp, $sp, -4    # stack frame
   sw   $ra, 0($sp)     # store $ra
   addi $a0, $0, 2      # $a0 = 2
   sw   $a0, f          # f = 2
   addi $a1, $0, 3      # $a1 = 3
   sw   $a1, g          # g = 3
   jal  sum             # call sum
   sw   $v0, y          # y = sum()
   lw   $ra, 0($sp)     # restore $ra
   addi $sp, $sp, 4     # restore $sp
   li   $v0, 10
   syscall
sum:
   add  $v0, $a0, $a1   # $v0 = a + b
   jr   $ra             # return
```

A sample program

- The code segment: .text
- The static data segment: .data
- Data declaration:

name : storage_type value(s)

Example

n : **.word** 0 #create a word and init value of 0

c : **.byte** ‘a’

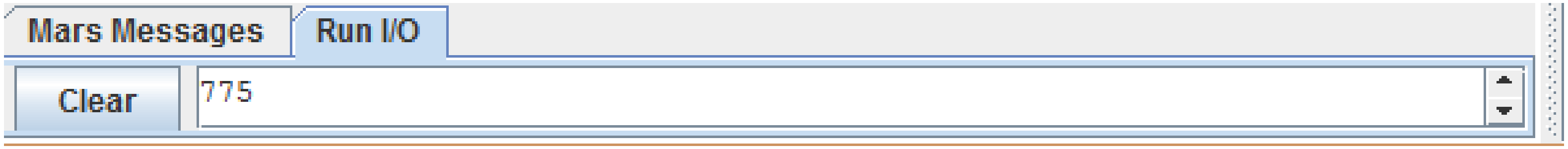
a1 : **.word** 10, 20

a2 : **.space** 100 #100 uninitialized bytes

Requesting kernel services: syscall

- **Print decimal** integer to standard output (the console).
- Argument(s):
  - \$v0 = 1
  - \$a0 = number to be printed
- Return value: none

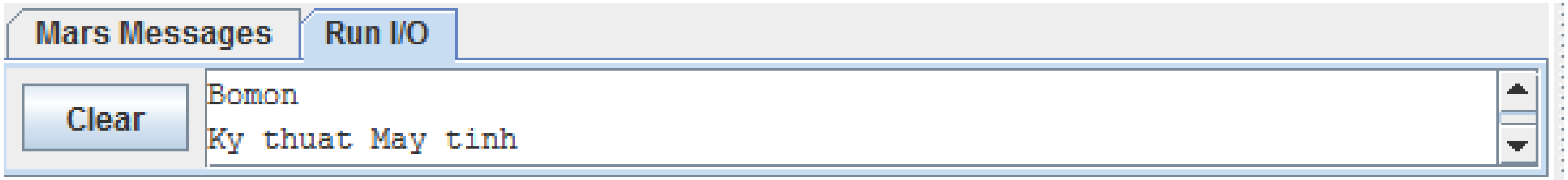
```
li   $v0, 1          # service 1 is print integer
li   $a0, 0x307      # the interger to be printed is 0x307
syscall              # execute
```



syscall

- **Print string** to standard output (the console).
- Argument(s)
  - $v0 = 4
  - $a0 = address of null terminated string to print
- Return value: none

```
.data
Message: .asciiz "Bomon \nKy thuat May tinh"
.text
   li  $v0, 4
   la  $a0, Message
   syscall
```



syscall

- **Read integer** from standard input (the console).
- Argument
  - \$v0 = 5
- Return value
  - \$v0 = contains integer read

```
li      $v0, 5
syscall
```

syscall

- **Read string** from standard input
- Argument(s):
  - $v0 = 8
  - $a0 = address of input buffer
  - $a1 = maximum number of characters to read
- Return value: none
- Note: for specified length n, string can be no longer than n-1.
  - If less than that, adds newline to end.
  - In either case, then pads with null byte
- String can be declared with *.space*

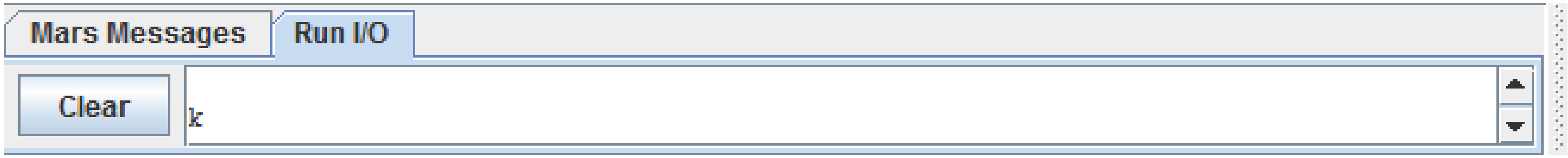
syscall

```
.data
Message: .space 100    # string with max len = 99
.text
   li  $v0, 8
   la  $a0, Message
   li  $a1, 100
   syscall
```

syscall

- **Print** a **character** to standard output.
