

Modulo Calcolatori Elettronici

Proff. Gian Luca Marcialis, Giulia Orrù, Lorenzo Putzu, Fabio Roli

Corsi di Laurea in Ingegneria Biomedica Ingegneria Elettrica, Elettronica ed Informatica

Contatti:

marcialis@unica.it, giulia.orrù@unica.it, lorenzo.putzu@unica.it

Capitolo 4

Linguaggio Assembly

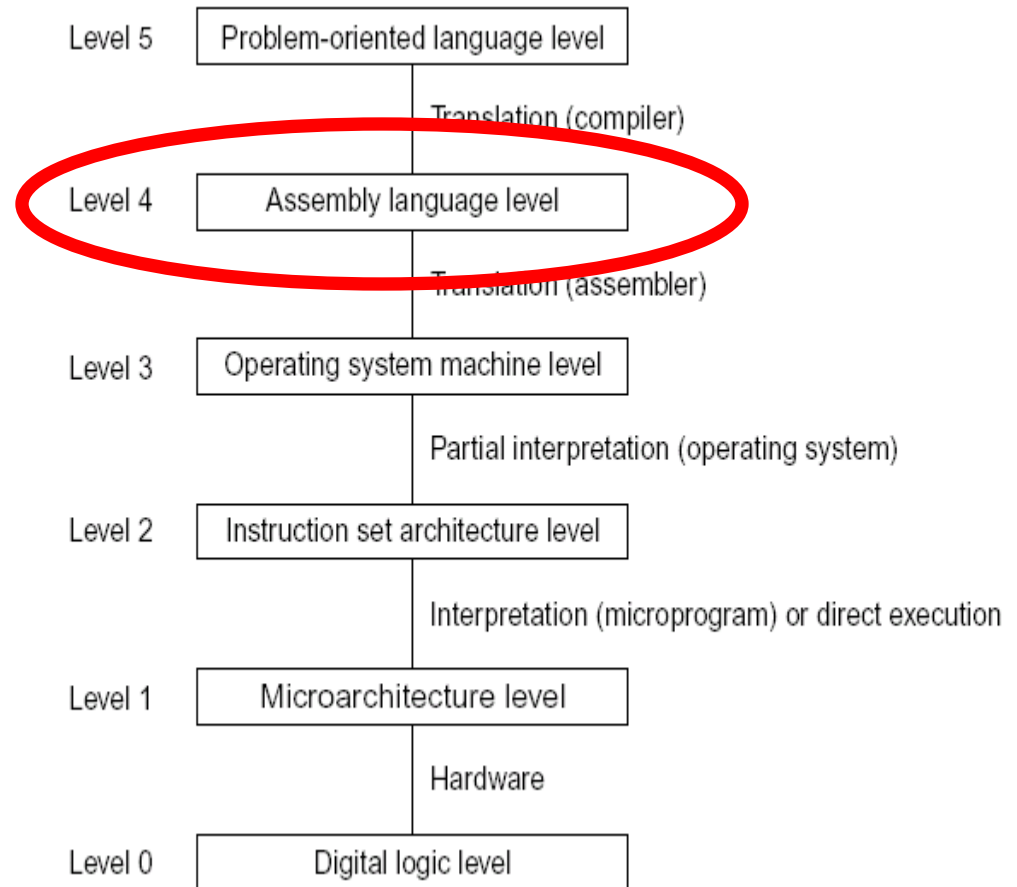
Fonti principali: Patterson, A.D., Hennessy, J., "Struttura, organizzazione e progetto dei calcolatori elettronici", Jackson Libri, Collana Università, 2000 (ISBN:8825615175), Cap. 3

Sommario

- Introduzione al linguaggio Assembly
 - il MIPS
- Le operazioni fondamentali
 - add, sub
- Le istruzioni di memorizzazione
 - lw, sw
- Salti condizionati e incondizionati
 - beq, bne
 - j
- Salti a sottoprogrammi
 - jal, jr

