



POLITECNICO
MILANO 1863

Architettura dei calcolatori e sistemi operativi

Set istruzioni e struttura del programma
Direttive all'Assemblatore

Capitolo 2 P&H

Sommario

Istruzioni

Formati istruzioni

Struttura del programma e direttive all'assemblatore



Sottoinsieme del linguaggio assembler MIPS

Operandi MIPS

Nome	Esempio	Commenti
32 registri	\$s0-\$s7, \$t0-\$t9, \$zero, \$a0-\$a3, \$v0-\$v1, \$gp, \$fp, \$sp, \$ra, \$at	Accesso veloce ai dati. Nel MIPS gli operandi devono essere contenuti nei registri per potere eseguire delle operazioni. Il registro \$zero contiene sempre il valore 0, e il registro \$at viene riservato all'assemblatore per la gestione di costanti molto lunghe.
2 ³⁰ parole di memoria	Memoria[0], Memoria[4], ..., Memoria[4294967292]	Alla memoria si accede solamente attraverso le istruzioni di trasferimento dati. Il MIPS utilizza l'indirizzamento al byte, perciò due parole consecutive hanno indirizzi in memoria a una distanza di 4. La memoria consente di memorizzare strutture dati, vettori, o il contenuto dei registri.

Linguaggio assembler MIPS

Tipo di istruzioni	Istruzioni	Esempio	Significato	Commenti
Aritmetiche	Somma	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Operandi in tre registri
	Sottrazione	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Operandi in tre registri
	Somma immediata	addi \$s1,\$s2,20	$\$s1 = \$s2 + 20$	Utilizzata per sommare delle costanti
Trasferimento dati	Lettura parola	lw \$s1,20(\$s2)	$\$s1 = \text{Memoria}[\$s2+20]$	Trasferimento di una parola da memoria a registro
	Memorizzazione parola	sw \$s1,20(\$s2)	$\text{Memoria}[\$s2+20] = \$s1$	Trasferimento di una parola da registro a memoria
	Lettura mezza parola	lh \$s1,20(\$s2)	$\$s1 = \text{Memoria}[\$s2+20]$	Trasferimento di una mezza parola da memoria a registro
	Lettura mezza parola, senza segno	lhu \$s1,20(\$s2)	$\$s1 = \text{Memoria}[\$s2+20]$	Trasferimento di una mezza parola da memoria a registro
	Memorizzazione mezza parola	sh \$s1,20(\$s2)	$\text{Memoria}[\$s2+20] = \$s1$	Trasferimento di una mezza parola da registro a memoria
	Lettura byte	lb \$s1,20(\$s2)	$\$s1 = \text{Memoria}[\$s2+20]$	Trasferimento di un byte da memoria a registro
	Lettura byte senza segno	lbu \$s1,20(\$s2)	$\$s1 = \text{Memoria}[\$s2+20]$	Trasferimento di un byte da memoria a registro
	Memorizzazione byte	sb \$s1,20(\$s2)	$\text{Memoria}[\$s2+20] = \$s1$	Trasferimento di un byte da registro a memoria
	Lettura di una parola e blocco	ll \$s1,20(\$s2)	$\$s1 = \text{Memoria}[\$s2+20]$	Caricamento di una parola come prima fase di un'operazione atomica
	Memorizzazione condizionata di una parola	sc \$s1,20(\$s2)	$\text{Memoria}[\$s2+20] = \$s1$; $\$s1 = 0$ oppure 1	Memorizzazione di una parola come seconda fase di un'operazione atomica
	Caricamento costante nella mezza parola superiore	lui \$s1,20	$\$s1 = 20 * 2^{16}$	Caricamento di una costante nei 16 bit più significativi
Logiche	And	and \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$	Operandi in tre registri; AND bit a bit
	Or	or \$s1,\$s2,\$s3	$\$s1 = \$s2 \mid \$s3$	Operandi in tre registri; OR bit a bit
	Nor	nor \$s1,\$s2,\$s3	$\$s1 = \sim(\$s2 \mid \$s3)$	Operandi in tre registri; NOR bit a bit

Tipo di istruzioni	Istruzioni	Esempio	Significato	Commenti
Logiche (segue)	And immediato	andi \$s1,\$s2,20	$\$s1 = \$s2 \& 20$	And bit a bit tra un operando in registro e una costante
	Or immediato	ori \$s1,\$s2,20	$\$s1 = \$s2 \mid 20$	OR bit a bit tra un operando in registro e una costante
	Scorrimento logico a sinistra	sll \$s1,\$s2,10	$\$s1 = \$s2 \ll 10$	Spostamento a sinistra del numero di bit specificato dalla costante
	Scorrimento logico a destra	srl \$s1,\$s2,10	$\$s1 = \$s2 \gg 10$	Spostamento a destra del numero di bit specificato dalla costante
Salti condizionati	Salta se uguale	beq \$s1,\$s2,25	Se $(\$s1 == \$s2)$ vai a PC+4+100	Test di uguaglianza; salto relativo al PC
	Salta se non è uguale	bne \$s1,\$s2,25	Se $(\$s1 != \$s2)$ vai a PC+4+100	Test di disuguaglianza; salto relativo al PC
	Poni uguale a 1 se minore	slt \$s1,\$s2,\$s3	Se $(\$s2 < \$s3)$ $\$s1 = 1$; altrimenti $\$s1 = 0$	Comparazione di minoranza; utilizzata con bne e beq
	Poni uguale a uno se minore, numeri senza segno	sltu \$s1,\$s2,\$s3	Se $(\$s2 < \$s3)$ $\$s1 = 1$; altrimenti $\$s1 = 0$	Comparazione di minoranza su numeri senza segno
	Poni uguale a uno se minore, immediato	slti \$s1,\$s2,20	Se $(\$s2 < 20)$ $\$s1 = 1$; altrimenti $\$s1 = 0$	Comparazione di minoranza con una costante
	Poni uguale a uno se minore, immediato e senza segno	sltiu \$s1,\$s2,20	Se $(\$s2 < 20)$ $\$s1 = 1$; altrimenti $\$s1 = 0$	Comparazione di minoranza con una costante, con numeri senza segno
	Salto incondizionato	j 2500	Vai a 10000	Salto all'indirizzo della costante
	Salto indiretto	jr \$ra	Vai all'indirizzo contenuto in \$ra	Salto all'indirizzo contenuto nel registro, utilizzato per il ritorno da procedura e per i costrutti switch
Salti incondizionati	Salta e collega	jal 2500	$\$ra = PC+4$; vai a 10000	Chiamata a procedura



Istruzioni aritmetico-logiche

In MIPS, un'istruzione aritmetico-logica ha *tre* operandi

- due **registri sorgente** contenenti i valori da elaborare
- un **registro destinazione** contenente il risultato

È quindi di tipo R e l'ordine degli operandi è **fisso**

❑ In assembler il formato istruzione è

OPCODE DEST, SORG1, SORG2

dove **DEST, SORG1, SORG2** sono registri referenziabili del MIPS



Istruzioni aritmetico-logiche (2)

