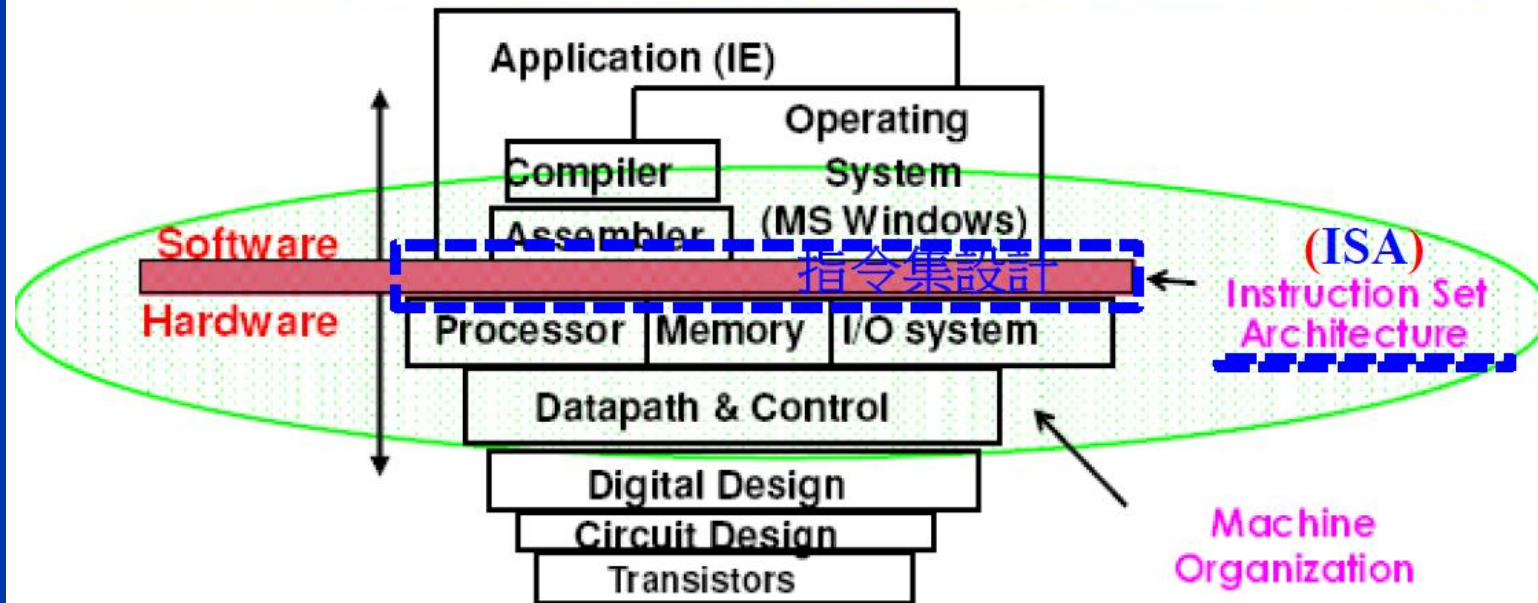


Chapter 2

Instructions: Language of the Computer

What is Computer Architecture?



**Computer Architecture =
Instruction Set Architecture
+ Machine Organization**

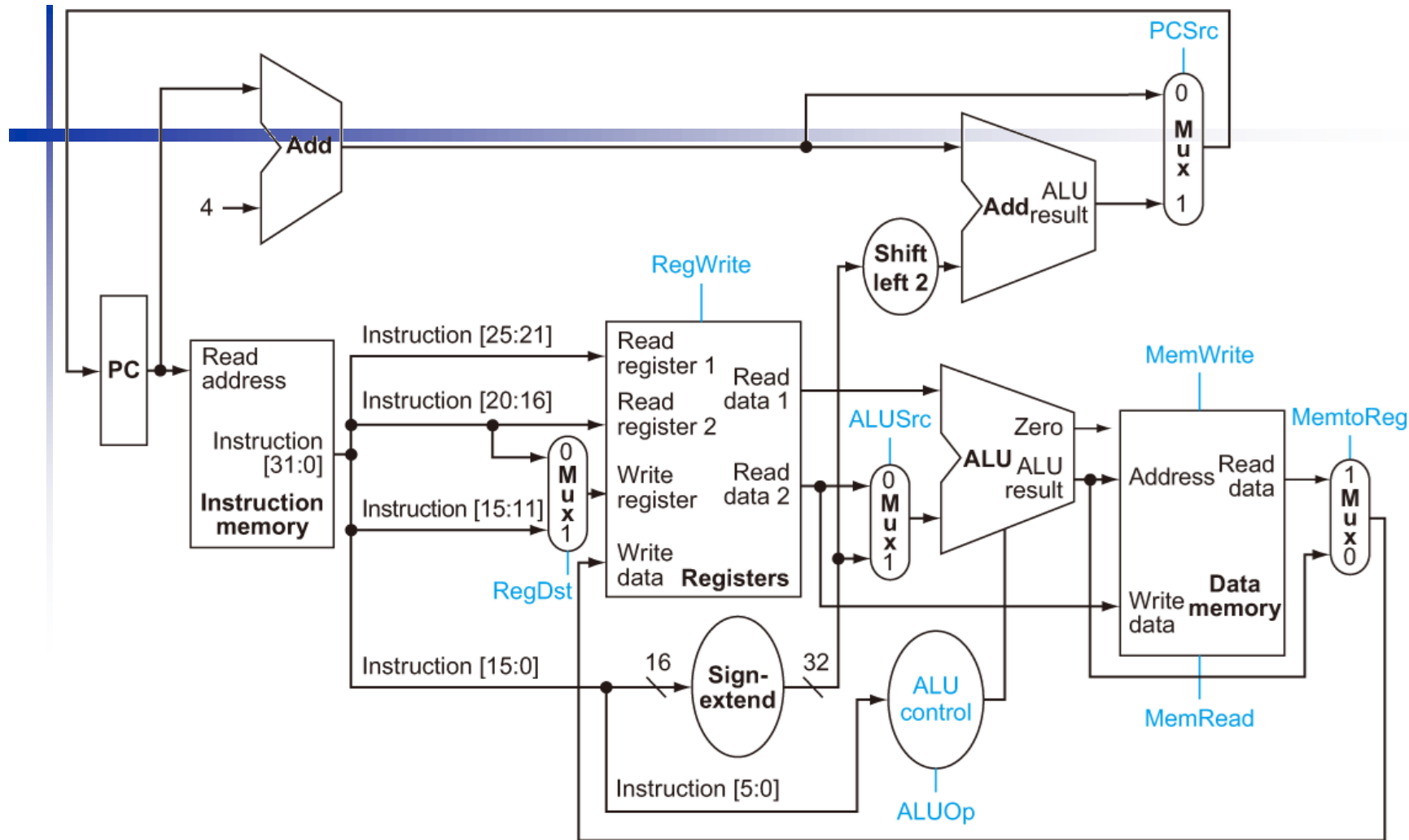


圖4.15 圖4.12 的數據通道以及所有需要的多工器及控制訊號

灰色的線表示控制訊號。ALU 控制區塊也已經加入圖中。因為PC 暫存器在每個時脈週期都會寫入一次，所以不需要寫入控制；分支控制邏輯決定寫入PC；暫存器的值是遞增後的PC 或是分支目標位址

The Processor: Datapath & Control

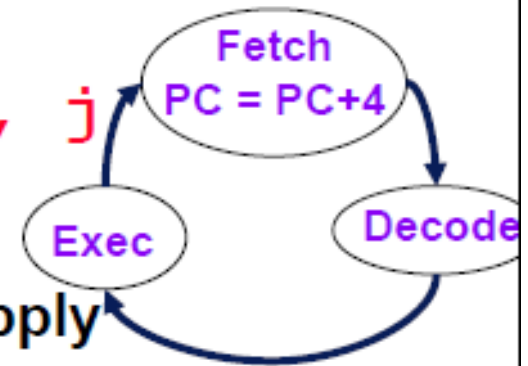
□ Our implementation of the MIPS is **simplified**

- memory-reference instructions: **lw, sw**
- arithmetic-logical instructions: **add, sub, and, or, slt**
- control flow instructions: **beq, j**

□ Generic implementation

- use the **program counter (PC)** to supply the instruction address and **fetch** the instruction from memory (and update the PC)
- **decode** the instruction (and read registers)
- **execute** the instruction

□ All instructions (except **j**) use the ALU after reading the registers



Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
 - But with many aspects in common
- Early computers had very simple instruction sets
 - Simplified implementation
- Many modern computers also have simple instruction sets

The MIPS Instruction Set

- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- Large share of embedded core market
 - Applications in consumer electronics, network/storage equipment, cameras, printers, ...
- Typical of many modern ISAs
 - See MIPS Reference Data tear-out card, and Appendixes B and E

MIPS (RISC) Design Principles (頁156)

- 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
 - limited 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**

Arithmetic Operations

- Add and subtract, three operands
 - Two sources and one destination

add a, b, c # a gets b + c
- All arithmetic operations have this form
- *Design Principle 1: Simplicity favors regularity* (簡潔利於規律性)
 - Regularity makes implementation simpler
 - Simplicity enables higher performance at lower cost

Arithmetic Example

- C code:

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

- Compiled MIPS code:

```
add t0, g, h    # temp t0 = g + h
add t1, i, j    # temp t1 = i + j
sub f, t0, t1   # f = t0 - t1
```

Register Operands

- Arithmetic instructions use register operands
- MIPS has a 32×32 -bit register file
 - Use for frequently accessed data
 - Numbered 0 to 31
 - 32-bit data called a “word”
- Assembler names
 - \$t0, \$t1, ..., \$t9 for temporary values
 - \$s0, \$s1, ..., \$s7 for saved variables
- *Design Principle 2: Smaller is faster*
 - c.f. main memory: millions of locations

Register Operand Example

- C code:

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

- f, \dots, j in $\$s0, \dots, \$s4$

- Compiled MIPS code:

add $\$t0, \$s1, \$s2$

add $\$t1, \$s3, \$s4$

sub $\$s0, \$t0, \$t1$

2.2 計算機硬體的運作

□ In the following forms

- written by **humans**

add \$t0, \$s1, \$s2

- read by **the computer**



<http://img.taopic.com/uploads/allimg/120717/201628-120GFG44651.jpg>

000000 10001 10010 01000 00000 100000



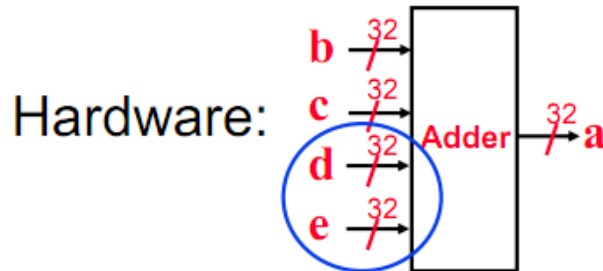

2.2 計算機硬體的運作

□ Of course this complicates some things...