Un po' di ripasso

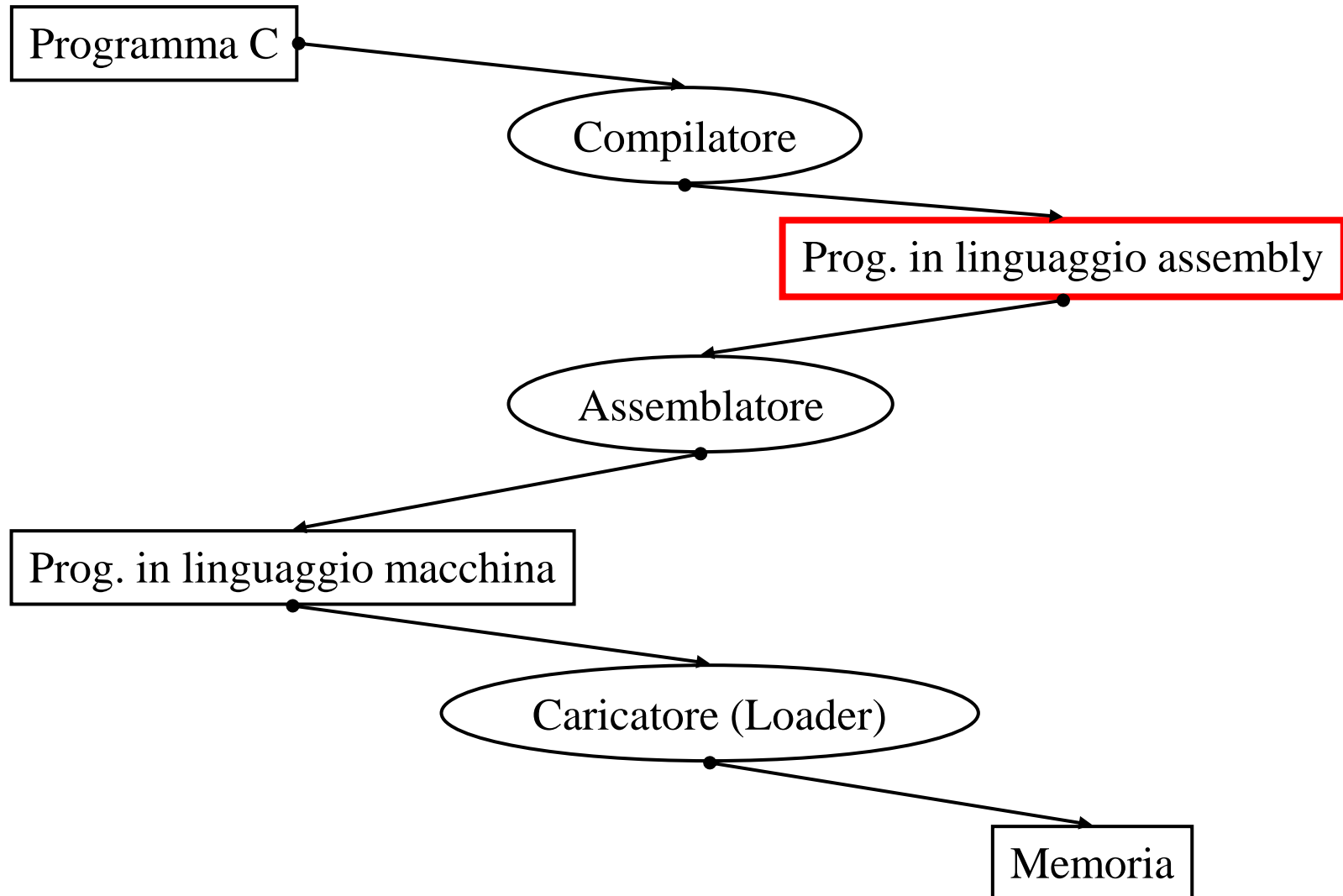
- Def. Architettura del calcolatore:
 - l'insieme dei suoi moduli e delle interconnessioni
 - aspetti "visibili" al programmatore
- L'architettura di un calcolatore è descrivibile a livelli di "dettaglio" molto differenti
- Uno di questi è il livello del linguaggio Assembly
- **In altri termini, in questo capitolo, l'architettura del calcolatore è l'insieme delle istruzioni assembly che esso è in grado di eseguire**



Il linguaggio Assembly

- Esprime in forma simbolica le istruzioni di macchina (descritte attraverso sequenze di bit)
- Un compilatore (“assemblatore”) traduce le istruzioni assembly in istruzioni di macchina
- Le istruzioni assembly sono a loro volta la “traduzione” in una forma semplice di un linguaggio ad alto livello (C, Python, Pascal)
- In questa sezione, impareremo a scrivere semplici programmi in un assembly che rappresenta le istruzioni del MIPS
 - architettura realizzata dalla MIPS Computer Company

Programmi: da linguaggio di alto livello a caricamento in memoria



Operazioni fondamentali del MIPS

- **Addizione:** `add a, b, c`
 - carica nel registro `a` la somma dei contenuti dei registri `b` e `c`
- **Sottrazione:** `sub a, b, c`
 - carica nel registro `a` la differenza dei contenuti dei registri `b` e `c`
- **Esempio:**
 - istruzione di alto livello (C): $a = b + c - d + e$
 - istruzioni assembly:

`add a, b, c`

`sub a, a, d`

`add a, a, e`

Gli operandi delle istruzioni add e sub

- Sono rappresentati da **registri**
- Il MIPS dispone di 32 registri che possono contenere parole di 32 bit
 - evitano accessi alla memoria
 - un numero maggiore provocherebbe ritardi eccessivi
- Poiché in un programma generalmente ci sono più di 32 variabili, non è possibile memorizzarle tutte in registri
- Un uso efficiente delle istruzioni MIPS comporta un uso efficiente dei registri
- Un registro si rappresenta attraverso il simbolo \$ + numero di registro: \$0, \$1, ..., \$31

Operazioni fondamentali del MIPS

- Istruzione di alto livello (C):
 - $a = b + c - d + e$
- Istruzione assembly, nell'ipotesi che:
 - $b \rightarrow \$8$
 - $c \rightarrow \$9$
 - $d \rightarrow \$10$
 - $e \rightarrow \$11$
 - Risultato (a) nel registro \$7

Sintassi

add \$7, \$8, \$9

sub \$7, \$7, \$10

add \$7, \$7, \$11

Semantica

$a \leftarrow b + c$

$a \leftarrow a - d$

$a \leftarrow a + e$

La pseudo-istruzione `move`

- `move R1, R2`
 - sposta il contenuto del registro `R2` nel registro `R1`
- L'istruzione `move` non fa parte del set di istruzioni del MIPS, per questo motivo ad essa è dato il nome di "pseudo-istruzione"
- Infatti l'assemblatore converte la `move` nell'istruzione effettiva:
`add R1, $0, R2`
- La `move` serve per migliorare la leggibilità del codice Assembly

Altre istruzioni aritmetiche

- `mul R1, R2, R3`
 - Salva nel registro `R1` il prodotto del contenuto di `R2` con il contenuto di `R3`
- Istruzioni con indirizzamento “immediato”
 - Il terzo operando non è un registro ma il valore che si intende elaborare
- `addi R1, R2, <Costante>`
 - salva nel registro `R1` la somma del contenuto di `R2` con il valore `<Costante>`
 - Esempio: `addi $29, $29, 4`
 - Nel formato dell’istruzione, al valore costante viene assegnato lo stesso spazio di un indirizzo di memoria (16 bit)
- `subi R1, R2, <Costante>`
- `muli R1, R2, <Costante>`

Registri speciali

- \$0
 - Il suo contenuto è posto a zero e non può essere alterato in nessun modo
- \$29
 - Contiene il puntatore allo stack di memoria (stack pointer, SP)
- \$31
 - Contiene la copia del contatore di programma (program counter, PC) durante i salti a procedura