Istruzioni di somma e sottrazione

`add rd, rs, rt` $\# \text{ rd} \leftarrow \text{rs} + \text{rt}$

`sub rd, rs, rt` $\# \text{ rd} \leftarrow \text{rs} - \text{rt}$

`addu` e `subu` lavorano in modo analogo su operandi senza segno (indirizzi) e non generano segnalazione di traboccamento (overflow)

Istruzioni logiche `and`, `or` e `nor` lavorano in modo analogo

`or rd, rs, rt` $\# \text{ rd} \leftarrow \text{rs or rt (OR bit a bit)}$

Istruzioni di scorrimento logico

`sll rd, rs, 10` $\# \text{ rd} \leftarrow \text{rs} \ll 10$

`srl rd, rs, 10` $\# \text{ rd} \leftarrow \text{rs} \gg 10$



Varianti con operando immediato (costante)

`addi rd, rs, imm` $\# \text{ rd} \leftarrow \text{rs} + \text{imm}$

`addiu rd, rs, imm` $\# \text{ rd} \leftarrow \text{rs} + \text{imm}$ (no overflow)



Qualche esempio

Codice C: **R = A + B**

Codice MIPS: **add \$s0, \$s1, \$s2**

nella traduzione da C
a linguaggio assembler
le variabili sono state associate
ai registri dal compilatore

Il fatto che ogni istruzione aritmetica abbia tre operandi sempre nella stessa posizione consente di semplificare lo HW, ma complica alcune cose...

Codice C: **A = B + C + D**

E = F - A

Codice MIPS: **add \$t0, \$s1, \$s2**

add \$s0, \$t0, \$s3

sub \$s4, \$s5, \$s0

A	\$s0
B	\$s1
C	\$s2
D	\$s3
E	\$s4
F	\$s5



Qualche esempio (2)

Espressioni con un numero di operandi maggiore di tre possono essere effettuate scomponendole in operazioni più semplici

Per esempio

Codice C: **A = B + C + D + E**

Codice MIPS: **add \$t0, \$s1, \$s2**
 add \$t0, \$t0, \$s3
 add \$s0, \$t0, \$s4



Istruzioni di trasferimento dati: *Load / Store*

MIPS fornisce due operazioni base per il trasferimento dei dati:

- **lw** (load word) per trasferire una parola di memoria in un registro
- **sw** (store word) per trasferire il contenuto di un registro in una parola di memoria

lw e *sw* hanno **due operandi**

- il registro destinazione (*lw*) o sorgente (*sw*) del trasferimento dei dati
- la parola di memoria coinvolta nel trasferimento (identificata dal suo indirizzo)



Istruzioni di trasferimento dati: *Load / Store* (2)

In **linguaggio macchina** MIPS l'indirizzo della parola di memoria coinvolta nel trasferimento viene sempre specificato secondo la modalità

offset(registro_base) dove

- *offset* è intendersi come spiazzamento su 16 bit e
- l'indirizzo della parola di memoria è dato dalla somma tra il valore immediato *offset* e il contenuto del *registro base*

In linguaggio macchina le istruzioni *lw / sw* hanno quindi ***tre argomenti***

- **registro coinvolto nel trasferimento**
- **offset** valore immediato (su 16 bit) che rappresenta lo spiazzamento rispetto al registro base, e può valere anche 0
- **registro base**



Istruzioni di trasferimento dati: *Load / Store* (3)

In *linguaggio assembler* MIPS l'indirizzo della parola di memoria coinvolta nel trasferimento può essere espresso in modo più *flessibile* come somma di

identificatore + espressione + registro

dove ciascuna parte può essere omessa e

- l'identificatore è costituito da un *simbolo rilocabile* che si riferisce all'area dati che contiene le *variabili globali* del programma; se è l'identificatore simbolico di una variabile scalare globale, la parte di indirizzo che deriva dell'identificatore è calcolata come (*\$gp*) + *offset* della variabile in area dati globale; il valore dell'offset è calcolato dal collegatore (linker)
- l'espressione può essere anche una costante con segno (*offset*)
- il registro è il *registro base*

```
lw $t0, ($a0)      # $t0 ← M[$a0 + 0]
lw $t0, 20($a0)     # $t0 ← M[$a0 + 20]
lw $t0, var1        # $t0 ← M[$gp + offset di var1 area dati globale]
```



Qualche esempio

```
lw $s1, 100($s2)    # $s1 ← M[$s2 + 100]
sw $s1, 100($s2)    # M[$s2 + 100] ← $s1
```

Codice C: **A[12] = h + A[8];**

- variabile **h** associata al registro **\$s2**
- indirizzo del primo elemento dell'array **A** (*base address*) contenuto nel registro **\$s3**

Codice MIPS:

```
lw    $t0, 32($s3)    # $t0 ← M[$s3 + 32]
add   $t0, $s2, $t0    # $t0 ← $s2 + $t0
sw    $t0, 48($s3)    # M[$s3 + 48] ← $t0
```



Ancora sugli array (vettori)

Sia A un array di 100 interi su 32 bit

Istruzione C: $g = h + A[i]$

- le variabili **g**, **h**, **i** siano associate rispettivamente ai registri **\$s1**, **\$s2**, ed **\$s4**
- l'indirizzo del primo elemento dell'array (*base address*) sia contenuto nel registro **\$s3**
- l'elemento **i-esimo** di un array si troverà nella locazione **base_address + 4 × i**
 - il fattore 4 dipende dall'indirizzamento al byte della memoria nel MIPS



Ancora sugli array (cont.)

L'elemento i -esimo dell'array si trova nella locazione di memoria di indirizzo $(\$s3 + 4 \times i)$

- *indirizzo di $A[i]$ nel registro temporaneo $\$t1$*

```
add $t1, $s4, $s4    # $t1 ← 2 × i
add $t1, $t1, $t1    # $t1 ← 4 × i
add $t1, $t1, $s3    # $t1 ← add. of A[i]
                    # that is ($s3 + 4 × i)
```

Oppure

```
sll $t1, $s4, 2      # $t1 ← 4 × i
add $t1, $t1, $s3
```

- *$A[i]$ nel registro temporaneo $\$t0$*

```
lw $t0, 0($t1)       # $t0 ← A[i]
```

- *somma di h e $A[i]$ e risultato in g*

```
add $s1, $s2, $t0    # g = h + A[i]
```



Costrutti di controllo e istruzioni di salto

Istruzioni di salto condizionato / incondizionato

- alterano l'ordine di esecuzione delle istruzioni
 - la prossima istruzione da eseguire non è necessariamente l'istruzione successiva all'istruzione corrente, ma quella individuata dalla **destinazione del salto**
- permettono di realizzare i costrutti di controllo condizionali e ciclici



Istruzioni di salto

In linguaggio assembler si specifica l'indirizzo dell'istruzione destinazione di salto tramite un *nome simbolico* che è costituito da un'*etichetta* (*label*) associata appunto all'istruzione destinazione

In *linguaggio macchina* MIPS

- le istruzioni di salto *condizionato* sono di tipo *I*: la modalità di indirizzamento è quella *relativa al PC* ($\text{new_PC} = \text{PC} + \text{offset}$) con spiazzamento (offset) di 16 bit
- le istruzioni di salto *incondizionato* sono di tipo *J*: la modalità di indirizzamento è quella *pseudo-diretta*

