POLITECNICO
MILANO 1863

Architettura dei calcolatori e sistemi operativi

Riassunto (pseudo)istruzioni e direttive importanti

Istruzioni e pseudo-istruzioni – aritmetica

ARITMETICA

add	$1, $2, $3	$1 := $2 + $3	addizione
addu	$1, $2, $3	$1 := $2 + $3	addizione naturale
addi	$1, $2, cost	$1 := $2 + cost	addizione di costante
addiu	$1, $2, cost	$1 := $2 + cost	addizione naturale di costante
sub	$1, $2, $3	$1 := $2 – $3	sottrazione
subu	$1, $2, $3	$1 := $2 – $3	sottrazione naturale
mult	$1, $2	hi|lo := $1 × $2	moltiplicazione (risultato a 64 bit)

ARITMETICA – pseudo-istruzioni

subi	$1, $2, cost	$1 := $2 – cost	sottrazione di costante
subiu	$1, $2, cost	$1 := $2 – cost	sottrazione naturale di costante
neg	$1, $2	$1 := –$2	negazione aritmetica (compl. a 2)

hanno tre o talvolta due argomenti, lavorano solo con registri oppure con registri e una sola costante a 16 bit e hanno formato numerico di tipo R o I

Istruzioni e pseudo-istruzioni – confronto

CONFRONTO

slt	$1, $2, $3	**if** $2 < $3 **then** $1 := 1 **else** $1 := 0	poni a 1 se strettamente minore
sltu	$1, $2, $3	**if** $2 < $3 **then** $1 := 1 **else** $1 := 0	poni a 1 se strettamente minore (aritmetica naturale)
slti	$1, $2, cost	**if** $2 < cost **then** $1 := 1 **else** $1 := 0	poni a 1 se strettamente minore di costante
sltiu	$1, $2, cost	**if** $2 < cost **then** $1 := 1 **else** $1 := 0	poni a 1 se strettamente minore di costante (aritmetica naturale)

hanno tre argomenti, lavorano solo con registri oppure con registri e una sola costante a 16 bit e hanno formato numerico di tipo R o I

Istruzioni e pseudo-istruzioni – logica

LOGICA

or	$1, $2, $3	$1 := $2 **or** $3	somma logica bit a bit
and	$1, $2, $3	$1 := $2 **and** $3	prodotto logico bit a bit
ori	$1, $2, cost	$1 := $2 **or** cost	somma logica bit a bit con costante
andi	$1, $2, cost	$1 := $2 **and** cost	prodotto logico bit a bit con costante
nor	$1, $2, $3	$1 := $2 **nor** $3	somma logica negata bit a bit
sll	$1, $2, cost	$1 := $2 << cost	scorrimento logico a sinistra (left) del numero di bit specificato da cost
srl	$1, $2, cost	$1 := $2 >> cost	scorrimento logico a destra (right) del numero di bit specificato da cost

LOGICA – pseudo-istruzione

not	$1, $2	$1 := **not** $2	negazione logica (not bit a bit)

hanno tre o talvolta due argomenti, lavorano solo con registri oppure con registri e una sola costante a 16 bit e hanno formato numerico di tipo R o I

Istruzioni e pseudo-istruzioni – salto

SALTO INCONDIZIONATO: ASSOLUTO, INDIRETTO E CON COLLEGAMENTO

j	indirizzo	pc := indirizzo (28 bit)	salto incondizionato assoluto
jr	$1	pc := $1 (32 bit)	salto incondizionato indiretto da registro
jal	indirizzo	pc := indirizzo (28 bit) e collega il registro $ra	salto incondizionato assoluto con collegamento (per chiamata a sottoprogramma)

SALTO CONDIZIONATO

beq	$1, $2, spi	**if** $1 = $2 salta relativo a PC	salto condizionato di uguaglianza
bne	$1, $2, spi	**if** $1 ≠ $2 salta relativo a PC	salto condizionato di disuguaglianza

SALTO CONDIZIONATO – pseudo-istruzioni

blt	$1, $2, spi	**if** $1 < $2 salta relativo a PC	salta se strettamente minore
bgt	$1, $2, spi	**if** $1 > $2 salta relativo a PC	salta se strettamente maggiore
ble	$1, $2, spi	**if** $1 ≤ $2 salta relativo a PC	salta se minore o uguale
bge	$1, $2, spi	**if** $1 ≥ $2 salta relativo a PC	salta se maggiore o uguale

hanno tre o talvolta uno argomenti e hanno formato numerico di tipo J o I

Istruzioni e pseudo-istruzioni – trasferimento

TRASFERIMENTO TRA PROCESSORE E MEMORIA

lw	$1, spi ($2)	$1 := mem ($2 + spi)	carica parola (a 32 bit)
sw	$1, spi ($2)	mem ($2 + spi) := $1	memorizza parola (a 32 bit)
lh, *lhu*	$1, spi ($2)	$1 := mem ($2 + spi)	carica mezza parola (a 16 bit)
sh	$1, spi ($2)	mem ($2 + spi) := $1	memorizza mezza parola (a 16 bit)
lb, *lbu*	$1, spi ($2)	$1 := mem ($2 + spi)	carica byte (a 8 bit)
sb	$1, spi ($2)	mem ($2 + spi) := $1	memorizza byte (a 8 bit)