C code: `a = b + c + d + e;`

MIPS code: ?

(1). `add a, b, c, d, e`



Complexity ↑ Cost ↑

More lines, more controls

2.2 計算機硬體的運作

❑ Of course this complicates some things...

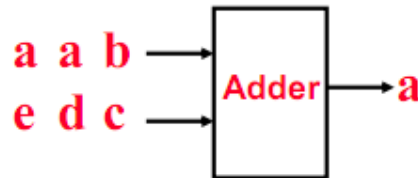
C code: `a = b + c + d + e;`

MIPS code:

(1). `add a, b, a, d, e`

(2). `add a, b, c #a=b+c`
`add a, a, d`
`add a, a, e`

Hardware:



頁63)

❑ Design Principle 1: simplicity favors regularity.

2.3 計算機硬體的運算元

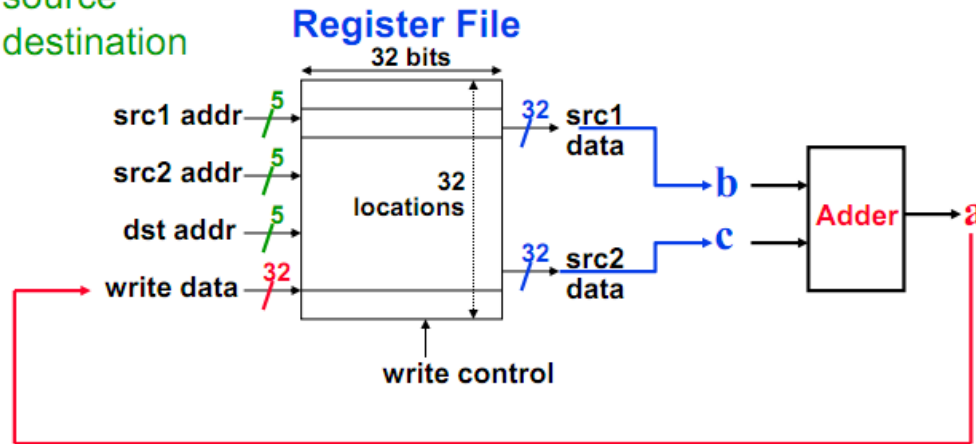
- 算術指令的運算元有其限制；它們必須是由直接建構在硬體中數量有限的暫存器(registers)而來
 - 暫存器的數量有限，在現今計算機中通常是32個
- MIPS 架構中暫存器的大小是32 位元
 - 一群群的32 位元如此經常地被使用，因此它們也在MIPS 架構中被稱為字組(word)

2.3 計算機硬體的運算元

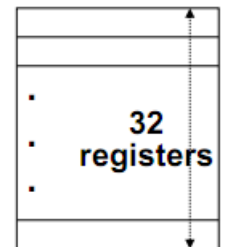
- ❑ MIPS arithmetic operands must be registers
- ❑ Holds thirty-two 32-bit registers
 - Two read ports and one write port

src: source

dst: destination



- ❑ Operands must be registers, only 32 registers provided



- ❑ Each register contains 32 bits
- MIPS-32 vs. MIPS-64**



2.3 計算機硬體的運算元

- 設計原則二：愈小愈快(smaller is faster)
 - 很大數量的暫存器單純地由於其內部的電子訊號必須傳遞更遠而需時更久、可能導致時脈週期時間增加
- 暫存器的有效運用對效能極為關鍵
(頁65-66)
- limited instruction set
- limited number of registers in register file
- limited number of addressing modes

2.3 計算機硬體的運算元

MIPS arithmetic (cont.)

頁66)

More Examples

C code: `d = a - e;`

MIPS code: **sub d, a, e** #d=a-e
 ↑ |
 └──────┘
 Comments

More Examples

C code: `f = (g + h) - (i + j);`

MIPS code: `add t0, g, h`
 `add t1, i, j`
 `sub f, t0, t1`

2.3 計算機硬體的運算元

MIPS arithmetic (cont.)

頁66)

- ❑ 32 registers, each is 32 bits wide
 - Groups of 32 bits called a **word** in MIPS
 - Registers are numbered from **0** to **31**
 - Each register also has a **name** to make it easier to code, e.g.,

\$16 - \$23 → **\$s0 - \$s7** (C variables)

\$8 - \$15 → **\$t0 - \$t7** (temporary)

Registers

R0	\$0
...	...
...	...
...	...
R28	\$28
R29	\$29
R30	\$30
R31	\$31

2.3 計算機硬體的運算元

Policy of Use Conventions		
Name	Register number	Usage
\$zero	0	the constant value 0
\$v0-\$v1	2-3	values for results and expression
\$a0-\$a3	4-7	arguments
\$t0-\$t7	8-15	temporaries
\$s0-\$s7	16-23	saved for C variables
\$t8-\$t9	24-25	more temporaries
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address
Register 1 (\$at) reserved for assembler, 26-27 for operating system		

使用暫存器來編譯C 賦值敘述

- 將程式中的變數與暫存器關聯起來的工作是由編譯器負責。例如，在先前例題中的賦值敘述：
 - $f = (g + h) - (i + j);$
- 令變數 f, g, h, i, j 分別存於暫存器 $\$s0, \$s1, \$s2, \$s3, \$s4$ 中
- 編譯出來的MIPS 碼為何？
 $f \rightarrow \$s0, g \rightarrow \$s1, h \rightarrow \$s2, i \rightarrow \$s3, j \rightarrow \$s4$

使用暫存器來編譯C 賦值敘述

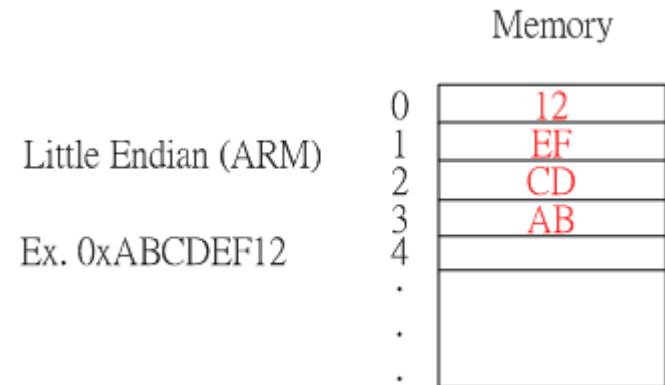
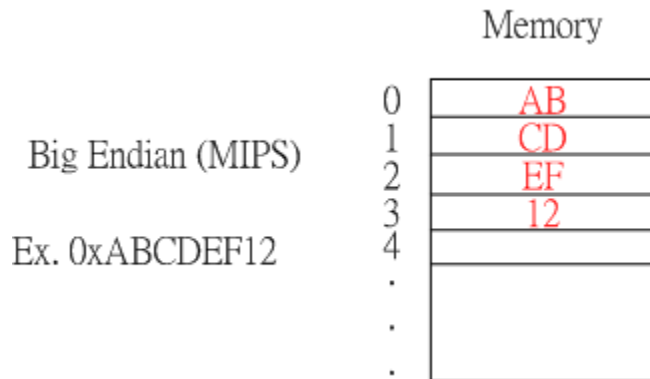
- 編譯出來的程式與前例中極相似，差異僅在於以上述暫存器名稱取代了各變數，以及兩個暫時暫存器即相對於之前的兩個暫時變數：
- `add $t0, $s1, $s2 # 暫存器$t0 內含 g+h`
- `add $t1, $s3, $s4 # 暫存器$t1 內含i+j`
- `sub $s0, $t0, $t1 # f 得到$t0-$t1, 亦即 (g+h) - (i+j)`

Memory Operands

- Main memory used for composite data
 - Arrays, structures, dynamic data
- To apply arithmetic operations
 - Load values from memory into registers
 - Store result from register to memory
- Memory is byte addressed
 - Each address identifies an 8-bit byte
- Words are aligned in memory
 - Address must be a multiple of 4
- MIPS is Big Endian
 - Most-significant byte at least address of a word
 - *c.f.* Little Endian: least-significant byte at least address

硬體／軟體介面

- 計算機的字組定址可分為兩種：
以最左邊或「大的端」(Big-Endian)(MIPS)，
或是最右邊或「小的端」(Little-Endian)(ARM)的位元組位址作為字組的位址



Memory Operand Example 1

- C code: (Byte addressing)

`g = h + A[8];`

- `g` in `$s1`, `h` in `$s2`, base address of `A` in `$s3`

- Compiled MIPS code:

- Index 8 requires offset of 32

- 4 bytes per word

```
lw    $t0, 32($s3)    # load word
add   $s1, $s2, $t0
```

offset

base register

Memory Operand Example 2

- C code: (word addressing)

`A[12] = h + A[8];`

- `h` in `$s2`, base address of `A` in `$s3`

- Compiled MIPS code:

- Index 8 requires offset of 32

```
lw    $t0, 32($s3)    # load word
add   $t0, $s2, $t0
sw    $t0, 48($s3)    # store word
```

硬體／軟體介面

Instructions

p.71 (頁69)

- ❑ Load and store instructions

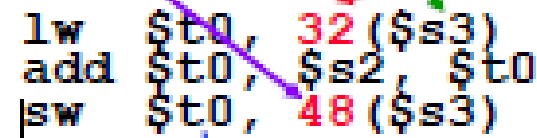
- ❑ Example:

C code:

$A[12] = h + A[8];$

MIPS code:

`lw $t0, 32($s3)`
`add $t0, $s2, $t0`
`sw $t0, 48($s3)`



- ❑ Can refer to registers by name (e.g., \$s2, \$t2) instead of number
- ❑ Store word has destination last
- ❑ Remember arithmetic operands are registers, not memory!

