Computer Architecture & Assembly Language

Chapter 6

ASSEMBLY LANGUAGE PROGRAMMING

VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY
UNIVERSITY OF SCIENCE

Thái Hùng Văn
thvan@fit.hcmus.edu.vn

X86

Instruction and Directive

- Consider the following statement:

  Start: MOV CX,5 ; initialize counter

  *** Instruction:** The operation and operands MOV CX , 5

  *** Directive:** The **label** (Start:) and the **comment**

- Some important instructions: MOV, ADD, SUB, CMP, JMP, J<condition>, INT,..
- Some important directives except labels and comments: .MODEL, .STACK, .DATA, .CODE, PROC, ENDP, END, DB/DW/DD/.., ...

The most basic x86 Instructions

- **MOV** <Dest>, <Source> : copies content of <Source> to <Dest>
- **ADD/SUB** < Destination>, <Source> : add /subtrast the content of <Dest> with <Source> , then copy the result to <Dest>
- **CMP** <Source1>, <Source2> : compare <Source1> with <Source2>
- **JMP/Jxxx** <Label> : jump /jump_if<**xxx**> to <Label>
- **INT** <InterruptNo> : call pre-defined procedure <*Interrupt Number*>

Instruction Arguments

A typical instruction has 2 operands

- **target operand** (left)
- **source operand** (right)

3 kinds of operands exists

- **immediate** : value
- **register** : AX, EBX, DL, etc.
- **memory location** : variable or pointer

Examples:

mov ax, 2

target operand
register

source operand
immediate

mov [buffer], ax

target operand
memory location

source operand
register

! Note that x86 processor does not allow both operands be memory locations.

~~mov [var1],[var2]~~

Move Instruction

***mov reg8/mem8**(16,32)**,reg8/imm8**(16,32)*

(copies content of register / immediate (source) to register / memory location)

***mov reg8**(16,32)**,reg8/mem8**(16,32)*

(copies content of register / memory location (source) to register (destination))

operands have to be of the same size

Examples:

mov eax, 2334AAFFh — reg32, imm32

mov [buffer], ax — mem16, reg16

*mov **word** [var], 2* — mem16, imm16

*Note that Asembler doesn't remember the types of variables you declare, and so you must explicitly code mov **word** [var], 2.*

Basic Arithmetical Instructions

- *<instruction> reg8/mem8*(16,32)*,reg8/imm8*(16,32)
  (source - register / immediate, destination- register / memory location)
- *<instruction> reg8*(16,32)*,reg8/mem8*(16,32)
  (source - register / immediate, destination - register / memory location)

ADD - add integers

Example:
add AX, BX ;(AX gets a value of AX+BX)

SUB - subtract integers

Example:
sub AX, BX ;(AX gets a value of AX-BX)

ADC - add integers with carry
(value of Carry Flag)

Example:
adc AX, BX ;(AX gets a value of AX+BX+CF)

SBB - subtract with borrow
(value of Carry Flag)

Example:
sbb AX, BX ;(AX gets a value of AX-BX-CF)

Basic Arithmetical Instructions

<instruction> reg8/mem8_(16,32)_

(source / destination - register / memory location)

INC - increment integer

Example:

inc AX ;(AX gets a value of AX+1)

DEC - increment integer

Example:

dec byte [buffer] ;([buffer] gets a value of [buffer] -1)

Basic Logical Instructions

<instruction> reg8/mem8(16,32)
(source / destination - register / memory location)

NOT – one's complement negation – inverts all the bits

Example:
mov al, 11111110_b
not al ;(AL gets a value of 00000001_b)
;(11111110_b + 00000001_b = 11111111_b)

NEG – two's complement negation - inverts all the bits, and adds 1

Example:
mov al, 11111110_b
neg al ;(AL gets a value of not(11111110_b)+1=00000001_b+1=00000010_b)
;(11111110_b + 00000010_b = 100000000_b = 0)

Basic Logical Instructions

<instruction> reg8/mem8(16,32),reg8/imm8(16,32)
(source - register / immediate, destination- register / memory location)

<instruction> reg8(16,32),reg8/mem8(16,32)
(source - register / immediate, destination - register / memory location)

OR – bitwise or – bit at index i of the destination gets '1' if bit at index i of source or destination are '1'; otherwise '0'

Example:

mov al, 11111100 $_b$
mov bl, 00000010$_b$
or AL, BL ;(AL gets a value 11111110$_b$)

AND– bitwise and – bit at index i of the destination gets '1' if bits at index i of both source and destination are '1'; otherwise '0'

Example:

or AL, BL ;(with same values of AL and BL as in previous example, AL gets a value 0)

Compare Instruction

CMP – Compare Instruction – compares integers

CMP performs a 'mental' subtraction - **affects the flags** as if the subtraction had taken place, but does not store the result of the subtraction.

cmp reg8/mem8(16,32),reg8/imm8(16,32)
(source - register / immediate, destination- register / memory location)

cmp reg8(16,32),reg8/mem8(16,32)
(source - register / immediate, destination - register / memory location)

Examples:

mov al, 11111100_b
mov bl, 00000010_b
cmp al, bl ;(ZF (zero flag) gets a value 0)

mov al, 11111100_b
mov bl, 11111100_b
cmp al, bl ;(ZF (zero flag) gets a value 1)

UnConditional Jump Instruction

JMP <Label>

JMP tells the processor that the next instruction to be executed is located at the label that is given as part of jmp instruction.