TRASFERIMENTO TRA PROCESSORE E MEMORIA – pseudo-istruzioni (vedi nota 1 sotto)

lw	$1, etichetta	$1 := mem ($gp + spi di etichetta)	carica parola (a 32 bit)
sw	$1, etichetta	mem ($gp + spi di etichetta) := $1	memorizza parola (a 32 bit)

Nota 1: anche le pseudo-istruzioni *lh*, *lhu*, *sh*, *lb*, *lbu* e *sb* con etichetta sottintendono il registro $gp.

hanno due argomenti e hanno formato numerico di tipo I

Istruzioni e pseudo-istruzioni – copia e inizializzazione

TRASFERIMENTO TRA REGISTRI (non referenziabili)

mflo	$1	$1 := lo	copia registro lo
mfhi	$1	$1 := hi	copia registro hi

TRASFERIMENTO TRA REGISTRI – pseudo-istruzione

move	$1, $2	$1 := $2	copia registro

CARICAMENTO DI COSTANTE IN REGISTRO

lui	$1, cost	$1 (16 bit più signif.) := cost	carica cost (in 16 bit più signif. di $1) (16 bit meno signif. di $1 posti a 0)

CARICAMENTO DI COSTANTE / INDIRIZZO IN REGISTRO – pseudo-istruzioni (vedi nota 2 sotto)

li	$1, cost	$1 := cost (32 bit)	carica costante a 32 bit
la	$1, indir	$1 := indir (32 bit)	carica indirizzo a 32 bit

Nota 2: entrambe le pseudo-istruzioni *li* e *la* caricano un valore a 32 bit in un registro, ma *li* va usata per caricare una costante e *la* per caricare un indirizzo.

Registri

REGISTRI REFERENZIABILI (nominabili come argomento di istruzione macchina)

0	*0*	costante 0 (registro denotabile anche come $zero)
1	*at*	uso riservato all'assembler-linker (per espandere pseudo-istruzioni e macro)
2 - 3	*v0 - v1*	valore restituito da funzione (sottoprogramma) (*v0* per dati di tipo scalare o puntatore, più *v1* per numeri reali di tipo *double*)
4 - 7	*a0 - a3*	argomenti in ingresso a funzione (max quattro argomenti)
8 - 15	*t0 - t7*	registri per valori temporanei (p. es. calcolo delle espressioni)
16 - 23	*s0 - s7*	registri usabili (se possibile) per var locali (scalari e punt) di sottoprogramma
24 - 25	*t8 - t9*	registri per valori temporanei (in aggiunta a *t0 - t7*), come i precedenti *tx*
26 - 27	*k0 - k1*	registri riservati per il nucleo (kernel) del Sistema Operativo
28	*gp*	global pointer (puntatore all'area dati globale)
29	*sp*	stack pointer (puntatore alla cima della pila)
30	*fp*	frame pointer (puntatore all'area di attivazione di sottoprogramma)
31	*ra*	return address (indirizzo di rientro da chiamata a sottoprogramma)

REGISTRI NON REFERENZIABILI (non nominabili come argomento di istruzione macchina)

	pc	program counter (contatore di programma)
	hi	registro per risultato di moltiplicazione e divisione (32 bit più significativi)
	lo	registro per risultato di moltiplicazione e divisione (32 bit meno significativi)

Direttive di assemblatore

Direttive fondamentali

`.align n`	allinea il dato dichiarato di seguito secondo l'argomento "n": n=0 byte, n=1 mezza parola, n=2 parola e n=3 parola lunga
`.ascii s`	riserva spazio per la stringa "s" nel segmento dati, senza aggiungere il terminatore di stringa
`.asciiz s`	riserva spazio per la stringa "s" nel segmento dati e aggiungi il terminatore di stringa
`.byte n1, …`	riserva spazio nel segmento dati per i byte elencati e inizializzali con i valori a 8 bit "n1", ...
`.data i`	dichiara il segmento dati con indirizzo iniziale "i" (default 0x 1000 0000)
`.eqv s, v`	dichiara un simbolo: il simbolo "s" ha valore numerico "v", senza allocare memoria (funziona come la #define in C ed è usato soprattutto per definire lo spiazzamento in pila) ATTENZIONE: il simbolo "s" dichiarato E' SOLO LOCALE AL MODULO
`.globl s`	esporta il simbolo "s" come etichetta iniziale del programma