Can't write: `add 48($s3), $s2, 32($s3)`

硬體/軟體介面

Fig. 2.2
p.68 (頁67)

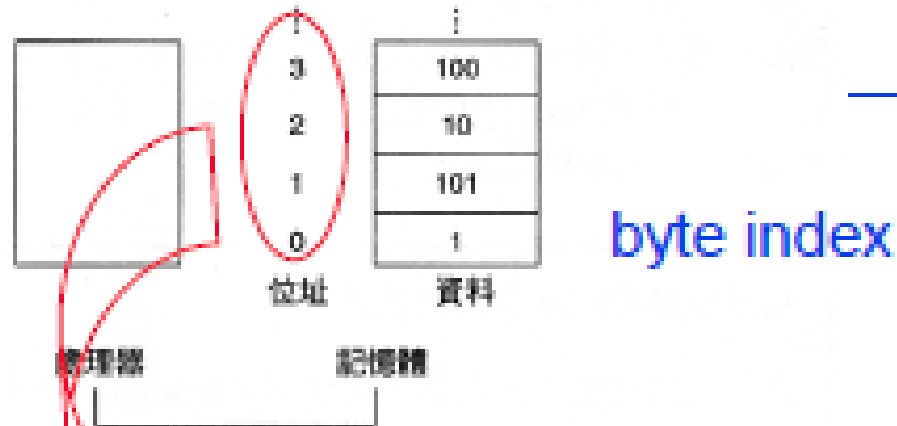


圖 2.2 記憶體位址及在這些位址的內容。這是 MIPS 定址的簡化，圖 2.3

Fig. 2.3
p.70 (頁69)

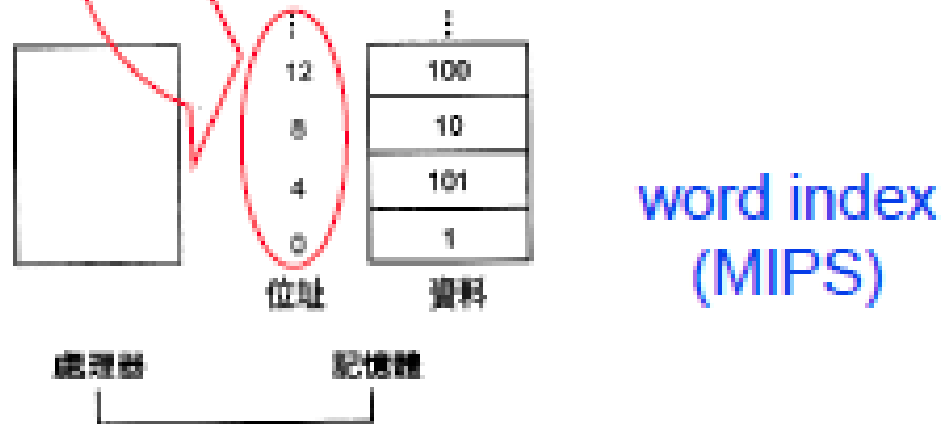


圖 2.3 真實的 MIPS 記憶體位址與內容。相對於圖 2.2，位址改變的部

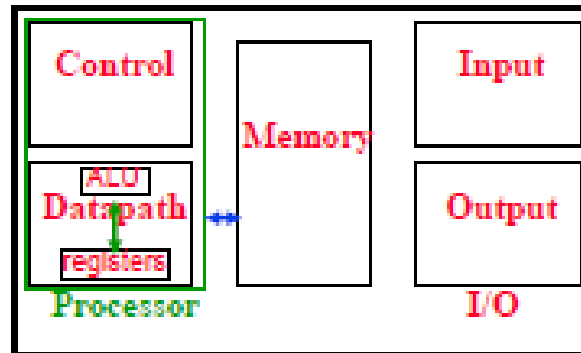
Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
 - More instructions to be executed
- Compiler must use registers for variables as much as possible
 - Only spill to memory for less frequently used variables
 - Register optimization is important!

硬體/軟體介面

Registers vs. Memory

- ❑ Arithmetic instructions operands must be **registers**,
 - **only 32** registers provided
- ❑ Compiler associates **variables** with registers
- ❑ What about programs with **lots of variables** ?



Immediate Operands

- Constant data specified in an instruction
`addi $s3, $s3, 4`
- No subtract immediate instruction
 - Just use a negative constant
`addi $s2, $s1, -1`
- *Design Principle 3: Make the common case fast*
 - Small constants are common
 - Immediate operand avoids a load instruction

The Constant Zero

- MIPS register 0 (\$zero) is the constant 0
 - Cannot be overwritten
- Useful for common operations
 - E.g., move between registers
`add $t2, $s1, $zero`

硬體/軟體介面

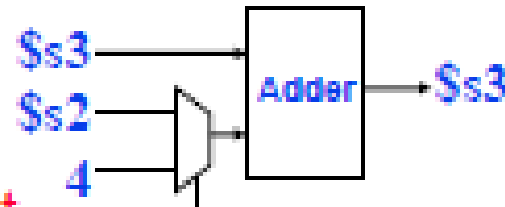
Immediate Operands

p.72 (頁71)

□ Example: $g \rightarrow \$s3$

C code: $g = g + 4;$ $\text{lw } \$t0, \frac{4}{0(\$s0)}$
 $\text{add } \$s3, \$s3, \$t0$

MIPS code: $\text{addi } \$s3, \$s3, 4$
 (add immediate)



□ Design Principle 3:

Make the common case fast

- arithmetic operands from the register file
 - allow instructions to contain immediate operands
- subtract immediate? X

圖2.1 本章中會出現的MIPS 組合語言指令³⁻¹

MIPS 運算元

名 稱	舉 例	註 解
32 個暫存器	\$s0-\$s7, \$t0-\$t9, \$zero, \$a0-\$a3, \$v0-\$v1, \$gp, \$fp, \$sp, \$ra, \$at	快速存取資料的地方。在 MIPS 中，資料必須要在暫存器裡才能執行運算，暫存器 \$zero 恆等於 0，而暫存器 \$at 則是保留給組譯器來處理值很大的常數。
2 ³⁰ 記憶體字組	Memory[0], Memory[4], ..., Memory[4294967292]	在 MIPS 裡，記憶體只能由資料傳輸指令來存取。MIPS 使用位元組位址，所以連續的字組位址相差 4。記憶體裡儲存資料結構、陣列和溢出暫存器 (spilled registers)。

MIPS 組合語言

分類	指 令	舉 例	意 義	註 解
算術	加法	add \$s1,\$s2,\$s3	\$s1=\$s2+\$s3	三個暫存器運算元
	減法	sub \$s1,\$s2,\$s3	\$s1=\$s2-\$s3	三個暫存器運算元
	加立即值	addi \$s1,\$s2,20	\$s1=\$s2+20	加上常數

圖2.1 本章中會出現的MIPS 組合語言指令³⁻²

MIPS 組合語言

分類	指 令	舉 例	意 義	註 解
資料 傳輸	載入字組	lw \$s1,20(\$s2)	\$s1=Memory[\$s2+20]	字組由記憶體載入至暫存器
	儲存字組	sw \$s1,20(\$s2)	Memory[\$s2+20]=\$s1	字組由暫存器儲存至記憶體
	載入半字組	lh \$s1,20(\$s2)	\$s1=Memory[\$s2+20]	半字組由記憶體載入至暫存器
	載入無號 半字組	lhu \$s1,20(\$s2)	\$s1=Memory[\$s2+20]	無號的半字組由記憶體載入至暫存器
	儲存半字組	sh \$s1,20(\$s2)	Memory[\$s2+20]=\$s1	半字組由暫存器儲存至記憶體
	載入位元組	lb \$s1,20(\$s2)	\$s1=Memory[\$s2+20]	位元組由記憶體載入至暫存器
	載入無號 位元組	lbu \$s1,20(\$s2)	\$s1=Memory[\$s2+20]	無號的位元組由記憶體載入至暫存器
	儲存位元組	sb \$s1,20(\$s2)	Memory[\$s2+20]=\$s1	位元組由暫存器儲存至記憶體
	載入連結的字元 組	ll \$s1,20(\$s2)	\$s1=Memory[\$s2+20]	作為不可分割的（記憶體與儲存器內容）交換中第一部份的載入字元組
	條件式儲存字元 組	sc \$s1,20(\$s2)	Memory[\$s2+20]=\$s1; \$s1=0 或 1	作為不可分割的（記憶體與暫存器內容）交換中第二部份的儲存字組
	載入上半部立即 值	lui \$s1,20	\$s1=20*2 ¹⁶	載入常數至較高的 16 位元
邏輯	及	and \$s1,\$s2,\$s3	\$s1=\$s2 & \$s3	三個暫存器運算元：逐位元的及運算
	或	or \$s1,\$s2,\$s3	\$s1=\$s2 \$s3	三個暫存器運算元：逐位元的或運算
	反或	nor \$s1,\$s2,\$s3	\$s1=~(\$s2 \$s3)	三個暫存器運算元：逐位元的反或運算
	及立即值	andi \$s1,\$s2,20	\$s1=\$s2 & 20	暫存器與常數做逐位元的及運算
	或立即值	ori \$s1,\$s2,20	\$s1=\$s2 20	暫存器與常數做逐位元的或運算
	邏輯左移	sll \$s1,\$s2,10	\$s1=\$s2<<10	左移常數個位元位置
	邏輯右移	srl \$s1,\$s2,10	\$s1=\$s2>>10	右移常數個位元位置

圖2.1 本章中會出現的MIPS 組合語言指令³⁻³

MIPS 組合語言

分類	指 令	舉 例	意 義	註 解
條件式分支	若等於則分支	beq \$s1,\$s2,25	若(\$s1==\$s2)則前往 PC+4+100	等於測試：PC 相對的分支
	若不等於則分支	bne \$s1,\$s2,25	若(\$s1!=\$s2)則前往 PC+4+100	不等於測試：PC 相對的分支
	若小於則設定	slt \$s1,\$s2,\$s3	若(\$s2<\$s3) \$s1=1; 否則 \$s1=0	小於比較；用於 beq、bne
	無號若小於則設定	sltu \$s1,\$s2,20	若(\$s2<\$s3) \$s1=1; 否則 \$s1=0	無號數的小於比較
	若小於立即值則設定	slti \$s1,\$s2,20	若(\$s2<20) \$s1=1; 否則 \$s1=0	小於某常數的比較
	無號若小於立即值則設定	sltiu \$s1,\$s2,20	若(\$s2<20) \$s1=1; 否則 \$s1=0	無號數的小於某常數的比較
非條件式跳躍	跳躍	j 2500	前往 10000	跳至目的位址
	透過暫存器跳躍	jr \$ra	前往 \$ra	用於 switch 敘述、程序反回
	跳躍並連結	jal 2500	\$ra=PC+4 前往 10000	用於程序呼叫

MIPS 組合語言

分類	指 令	舉 例	意 義	註 解
算術	加法	add \$s1,\$s2,\$s3	\$s1=\$s2+\$s3	三個暫存器運算元
	減法	sub \$s1,\$s2,\$s3	\$s1=\$s2-\$s3	三個暫存器運算元
	加立即值	addi \$s1,\$s2,20	\$s1=\$s2+20	加上常數

2.4 有號及無號數字

- 任何數字基底下，第 i 個數元代表的值是：

$$d \times \text{基底}^i$$

- 其中 i 的值由小數點的左方為0開始，並由右至左逐位加1，由左至右逐位減1
- 以下圖示是MIPS的字組中位元的編號方法以及1011₂的放置法：



- 最不重要位元(least significant bit) 表示最右側的位元(上圖的位元0)，而最重要位元(most significant bit) 表示最左側的位元(位元31)

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

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

- Example

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

- Using 32 bits

- 0 to +4,294,967,295

2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

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

- Example

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

- Using 32 bits

- $-2,147,483,648$ to $+2,147,483,647$

2s-Complement Signed Integers

- Bit 31 is sign bit
 - 1 for negative numbers
 - 0 for non-negative numbers
- $-(-2^n - 1)$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
 - 0: 0000 0000 ... 0000
 - -1: 1111 1111 ... 1111
 - Most-negative: 1000 0000 ... 0000
 - Most-positive: 0111 1111 ... 1111