Example:

```
mov eax, 1
inc_again:
    inc eax
    jmp inc_again
    mov ebx, eax
```

this is infinite loop !

this instruction is never reached from this code

Conditional Jump Instructions

J<Condition> <Label>

- execution is transferred to the target instruction only if the specified condition is satisfied
- usually, the condition being tested is the result of the last arithmetic or logic operation

Example:

```
mov eax, 1
inc_again:
    inc eax
    cmp eax, 10
    je end_of_loop      ; if eax = = 10, jump to end_of_loop
    jmp inc_again       ; go back to loop
end_of_loop:
```

```
mov eax, 1
inc_again:
    inc eax
    cmp eax, 10
    jne inc_again       ; if eax ! = 10, go back to loop
```

Conditional Jump Instructions

Instruction	Description	Flags
JO	Jump if overflow	OF = 1
JNO	Jump if not overflow	OF = 0
JS	Jump if sign	SF = 1
JNS	Jump if not sign	SF = 0
JE /JZ	Jump if equal / Jump if zero	ZF = 1
JNE/JNZ	Jump if not equal / Jump if not zero	ZF = 0
JB/JNAE/JC	Jump if below / Jump if not above or equal / Jump if carry	CF = 1
JNB/JAE/JNC	Jump if not below /Jump if above or equal /Jump if not carry	CF = 0
JBE / JNA	Jump if below or equal / Jump if not above	CF = 1 or ZF = 1
JA / JNBE	Jump if above / Jump if not below or equal	CF = 0 and ZF = 0
JL / JNGE	Jump if less / Jump if not greater or equal	SF <> OF
JGE / JNL	Jump if greater or equal / Jump if not less	SF = OF
JLE / JNG	Jump if less or equal / Jump if not greater	ZF = 1 or SF <> OF
JG / JNLE	Jump if greater / Jump if not less or equal	ZF = 0 and SF = OF
JP / JPE	Jump if parity / Jump if parity even	PF = 1
JNP / JPO	Jump if not parity / Jump if parity odd	PF = 0
JCXZ / JECXZ	Jump if CX register is 0 / Jump if ECX register is 0	CX=0 / ECX=0

Declare Initialized Data

D<size> [InitialValue]

Pseudo-instruction	<size> filed	<size> value
DB	byte	1 byte
DW	word	2 bytes
DD	double word	4 bytes
DQ	quad word	8 bytes
DT	ten byte	10 bytes
DDQ	double quadword	16 bytes
DO	octoword	16 bytes

Examples:

var: db 0x55 ; define a variable 'var' of size byte, initialized by 0x55

var: db 0x55,0x56,0x57 ; three bytes in succession

var: db 'a' ; character constant 0x61 (ascii code of 'a')

var: db 'hello',13,10,'$' ; string constant

var: dw 0x1234 ; 0x34 0x12

var: dw 'A' ; 0x41 **0x00 – complete to word**

var: dw 'AB' ; 0x41 0x42

var: dw 'ABC' ; 0x41 0x42 0x43 **0x00 – complete to word**

var: dd 0x12345678 ; 0x78 0x56 0x34 0x12

Arrays

- Any consecutive storage locations of the same size can be called an array

```
X DW 40CH,10B,-13,0 ;  Components of X are at X, X+2, X+4, X+6
Y DB 'This is an array' ; Components of Y are at Y, Y+1, …, Y+15
Z DD -2019, FFFFFh, 100b ; Components of Z are at Z, Z+4, Z+8
```

- **DUP** allows a sequence of storage locations to be defined
- Only used as an operand of a define directive

```
DB    40   DUP (?)      ; 40 Bytes, uninitialized
Table1   DW   10   DUP (?) ; 10 Words, uninitialized
Mes   DB   3   DUP ('ole') ; 9 Bytes, initialized as oleoleole
a   DB   30   DUP ('?') ; 30 Bytes, each initialized to ?
DD 9 DUP (0)    ; 9 DWords, initialized as 0
Matrix   DW   3 DUP (5 DUP (?)) ; defines a 3x5 matrix
```

Segment Directives

There are 5 most Directives : .Model, .Code, .Data, .Stack, End

.MODEL

- **.Model statement followed by the size of the memory system designates the Memory Model.**

.CODE

- **Designates the beginning of the CODE segment in the program.**

END

- **Designates the ending of the CODE segment in the program.**

.DATA

- **Designates the beginning of the DATA segment in the program**

.STACK

- **Defines STACK segment in the program.**
- **Syntax : .STACK [memory-size] ;memory-size is optional**
- **Default memory size for stack segment is 1KB.**
- **Initializes Stack Segment(SS), Stack Pointer(SP) and Base Pointer(BP).**

Memory Models

- .Model *memory_model*
  - tiny: code+data <= 64K (.com program)
  - small: code<=64K, data<=64K, one of each
  - medium: data<=64K, one data segment
  - compact: code<=64K, one code segment
  - large: multiple code and data segments
  - huge: allows individual arrays to exceed 64K
  - flat: no segments, 32-bit addresses, protected mode only (80386 and higher)

Memory Organization

- The assembler uses two basic formats for developing software:

1. **Using Memory Models:**
   - Memory models are unique to the MASM assembler program.
   - The TASM assembler also uses memory models, but they differ somewhat from the MASM models.
   - The models are easier to use for simple tasks.

2. **Using full-segment definitions:**
   - The full-segment definitions are common to most assemblers, including the Intel assembler, and are often used for software development.
   - The full-segment definitions offer better control over the assembly language task and are recommended for complex programs.

Some form of Assembly Program

```
.Model Small   ;Select a memory model
.Stack 100h    ;Define the stack size
.Code
        ;code
End
```

```
.Model Small
.Stack 100h
.Data
    ;  Declare variables
.Code
Main proc
    ;code
Main endp
    ;other procs
End Main
```

Standard I/O

- DOS function calls are used for Standard Input /Output in Assembly language (8086).
- To use a DOS function call in a DOS program,
  1. **Place the function number in AH (8 bit register) and other data that might be necessary in other registers.**
  2. **Once everything is loaded, execute the INT 21h instruction to perform the task.**
- After execution of a DOS function, it may return results in some specific registers.