Trasferimento dati dalla memoria ai registri

- Poiché non tutte le variabili vengono inserite in registri, il MIPS dispone della seguente istruzione in grado di caricare dati dalla memoria e inserirli in registri
- **lw <T-reg>, <start-address (offset-reg)>**
 - carica nel registro T-reg la parola contenuta nell'indirizzo di memoria start-address a cui viene sommato il contenuto del registro offset-reg
 - lw = load word
- **Esempio:**
 - lw \$8, Astart(\$19)
 - carica in \$8 la parola contenuta nell'indirizzo di memoria Astart+\$19
 - Astart è un'etichetta

Trasferimento dati dai registri alla memoria

- Il MIPS dispone della seguente istruzione in grado di copiare dati dai registri in memoria
- **sw** <S-reg>, <start-address (offset-reg)>
 - copia la parola dal registro S-reg all'indirizzo di memoria start-address a cui viene sommato il contenuto del registro offset-reg
 - sw = store word
- **Esempio:**
 - sw \$8, Astart(\$19)
 - scarica la parola contenuta in \$8 nell'indirizzo di memoria Astart+\$19
 - Astart è un'etichetta

Etichette (label)

- E' possibile rappresentare un indirizzo di memoria sotto forma di stringa di caratteri
 - Es. l'indirizzo di memoria 1024 (in decimale) potrebbe essere rappresentato dalla stringa "Astart"
- L'assemblatore (assembler) provvede alla codifica opportuna delle etichette
- L'uso di etichette semplifica la lettura e la scrittura del codice
- Lo vedremo nel caso delle istruzioni di salto e dei trasferimenti in memoria

Organizzazione della memoria nel MIPS

- La memoria viene organizzata in parole da 4 byte, indirizzabili a gruppi attraverso opportune istruzioni. Di conseguenza, ogni parola è indirizzata con un multiplo del valore **4**.
- Esempio: le istruzioni `lw` e `sw` indirizzano solo gruppi di 4 byte.
 - `lw $8, 4($0)`
 - **OK:** Preleva la parola presente all'indirizzo 4 e la copia nel registro \$8
 - `lw $8, 1($0)`
 - **ERRORE:** stiamo cercando di indirizzare raggruppamenti di byte (blocchi) non allineati

Esempio di uso lw e sw : i vettori

- Un **vettore** è una sequenza di locazioni di memoria contigue a partire da un dato **indirizzo base**
- Si supponga di esprimere l'indirizzo base di un generico vettore con il simbolo A
- A quanto corrisponde, in Assembly MIPS, l'indirizzo di questa locazione?

Memoria	
A	36
$A+4$	10
$A+8$	98
...	...
Locazione h-esima?	67
...	8
	4
...	...

$$A + 4 * (h - 1)$$

Esercizio

- Si consideri un vettore memorizzato a partire dalla locazione A .
- La sequenza di locazioni a partire da A è costituita da N elementi
- **Domanda 1.** Si prelevi con un'opportuna istruzione MIPS l'elemento memorizzato nella locazione h -esima e lo si memorizzi nel registro $\$10$
 - Nello scrivere la soluzione si faccia uso dei registri MIPS che si riterrà opportuni, e si supponga che h sia memorizzato nel registro $\$5$.
- **Domanda 2.** Si scriva quali indirizzi di memoria **non** contengono elementi del vettore, sapendo che V coincide con il valore 1024 ed N con il valore 50 e che le parole sono indirizzate con 32 bit.

Domanda 1: soluzione

- Dobbiamo accedere alla locazione $A+4(h-1)$
- Dobbiamo quindi memorizzare in un registro il valore $4(h-1)$; utilizziamo il registro $\$5$ che già contiene il valore di h .
- Otteniamo la sequenza:

```
addi $5, $5, -1
```

```
mulh $5, $5, 4
```

```
lw $10, A($5)
```

Memoria	
A	36
A+4	10
A+8	98
...	...
A+4(h-1)	67
...	8
	4
...	...

Domanda 2: soluzione

- Sulla base di quanto espresso prima, gli indirizzi «liberi» sono da 0, 4, 8, 12,..., A-4, e da...
- ... $A+4(N-1) + 4 = A+4N$...
- ...al massimo indirizzo ottenibile con parole da 32 bit, ovvero, $2^{32}-4$
- Quindi se $A=1024$ e $N=50$, le locazioni libere vanno da 0 a 1020, e da $1024+50*4=1224$ a $2^{32}-4$

Un breve esercizio

- Dato lo stato della memoria e dei registri in figura, si indichi lo stato della memoria dopo le seguenti operazioni:

```
lw $8, 0($20)
add $8, $8, $9
add $20, $20, $20
sw $8, 0($20)
```

Stato iniziale

0	36
4	10
8	98
...	...

\$8	67
-----	----

\$9	8
-----	---

\$20	4
------	---

Soluzione

- Esecuzione di:

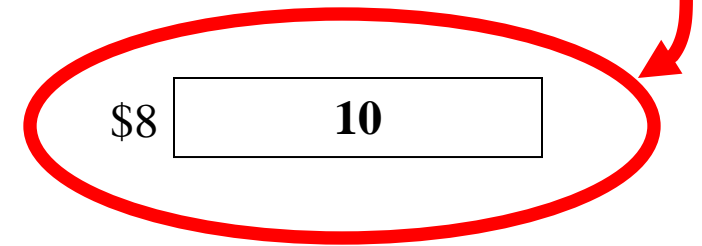
`lw $8, 0($20)`

- Passi intermedi:

- Estrazione del contenuto di \$20
 - $(\$20) \rightarrow 4$
- Somma del contenuto di \$20 con il valore fuori parentesi
 - $4 + 0 = 4$
- Il valore precedente indica l'indirizzo di memoria da cui estrarre il dato
 - $[4] \rightarrow 10$
- Memorizzazione del dato nel registro \$8
 - $10 \rightarrow (\$8)$

Stato memoria

0	36
4	10
8	98
...	...