Direttive di assemblatore

`.half h1, …`	riserva spazio nel segmento dati per le mezze parole elencate e inizializzale con i valori a 16 bit "h1", ..., allneandole a indirizzi pari (con .align si può cambiare allineamento)
`.space n`	riserva spazio nel segmento dati per "n" byte, senza inizializzarli (si usa principalmente per vettore e struct, e in luogo di .word, .half, .byte se il valore iniziale manca)
`.text i`	dichiara il segmento testo (codice) con indirizzo iniziale "i" (default 0x 0040 0000)
`.word w1, …`	riserva spazio nel segmento dati per le parole elencate e inizializzale con i valori a 32 bit "w1", ..., allneandole a indirizzi multipli di quattro (.align cambia allineamento)

Altre direttive

`.kdata i`	dichiara il segmento dati di kernel con indirizzo iniziale "i"
`.ktext i`	dichiara il segmento testo di kernel con indirizzo iniziale "i"
`.float f1, …`	riserva spazio nel seg. dati per i numeri reali elencati
`.macro a1, …`	dichiarazione di macro con argomenti "a1", ...
`.end_macro`	fine dichiarazione di macro

Simboli locali e globali

- Per il collegatore (linker) un simbolo (etichetta di variabile o di istruzione) dichiarato in un modulo può essere:
  - locale, cioè visibile solo nel modulo dove è dichiarato
    - non va nella tabella dei simboli del modulo (e neppure globale)
  - globale, cioè visibile in tutti i moduli da collegare
    - va nella tabella dei simboli del modulo e poi in quella globale
- Se i simboli siano automaticamente locali o globali, in genere dipende dal particolare collegatore in uso
- Per MIPS si suppone che i simboli siano sempre globali
  - altrimenti ci sarebbero direttive per «esportare» e «importare» un simbolo, per esempio le direttive **.globl** ed **.extern**
- Ma un simbolo dichiarato tramite **.eqv** è solo LOCALE
  - ha lo stesso uso della direttiva **#define** del preprocessore C

Modello di programma MIPS completato

gli indirizzi iniziali (o di impianto) dei segmenti sono virtuali (non fisici)

```
// costanti
#define COST VAL
...
// dichiarazioni
...
...
...
// parti esecutive
funz (...) {
   // corpo
   ...
}
main (...) {
   // corpo
   ...
}
```

```
      // costanti (non ingombrano spazio di memoria)
      .eqv COST, VAL   // costanti LOCALI del modulo
      ...
      // segmento dati (ingombra spazio di memoria)
...   .data ind. iniz. dati   // default 0x10000000
...   // varglob dichiarate tramite le direttive
...   // .space, .word, .half, .byte, .ascii, ecc
      // segmento codice (ingombra spazio di memoria)
      .text ind. iniz. codice // default 0x00400000
      // codice di main e delle funzioni utente
      .globl MAIN // esporta MAIN come etichetta iniz.
FUNZ: ...         // codice funzione utente
      ...
MAIN: ...         // codice programma principale
      ...
```

Riassunto su come accedere a variabili

Variabile globale (nel segmento dati)

C	assemblatore semplificato	espansione in assemblatore nativo (*gcc*)

`int a`
 	
`leggi a`
	`A:  .space 4 (.word se iniz.)`
	
	`lw  $t0, A`
	`// var. a caricata in t0`
		`A:  space 4 (.word se iniz.)`
		
		`lw  $t0, A($gp)`
		`// var. a caricata in t0`

`int vet [5]`
 	
`leggi vet [1]`
	`VET: .space 20`
	
	`la  $t0, VET`
	
	`lw  $t1, 4($t0)`
	`// elem. vet[1] caricato in t1`
		`VET: .space 20`
		
		`lui  $t0, VET`
		`addiu $t0, $t0, VET`
		`lw  $t1, 4($t0)`
		`// elem. vet[1] caricato in t1`

`int i`
 	
`int vet [5]`
 	
`leggi vet [i]`
	`// var. i già caricata in t1`
	
	`VET: .space 20`
	
	`la  $t0, VET`
	
	`sll  $t1, $t1, 2`
	`addu $t0, $t0, $t1`
	`lw  $t2, ($t0)`
	`// elem. vet[i] caricato in t2`
		`// var. i già caricata in t1`
		
		`VET: .space 20`
		
		`lui  $t0, VET`
		`addiu $t0, $t0, VET`
		`sll  $t1, $t1, 2`
		`addu $t0, $t0, $t1`
		`lw  $t2, ($t0)`
		`// elem. vet[i] caricato in t2`

per scrivere usa «sw» invece di «lw»