Quindi partendo da *etichetta* l'operazione di traduzione in linguaggio macchina dell'indirizzo destinazione sarà diversa nei due casi



Istruzioni di salto condizionato

Istruzioni di **salto condizionato** (*conditional branch*): il salto viene eseguito solo se una certa condizione risulta soddisfatta

`beq` (*branch on equal*)

`bne` (*branch on not equal*)

```
beq r1, r2, label_1      # go to label_1 if (r1 == r2)
```

```
bne r1, r2, label_1      # go to label_1 if (r1 != r2)
```

È possibile utilizzarle insieme ad altre istruzioni per realizzare salti condizionati su esito di maggioranza o minoranza (vedi più avanti)



Istruzioni di salto incondizionato

Istruzioni di **salto incondizionato** (*unconditional jump*):
il salto viene sempre eseguito

<code>j</code>	(<i>jump</i>)
<code>jr</code>	(jump register – ritorno da sottoprogramma)
<code>jal</code>	(jump and link – chiamata di sottoprogramma)

<code>j</code>	<code>L1</code>	<code># go to L1</code>
<code>jr</code>	<code>\$ra</code>	<code># go to address contained in \$ra</code>
<code>jal</code>	<code>L1</code>	<code># go to L1. Save add. of next</code>
		<code># instruction in reg. \$ra</code>



Esempio if ... then ... else

Codice C:

```
if (i == j) f = g + h;  
else f = g - h;
```

Si suppone che le variabili **f**, **g**, **h**, **i** e **j** siano associate rispettivamente ai registri **\$s0**, **\$s1**, **\$s2**, **\$s3** e **\$s4**

Codice MIPS:

```
        bne      $s3, $s4, Else      # go to ELSE if i≠j  
        add      $s0, $s1, $s2      # f=g+h (skipped if i ≠ j)  
        j        END_IF             # go to END_IF  
ELSE:    sub      $s0, $s1, $s2      # f=g-h (skipped if i = j)  
END_IF:  ...
```



Condizioni di salto

registro `$zero`

spesso la verifica di uguaglianza richiede il confronto con il valore 0 per rendere più veloce il confronto, in MIPS il registro `$zero` contiene il valore 0 e non può mai essere utilizzato per contenere altri valori

Il processore tiene traccia di alcune informazioni sui risultati di operazioni per usarle nelle condizioni di successive istruzioni di salto condizionato

- queste informazioni sono memorizzate in bit o flag denominati *codici di condizione*
- il registro di stato o registro dei codici di condizione contiene i flag dei codici di condizione

I codici di condizione più usati sono:

N (negativo)	# posto a 1 se il risultato è negativo; # altrimenti posto a 0
Z (zero)	# posto a 1 se il risultato è zero; # altrimenti posto a 0
V (overflow)	# posto a 1 se si verifica un overflow aritmetico; # altrimenti posto a 0
C (riporto)	# posto a 1 se dall'operazione risulta un riporto; # altrimenti posto a 0



Ancora sulle istruzioni di salto condizionato

Spesso è utile condizionare l'esecuzione di una istruzione al fatto che una variabile sia minore di una altra

```
slt $s1, $s2, $s3      # set on less than
```

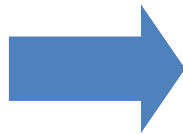
assegna il valore **1** a **\$s1** se **\$s2 < \$s3**; altrimenti assegna il valore **0**

Con **slt**, **beq** e **bne** si possono realizzare i test sui valori di due variabili (**=**, **!=**, **<**, **<=**, **>**, **>=**)



Esempio

```
if (i < j)  k = i + j;  
else k = i - j;
```



#\$s0 ed \$s1 contengono i e j
#\$s2 contiene k

```
    slt $t0, $s0, $s1  
    beq $t0, $zero, ELSE  
    add $s2, $s0, $s1  
    j    EXIT  
ELSE: sub $s2, $s0, $s1  
EXIT:
```



Pseudo-istruzioni

Per semplificare la programmazione, MIPS fornisce un insieme di *pseudo-istruzioni*

- ❑ le pseudoistruzioni sono un modo compatto e intuitivo di specificare *un insieme di istruzioni*
- ❑ la traduzione della pseudo-istruzione nelle istruzioni equivalenti è attuata automaticamente dall'assemblatore

```
move $t0, $t1
```

```
▪ add $t0, $zero, $t1
```

```
mul $s0, $t1, $t2
```

```
▪ mult $t1, $t2
```

```
▪ mflo $s0
```



ISTRUZIONI E PSEUDOISTRUZIONI

ARITMETICA

add	\$s1, \$s1, \$s3	$s1 := s2 + s3$	addizione
addu	\$s1, \$s1, \$s3	$s1 := s2 + s3$	addizione naturale
addi	\$s1, \$s2, cost	$s1 := s2 + \text{cost}$	addizione di costante
addiu	\$s1, \$s2, cost	$s1 := s2 + \text{cost}$	addizione naturale di costante
sub	\$s1, \$s2, \$s3	$s1 := s2 - s3$	sottrazione
subu	\$s1, \$s2, \$s3	$s1 := s2 - s3$	sottrazione naturale

ARITMETICA – pseudoistruzioni

subi	\$s1, \$s2, cost	$s1 := s2 - \text{cost}$	sottrazione di costante
subiu	\$s1, \$s2, cost	$s1 := s2 - \text{cost}$	sottrazione naturale di costante
neg	\$s1, \$s2	$s1 := -s2$	negazione aritmetica

CONFRONTO

slt	\$s1, \$s2, \$s3	if $s2 < s3$ then $s1 := 1$ else $s1 := 0$	poni a 1 se minore stretto
sltu	\$s1, \$s2, \$s3	if $s2 < s3$ then $s1 := 1$ else $s1 := 0$	poni a 1 se minore str. nat.
slti	\$s1, \$s2, cost	if $s2 < \text{cost}$ then $s1 := 1$ else $s1 := 0$	poni a 1 se minore str. cost.
sltiu	\$s1, \$s2, cost	if $s2 < \text{cost}$ then $s1 := 1$ else $s1 := 0$	poni a 1 se min. str. cost. nat.