E.g:

```
MOV AH,01H        ;load DOS function number in AH
INT 21H           ;access DOS
                  ;returns with AL = ASCII key code
```

Standard I/O

- AH=1 # Int 21h: Read the Keyboard
  - This function waits until a character is input from the keyboard.
  - Returns ASCII key code of character in AL register.
- AH=2 # Int 21h : Write to Standard Output device
  - This function displays single character on the video display.
  - ASCII code of the character to be displayed must be loaded in DL register

E.g
```
MOV DL, 'A'      ;load ASCII key code of Character 'A' in DL
MOV AH, 2        ;load DOS function number in AH
INT 21h          ;access DOS Interrupt 21h procedure
```

- AH=2 # Int 21h : Display a character String
  - This function displays a character string on the video display.
  - The character string must end with '$' 24H). The string can be of any length and may contain control characters such as CR and LF.
  - DX must contain address of the character string.

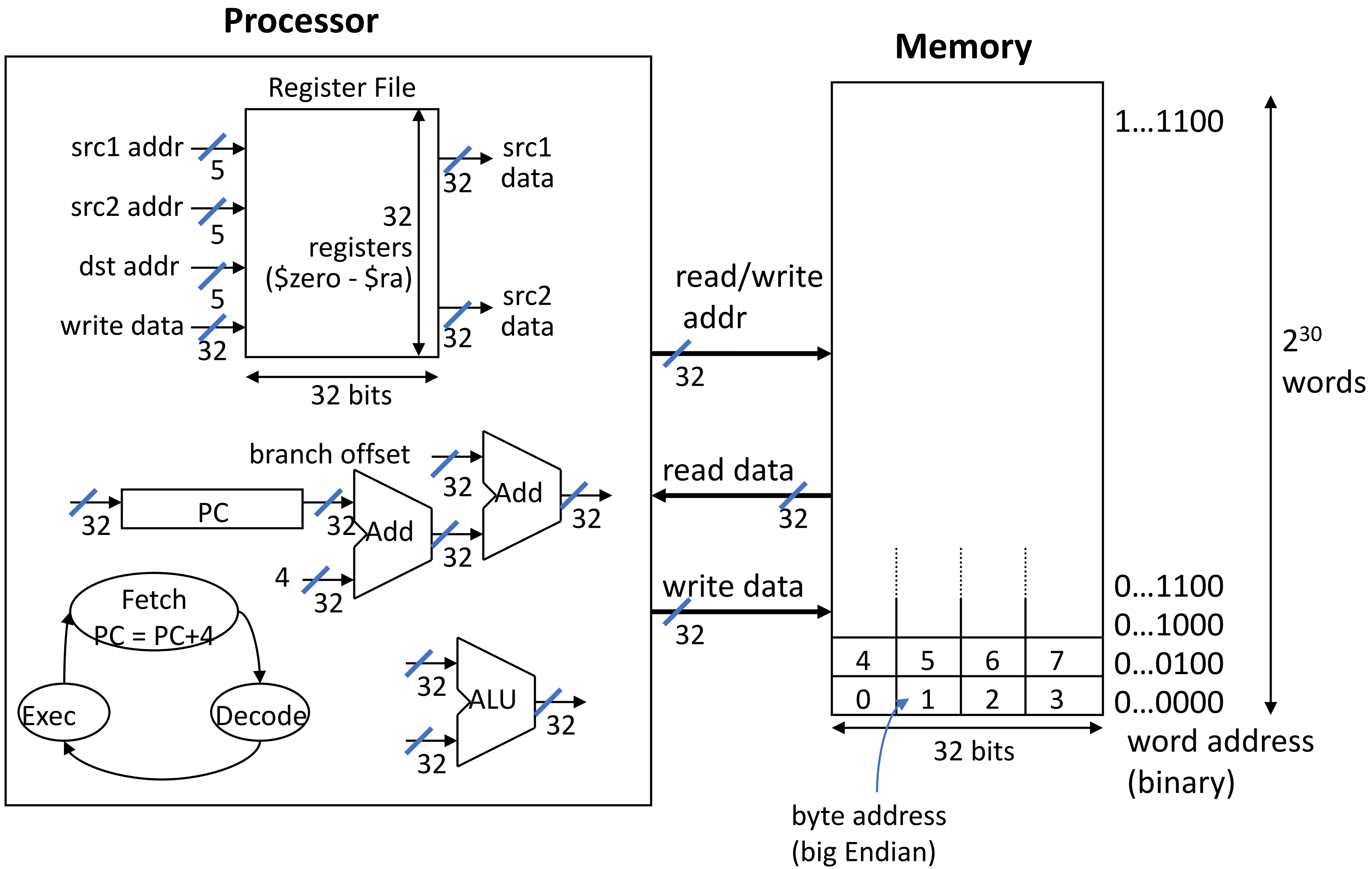
E.g
```
Buf DB "Hello World$"      ;define character string
MOV DX, offset Buf         ;load address of the string in DX
MOV AH, 9
INT 21h
```

MIPS

MIPS (RISC) Design Principles

- Simplicity favors regularity
  - fixed size instructions – 32-bits
  - small number of instruction formats
  - opcode always the first 6 bits
- Good design demands good compromises
  - three instruction formats
- Smaller is faster
  - limited instruction set
  - compromise on number of registers in register file
  - limited number of addressing modes
- Make the common case fast
  - arithmetic operands from the register file (load-store machine)
  - allow instructions to contain immediate operands

MIPS Organization



MIPS R3000 Instruction Set Architecture (ISA)

- Instruction Categories
  - Computational
  - Load/Store
  - Jump and Branch
  - Floating Point
    - coprocessor
  - Memory Management
  - Special