\$9	8
-----	---

\$20	4
------	---

Soluzione

- Esecuzione di:

`add $8, $8, $9`

- Passi intermedi:

- Estrazione del contenuto di \$8
 - $(\$8) \rightarrow 10$
- Estrazione del contenuto di \$9
 - $(\$9) \rightarrow 8$
- Somma dei due valori
 - $10 + 8 = 18$
- Memorizzazione del dato nel registro \$8
 - $18 \rightarrow (\$8)$

Stato memoria

0	36
4	10
8	98
...	...

\$8	18
-----	-----------

\$9	8
-----	---

\$20	4
------	---

Soluzione

- Esecuzione di:

`add $20, $20, $20`

- Passi intermedi:

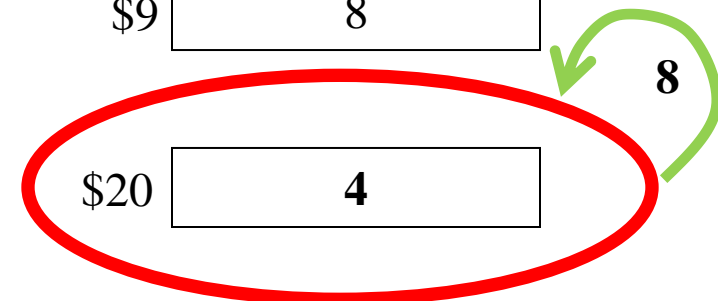
- Estrazione del contenuto di \$20
 - $(\$20) \rightarrow 4$
- Estrazione del contenuto di \$20
 - $(\$20) \rightarrow 4$
- Somma dei due valori
 - $4 + 4 = 8$
- Memorizzazione del dato nel registro \$20
 - $8 \rightarrow (\$20)$

Stato memoria

0	36
4	10
8	98
...	...

\$8	18
-----	----

\$9	8
-----	---



Soluzione

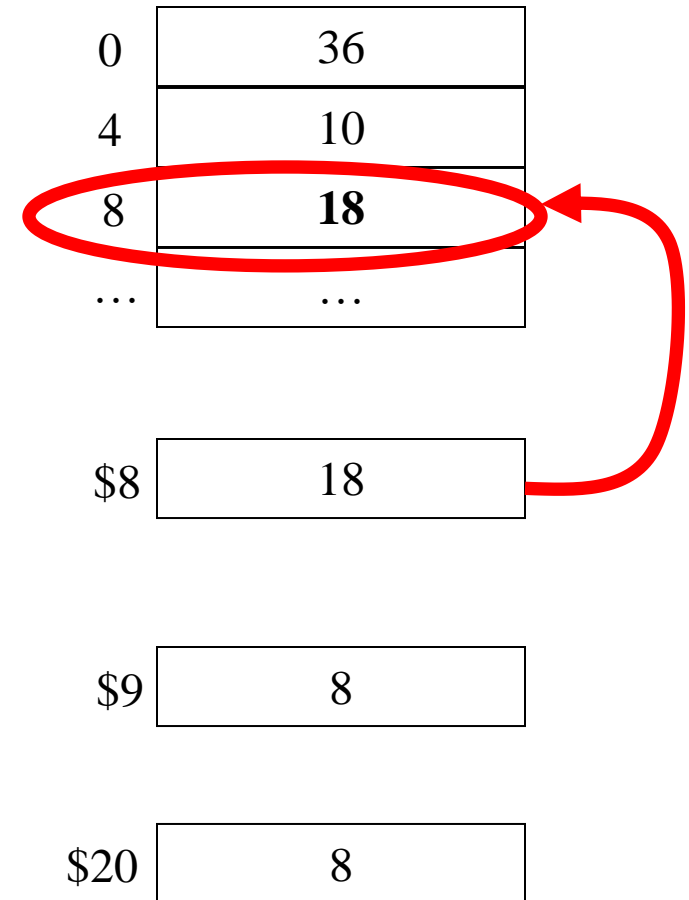
- Esecuzione di:

`SW $8, 0($20)`

- Passi intermedi:

- Estrazione del contenuto di \$20
 - $(\$20) \rightarrow 8$
- Somma del contenuto di \$20 con il valore fuori parentesi
 - $8 + 0 = 8$
- Memorizzazione del contenuto del registro \$8 nella locazione di memoria indicata dal valore precedente
 - $(\$8) \rightarrow 18 \rightarrow [8]$

Stato memoria



“Morale” dell’esercizio

- **QUALSIASI ELABORAZIONE PASSA PER I REGISTRI**
- I dati in memoria devono essere prima **copiati** nei registri
- Li si può elaborare e si mette il risultato **in altri registri**
- Il risultato può essere **copiato** in locazioni di memoria
- I dati prelevati dalla memoria ed alterati sui registri **NON** vengono alterati **anche** in memoria
 - Occorre quindi **copiarli in memoria**

Istruzioni di «salto» o «branching»

- Nell'eseguire una certa operazione può capitare di dover prendere decisioni che possono condurre ad istruzioni da eseguire le une **in alternativa** alle altre
- Si consideri ad esempio il seguente algoritmo

Se il valore `x` è maggiore del valore `y`
allora

scrivi `x` nella locazione 1024

altrimenti

scrivi `y` nella locazione 1024

Istruzioni di salto

- `beq R1, R2, L`
 - salta all'istruzione di indirizzo etichettato con `L` se i contenuti dei registri `R1` e `R2` **si eguagliano**
 - `beq` = branch if equal
- `bne R1, R2, L`
 - salta all'istruzione di indirizzo etichettato con `L` se i contenuti dei registri `R1` e `R2` **non** si eguagliano
 - `bne` = branch if not equal
- `j L`
 - salta all'istruzione etichettata con `L`
 - `j` = jump

Esempio di costrutto `if ... then ... else`

- Scrivere cosa accade dopo l'esecuzione di questa sequenza, usando il simbolismo nel riquadro in giallo:

```
bne $19, $20, Else  
add $16, $17, $18  
j Exit
```

```
Else:    sub $16, $17, $18
```

```
Exit:    ...
```