- Arguments
  - $v0 = 11
  - $a0 = character to print (at LSB)
- Return value: none

```
li  $v0, 11
li  $a0, 'k'
syscall
```



syscall

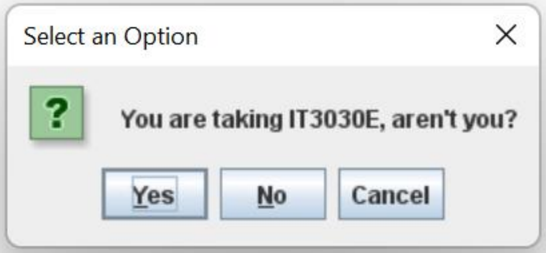
- **Read** a **character** from standard input.
- Argument(s):
  - $v0 = 12
- Return value:
  - $v0 contains the character read

syscall

- ConfirmDialog
- Argument(s):
  - $v0 = 50
  - $a0 = address of the null-terminated message string
- Return value: $a0 = value of selected option

  0: Yes 1: No 2: Cancel

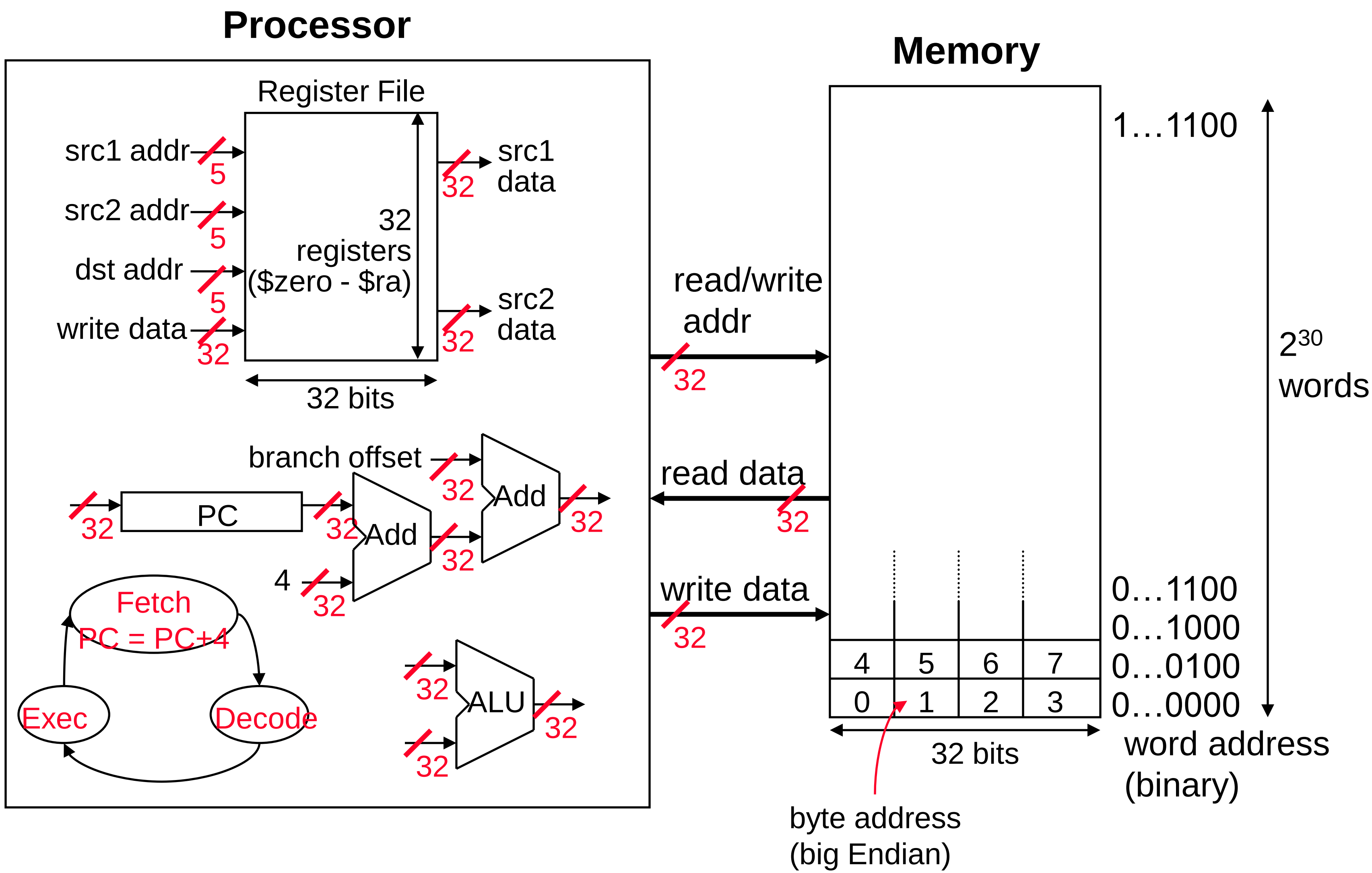
```
.data
Message: .asciiz "You are taking IT3030E, aren't you?"