Registers

R0 - R31

PC

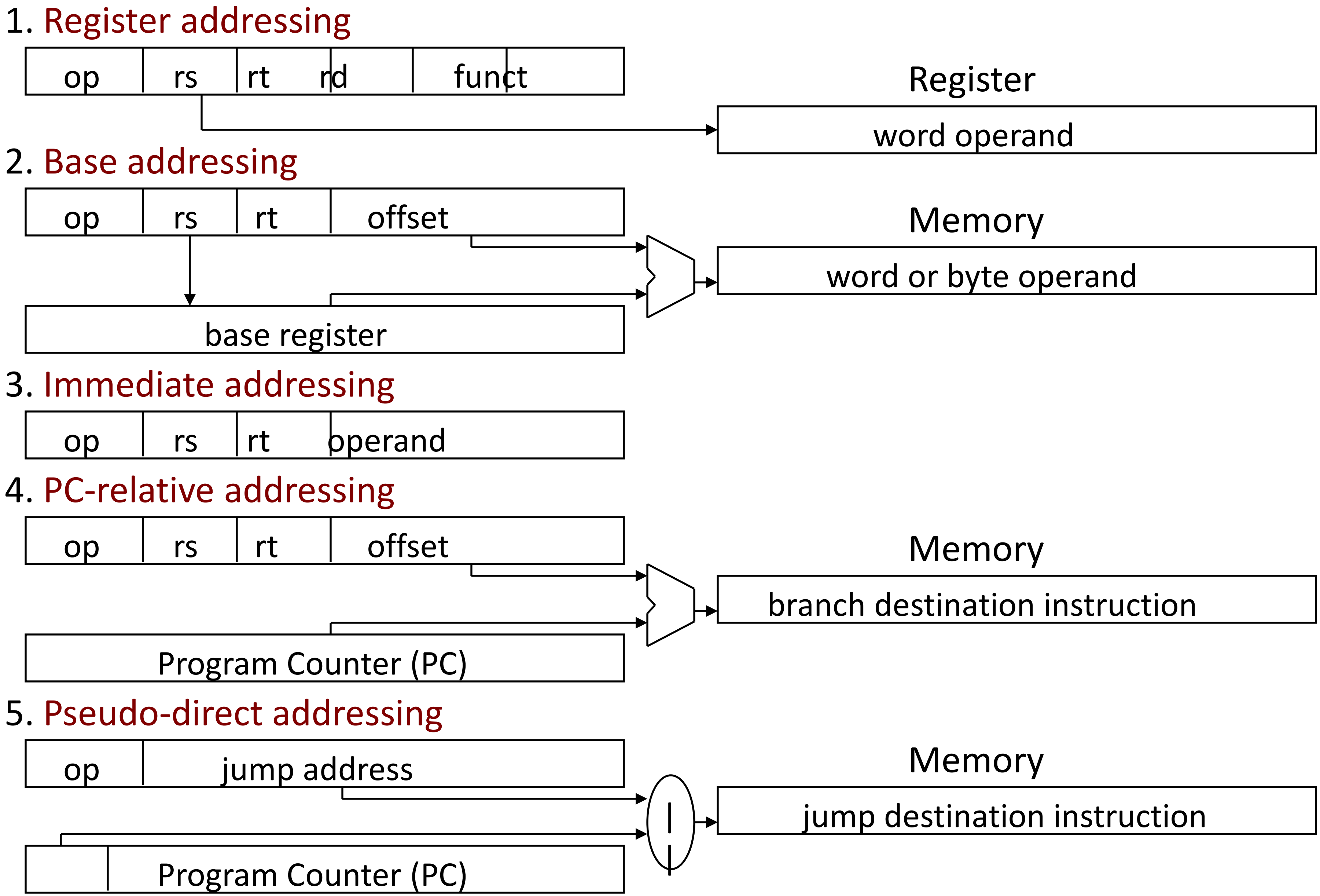
HI

LO

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

MIPS Addressing Modes



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

A MIPS Sample Program

C program

```c
#include <stdio.h>

int
main (int argc, char *argv[])
{
    int i;
    int sum = 0;

    for (i = 0; i <= 100; i = i + 1) sum = sum + i * i;
    printf ("The sum from 0 .. 100 is %d\n", sum);
}
```

MIPS Assembly Program

```
        .text
        .align  2
        .globl  main
main:
        subu    $sp, $sp, 32
        sw      $ra, 20($sp)
        sd      $a0, 32($sp)
        sw      $0,  24($sp)
        sw      $0,  28($sp)
loop:
        lw      $t6, 28($sp)
        mul     $t7, $t6, $t6
        lw      $t8, 24($sp)
        addu    $t9, $t8, $t7
        sw      $t9, 24($sp)
        addu    $t0, $t6, 1
        sw      $t0, 28($sp)
        ble     $t0, 100, loop
        la      $a0, str
        lw      $a1, 24($sp)
        jal     printf
        move    $v0, $0
        lw      $ra, 20($sp)
        addu    $sp, $sp, 32
        jr      $ra


        .data
        .align  0
str:
        .asciiz "The sum from 0 .. 100 is %d\n"
```

Machine code Memory Dump

```
00100111101111011111111111100000
10101111101111110000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
10001111101110000000000000011000
00000001110011100000000000011001
00100101110010000000000000000001
00101001000000010000000001100101
10101111101010000000000000011100
00000000000000000111100000010010
00000011000011111100100000100001
00010100001000001111111111110111
10101111101110010000000000011000
00111100000001000001000000000000
10001111101001010000000000011000
00001100000100000000000011101100
00100100100001000000010000110000
10001111101111110000000000010100
00100111101111010000000000100000
00000011111000000000000000001000
00000000000000000001000000100001
```