Signed Negation

- Complement and add 1
 - Complement means $1 \rightarrow 0, 0 \rightarrow 1$

$$x + \bar{x} = 1111 \dots 111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Example: negate +2
 - $+2 = 0000 \ 0000 \dots 0010_2$
 - $-2 = 1111 \ 1111 \dots 1101_2 + 1$
 $= 1111 \ 1111 \dots 1110_2$

Sign Extension

- Representing a number using more bits
 - Preserve the numeric value
- In MIPS instruction set
 - `addi`: extend immediate value
 - `lb`, `lh`: extend loaded byte/halfword
 - `beq`, `bne`: extend the displacement
- Replicate the sign bit to the left
 - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
 - `+2`: 0000 0010 => 0000 0000 0000 0010
 - `-2`: 1111 1110 => 1111 1111 1111 1110

2.4 有號及無號數字

MIPS Number Representations

32-bit signed numbers (2's complement):

0000 0000 0000 0000 0000 0000 0000 0000	$= 0_{\text{ten}}$	
0000 0000 0000 0000 0000 0000 0000 0001	$= +1_{\text{ten}}$	
...		
0111 1111 1111 1111 1111 1111 1111 1110	$= +2,147,483,646_{\text{ten}}$	
0111 1111 1111 1111 1111 1111 1111 1111	$= +2,147,483,647_{\text{ten}}$	<i>maxint</i>
1000 0000 0000 0000 0000 0000 0000 0000	$= -2,147,483,648_{\text{ten}}$	
1000 0000 0000 0000 0000 0000 0000 0001	$= -2,147,483,647_{\text{ten}}$	
...		
1111 1111 1111 1111 1111 1111 1111 1110	$= -2_{\text{ten}}$	
1111 1111 1111 1111 1111 1111 1111 1111	$= -1_{\text{ten}}$	<i>minint</i>

MSB (Most Significant Bit) is indicated by a blue oval around the first bit of each row.
LSB (Least Significant Bit) is indicated by a blue oval around the last bit of each row.

Converting < 32-bit values into 32-bit values

- copy the most significant bit (the sign bit) into the "empty" bits

0010 \rightarrow 0000 0010

1010 \rightarrow 1111 1010

- sign extend** versus **zero extend**

(**lb** vs. **lbu**) (**lh** vs. **lhu**)

2.4 有號及無號數字

Sign Extension

□ Examples: 8-bit to 16-bit

● +2: 0000 0010 => 0000 0000 0000 0010

● -2: 1111 1110 => 1111 1111 1111 1110

□ Replicate the sign bit to the left

● unsigned values: extend with 0s
(zero extend)

2.4 有號及無號數字

Negation Shortcut (2's complement) p.77 (頁77)

❑ Negate

1011

and add a 1

1010

complement all the bits

❑ Note: negate and invert are different!

$$\begin{aligned} -2^3 &= \\ -(2^3 - 1) &= \end{aligned}$$

$$2^3 - 1 =$$

2'sc binary	decimal
1000	-8
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
0111	7

2.4 有號及無號數字

將16位元的2(10進位)及- 2(10進位)轉換為32位元的數字?(using the 2s-Complement Signed Integers)

Representing Instructions

- Instructions are encoded in binary
 - Called machine code
- MIPS instructions
 - Encoded as 32-bit instruction words
 - Small number of formats encoding operation code (opcode), register numbers, ...
 - **Regularity!**
- Register numbers
 - \$t0 – \$t7 are reg's 8 – 15
 - \$t8 – \$t9 are reg's 24 – 25
 - \$s0 – \$s7 are reg's 16 – 23

2.3 計算機硬體的運算元

Policy of Use Conventions		
Name	Register number	Usage
\$zero	0	the constant value 0
\$v0-\$v1	2-3	values for results and expression
\$a0-\$a3	4-7	arguments
\$t0-\$t7	8-15	temporaries
\$s0-\$s7	16-23	saved for C variables
\$t8-\$t9	24-25	more temporaries
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address
Register 1 (\$at) reserved for assembler, 26-27 for operating system		

2.5 在計算機中表示指令

□ Instruction Categories

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

Registers

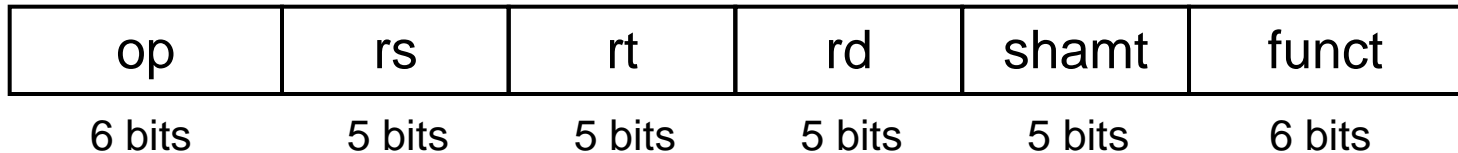
R0 - R31

PC

+ 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 R-format Instructions



■ Instruction fields

- op: operation code (opcode)
- rs: first source register number
- rt: second source register number
- rd: destination register number
- shamt: shift amount (00000 for now)
- funct: function code (extends opcode)

R-format Example

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

add \$t0, \$s1, \$s2

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

$00000010001100100100000000100000_2 = 02324020_{16}$

將一道MIPS 指令轉換成機器指令

- 十進位表示法是：

0	17	18	8	0	32
---	----	----	---	---	----

- 一指令中每一該等片斷稱為一個欄位(field)。第一以及最後一個欄位(上例中的0 以及32)合併起來告知MIPS 計算機該指令為加。第二欄位指出加法運算的第一個來源運算元(source operand)的暫存器編號(17=\$s1)，以及第三欄位指出另一個來源運算元(18=\$s2)。第四欄位指出將接受和的暫存器的編號(8=\$t0)。第五欄位在該指令中沒有用到，因此被給予0 這個值。因此，該指令將暫存器\$s1 及\$s2 相加並將和置於暫存器\$t0 中

Hexadecimal (Review)

- 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

Representing Instructions

- Instructions are encoded in binary
 - Called machine code
- MIPS instructions
 - Encoded as 32-bit instruction words
 - Small number of formats encoding operation code (opcode), register numbers, ...
 - Regularity!
- Register numbers
 - \$t0 – \$t7 are reg's 8 – 15
 - \$t8 – \$t9 are reg's 24 – 25
 - \$s0 – \$s7 are reg's 16 – 23

MIPS R-format Instructions



■ Instruction fields

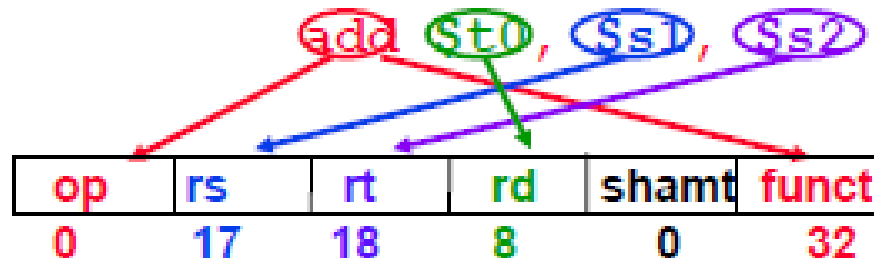
- op: operation code (opcode)
- rs: first source register number
- rt: second source register number
- rd: destination register number
- shamt: shift amount (00000 for now)
- funct: function code (extends opcode)

R-format Example

2.5 Representing instructions in the computer

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

p.80 (頁80)



- 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

2.5 在計算機中表示指令

- 這種指令的擺置方式稱為指令格式(instruction format)。
 - 遵循「規律性易導致簡單的設計」的設計原則，所有**MIPS** 指令的長度均為**32** 個位元，並僅使用少數指令格式
 - 稱呼該種數字形式的指令為機器語言(machine language)指令以及一串該等指令為機器碼

MIPS 欄位

- 對MIPS 各欄位均給予名稱，以方便對它們

op	rs	rt	rd	shamt	funct
6 個位元	5 個位元	5 個位元	5 個位元	5 個位元	6 個位元

- 以下是MIPS 指令中各欄位每一個名稱的意義：

op：該指令的基本運作，傳統上稱其為運作碼 (opcode)

rs：第一個來源運算元暫存器

rt：第二個來源運算元暫存器

rd：目的運算元暫存器。其會得到運作的結果

MIPS 欄位

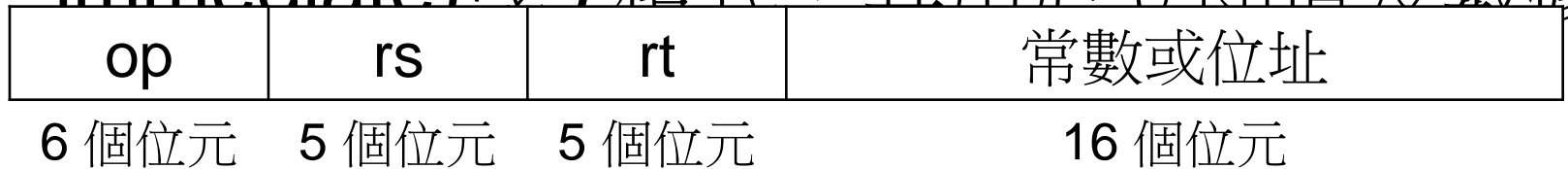
shamt：位移量(shift amount)。 (2.6 節說明位移指令及本名詞；其在該節之前將不會被使用到，因此本節中該欄位內均含0。)

funct：功能。本欄位常被稱之為功能碼(function code)，用以選取op 欄位的運作的特定變型(variant)。

- 希望保持所有指令的長度相同及希望使用單一指令格式之間有了衝突

MIPS 欄位

- 設計原則4：好的設計需要有好的折衷
(good design demands good compromises)
 - MIPS 設計者選擇的折衷是保持所有指令長度相同，因此對不同種類的指令有不同格式的需求
 - 上述格式稱為R 型(指暫存器register)或R 格式
- 第二種指令格式稱為I 型(指立即值 immediate)或 I 格式，其用於立即值及數據