.text
   li   $v0, 50
   la   $a0, Message
   syscall
```



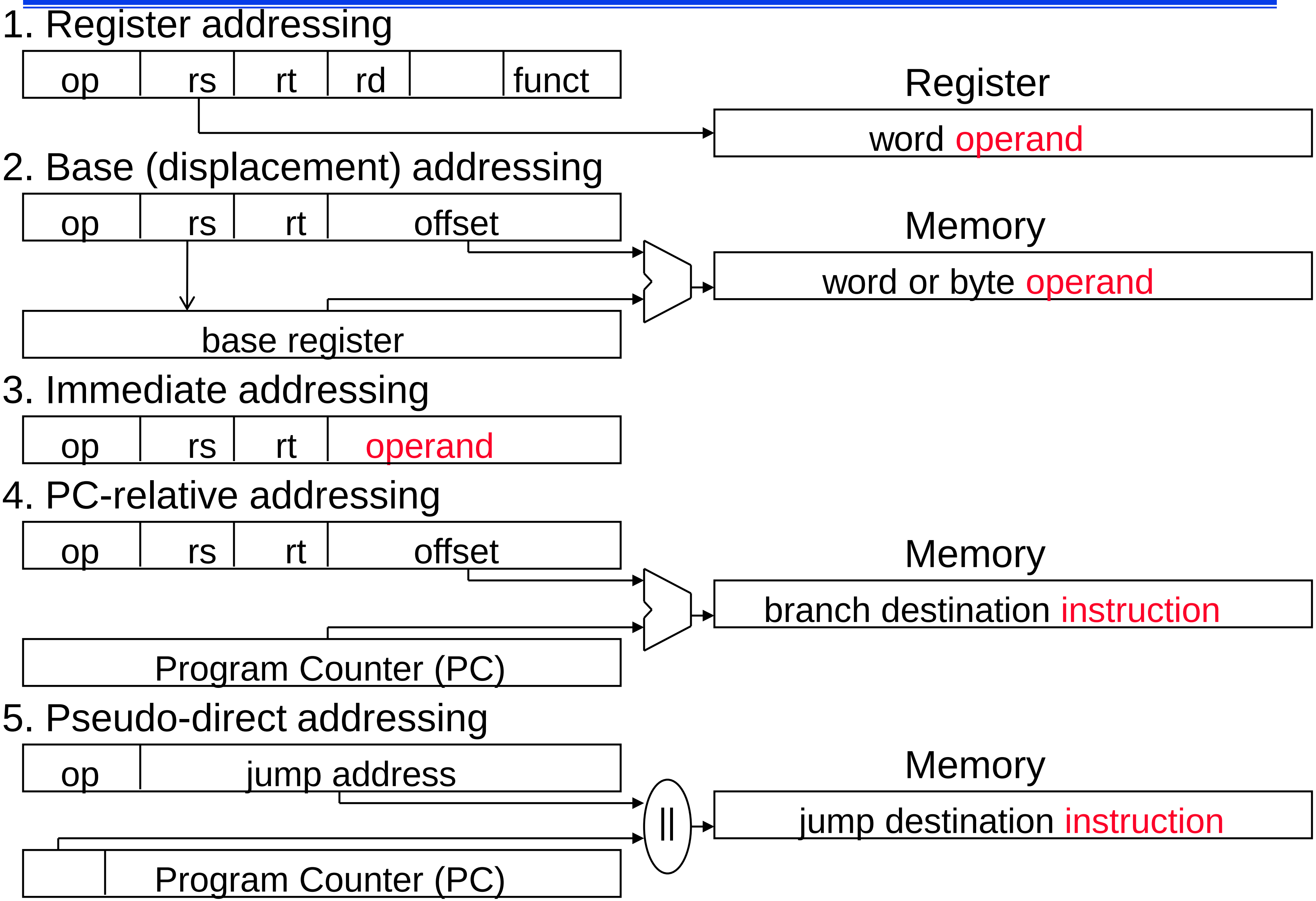
syscall

- exit: terminate the program
- Argument
  - $v0 = 10
- Return value: none

MIPS Organization



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)