Reverse Engineered Code

```
addiu   $29, $29, -32
sw      $31, 20($29)
sw      $4,  32($29)
sw      $5,  36($29)
sw      $0,  24($29)
sw      $0,  28($29)
lw      $14, 28($29)
lw      $24, 24($29)
multu   $14, $14
addiu   $8,  $14, 1
slti    $1,  $8, 101
sw      $8,  28($29)
mflo    $15
addu    $25, $24, $15
bne     $1,  $0, -9
sw      $25, 24($29)
lui     $4,  4096
lw      $5,  24($29)
jal     1048812
addiu   $4,  $4, 1072
lw      $31, 20($29)
addiu   $29, $29, 32
jr      $31
move    $2,  $0
```

Supporting Procedures

Process:

- Place parameters where procedure can access them
- Transfer control to the procedure
- Acquire storage resources for the procedure
- Perform the task
- Place result where calling program can access it
- Return control to calling program

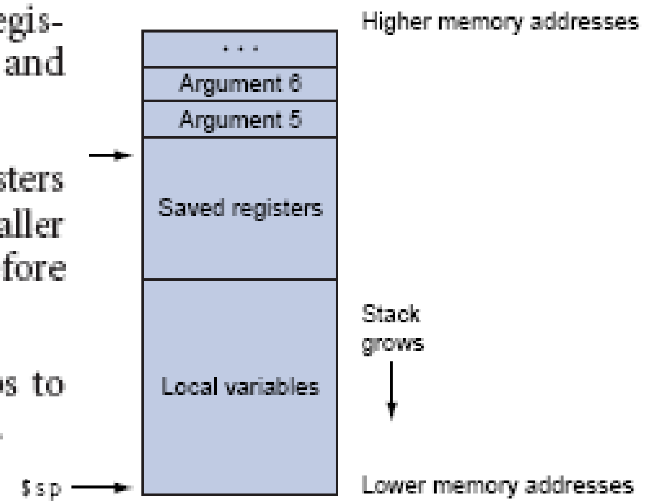
Support structure:

- $a0-$a3 argument passing registers
- $v0-$v1 return value registers
- $ra return address register

Procedure Call Convention

First the caller must:

1. Pass arguments. By convention, the first four arguments are passed in registers `$a0`–`$a3`. Any remaining arguments are pushed on the stack and appear at the beginning of the called procedure's stack frame.
2. Save caller-saved registers. The called procedure can use these registers (`$a0`–`$a3` and `$t0`–`$t9`) without first saving their value. If the caller expects to use one of these registers after a call, it must save its value before the call.
3. Execute a `jal` instruction (see Section 2.7 of Chapter 2), which jumps to the callee's first instruction and saves the return address in register `$ra`.



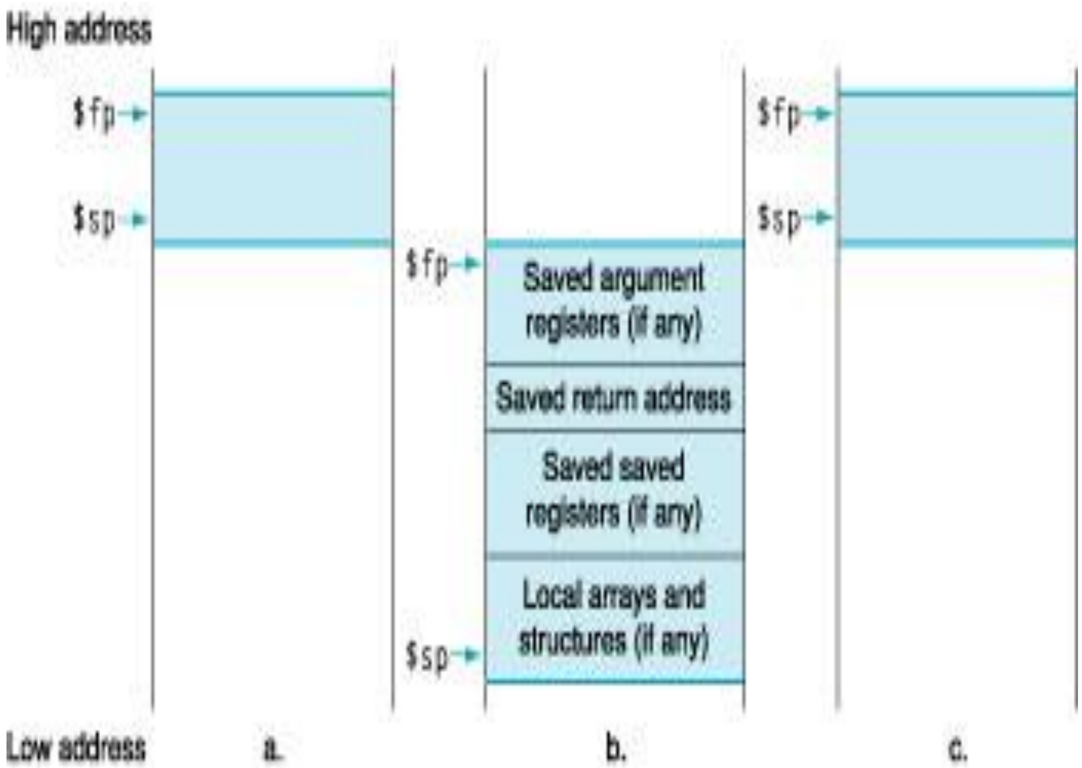
Before a called routine starts running, it must take the following steps to set up its stack frame:

1. Allocate memory for the frame by subtracting the frame's size from the stack pointer.
2. Save callee-saved registers in the frame. A callee must save the values in these registers (`$s0`–`$s7`, `$fp`, and `$ra`) before altering them since the caller expects to find these registers unchanged after the call. Register `$fp` is saved by every procedure that allocates a new stack frame. However, register `$ra` only needs to be saved if the callee itself makes a call. The other callee-saved registers that are used also must be saved.
3. Establish the frame pointer by adding the stack frame's size minus 4 to `$sp` and storing the sum in register `$fp`.