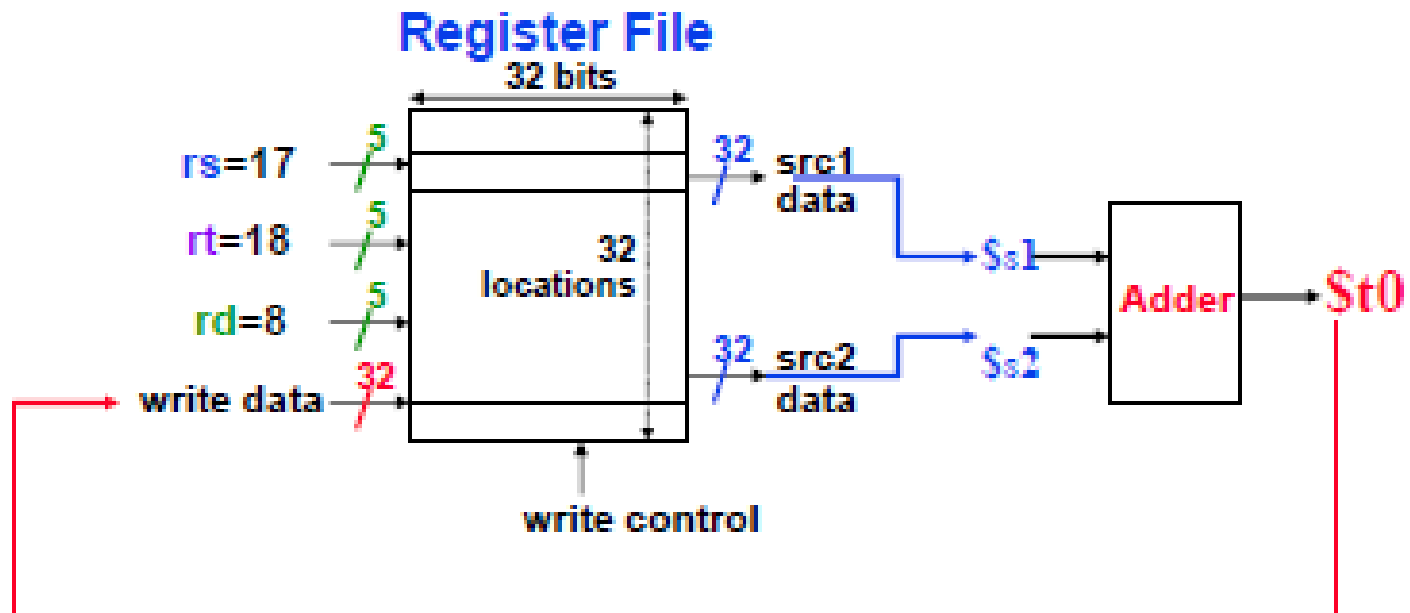
MIPS I-format Instructions



- Immediate arithmetic and load/store instructions
 - rt: destination or source register number
 - Constant: -2^{15} to $+2^{15} - 1$
 - Address: offset added to base address in rs
- *Design Principle 4: Good design demands good compromises*
 - Different formats complicate decoding, but allow 32-bit instructions uniformly
 - Keep formats as similar as possible

MIPS Register File

- MIPS arithmetic operands must be registers
- Holds thirty-two 32-bit registers
 - Two read ports and one write port



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	no
\$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

Machine Language (頁80)

□ Instructions, like registers and words of data, are also 32 bits long

- Example: `add $t0, $s1, $s2`

- registers have numbers, `$t0=8`, `$s1=17`, `$s2=18`

op	rs	rt	rd	shamt	funct
0	17	18	8	0	32

Machine
Language

□ Instruction Format:

000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct

Machine
Code

Machine Language (cont.) (頁83)

- Consider the load-word and store-word instructions,

op	rs	rt	rd	shamt	funct
----	----	----	----	-------	-------

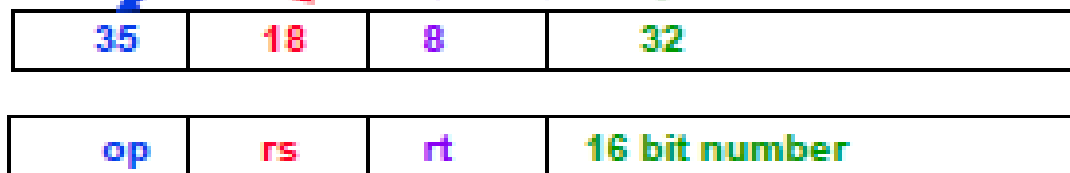
? **R-type**

- What would the regularity principle have us do?
- New principle:** Good design demands a compromise

- Introduce a new type of instruction format

- I-type** for data transfer instructions
- other format was **R-type** for register

- Example: **lw \$t0, 32(\$s2)**



I-type

- Where's the compromise?

(頁83)所有指令相同

MIPS (RISC) Design Principles (頁156)

- 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
 - limited 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 instruction encoding (頁84)

n.a. = not available

指令	格式	op	rs	rt	rd	shamt	funct	address
add (加法)	R	0	reg	reg	reg	0	32 ₁₀	—n.a.—
sub (減法)	R	0	reg	reg	reg	0	34 ₁₀	—n.a.—
addi (立即加法)	I	8 ₁₀	reg	reg	n.a.	n.a.	n.a.	常數
lw (載入字組)	I	35 ₁₀	reg	reg	n.a.	n.a.	n.a.	位址
sw (儲存字組)	I	43 ₁₀	reg	reg	n.a.	n.a.	n.a.	位址

R-type

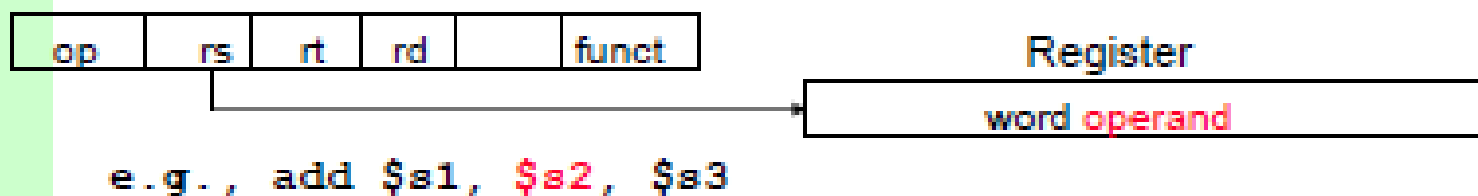
op	rs	rt	rd	shamt	funct
----	----	----	----	-------	-------

I-type

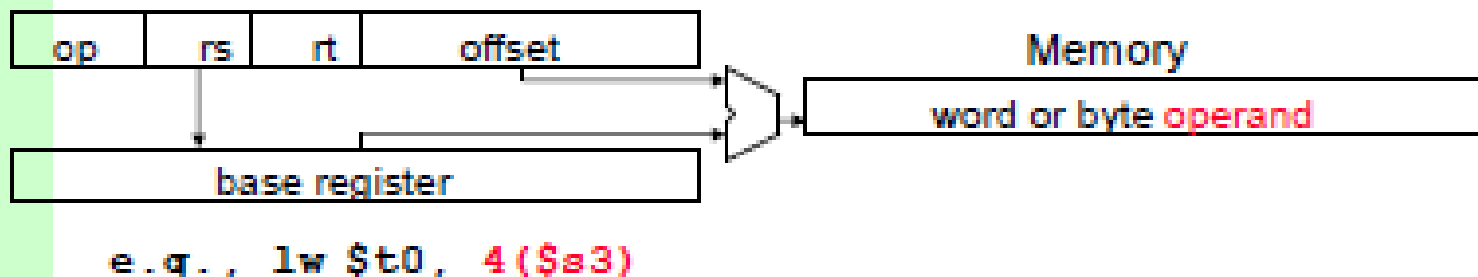
op	rs	rt	16 bit offset
----	----	----	---------------

Review of MIPS Operand Addressing Modes

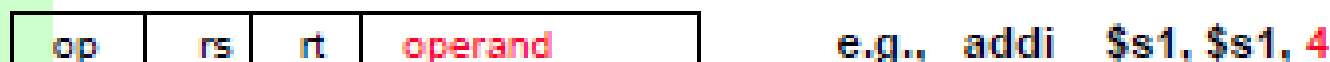
- Register addressing – operand is in a register



- Base (displacement) addressing – operand is at the memory location whose address is the sum of a register and a 16-bit constant contained within the instruction



- Immediate addressing – operand is a 16-bit constant contained within the instruction



MIPS 欄位(頁85)

- 第一個欄位的內容決定了所採用的格式：
 - 由此硬體即可得知如何解讀指令的後半

MIPS 機器語言

名 稱	型式	舉 例						註 解
加法	R	0	18	19	17	0	32	add \$s1,\$s2,\$s3
減法	R	0	18	19	17	0	34	sub \$s1,\$s2,\$s3
加立即值	I	8	18	17	100			addi \$s1,\$s2,100
載入字組	I	35	18	17	100			lw \$s1,100(\$s2)
儲存字組	I	43	18	17	100			sw \$s1,100(\$s2)
欄位大小		6 位元	5 位元	5 位元	5 位元	5 位元	6 位元	所有 MIPS 指令的長度都是 32
R 型式	R	op	rs	rt	rd	shamt	funct	算術指令型式
I 型式	I	op	rs	rt	address			資料傳輸型式

圖2.6 到第2.5 節為止出現過的MIPS 架構。

R 和I 是到目前為止的二種MIPS 指令型式。這兩種指令格式前十六位元相同；兩者都包含`op` 欄位，說明是哪一種運算；`rs` 欄位，指出一個來源運算元及`rt` 欄位是另一個來源運算元，而在載入字組指令`rt` 則是指目的運算元。R 型式將後十六位元分成`rd` 欄位，指出目的暫存器；`shamt` 欄位會在第2.6 節解釋；`funct` 欄位指出R 型式指令的運作。I 格式則將後十六位元當作`address` 欄位

- ➡ 現今計算機均根據二個主要原則來建構：
 1. 指令均以數字來代表
 2. 程式儲存於可以被讀或寫的記憶體中，如同數字一般
- ➡ 此等原則導引出內儲程式(stored-program)的概念；
 - 圖2.7 表示該概念的威力；
 - 具體而言，記憶體可存放編輯(editor)程式的原始碼、相對應的編譯過的(compiled)機器碼、該編譯過的程式所使用的文字(text)，甚至於產生該機器碼的編譯器

R-format Example

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

add \$t0, \$s1, \$s2

special	\$s1	\$s2	\$t0	0	add
---------	------	------	------	---	-----

0	17	18	8	0	32
---	----	----	---	---	----

000000	10001	10010	01000	00000	100000
--------	-------	-------	-------	-------	--------

$00000010001100100100000000100000_2 = 02324020_{16}$

MIPS I-format Instructions



- Immediate arithmetic and load/store instructions
 - rt: destination or source register number
 - Constant: -2^{15} to $+2^{15} - 1$
 - Address: offset added to base address in rs
- *Design Principle 4: Good design demands good compromises*
 - Different formats complicate decoding, but allow 32-bit instructions uniformly
 - Keep formats as similar as possible

Check Yourself? (頁87)

What MIPS instruction does this represent? Choose from one of the four options below.