ISTRUZIONI E PSEUDOISTRUZIONI

LOGICA

or	\$s1, \$s2, \$s3	s1 := s2 or s3	somma logica bit a bit
and	\$s1, \$s2, \$s3	s1:= s2 and s3	prodotto logico bit a bit
ori	\$s1, \$s2, cost	s1:= s2 or cost	somma logica bit a bit costante
andi	\$s1, \$s2, cost	s1:= s2 or cost	prodotto logico bit a bit costante
nor	\$s1, \$s2, \$s3	s1:= s2 nor s3	somma logica negata bit a bit
sll	\$s1, \$s2, cost	s1:= s2 << cost	scorrimento a sinistra (left) del n° di bit specificato da cost
srl	\$s1, \$s2, cost	s1:= s2 >> cost	scorrimento a destra (right) del n° di bit specificato da cost

LOGICA – pseudoistruzioni

not	\$s1, \$s2	s1 = not s2	(p) negazione logica
-----	------------	--------------------	----------------------



ISTRUZIONI E PSEUDOISTRUZIONI

SALTO INCONDIZIONATO E CON COLLEGAMENTO

j	indir	PC := cost (28 bit)	salto incondizionato assoluto
jr	\$r	PC := r (32 bit)	salto indiretto da registro
jal	indir	PC := cost (28 bit) e collega \$ra	salto assoluto e collegamento

SALTO CONDIZIONATO

beq	\$s1, \$s2, spi	if s2 = s1 salta rel. a PC	salto cond. di uguaglianza
bne	\$s1, \$s2, spi	if s2 ≠ s1 salta rel. a PC	salto cond. di disuguaglianza

SALTO CONDIZIONATO - pseudoistruzioni

blt	\$s1, \$s2, spi	if s2 < s1 salta rel. a PC	salta se minore stretto
bgt	\$s1, \$s2, spi	if s2 > s1 salta rel. a PC	salta se maggiore stretto
ble	\$s1, \$s2, spi	if s2 ≤ s1 salta rel. a PC	salta se minore o uguale
bge	\$s1, \$s2, spi	if s2 ≥ s1 salta rel. a PC	salta se maggiore o uguale



ISTRUZIONI E PSEUDOISTRUZIONI

TRASFERIMENTO MEMORIA

lw	\$s1, spi (\$s2)	$s1 := \text{mem}(s2 + \text{spi})$	carica parola (a 32 bit)
sw	\$s1, spi (\$s2)	$\text{mem}(s2 + \text{spi}) := s1$	memorizza parola (a 32 bit)
lh, lhu	\$s1, spi (\$s2)	$s1 := \text{mem}(s2 + \text{spi})$	carica mezza parola (a 16 bit)
sh	\$s1, spi (\$s2)	$\text{mem}(s2 + \text{spi}) := s1$	memor. mezza parola (a 32 bit)
lb, lbu	\$s1, spi (\$s2)	$s1 := \text{mem}(s2 + \text{spi})$	carica byte (a 8 bit)
sb	\$s1, spi (\$s2)	$\text{mem}(s2 + \text{spi}) := s1$	memorizza byte (a 8 bit)

TRASFERIMENTO in registro di COSTANTE

lui	\$s1, cost	$s1 \text{ (16 bit più signif.)} := \text{cost}$	carica cost (in 16 bit più signifi)
-----	------------	--	-------------------------------------

TRASFERIMENTI tra REGISTRI e di COSTANTI/INDIRIZZI – pseudo istruzioni

move	\$d, \$s	$d := s$	copia registro
la	\$d, indir	$d := \text{indir (32 bit)}$	carica indirizzo a 32 bit
li	\$d, cost	$d := \text{cost (32 bit)}$	carica costante a 32 bit



REGISTRI

REGISTRI REFERENZIABILI

0	0	costante 0
1	at	uso riservato all'assembler-linker
2-3	v0 - v1	valore restituito da funzione
4-7	a0-a3	argomenti in ingresso a funzione
8-15	t0-t7	registri per valori temporanei
16-23	s0-s7	registri
24-25	t8-t9	registri per valori temporanei (in aggiunta a $t0-t7$), come i precedenti t
26-27	k0-k1	registri riservati per il nucleo del SO
28	gp	global pointer (puntatore all'area dati globale)
29	sp	stack pointer (puntatore alla pila)
30	fp	frame pointer (puntatore area di attivazione)
31	ra	registro return address

REGISTRI NON REFERENZIABILI

	pc	Program Counter
	hi	Registro per risultato moltiplicazioni e divisioni
	lo	Registro per risultato moltiplicazioni e divisioni



FORMATI ISTRUZIONE



Formato istruzioni di tipo R - aritmetiche

Formato usato per istruzioni aritmetico-logiche

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

Ai vari campi sono stati assegnati dei nomi mnemonici:

- **op: (opcode)** identifica il tipo di istruzione (**0**)
- **rs:** registro contenente il primo operando sorgente
- **rt:** registro contenente il secondo operando sorgente
- **rd:** registro destinazione contenente il risultato
- **shamt:** shift amount (scorrimento)
- **funct:** indica la variante specifica dell'operazione



Istruzioni di tipo R: esempi

add \$s1, \$s2, \$s3

Nome campo	op	rs	rt	rd	shamt	funct
Dimensione	6-bit	5-bit	5-bit	5-bit	5-bit	6-bit
add \$s1, \$s2, \$s3	000000	10010	10011	10001	00000	100000

Nome campo	op	rs	rt	rd	shamt	funct
Dimensione	6-bit	5-bit	5-bit	5-bit	5-bit	6-bit
sub \$s1, \$s2, \$s3	000000	10010	10011	10001	00000	100010



Formato istruzioni di tipo I – lw/sw

op	rs	rt	indirizzo
6 bit	5 bit	5 bit	16 bit

Nel caso di istruzioni load/store, i campi hanno il seguente significato:

- **op** (**opcode**) identifica il tipo di istruzione; (**35 o 43**)
- **rs** indica il registro base;
- **rt** indica il registro destinazione dell'istruzione load o il registro sorgente dell'istruzione store;
- **indirizzo** riporta lo spiazzamento (offset)

Con questo formato, un'istruzione **lw (sw)** può indirizzare parole nell'intervallo -2^{15} $+2^{15}-1$ rispetto all'indirizzo base



Istruzioni di tipo I: esempi lw e sw

lw \$t0, 32(\$s3)

Nome campo	op	rs	rt	indirizzo			
Dimensione	6-bit	5-bit	5-bit	16-bit			
lw \$t0, 32 (\$s3)	100011	10011	01000	0000	0000	0010	0000

Nome campo	op	rs	rt	indirizzo			
Dimensione	6-bit	5-bit	5-bit	16-bit			
sw \$t0, 32 (\$s3)	101011	10011	01000	0000	0000	0010	0000



Formato istruzioni di tipo I – con immediato

op	rs	rt	indirizzo
6 bit	5 bit	5 bit	16 bit

Nel caso di istruzioni **con immediati**, i campi hanno il seguente significato:

- **op** (**opcode**) identifica il tipo di istruzione;
- **rs** indica il registro sorgente;
- **rt** indica il registro destinazione;
- **indirizzo** contiene il valore dell'operando immediato

Con questo formato, un'istruzione con immediato può contenere costanti nell'intervallo -2^{15} $+2^{15}-1$



Istruzioni di tipo I: esempi con operando immediato

addi \$s1, \$s1, 4

Nome campo	op	rs	rt	indirizzo			
Dimensione	6-bit	5-bit	5-bit	16-bit			
addi \$s1, \$s1, 4	001000	10001	10001	0000	0000	0000	0100

Nome campo	op	rs	rt	indirizzo			
Dimensione	6-bit	5-bit	5-bit	16-bit			
slti \$t0, \$s2, 8	001010	10010	01000	0000	0000	0000	1000

\$t0 =1 if \$s2 < 8

slti \$t0, \$s2, 8



Formato istruzioni di tipo I - branch

op	rs	rt	indirizzo
6 bit	5 bit	5 bit	16 bit

Nel caso di salti condizionati, i campi hanno il seguente significato:

- **op (opcode)** identifica il tipo di istruzione (**4 = beq**)
- **rs** indica il primo registro;
- **rt** indica il secondo registro;
- **indirizzo** riporta lo spiazzamento (offset)

Per l'offset si hanno a disposizione solo 16-bit del campo **indirizzo** \Rightarrow rappresentano un indirizzo di **parola** relativo al PC (**PC-relative word address**)



Istruzioni di salto condizionato (tipo I)

I 16-bit del campo indirizzo esprimono l'**offset** rispetto al PC rappresentato in complemento a due per permettere salti in avanti e all'indietro

L'offset varia tra -2^{15} e $+2^{15}-1$

Esempio: `bne $s0, $s1, L1`

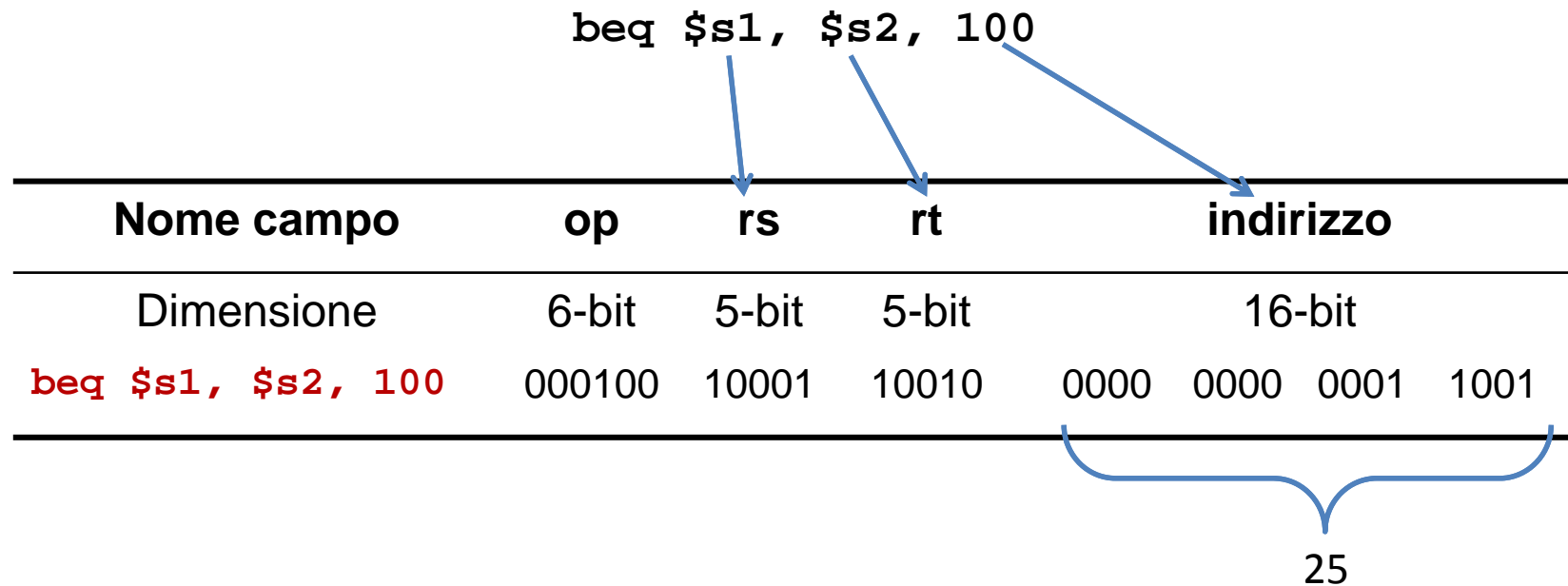
L'assemblatore sostituisce l'etichetta `L1` con l'offset **di parola** relativo a PC: **$(L1 - PC)/4$**

- PC contiene già l'indirizzo dell'istruzione successiva al salto
- La divisione per 4 serve per calcolare l'offset di parola

Il valore del campo **indirizzo** può essere negativo (salti all'indietro)

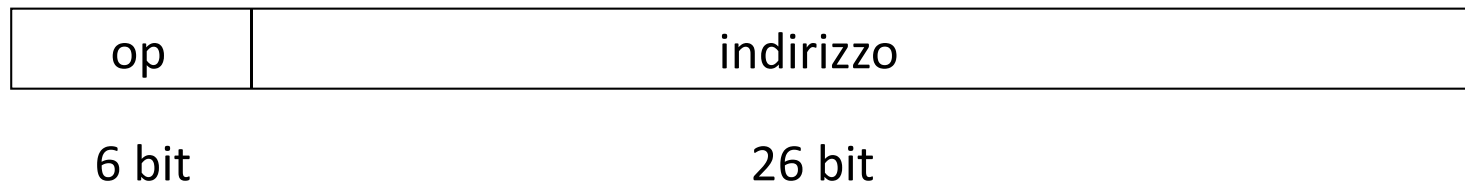


Istruzioni di tipo I: esempio



Formato istruzioni di tipo J

È il formato usato per le istruzioni di salto incondizionato (*jump*), per esempio `j L1`



In questo caso, i campi hanno il seguente significato:

- **op (opcode)** indica il tipo di operazione **(2)**
- **indirizzo** (composto da **26-bit**) riporta una parte (26 bit su 32) dell'indirizzo **assoluto** di destinazione del salto

I 26-bit del campo **indirizzo** rappresentano un indirizzo di parola (**word address**)



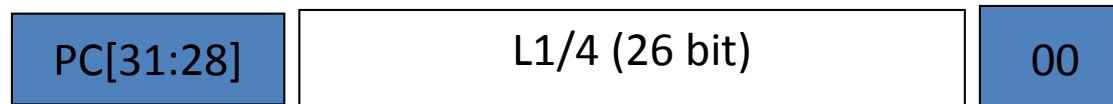
Istruzioni di salto incondizionato (tipo J)

L'assemblatore sostituisce l'etichetta **L1** con i 28 bit meno significativi traslati a destra di 2 (divisione per 4 per calcolare l'indirizzo di parola) per ottenere 26-bit

- in pratica elimina i due 0 finali
- si amplia lo spazio di salto:
 - si salta tra 0 e 2^{28} byte (2^{26} word)

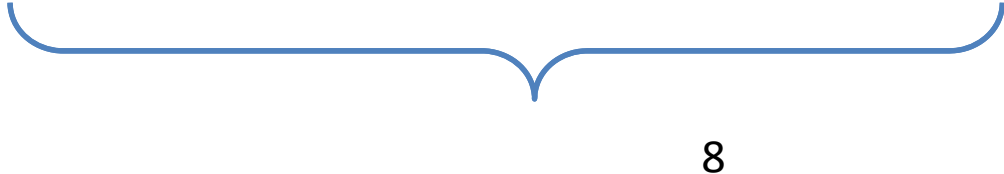
I 26-bit di indirizzo nelle jump rappresentano un indirizzo di parola (word address) \Rightarrow corrispondono ad un indirizzo di byte (byte address) composto da 28 bit.

Poiché il registro PC è composto da 32 bit \Rightarrow l'istruzione jump rimpiazza solo i 28 bit meno significativi del PC, lasciando inalterati i rimanenti 4 bit più significativi.