Finally, the callee returns to the caller by executing the following steps:

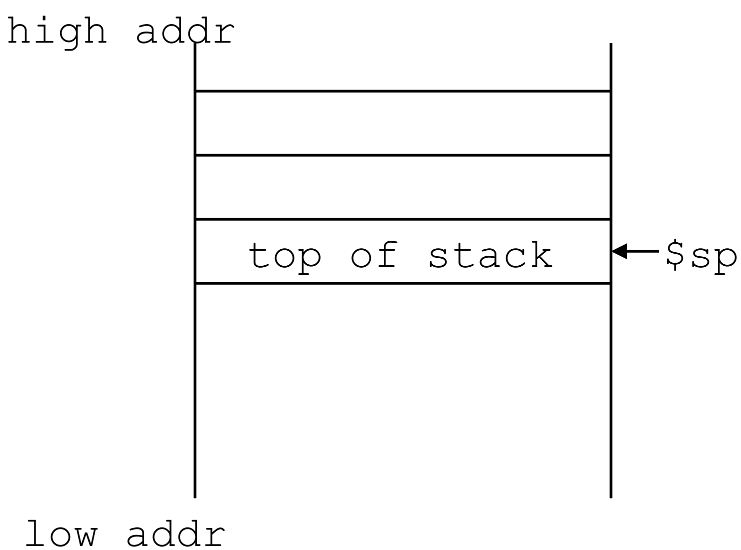
1. If the callee is a function that returns a value, place the returned value in register `$v0`.
2. Restore all callee-saved registers that were saved upon procedure entry.
3. Pop the stack frame by adding the frame size to `$sp`.
4. Return by jumping to the address in register `$ra`.

MIPS 32 Context Frames



Calling Procedure: Spilling Registers

- What if the callee needs more registers? What if the procedure is recursive?
  - uses a stack – a last-in-first-out queue – in memory for passing additional values or saving (recursive) return address(es)



- $sp, is used to address the stack (which "grows" from high address to low address)
  - add data onto the stack – push

    $sp = $sp – 4 data on stack at new $sp

  - remove data from the stack – pop

    data from stack at $sp
    $sp = $sp + 4

MIPS Arithmetic Instructions

- MIPS assembly language arithmetic statement

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

- Each arithmetic instruction performs only one operation

- Each arithmetic instruction fits in 32 bits and specifies exactly three operands

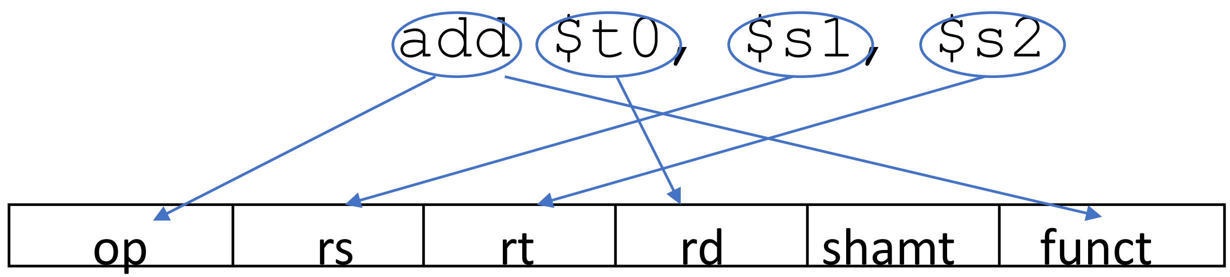
destination ← source1 op source2

- Operand order is fixed (destination first)

- Those operands are all contained in the datapath's register file (`$t0,$s1,$s2`) – indicated by $

Machine Language - Add Instruction

- Instructions, like registers and words of data, are 32 bits long
- Arithmetic Instruction Format (R format):



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

MIPS Immediate Instructions

- Small constants are used often in typical code
- Possible approaches?
  - put "typical constants" in memory and load them
  - create hard-wired registers (like $zero) for constants like 1
  - have special instructions that contain constants !

```
addi  $sp, $sp, 4      #$sp = $sp + 4
slti $t0, $s2, 15      #$t0 = 1 if $s2<15
```

- Machine format (I format):

op	rs	rt	16 bit immediate

I format

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

How About Larger Constants?

- We'd also like to be able to load a 32 bit constant into a register, for this we must use two instructions
- a new "load upper immediate" instruction

```
lui $t0, 1010101010101010
```