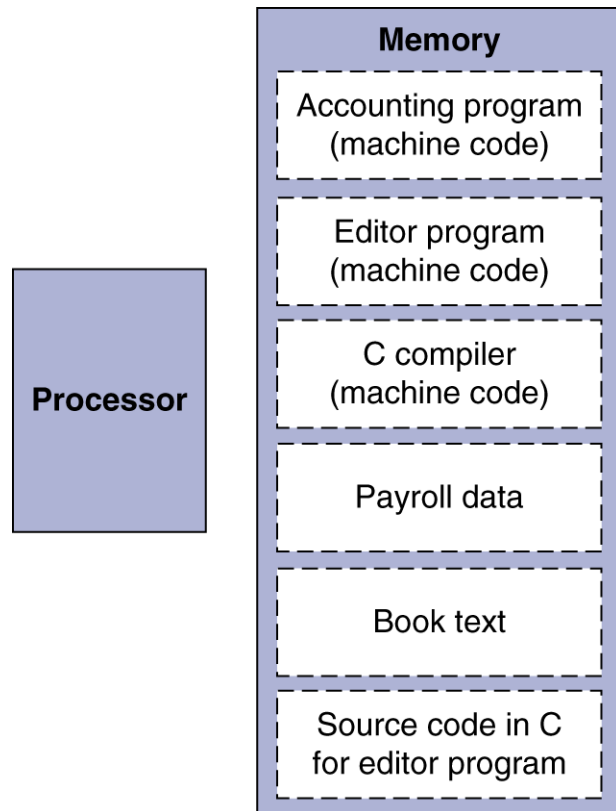
**Check
Yourself**

op	rs	rt	rd	shamt	funct
0	8	9	10	0	34

1. `sub $t0, $t1, $t2`
2. `add $t2, $t0, $t1`
3. `sub $t2, $t1, $t0`
4. `sub $t2, $t0, $t1`

Stored Program Computers

The BIG Picture



- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
 - e.g., compilers, linkers, ...
- Binary compatibility allows compiled programs to work on different computers
 - Standardized ISAs

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	MIPS
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR			or, ori
Bitwise NOT	~	~	nor

- Useful for extracting and inserting groups of bits in a word

Shift Operations

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

- shamt: how many positions to shift
- Shift left logical
 - Shift left and fill with 0 bits
 - `sll` by i bits multiplies by 2^i
- Shift right logical
 - Shift right and fill with 0 bits
 - `srl` by i bits divides by 2^i (unsigned only)

Shift left logical (sll)

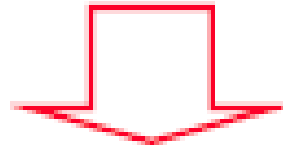
□ Example:

sll \$t2, \$s0, 4

□ Instruction Format (R-format):

Why not I-format?

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0



000000 00000 10000 01010 00100 000000

AND Operations

- Useful to mask bits in a word
 - Select some bits, clear others to 0
- and \$t0, \$t1, \$t2

\$t2	0000 0000 0000 0000 0000 1101 1100 0000
\$t1	0000 0000 0000 0000 0011 1100 0000 0000
\$t0	0000 0000 0000 0000 0000 1100 0000 0000

OR Operations

- Useful to include bits in a word
 - Set some bits to 1, leave others unchanged
- or \$t0, \$t1, \$t2

\$t2	0000 0000 0000 0000 0000 1101 1100 0000
\$t1	0000 0000 0000 0000 0011 1100 0000 0000
\$t0	0000 0000 0000 0000 0011 1101 1100 0000

NOT Operations

- Useful to invert bits in a word
 - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
 - $a \text{ NOR } b == \text{NOT} (a \text{ OR } b)$

`nor $t0, $t1, $zero` ←

Register 0: always
read as zero

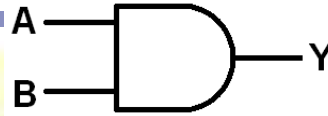
\$t1

0000 0000 0000 0000 0011 1100 0000 0000

\$t0

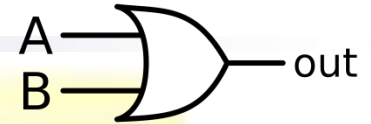
1111 1111 1111 1111 1100 0011 1111 1111

Review: Variety of Logic Gates



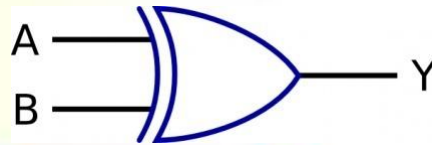
AND

x	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1



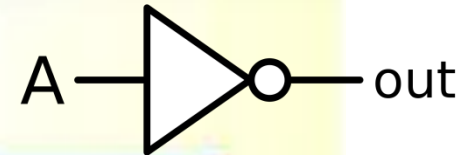
OR

x	y	x OR y
0	0	0
0	1	1
1	0	1
1	1	1



XOR

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0



Not

x	NOT x
0	1
1	0

Logical Operations (cont.) (頁62)

及運算	<u>and \$s1,\$s2,\$s3</u>	\$s1=\$s2 & \$s3
或運算	<u>or \$s1,\$s2,\$s3</u>	\$s1=\$s2 \$s3
反或運算	<u>nor \$s1,\$s2,\$s3</u>	\$s1=-(\$s2 <u>\$s3</u>)  Not \$zero
立即 及運算	<u>andi \$s1,\$s2,100</u>	\$s1=\$s2 & 100
立即 或運算	<u>ori \$s1,\$s2,100</u>	\$s1=\$s2 100
向左位移	<u>sll \$s1,\$s2,10</u>	\$s1=\$s2<<10 Shifts
向右位移	<u>srl \$s1,\$s2,10</u>	\$s1=\$s2>>10

Conditional Operations

- Branch to a labeled instruction if a condition is true
 - Otherwise, continue sequentially
- `beq rs, rt, L1`
 - if (`rs == rt`) branch to instruction labeled L1;
- `bne rs, rt, L1`
 - if (`rs != rt`) branch to instruction labeled L1;
- `j L1`
 - unconditional jump to instruction labeled L1

Compiling If Statements

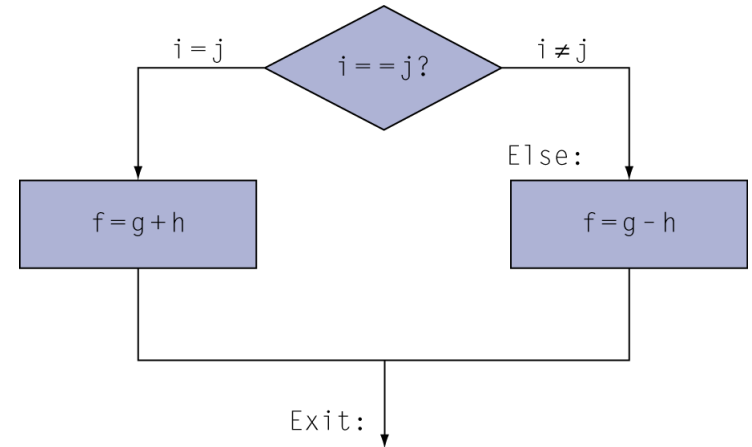
- C code:

```
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 addresses

Compiling Loop Statements

- C code:

```
while (save[i] == k) i += 1;
```

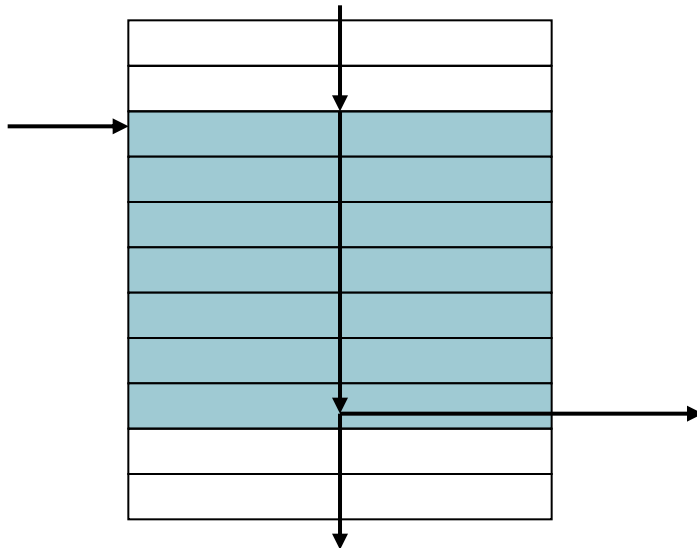
- i in \$s3, k in \$s5, address of save in \$s6

- Compiled MIPS code:

```
Loop:  sll    $t1, $s3, 2
       add    $t1, $t1, $s6
       lw     $t0, 0($t1)
       bne    $t0, $s5, Exit
       addi   $s3, $s3, 1
       j      Loop
Exit:  ...
```

Basic Blocks

- A basic block is a sequence of instructions with
 - No embedded branches (except at end)
 - No branch targets (except at beginning)



- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

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, constant`
 - if ($rs < \text{constant}$) $rt = 1$; else $rt = 0$;
- Use in combination with `beq`, `bne`

```
    slt $t0, $s1, $s2    # if ($s1 < $s2)
    bne $t0, $zero, L    #   branch to L
```

Branch Instruction Design

- Why not b1t, bge, etc?
- Hardware for $<$, \geq , ... slower than $=$, \neq
 - Combining with branch involves more work per instruction, requiring a slower clock
 - All instructions penalized!
- beq and bne are the common case
- This is a good design compromise

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`
 - $-1 < +1 \Rightarrow \$t0 = 1$
 - `sltu $t0, $s0, $s1 # unsigned`
 - $+4,294,967,295 > +1 \Rightarrow \$t0 = 0$

2.8 Supporting Procedures in Computer Hardware

- ❑ MIPS procedure call instruction (jump-and-link):

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

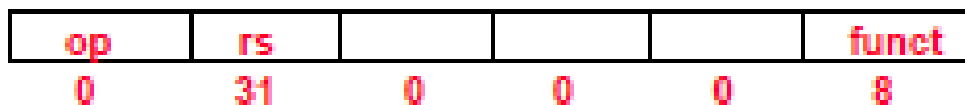
- ❑ Saves PC+4 in register \$ra to have a link to the next instruction for the procedure return
- ❑ Machine format (J-format):



- ❑ Then can do procedure return with a jump register instruction