Istruzioni di tipo J: esempio

Nome campo	op	indirizzo				
Dimensione	6-bit	26-bit				
j 32	000010	00 0000	0000	0000 0000	0000	1000



Gestione costanti su 32 bit

Le istruzioni di **tipo I** consentono di rappresentare costanti esprimibili in 16 bit (valore massimo 65535 unsigned)

Se si hanno costanti da 32, l'assemblatore (o il compilatore) deve fare due passi per caricarla, suddividendo il valore della costante in due parti da 16 bit che vengono trattate separatamente come due *immediati* in due istruzioni successive:

- si utilizza l'istruzione **lui** (*load upper immediate*) per caricare il *primo immediato* (che rappresenta i 16 bit più significativi della costante) nei 16 bit più significativi di un registro; i rimanenti 16 bit meno significativi del registro sono posti a 0
- una successiva istruzione (ad esempio **ori** o anche **addi**) specifica tramite il *secondo immediato* i 16 bit meno significativi della costante



Istruzioni di tipo I: istruzione lui

lui \$s0, 61

\$zero

Nome campo	op	rs	rt	indirizzo			
Dimensione	6-bit	5-bit	5-bit	16-bit			
lui \$s0, 61	001111	00000	10000	0000	0000	0011	1101



Esempio

Per caricare in \$t0 il valore

0000 0000 1111 1111 0000 1001 0000 0000

lui \$t0, 255

ori \$t0, \$t0, 2034

255 (dec) = 0000 0000 1111 1111

2034 (dec) = 0000 1001 0000 0000

La versione in linguaggio macchina di lui \$t0, 255 # \$t0 è il registro 8:

001111	00000	01000	0000 0000 1111 1111
--------	-------	-------	---------------------

Il contenuto del registro \$t0 dopo avere eseguito lui \$t0, 255:

0000 0000 1111 1111	0000 0000 0000 0000
---------------------	---------------------



Pseudoistruzione li

Consideriamo la costante a 32 bit: 118345_{10} ($0x1CE49$)

0000 0000 0000 0001 1100 1110 0100 1001

16 bit più significativi

corrispondenti al valore 1_{10}

16 bit meno significativi

corrispondenti al valore 52809_{10}

La pseudoistruzione *load immediate* consente di caricarla nel registro

`li $t1, 118345`

`# $t1 ← 118345`



Pseudoistruzione li: esempio

L'assemblatore sostituisce la pseudoistruzione *li* con le seguenti istruzioni:

```
lui $at, 1                # 1 = 0000 0000 0000 0001
```

valore di \$at:

```
0000 0000 0000 0001 0000 0000 0000 0000
```

```
ori $t1, $at, 52809      # $t1 ← $at or 52809
```

valore di \$t1:

```
0000 0000 0000 0001 1100 1110 0100 1001
```

ori esegue senza estensione di segno



Pseudoistruzione *la*

Per caricare in un registro il valore dell'indirizzo di una variabile

ADDR = **indirizzo simbolico** (su 32 bit)

è disponibile la pseudoistruzione *load address* che lavora in modo simile alla *li* vista prima

```
la $t1, ADDR          # $t1 ← ADDR
```

L'indirizzo su 32 può essere visto come la giustapposizione di due parti da 16 bit ciascuna

ADDR_{High} **ADDR**_{Low}

L'assemblatore e il collegatore espandono in modo opportuno (e più complicato ... rispetto a *li*) la pseudo-istruzione sopra in un modo simile al seguente

```
lui reg, ADDR_HIGH    #i 16 bit più significativi dell'ind.  
                      # sono caricati in reg e i rimanenti di reg posti a 0
```

```
ori reg, reg, ADDR_LOW # i 16 bit meno significativi dell'ind.  
                      # vengono giustapposti
```

```
ori esegue senza estensione di segno così come addiu
```



***Struttura del programma
e
direttive all'assemblatore***



Come definiamo la struttura del programma

- Seguiamo uno schema di compilazione, ispirato a GCC, per tradurre da linguaggio sorgente C a linguaggio macchina MIPS
- Presuppone di conoscere MIPS: banco di registri, classi d'istruzioni, modi d'indirizzamento e organizzazione del sottoprogramma (chiamata rientro e passaggio parametri)
- Consiste in vari insiemi di convenzioni e regole per
 - segmentare il programma
 - dichiarare le variabili
 - usare (leggere e scrivere) le variabili
 - rendere le strutture di controllo
- Non attua necessariamente la traduzione più efficiente
- Sono possibili varie ottimizzazioni “ad hoc” del codice



Come definiamo la struttura del programma

- dobbiamo definire un modello di architettura “run-time” per memoria e processore
- le convenzioni del modello run-time comprendono
 - collocazione e ingombro delle diverse classi di variabile
 - destinazione di uso dei registri
- il modello di architettura run-time consente interoperabilità tra porzioni di codice di provenienza differente, come per esempio codice utente e librerie standard precompilate
- esempio tipico in linguaggio C è la libreria standard di IO



Struttura del programma e modello di memoria

- ❑ un programma in esecuzione (processo per il SO) ha tre segmenti essenziali
 - codice main e funzioni utente
 - dati variabili globali e dinamiche
 - pila aree di attivazione con indirizzi, parametri, registri salvati e variabili locali
- ❑ codice e dati sono segmenti dichiarati nel programma
- ❑ il segmento di pila viene creato al lancio del processo

Si possono avere anche

- due o più segmenti codice o dati
- segmenti di dati condivisi
- segmenti di libreria dinamica
- e altre peculiarità ...

questo modello di memoria è valido in generale



Dichiarare i segmenti

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

<pre>// var. glob. ... // funzioni ... main (...) { // corpo ... }</pre>	<pre>// segmento dati .data // indir. iniziale dati 0x 1000 0000 ... // var globali e dinamiche // segmento codice .text // indir. iniziale codice 0x 0040 0000 .globl main main: ... // codice programma</pre>
---	--

non occorre dichiarare esplicitamente il segmento di pila
implicitamente esso inizia all'indirizzo 0x 7FFF FFFF
e cresce verso gli indirizzi minori (ossia verso 0)



Direttive dell'assemblatore

Sono direttive inserite nel *file sorgente* che vengono interpretate dall'assemblatore per generare il *file oggetto* e danno indicazioni su:

- variabili e strutture dati da allocare (eventualmente inizializzati) nel segmento dati del modulo (file) di programma (*data segment* – parte globale/statica) da assemblare
- istruzioni da allocare nel segmento codice (*text segment*) del modulo da assemblare
- indicazione di simboli globali definiti nel modulo, cioè referenziabili anche da altri moduli
-



Convenzioni per le variabili

- ❑ in generale le variabili del programma sono collocate
 - globali in memoria a indirizzo fissato (assoluto)
 - locali, e parametri, nei registri del processore o nell'area di attivazione in pila (da precisare in seguito)
 - dinamiche in memoria (qui non sono considerate)
- ❑ le istruzioni aritmetico-logiche operano su registri
- ❑ dunque per operare sulla variabile (glob – loc – din)
 1. prima caricala in un registro libero (*load*)
 2. poi elaborala nel registro (confronto e aritmetica-logica)
 3. infine riscrivila in memoria (*store*)