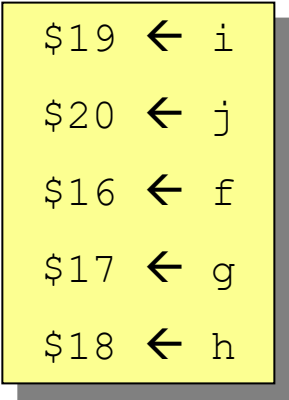
- Soluzione:

Se `i` **è diverso da** `j`

`f` = `g` - `h`

altrimenti

`f` = `g` + `h`



```
$19 ← i  
$20 ← j  
$16 ← f  
$17 ← g  
$18 ← h
```

I «cicli»

- In molti casi si deve ripetere una stessa operazione vincolando la ripetizione ad una certa condizione → **ciclo**
- Es. Calcolo del valore $z = b^n$
- Un possibile modo di calcolare è:

$z = 1$

Ripeti n volte

$z = z * b$

Esempio di ciclo

- Interpretare il seguente codice Assembly MIPS

```
Loop:      lw $8, A($9)
           add $17, $17, $8
           add $9, $9, $20
           bne $9, $10, Loop
```

\$8	←	x
\$9	←	i
\$10	←	j
\$17	←	z
\$20	←	h

- Possibile interpretazione:

Ripeti

$x = A[i]$

$z = z + x$

$i = i + h$

finché i **è diverso da** j

Istruzioni di salto con registro e confronto

- `jr R`
 - salto incondizionato basato sull'indirizzo specificato nel registro `R`
 - copia nel registro PC il valore di `R`
 - `jr` = jump register
- `slt R1, R2, R3`
 - `R1, R2, R3` sono registri
 - imposta a 1 il contenuto di `R1` se il valore in `R2` è minore di quello in `R3`, altrimenti lo imposta a 0
 - in combinazione con `beq` o `bne` realizza confronti di minoranza o maggioranza e relative istruzioni di salto condizionato
 - `slt` = set on less than

Istruzioni in modalità di indirizzamento “immediato”

- Sono istruzioni in cui uno degli operandi non è un registro od un indirizzo di memoria, bensì una costante numerica
- `slti R1, R2, <Costante>`
- `lui R, <Costante>`
 - salva nella metà più significativa del registro `R` il valore `<Costante>`
 - `lui` = load upper immediate
 - Le istruzioni precedenti sono in grado di gestire soltanto costanti da 16 bit (ampiezza del campo `<Costante>`)
 - Scopo: si caricano i 16 bit più significativi di un valore numerico in un registro, consentendo a un'istruzione successiva di specificare i 16 bit meno significativi dello stesso valore

Utilizzo di `lui`

- Si supponga di dovere caricare nel registro `$16` la seguente costante di 32 bit:

0000 0000 0011 1101 0000 1001 0000 0000

- $(61)_{10} = (0000\ 0000\ 0011\ 1101)_2$

- Parte più significativa della costante

- $(2304)_{10} = (0000\ 1001\ 0000\ 0000)_2$

- Parte meno significativa della costante

- Utilizzando l'istruzione `lui` è assai semplice:

`move $16, $0`

`lui $16, 61`

`addi $16, $16, 2304`

Esercizi

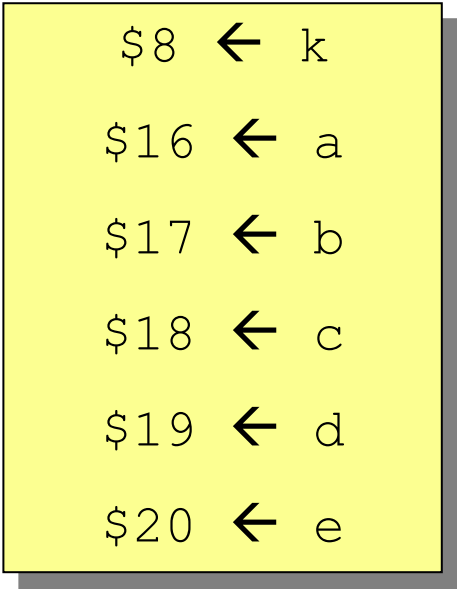
- Scrivere il codice Assembly MIPS che memorizza nella locazione di memoria 1024 il maggiore tra due numeri x e y contenuti rispettivamente nei registri \$4 e \$5.
- Scrivere il codice Assembly MIPS che memorizza nella locazione di memoria 2^{16} il valore x^n con x contenuto nel registro \$4 e n nel registro \$5.
- Scrivere il codice Assembly MIPS che memorizza nel registro \$8 la somma dei valori contenuti nel vettore di indirizzo iniziale v costituito da N elementi. Il valore N è memorizzato nel registro \$9.

Esercizio

- Si vuole tradurre in Assembly MIPS il seguente codice C:

```
switch (k)
{
    case 0:    a = d + e;
               break;
    case 1:    a = b + c;
               break;
    case 2:    a = b - c;
               break;
    case 3:    a = d - e;
}

```



```
$8 ← k
$16 ← a
$17 ← b
$18 ← c
$19 ← d
$20 ← e

```

Soluzione banale

```
switch:      move $7, $0
             beq $8, $7, case0
             addi $7, $7, 1
             beq $8, $7, case1
             addi $7, $7, 1
             beq $8, $7, case2
             addi $7, $7, 1
             beq $8, $7, case3
exit:        sw $16, a($0)
             ...
```

```
case0: add $16, $19, $20
       j exit
case1: add $16, $17, $18
       j exit
case2: sub $16, $17, $18
       j exit
case3: sub $16, $19, $20
       j exit
```

E' possibile scrivere una versione più efficiente nel caso per esempio che k (\$8) possa avere più di 4 valori possibili?