Variabile locale (nell'area di attivazione)

C	assemblatore semplificato	espansione in assemblatore nativo (*gcc*)

char c
leggi **c**
 	
scrivi **c**
	C: **.space** 1 (**.byte** se iniz.)
	lbu $t0, **C**
	// var. c caricata in t0
	sb $t0, **C**
	// t0 memorizzato in var. c
		C: **.space** 1 (**.byte** se iniz.)
		lbu $t0, **C**($gp)
		// var. c caricata in t0
		sb $t0, **C**($gp)
		// t0 memorizzato in var. c

char vet [5] leggi **vet** [**1**]	**VET:** **.space** 5 **la** $t0, **VET** **lbu** $t1, **1**($t0) // elem. vet[1] caricato in t1	**VET:** **.space** 5 **lui** $t0, **VET** **addiu** $t0, $t0, **VET** **lbu** $t1, **1**($t0) // elem. vet[1] caricato in t1

int i
char vet [5]
leggi **vet** [**i**]
	// var. i già caricata in t1
	VET: **.space** 5
	la $t0, **VET**
	
	addu $t0, $t0, **$t1**
	lbu $t2, ($t0)
	// elem. vet[i] caricato in t2
		// var. i già caricata in t1
		VET: **.space** 5
		lui $t0, **VET**
		addiu $t0, $t0, **VET**
		addu $t0, $t0, **$t1**
		lbu $t2, ($t0)
		// elem. vet[i] caricato in t2

per scrivere usa «sw» invece di «lw»

Variabile locale (nell'area di attivazione) - ottimizzato

C	assemblatore ottimizzato (nativo)	spiegazione

`void f ( ) {`
`  int a`
 	
`  leggi a`
`}`
	`// spi. di a in area attiv.`
	`.eqv  A, ...`
	
	`lw  $t0, A($sp)`
	`// var. a caricata in t0`	

`void f ( ) {`
`  int vet [5]`
 	
`  leggi vet [1]`
 	
`}`
	`// spi. di vet in area attiv.`
	`.eqv  VET, ...`
	
	`addiu  $t0, $sp, VET`
	`lw  $t1, 4($t0)`
	`// elem. vet[1] caricato in t1`
		pseudo-istruzione «move» per trasferire lo stack pointer `$sp` rimossa e sua operazione unificata con l'istruzione «addiu» consecutiva

`int i`
`void f ( ) {`
`  int vet [5]`
 	
`  leggi vet [i]`
 	
`}`
	`// var. i già caricata in t1`
	`// spi. di vet in area attiv.`
	`.eqv  VET, ...`
	
	`addiu  $t0, $sp, VET`
	`sll  $t1, $t1, 2`
	`addu  $t0, $t0, $t1`
	`lw  $t2, ($t0)`
	`// elem. vet[i] caricato in t2`
		pseudo-istruzione «move» per trasferire lo stack pointer `$sp` rimossa e sua operazione unificata con l'istruzione «addiu» consecutiva

per scrivere usa «sw» invece di «lw»

Variabile globale (nel segmento dati) - carattere

C	assemblatore semplificato	espansione in assemblatore nativo (*gcc*)

char c
leggi **c**
 	
scrivi **c**
	C: **.space** 1 (**.byte** se iniz.)
	lbu $t0, **C**
	// var. c caricata in t0
	sb $t0, **C**
	// t0 memorizzato in var. c
		C: **.space** 1 (**.byte** se iniz.)
		lbu $t0, **C**($gp)
		// var. c caricata in t0
		sb $t0, **C**($gp)
		// t0 memorizzato in var. c

char vet [5] leggi **vet** [**1**]	**VET:** **.space** 5 **la** $t0, **VET** **lbu** $t1, **1**($t0) // elem. vet[1] caricato in t1	**VET:** **.space** 5 **lui** $t0, **VET** **addiu** $t0, $t0, **VET** **lbu** $t1, **1**($t0) // elem. vet[1] caricato in t1

int i
char vet [5]
leggi **vet** [**i**]
	// var. i già caricata in t1
	VET: **.space** 5
	la $t0, **VET**
	
	addu $t0, $t0, **$t1**
	lbu $t2, ($t0)
	// elem. vet[i] caricato in t2
		// var. i già caricata in t1
		VET: **.space** 5
		lui $t0, **VET**
		addiu $t0, $t0, **VET**
		addu $t0, $t0, **$t1**
		lbu $t2, ($t0)
		// elem. vet[i] caricato in t2

l'istruzione «lbu» carica solo un byte di memoria e non modifica i tre byte più significativi del registro destinazione; l'istruzione «sb» memorizza solo il byte meno significativo del registro sorgente

vedi elenco per altre varianti di «lw» e «sw»