```
jr $ra                #return
```

- ❑ Instruction format (R-format):



Procedure Calling

- Steps required
 1. Place parameters in registers
 2. Transfer control to procedure
 3. Acquire storage for procedure
 4. Perform procedure's operations
 5. Place result in register for caller
 6. Return to place of call

Register Usage

- \$a0 – \$a3: arguments (reg's 4 – 7)
- \$v0, \$v1: result values (reg's 2 and 3)
- \$t0 – \$t9: temporaries
 - Can be overwritten by callee
- \$s0 – \$s7: saved
 - Must be saved/restored by callee
- \$gp: global pointer for static data (reg 28)
- \$sp: stack pointer (reg 29)
- \$fp: frame pointer (reg 30)
- \$ra: return address (reg 31)

Procedure Call Instructions

- Procedure call: jump and link
`jal ProcedureLabel`
 - Address of following instruction put in `$ra`
 - Jumps to target address
- Procedure return: jump register
`jr $ra`
 - Copies `$ra` to program counter
 - Can also be used for computed jumps
 - e.g., for case/switch statements

Leaf Procedure Example

- 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
- f in \$s0 (hence, need to save \$s0 on stack)
- Result in \$v0

Leaf Procedure Example

- MIPS code:

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

Non-Leaf Procedures

- Procedures that call other procedures
- 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

Non-Leaf Procedure Example

- C code:

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

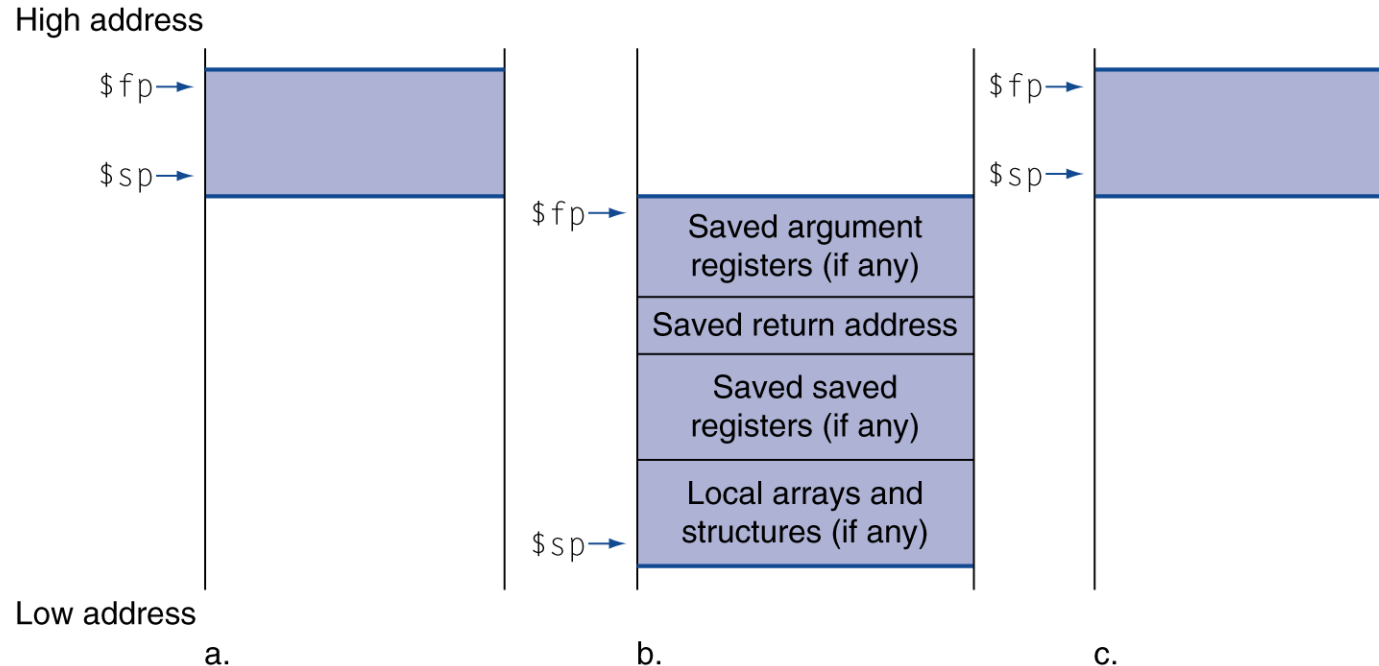
- Argument n in \$a0
- Result in \$v0

Non-Leaf Procedure Example

- 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

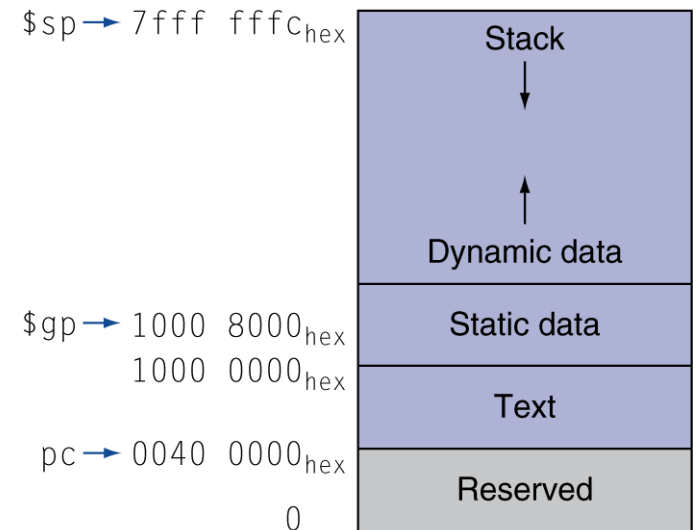
Local Data on the Stack



- Local data allocated by callee
 - e.g., C automatic variables
- Procedure frame (activation record)
 - Used by some compilers to manage stack storage

Memory Layout

- Text: program code
- Static data: global variables
 - e.g., static variables in C, constant arrays and strings
 - \$gp initialized to address allowing \pm offsets into this segment
- Dynamic data: heap
 - E.g., malloc in C, new in Java
- Stack: automatic storage



Character Data

- Byte-encoded character sets
 - ASCII: 128 characters
 - 95 graphic, 33 control
 - Latin-1: 256 characters
 - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
 - Used in Java, C++ wide characters, ...
 - Most of the world's alphabets, plus symbols
 - UTF-8, UTF-16: variable-length encodings

Byte/Halfword Operations

- Could use bitwise operations
- MIPS byte/halfword load/store
 - String processing is a common case

`lb rt, offset(rs)` `lh rt, offset(rs)`

- Sign extend to 32 bits in `rt`

`lbu rt, offset(rs)` `lhu rt, offset(rs)`

- Zero extend to 32 bits in `rt`

`sb rt, offset(rs)` `sh rt, offset(rs)`

- Store just rightmost byte/halfword

String Copy Example

- C code (naïve):

- 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

String Copy Example

- MIPS code:

strcpy:		
	addi \$sp, \$sp, -4	# adjust stack for 1 item
	sw \$s0, 0(\$sp)	# save \$s0
	add \$s0, \$zero, \$zero	# i = 0
L1:	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, L2	# exit loop if y[i] == 0
	addi \$s0, \$s0, 1	# i = i + 1
	j L1	# next iteration of loop
L2:	lw \$s0, 0(\$sp)	# restore saved \$s0
	addi \$sp, \$sp, 4	# pop 1 item from stack
	jr \$ra	# and return

32-bit Constants

- Most constants are small
 - 16-bit immediate is sufficient
- For the occasional 32-bit constant
 - Copies 16-bit constant to left 16 bits of rt
 - Clears right 16 bits of rt to 0

`lhi $s0, 61`

0000 0000 0111 1101	0000 0000 0000 0000
---------------------	---------------------

`ori $s0, $s0, 2304`

0000 0000 0111 1101	0000 1001 0000 0000
---------------------	---------------------

Branch Addressing

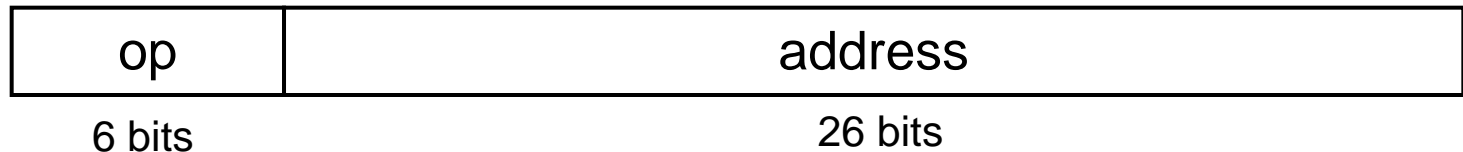
- Branch instructions specify
 - Opcode, two registers, target address
- Most branch targets are near branch
 - Forward or backward



- PC-relative addressing
 - Target address = $PC + \text{offset} \times 4$
 - PC already incremented by 4 by this time

Jump Addressing

- Jump (j and jal) targets could be anywhere in text segment
 - Encode full address in instruction



- (Pseudo)Direct jump addressing
 - Target address = $PC_{31...28} : (\text{address} \times 4)$

Target Addressing Example

- Loop code from earlier example
 - Assume Loop at location 80000

Loop: sll	\$t1, \$s3, 2	80000	0	0	19	9	4	0
add	\$t1, \$t1, \$s6	80004	0	9	22	9	0	32
lw	\$t0, 0(\$t1)	80008	35	9	8	0		
bne	\$t0, \$s5, Exit	80012	5	8	21	2		
addi	\$s3, \$s3, 1	80016	8	19	19	1		
j	Loop	80020	2	20000				
Exit: ...		80024						

Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

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

↓