Soluzione efficiente: utilizzo di una vettore di salto «jumpTable»

```
case0: add $16, $19, $20
        j exit
case1: add $16, $17, $18
        j exit
case2: sub $16, $17, $18
        j exit
case3: sub $16, $19, $20
        j exit
```

```
jumpTable: case0
           case1
           case2
           case3
```

jumpTable è un vettore, il cui contenuto non è altro che l'indirizzo da ciascun sottoinsieme di istruzioni a cui saltare.

Ricordare che case0, ..., case3 e jumpTable sono tutti **indirizzi di memoria espressi tramite etichette**.

Costrutto “switch” con jumpTable

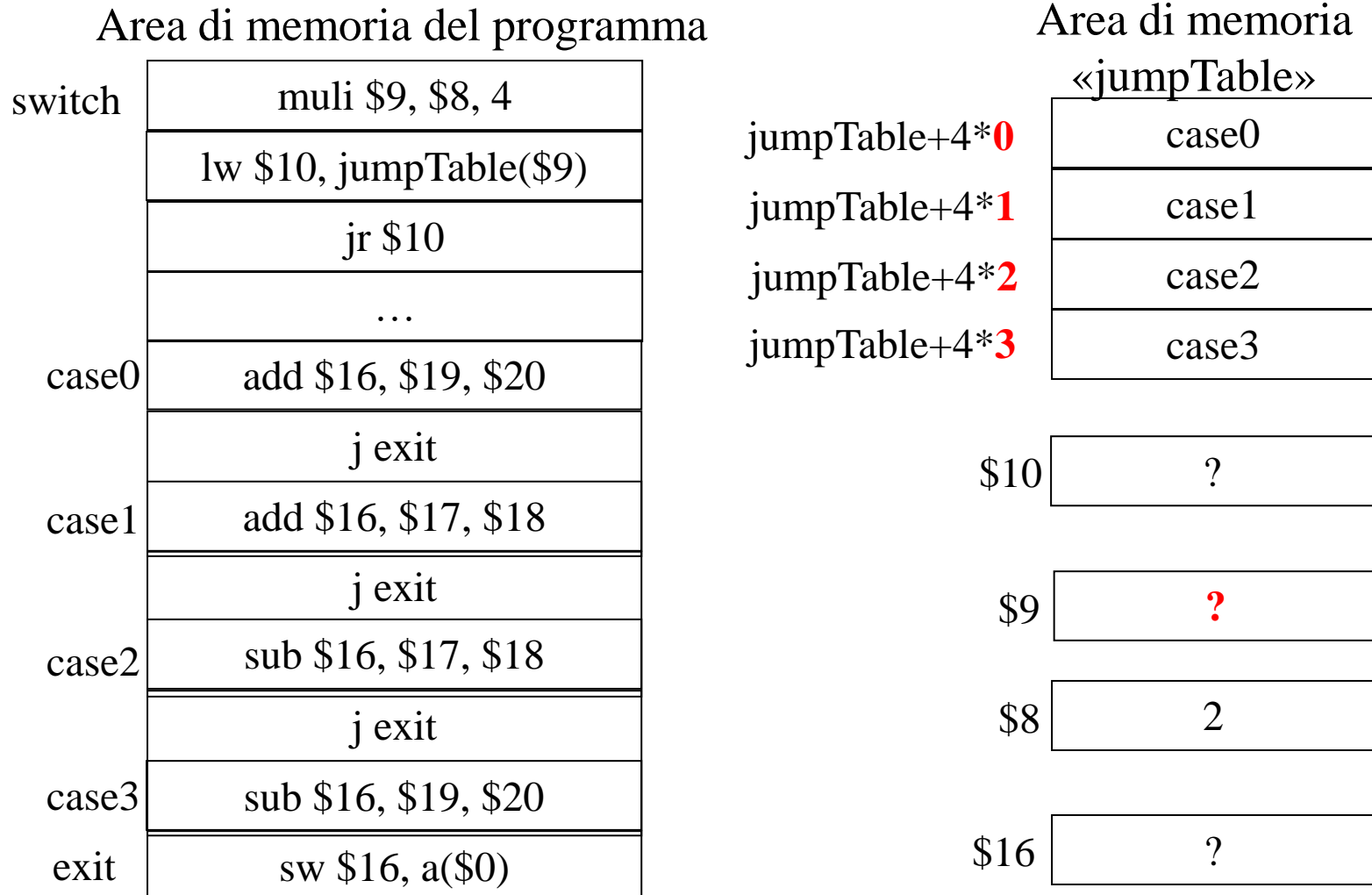
\$8 ← k

```
switch:      muli $9, $8, 4
             lw  $10, jumpTable($9)
             jr  $10
```

```
case0: add $16, $19, $20
      j  exit
case1: add $16, $17, $18
      j  exit
case2: sub $16, $17, $18
      j  exit
case3: sub $16, $19, $20
      j  exit
```

```
jumpTable:  case0
              case1
              case2
              case3
```

Funzionamento del programma «switch»



Funzionamento del programma «switch»

Area di memoria del programma

switch	muli \$9, \$8, 4
	lw \$10, jumpTable(\$9)
	jr \$10
	...
case0	add \$16, \$19, \$20
	j exit
case1	add \$16, \$17, \$18
	j exit
case2	sub \$16, \$17, \$18
	j exit
case3	sub \$16, \$19, \$20
exit	sw \$16, a(\$0)

Area di memoria

«jumpTable»

jumpTable+4***0**

jumpTable+4***1**

jumpTable+4***2**

jumpTable+4***3**

case0
case1
case2
case3

\$10

?