Se variabile locale allocata in registro \Rightarrow salta (1) e (3)



Come dichiarare le diverse classi di variabili

- ❑ in C la variabile è un oggetto formale e ha
 - nome per identificarla e farne uso
 - tipo per stabilirne gli usi ammissibili

- ❑ in MIPS la variabile è un elemento di memoria (byte parola o regione di mem) e ha una “collocazione” con
 - nome per identificarla
 - modo per indirizzarla

- ❑ in MIPS la variabile viene manipolata tramite indirizzo simbolico o nome di registro

occorre però distinguere tra diverse classi di variabile
(var globale – parametro – var locale)



Variabile globale nominale

- la variabile globale è collocata in memoria a indirizzo fisso stabilito dall'assemblatore e dal linker
- per comodità l'indirizzo simbolico della variabile globale coincide con il nome della variabile
- gli ordini di dichiarazione e disposizione in memoria delle variabili globali coincidono



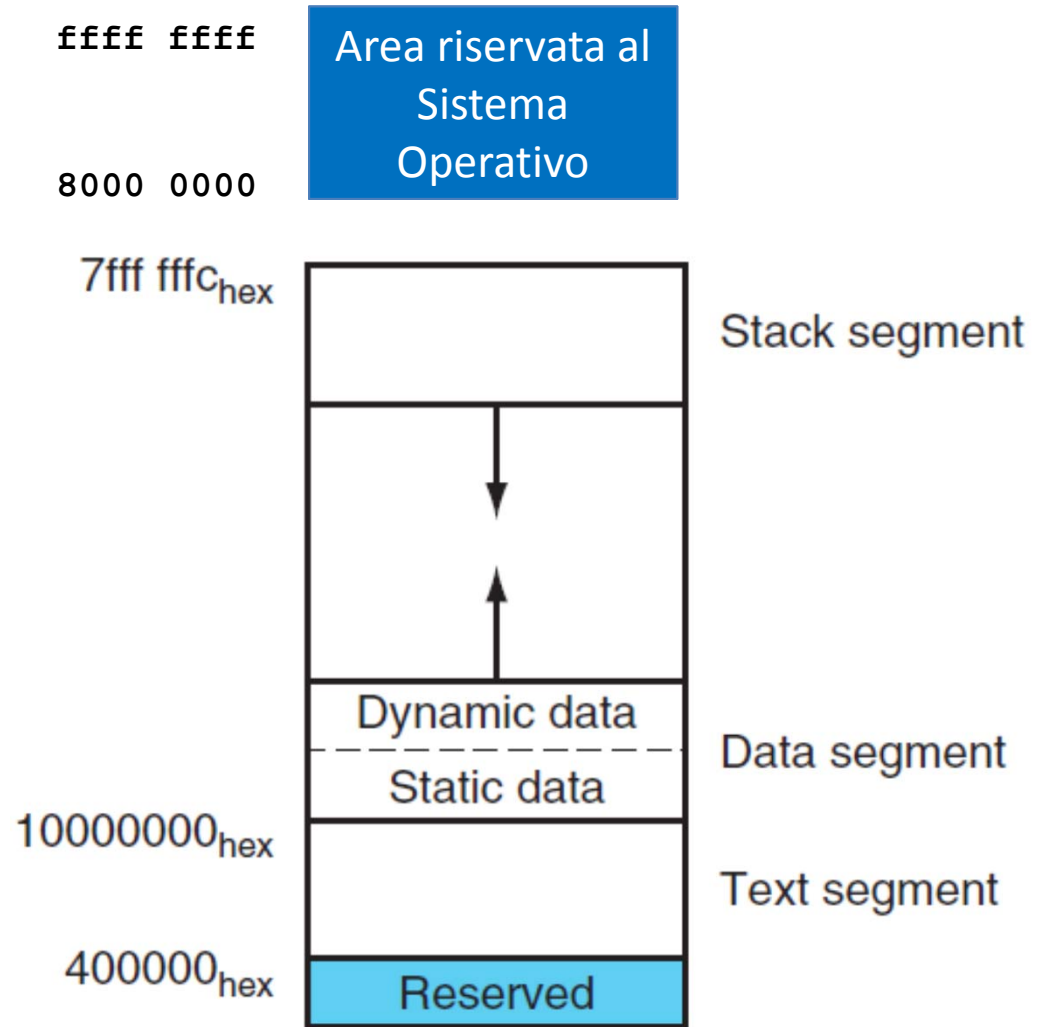
Variabili globali e partizione della memoria

le variabili globali sono allocate a partire dall'indirizzo:
0x 1000 0000

per puntare al segmento dati statici si può usare il registro *global pointer*: **\$gp**

il registro **\$gp** è inizializzato all'indirizzo:
0x 1000 8000

con un *offset* di 16 bit rispetto al **\$gp** si indirizzano 64kbyte $gp + 32\text{kbyte}$ e $gp - 32\text{kbyte}$



Esempio di dichiarazione di variabili globali

intero ordinario e corto a 32 e 16 bit rispettivamente

```
char c = `@`  
int a = 1  
short int B  
int vet [10]  
int * punt1  
char * punt2
```

```
        .data  
C:      .byte   64    // @ valore iniziale  
A:      .word   1     // 1 valore iniziale  
B:      .half           // non inizializzato  
VET:    .space  40    // 10 elem × 4 byte)  
PUNT1:  .word   0     // iniz. a NULL  
PUNT2:  .word           // non inizializzato
```

il valore iniziale è facoltativo

il puntatore è sempre una parola a 32 bit,
indipendentemente dal tipo di oggetto puntato



Direttive - 1

.data <addr>

Gli elementi successivi sono memorizzati nel segmento dati a partire dall'indirizzo **addr**

.asciiz ''str''

Memorizza la stringa **str** terminandola con il carattere **Null**

.ascii ''str''

ha lo stesso effetto, ma non aggiunge alla fine il carattere **Null**

.byte b1,...,bn

Memorizza gli **n** valori **b1**, .., **bn** in byte consecutivi di memoria

.word w1, ...,wn

Memorizza gli **n** valori su 32-bit **w1**, ..., **wn** in parole consecutive di memoria

.half h1, ...,hn

Memorizza gli **n** valori su 16-bit **h1**, ..., **hn** in halfword (mezze parole) consecutive di memoria

.space n

Alloca uno spazio pari ad **n** byte nel segmento dati



Direttive - 2

.text <addr>

Memorizza gli elementi successivi nel segmento testo dell'utente a partire dall'indirizzo

.globl sym

Dichiara **sym** come etichetta globale (ad essa è possibile fare riferimento da altri file)

.align n

Allinea il dato successivo a blocchi di **2ⁿ** byte: ad esempio

.align 2 = .word allinea alla parola (indirizzo multiplo di 4) il valore successivo

.align 1 = .half allinea alla mezza parola il valore successivo

.align 0 elimina l'allineamento automatico delle direttive **.half**, **.word**,

.float, e **.double** fino a quando compare la successiva direttiva **.data**

.eqv

Sostituisce il secondo operando al primo. Il primo operando è un simbolo, mentre il secondo è un'espressione (direttiva **#define** del C)