```
bne $s0,$s1, L2
```

```
j L1
```

```
L2: ...
```

Addressing Mode Summary

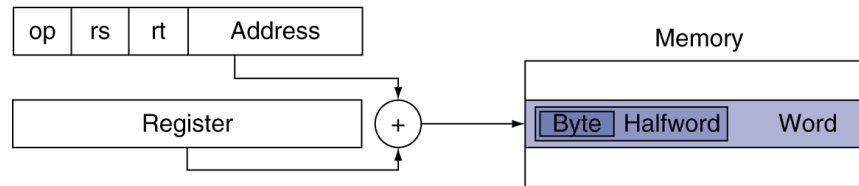
1. Immediate addressing



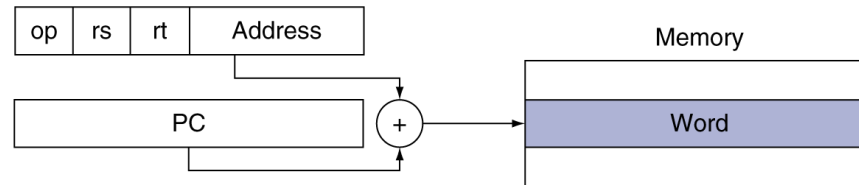
2. Register addressing



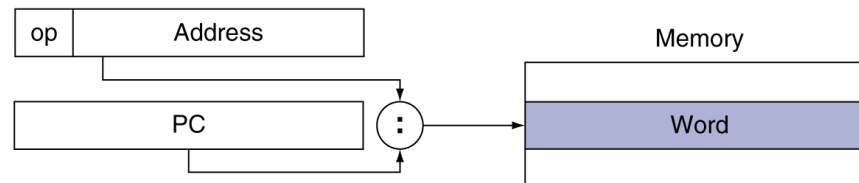
3. Base addressing



4. PC-relative addressing



5. Pseudodirect addressing



Synchronization

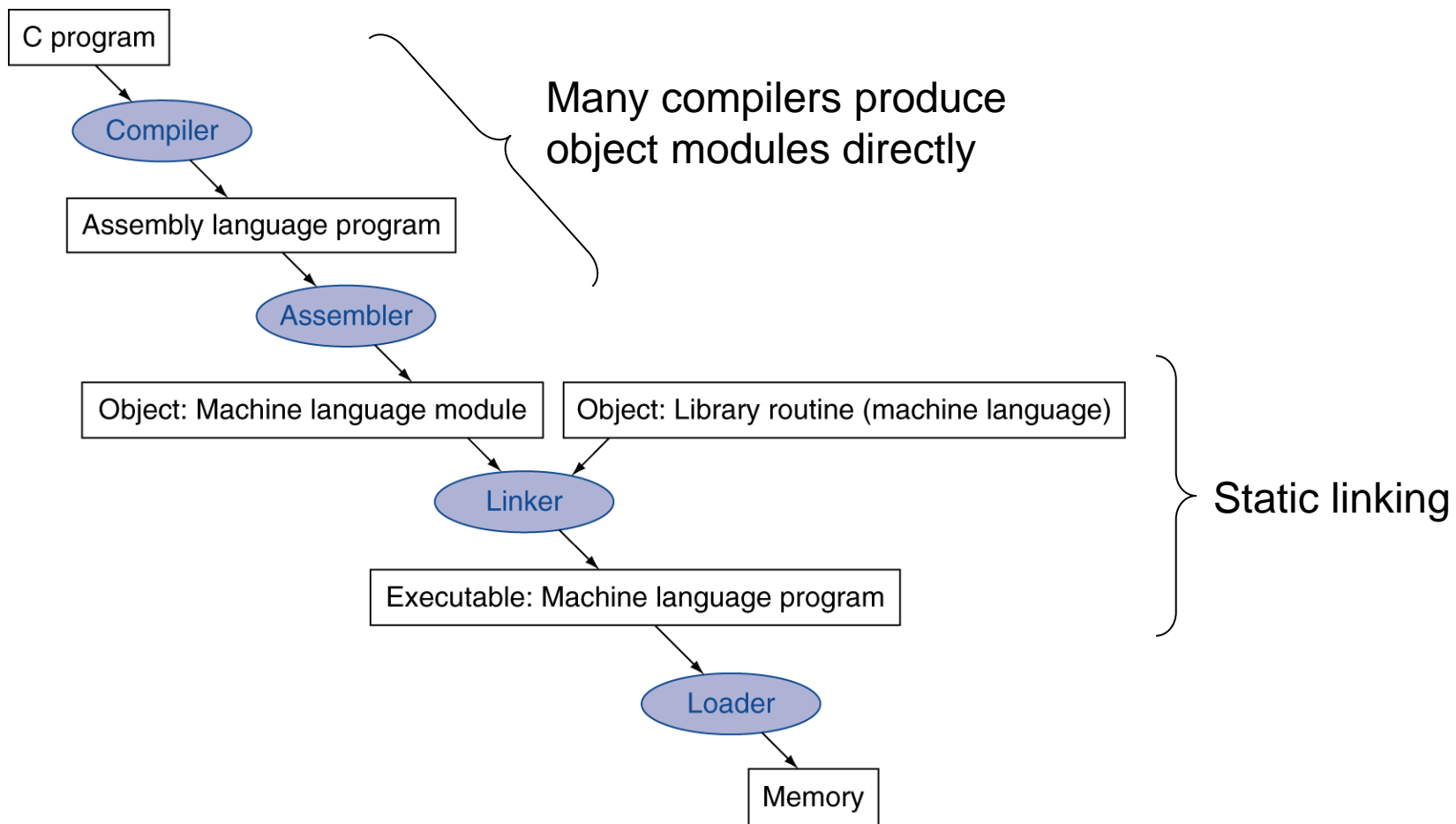
- Two processors sharing an area of memory
 - P1 writes, then P2 reads
 - Data race if P1 and P2 don't synchronize
 - Result depends of order of accesses
- Hardware support required
 - Atomic read/write memory operation
 - No other access to the location allowed between the read and write
- Could be a single instruction
 - E.g., atomic swap of register \leftrightarrow memory
 - Or an atomic pair of instructions

Synchronization in MIPS

- Load linked: `ll rt, offset(rs)`
- Store conditional: `sc rt, offset(rs)`
 - Succeeds if location not changed since the `ll`
 - Returns 1 in `rt`
 - Fails if location is changed
 - Returns 0 in `rt`
- Example: atomic swap (to test/set lock variable)

```
try: add $t0,$zero,$s4 ;copy exchange value
      ll $t1,0($s1)    ;load linked
      sc $t0,0($s1)    ;store conditional
      beq $t0,$zero,try ;branch store fails
      add $s4,$zero,$t1 ;put load value in $s4
```


Translation and Startup



Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: figments of the assembler's imagination

`move $t0, $t1` \rightarrow `add $t0, $zero, $t1`

`blt $t0, $t1, L` \rightarrow `slt $at, $t0, $t1`
 `bne $at, $zero, L`

- `$at` (register 1): assembler temporary

Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
 - Header: described contents of object module
 - Text segment: translated instructions
 - Static data segment: data allocated for the life of the program
 - Relocation info: for contents that depend on absolute location of loaded program
 - Symbol table: global definitions and external refs
 - Debug info: for associating with source code

Linking Object Modules

- Produces an executable image
 1. Merges segments
 2. Resolve labels (determine their addresses)
 3. Patch location-dependent and external refs
- Could leave location dependencies for fixing by a relocating loader
 - But with virtual memory, no need to do this
 - Program can be loaded into absolute location in virtual memory space

Loading a Program

- Load from image file on disk into memory
 1. Read header to determine segment sizes
 2. Create virtual address space
 3. Copy text and initialized data into memory
 - Or set page table entries so they can be faulted in
 4. Set up arguments on stack
 5. Initialize registers (including \$sp, \$fp, \$gp)
 6. Jump to startup routine
 - Copies arguments to \$a0, ... and calls main
 - When main returns, do exit syscall

C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)

```
void swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

- v in \$a0, k in \$a1, temp in \$t0

The Procedure Swap

swap: sll \$t1, \$a1, 2	# \$t1 = k * 4
add \$t1, \$a0, \$t1	# \$t1 = v+(k*4)
	# (address of v[k])
lw \$t0, 0(\$t1)	# \$t0 (temp) = v[k]
lw \$t2, 4(\$t1)	# \$t2 = v[k+1]
sw \$t2, 0(\$t1)	# v[k] = \$t2 (v[k+1])
sw \$t0, 4(\$t1)	# v[k+1] = \$t0 (temp)
jr \$ra	# return to calling routine

The Sort Procedure in C

- Non-leaf (calls swap)

```
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);
        }
    }
}
```

- v in \$a0, k in \$a1, i in \$s0, j in \$s1

The Procedure Body

<pre> move \$s2, \$a0 # save \$a0 into \$s2 move \$s3, \$a1 # save \$a1 into \$s3 </pre>			Move params
<pre> move \$s0, \$zero # i = 0 for1tst: slt \$t0, \$s0, \$s3 # \$t0 = 0 if \$s0 ≥ \$s3 (i ≥ n) </pre>			Outer loop
<pre> for2tst: beq \$t0, \$zero, exit1 # go to exit1 if \$s0 ≥ \$s3 (i ≥ n) addi \$s1, \$s0, -1 # j = i - 1 slti \$t0, \$s1, 0 # \$t0 = 1 if \$s1 < 0 (j < 0) bne \$t0, \$zero, exit2 # go to exit2 if \$s1 < 0 (j < 0) sll \$t1, \$s1, 2 # \$t1 = j * 4 add \$t2, \$s2, \$t1 # \$t2 = v + (j * 4) lw \$t3, 0(\$t2) # \$t3 = v[j] lw \$t4, 4(\$t2) # \$t4 = v[j + 1] slt \$t0, \$t4, \$t3 # \$t0 = 0 if \$t4 ≥ \$t3 beq \$t0, \$zero, exit2 # go to exit2 if \$t4 ≥ \$t3 </pre>			Inner loop
<pre> move \$a0, \$s2 # 1st param of swap is v (old \$a0) move \$a1, \$s1 # 2nd param of swap is j jal swap # call swap procedure </pre>			Pass params & call
<pre> addi \$s1, \$s1, -1 # j -= 1 j for2tst # jump to test of inner loop </pre>			Inner loop
<pre> exit2: addi \$s0, \$s0, 1 # i += 1 j for1tst # jump to test of outer loop </pre>			Outer loop

The Full Procedure

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
...		
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
	jr \$ra	# return to calling routine

Arrays vs. Pointers

- Array indexing involves
 - Multiplying index by element size
 - Adding to array base address
- Pointers correspond directly to memory addresses
 - Can avoid indexing complexity

Example: Clearing and Array

```
clear1(int array[], int size) {  
    int i;  
    for (i = 0; i < size; i += 1)  
        array[i] = 0;  
}
```

```
        move $t0,$zero    # i = 0  
loop1: sll $t1,$t0,2      # $t1 = i * 4  
        add $t2,$a0,$t1   # $t2 =  
                           # &array[i]  
        sw $zero, 0($t2)  # array[i] = 0  
        addi $t0,$t0,1    # i = i + 1  
        slt $t3,$t0,$a1   # $t3 =  
                           # (i < size)  
        bne $t3,$zero,loop1 # if (...)  
                           # goto loop1
```

```
clear2(int *array, int size) {  
    int *p;  
    for (p = &array[0]; p < &array[size];  
        p = p + 1)  
        *p = 0;  
}
```

```
        move $t0,$a0      # p = & array[0]  
        sll $t1,$a1,2      # $t1 = size * 4  
        add $t2,$a0,$t1   # $t2 =  
                           # &array[size]  
loop2: sw $zero,0($t0)    # Memory[p] = 0  
        addi $t0,$t0,4     # p = p + 4  
        slt $t3,$t0,$t2   # $t3 =  
                           # (p < &array[size])  
        bne $t3,$zero,loop2 # if (...)  
                           # goto loop2
```

Comparison of Array vs. Ptr

- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
 - Part of index calculation for incremented i
 - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
 - Induction variable elimination
 - Better to make program clearer and safer

ARM & MIPS Similarities

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

	ARM	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

Compare and Branch in ARM

- 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

Basic x86 Registers

Name	31	0	Use
EAX			GPR 0
ECX			GPR 1
EDX			GPR 2
EBX			GPR 3
ESP			GPR 4
EBP			GPR 5
ESI			GPR 6
EDI			GPR 7
	CS		Code segment pointer
	SS		Stack segment pointer (top of stack)
	DS		Data segment pointer 0
	ES		Data segment pointer 1
	FS		Data segment pointer 2
	GS		Data segment pointer 3
EIP			Instruction pointer (PC)
EFLAGS			Condition codes

Basic x86 Addressing Modes

- Two operands per instruction

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

- Memory addressing modes
 - Address in register
 - $\text{Address} = R_{\text{base}} + \text{displacement}$
 - $\text{Address} = R_{\text{base}} + 2^{\text{scale}} \times R_{\text{index}}$ (scale = 0, 1, 2, or 3)
 - $\text{Address} = R_{\text{base}} + 2^{\text{scale}} \times R_{\text{index}} + \text{displacement}$

ARM v8 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
 - Dropped load/store multiple
 - 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