16	0	8	1010101010101010

- Then must get the lower order bits right, use

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

1010101010101010	0000000000000000
0000000000000000	1010101010101010

1010101010101010	1010101010101010

MIPS Memory Access Instructions

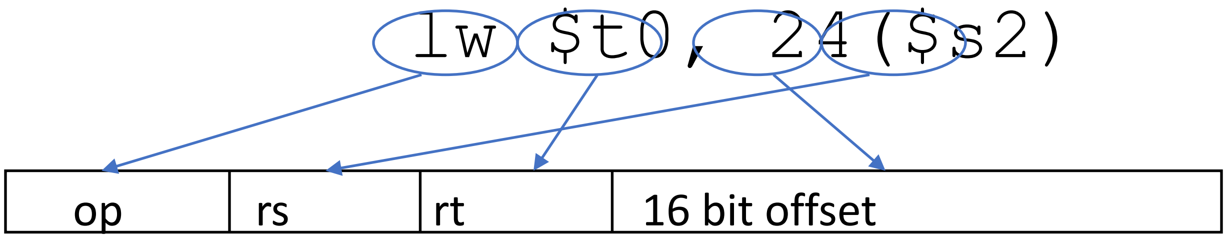
- MIPS has two basic data transfer instructions for accessing memory

```
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 – a 5 bit address
- The memory address – a 32 bit address – is formed by adding the contents of the base address register to the offset value
  - A 16-bit field meaning access is limited to memory locations within a region of $\pm 2^{13}$ or 8,192 words ($\pm 2^{15}$ or 32,768 bytes) of the address in the base register
  - Note that the offset can be positive or negative

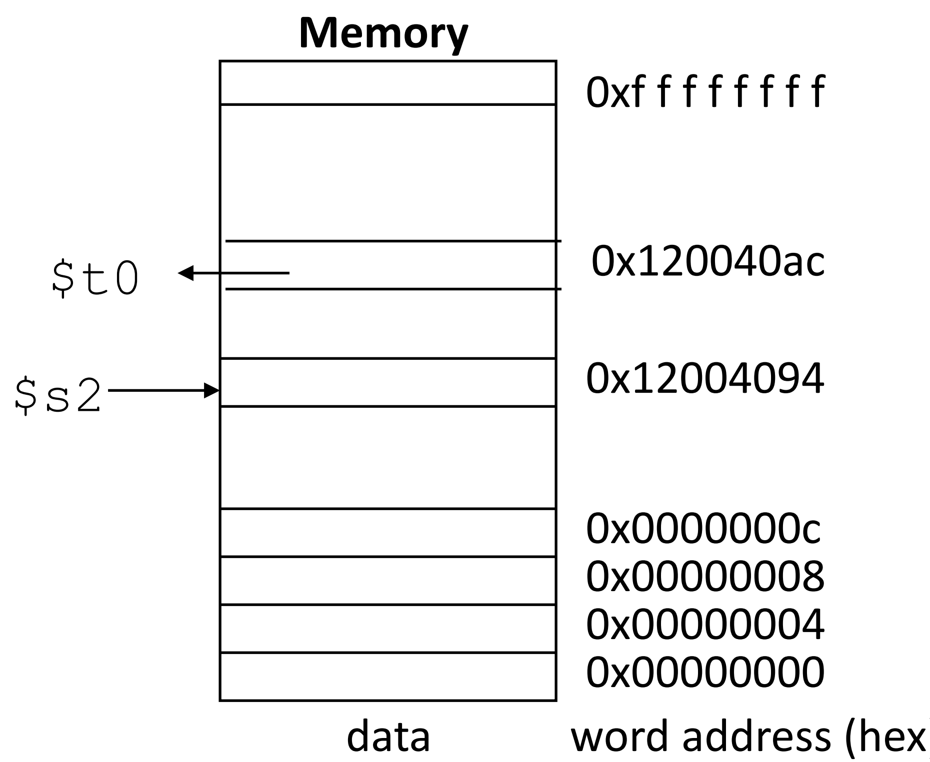
Machine Language - Load Instruction

- Load/Store Instruction Format (I format):



24_{10} + $s2 =

0x00000018
+ 0x12004094
0x120040ac

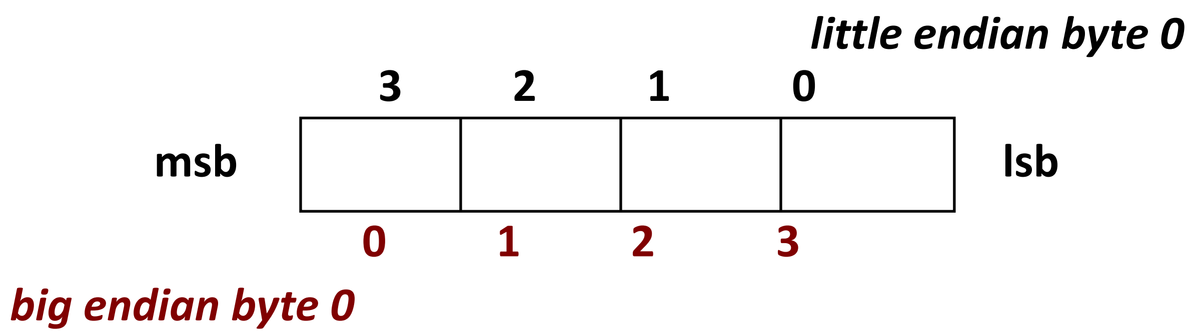


Byte Addresses

- Since 8-bit bytes are so useful, most architectures address individual bytes in memory
  - The memory address of a word must be a multiple of 4 (alignment restriction)
- Big Endian: leftmost byte is word address

  IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian: rightmost byte is word address

  Intel x86 /x64, DEC Vax, DEC Alpha (Windows NT)



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
```

op	rs	rt	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?

MIPS Control Flow Instructions

- MIPS conditional branch instructions:

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

  - Ex: `if (i==j) h = i + j;`

```
        bne $s0, $s1, Lbl1
        add $s3, $s0, $s1
Lbl1:   ...
```