\$9

4*2

\$8

2

\$16

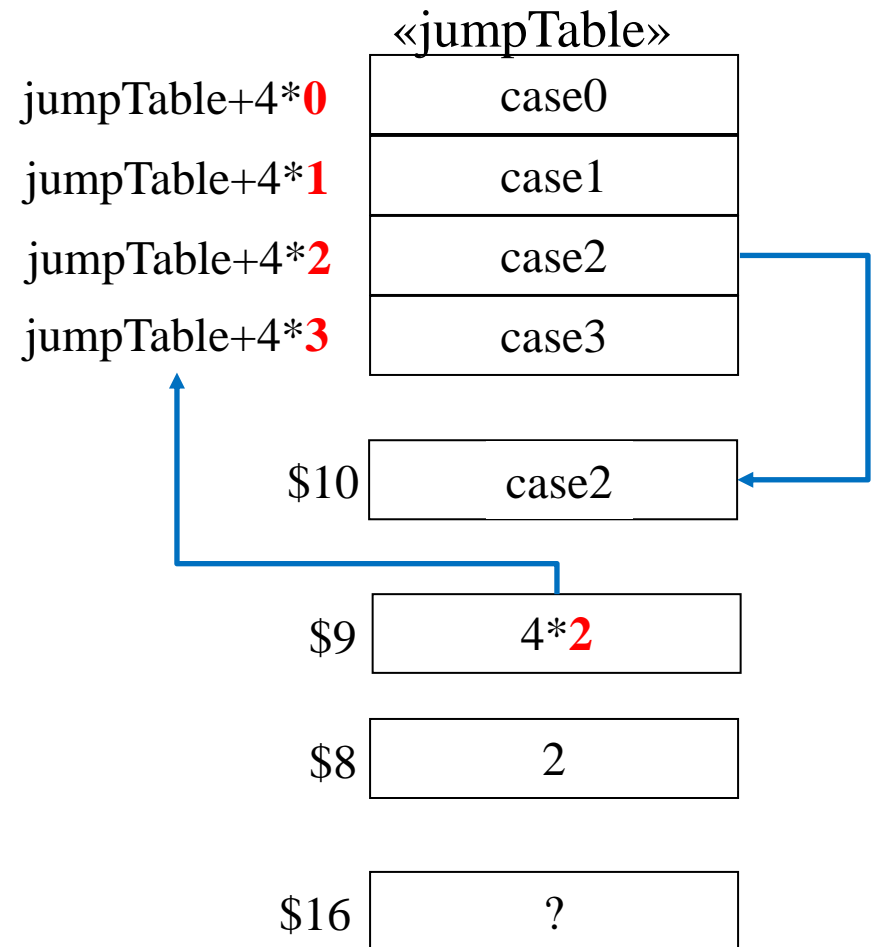
?

Funzionamento del programma «switch»

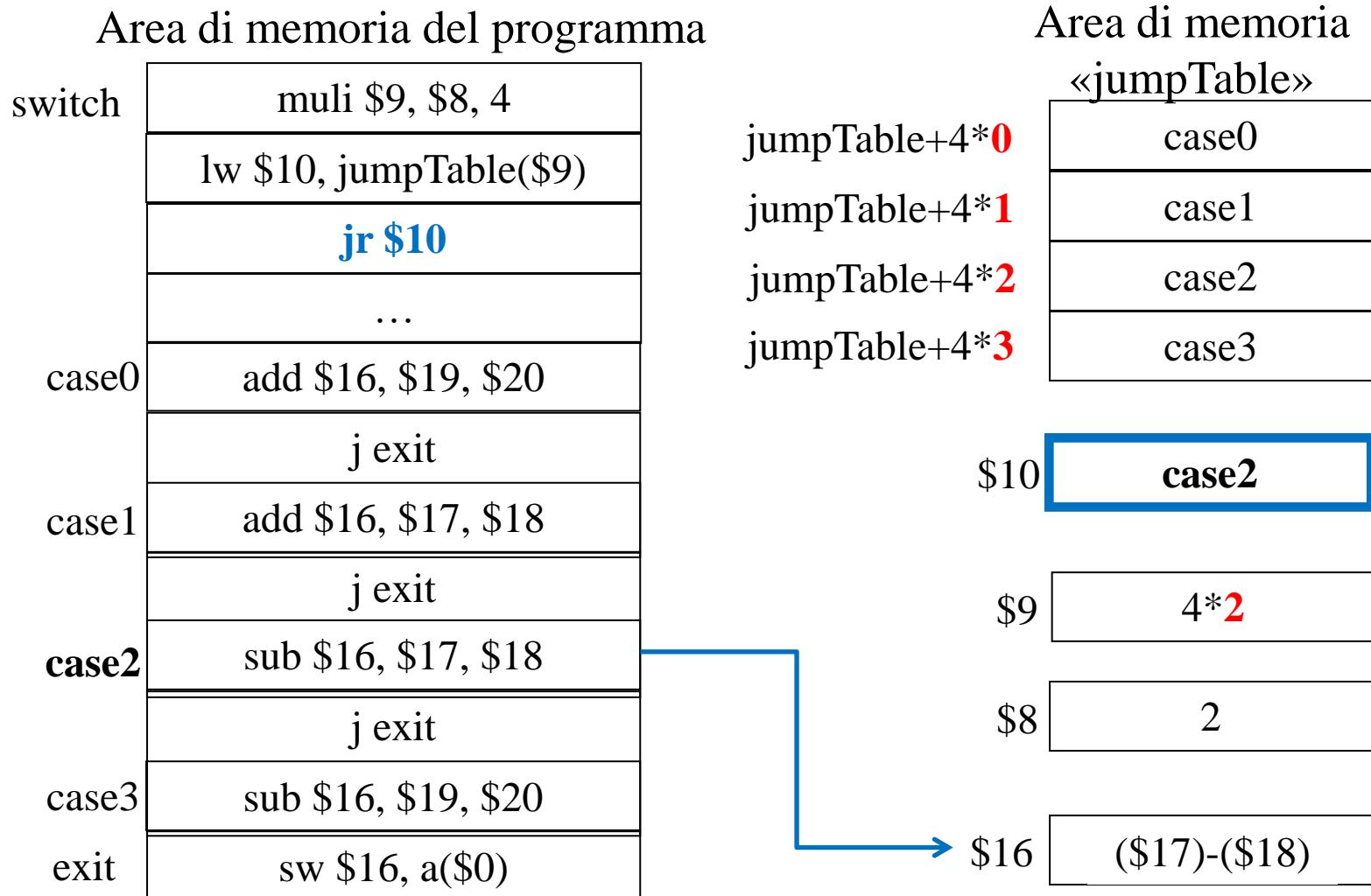
Area di memoria del programma

switch	mul \$9, \$8, 4
	lw \$10, jumpTable(\$9)
	jr \$10
	...
case0	add \$16, \$19, \$20
	j exit
case1	add \$16, \$17, \$18
	j exit
case2	sub \$16, \$17, \$18
	j exit
case3	sub \$16, \$19, \$20
exit	sw \$16, a(\$0)