Tutte le direttive che memorizzano valori o allocano spazio di memoria sono precedute da un'etichetta simbolica che rappresenta l'indirizzo iniziale



Direttive: esempio

```
# Somma valori in un array
.data
array: .word 1,2,3,4,5,6,7,8,9,10 # dichiarazione array
.text
.globl main
main:
    li $s0,10           # $s0 ← numero elementi
    la $s1,array        # $s1 ← registro base per array
    li $s2,0            # azzero $s2 contatore cicli
    li $t2,0            # azzero $t2 accumulatore
loop: lw $t1,0($s1)      # accesso all'array
    add $t2,$t1,$t2      # calcolo risultato nell'acc. $t2
    addi $s1,$s1,4       # $s1 ← ind. del prossimo
                        # elemento dell'array
    addi $s2,$s2,1       # incremento $s2 contatore cicli
    bne $s2,$s0,loop     # test terminazione
```



Parametri in ingresso a una funzione e valore restituito

i primi quattro parametri vanno passati nei registri *a0* (primo nella testata), *a1*, *a2* e *a3* (quarto)

- se sono di tipo scalare o puntatore (a 32 bit)
- il nome di vettore è considerato puntatore (al primo elem.)
- per passare una *struct* si veda la ABI del compilatore*

gli eventuali parametri rimanenti vanno impilati a cura del chiamante, sempre come valori a 32 bit

- è raro che una funzione abbia più di quattro parametri

il valore in uscita (a 32 bit) va restituito nel registro *v0*

- se è di tipo scalare o puntatore
- il nome di vettore è considerato puntatore (al primo elem.)
- per restituire una *struct* si veda la ABI del compilatore*

se il valore in uscita è di tipo *double*, si usa anche *v1*

* la ABI di *gcc* impone di passare l'indirizzo dell'inizio della *struct*



Variabile locale nominale

la variabile locale può essere gestita in vari modi, secondo il tipo di variabile e il grado di ottimizzazione del codice, e anche in dipendenza di come la variabile viene utilizzata

variabile di tipo scalare o puntatore – due modi

- in un registro del blocco $s0 - s7$, se per altro motivo non deve avere un indirizzo di memoria – vedi precisazioni sotto
- altrimenti nell'area di attivazione della funzione

variabile di tipo scalare, ma che viene anche acceduta tramite un puntatore – un solo modo

- nell'area di attivazione della funzione, perché deve avere un indirizzo

variabile di tipo *array* (o *struct*) – un solo modo

- sempre nell'area di attivazione della funzione



Convenzioni

Come usare le diverse classi di variabili scalari

Come rendere le strutture di controllo

Come usare una variabile strutturata

Vedi Come tradurre da C a MIPS

Attenzione alle **convenzioni ACSO**: per esempio

➤ Array (vettori)

- indirizzo base (= nome array) caricato in un registro
- indirizzo effettivo di un generico elemento calcolato tramite il registro

➤ Variabili locali

- vedi prima



Vettore – scansione sequenziale con indice

```
// variabili globali
int v [5] // vettore
int a      // indice
...
// testata del ciclo
for (a = 2; a <= 5; a++) {
    v[a] = v[a] + 6
} /* end for */
// seguito del ciclo
...
```

```
V:    .space 20           // 20 byte per v
A:    .word               // mem per a
      // assegna a = 2 già visto
FOR:  li    $t0, 5         // inizializza $t0
      lw    $t1, A         // carica a
      bgt   $t1, $t0, END  // se .. va' a END
      li    $t0, 6         // inizializza $t0
      la    $t1, V         // ind. iniz. di v
      lw    $t2, A         // carica a
      sll   $t2, $t2, 2     // allinea indice
      addu  $t1, $t1, $t2  // indir. di v[a]
      lw    $t3, 0($t1)    // carica v[a]
      add   $t3, $t3, $t0  // v[a] + 6
      sw    $t3, 0($t1)    // memorizza v[a]
      // aggiorna a++ già visto
      j     FOR            // torna a FOR
END:  ...                  // seguito ciclo
```

ottimizzazioni possibili trattando costanti e aritmetica



Approfondimento – System Call



System Call

Sono disponibili delle **chiamate di sistema (system call)** predefinite che implementano particolari servizi (per esempio: stampa a video)

Ogni system call ha:

- un codice
- degli argomenti (opzionali)
- dei valori di ritorno (opzionali)



System Call: qualche esempio

print_int: stampa sulla console il numero intero che le viene passato come argomento;

print_string: stampa sulla console la stringa che le è stata passata come argomento terminandola con il carattere **Null**;

read_int: legge una linea in ingresso fino al carattere a capo incluso (i caratteri che seguono il numero sono ignorati);

read_string: legge una stringa di caratteri di lunghezza **\$a1** da una linea in ingresso scrivendoli in un buffer (**\$a0**) e terminando la stringa con il carattere **Null** (se ci sono meno caratteri sulla linea corrente, li legge fino al carattere a capo incluso e termina la stringa con il carattere **Null**);

sbrk restituisce il puntatore (indirizzo) ad un blocco di memoria;

exit interrompe l'esecuzione di un programma;

E anche altre ...



System Call

Nome	Codice	Argomenti	Risultato
print_int	1	\$a0	
print_float	2	\$f12	
print_double	3	\$f12	
print_string	4	\$a0	
read_int	5		\$v0
read_float	6		\$f0
read_double	7		\$f0
read_string	8	\$a0,\$a1	
sbrk	9	\$a0	\$v0
exit	10		



System Call

Per richiedere un servizio a una chiamata di sistema (**syscall**) occorre:

- caricare il **codice** della **syscall** nel registro **\$v0**
- caricare gli **argomenti** nei registri **\$a0** - **\$a3** (oppure nei registri **\$f12** - **\$f15** nel caso di valori in virgola mobile)
- eseguire **syscall**
- l'eventuale **valore di ritorno** è caricato nel registro **\$v0** (**\$f0**)



Esempio

#Programma che stampa: la risposta è 5

```
    .data  
str:  .ascii "la risposta è"  
    .text
```

```
    li $v0, 4           # $v0 ← codice della print_string  
    la $a0, str         # $a0 ← indirizzo della stringa  
    syscall            # stampa della stringa
```

```
    li $v0, 1           # $v0 ← codice della print_integer  
    li $a0, 5           # $a0 ← intero da stampare  
    syscall            # stampa dell'intero
```

```
    li $v0, 10          # $v0 ← codice della exit  
    syscall            # esce dal programma
```



Esempio

```
#Programma che stampa "Dammi un intero: "  
# e che legge un intero  
    .data  
prompt:.asciiz "Dammi un intero: "  
    .text  
    .globl main  
main:  
    li $v0, 4          # $v0 ← codice della print_string  
    la $a0, prompt     # $a0 ← indirizzo della stringa  
    syscall            # stampa la stringa  
  
    li $v0, 5          # $v0 ← codice della read_int  
    syscall            # legge un intero e lo carica in $v0  
  
    li $v0, 10         # $v0 ← codice della exit  
    syscall            # esce dal programma
```