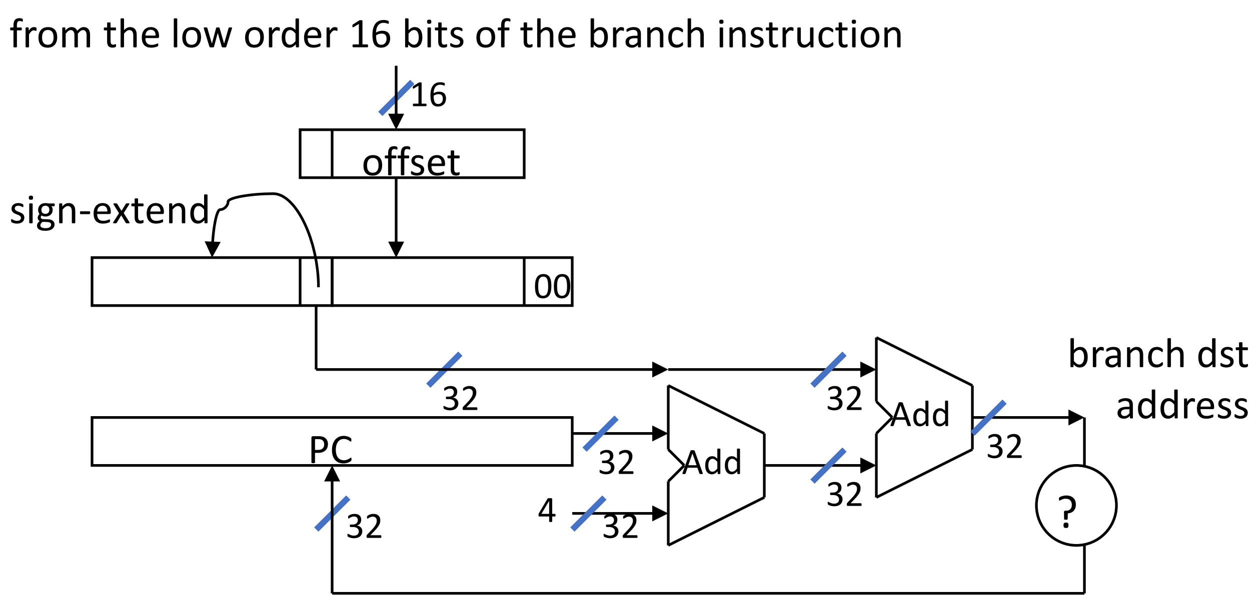
- Instruction Format (I format):

op	rs	rt	16 bit offset

- How is the branch destination address specified?

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 $+2^{15}-1$ instructions from the (instruction after the) branch instruction, but most branches are local anyway



Other Control Flow Instructions

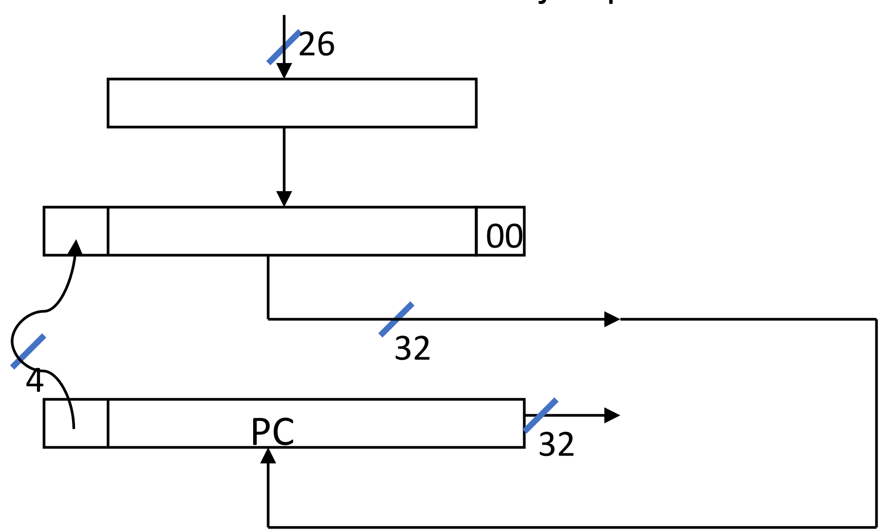
- MIPS also has an unconditional branch instruction or jump instruction:

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

- Instruction Format (J Format):

op	26-bit address

from the low order 26 bits of the jump instruction



Branching Far Away

- What if the branch destination is further away than can be captured in 16 bits?

- The assembler comes to the rescue – it inserts an unconditional jump to the branch target and inverts the condition

```
    beq  $s0, $s1, L1
```

becomes

```
    bne  $s0, $s1, L2
    j  L1
L2:
```

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):

op	26 bit address

- Then can do procedure return with a

```
jr   $ra               #return
```

- Instruction format (R format):

op	rs			funct	

MIPS ISA – First look

Category	Instr	Op Code	Example	Meaning
Arithmetic (R & I format)	add	0 and 32	add $s1, $s2, $s3	$s1 = $s2 + $s3
	subtract	0 and 34	sub $s1, $s2, $s3	$s1 = $s2 - $s3
	add immediate	8	addi $s1, $s2, 6	$s1 = $s2 + 6
	or immediate	13	ori $s1, $s2, 6	$s1 = $s2 v 6
Data Transfer (I format)	load word	35	lw $s1, 24($s2)	$s1 = Memory($s2+24)
	store word	43	sw $s1, 24($s2)	Memory($s2+24) = $s1
	load byte	32	lb $s1, 25($s2)	$s1 = Memory($s2+25)
	store byte	40	sb $s1, 25($s2)	Memory($s2+25) = $s1
	load upper imm	15	lui $s1, 6	$s1 = 6 * 2^{16}
Cond. Branch (I & R format)	br on equal	4	beq $s1, $s2, L	if ($s1==$s2) go to L
	br on not equal	5	bne $s1, $s2, L	if ($s1 !=$s2) go to L
	set on less than	0 and 42	slt $s1, $s2, $s3	if ($s2<$s3) $s1=1 else $s1=0
	set on less than immediate	10	slti $s1, $s2, 6	if ($s2<6) $s1=1 else $s1=0
Uncond. Jump (J & R format)	jump	2	j 2500	go to 10000
	jump register	0 and 8	jr $t1	go to $t1
	jump and link	3	jal 2500	go to 10000; $ra=PC+4