Area di memoria

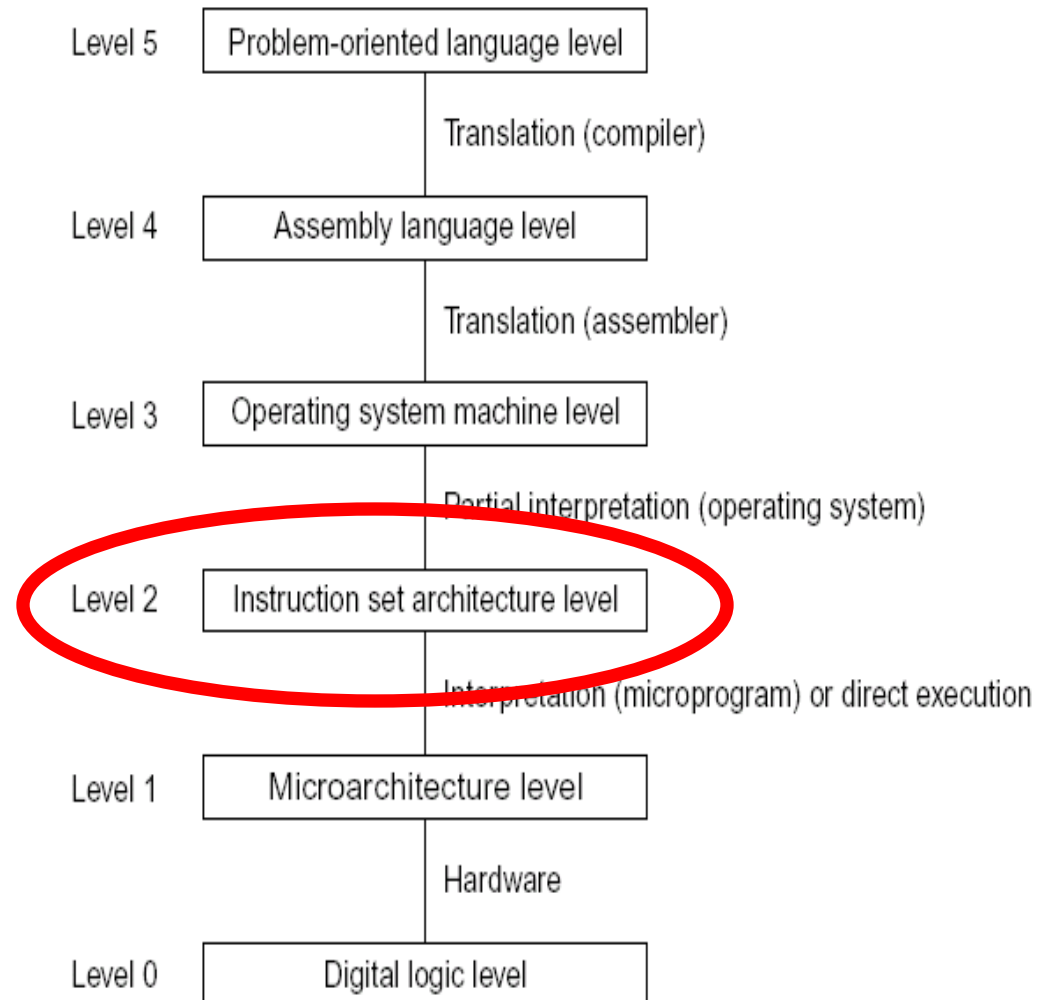


Funzionamento del programma «switch»



Da Assembly al linguaggio macchina

- Nelle prossime slide, vediamo come il calcolatore rappresenta alcune delle istruzioni Assembly MIPS
- Scenderemo cioè al livello architetturale descritto dal **set di istruzioni di macchina**



Formato delle istruzioni MIPS

- Ogni istruzione MIPS è descritta da una parola di 32 bit raggruppati nei seguenti campi
- Tre tipi di istruzioni
 - indirizzamento a registro
 - indirizzamento misto registro-diretto
 - indirizzamento diretto

op(6)	rs(5)	rt(5)	rd(5)	shamt(5)	funct(6)
-------	-------	-------	-------	----------	----------

op(6)	rs(5)	rt(5)	address(16)
-------	-------	-------	-------------

op(6)	address(26)
-------	-------------

Esempi di istruzioni MIPS (R-type)

- Istruzioni `add` e `sub`:

op(6)	rs(5)	rt(5)	rd(5)	shamt(5)	funct(6)
-------	-------	-------	-------	----------	----------

- op: operazione effettuata dall'istruzione
- rs: registro del primo operando sorgente
- rt: registro del secondo operando sorgente
- rd: registro dell'operando destinazione
- shamt: "**shift amount**"
- funct: variante dell'operazione specificata da op

Esempi di istruzioni MIPS (I-type)

- Istruzioni `lw` e `sw`:

op(6)	rs(5)	rt(5)	address(16)
-------	-------	-------	-------------

- rs: registro che contiene l'offset (registro indice)
- rt: registro da/su cui prelevare/immettere i dati
- address: contiene l'indirizzo iniziale di memoria (start-address) a cui sommare l'offset

Istruzione	op	rs	rt	rd	shamt	funct
add	0	reg	reg	reg	0	32
sub	0	reg	reg	reg	0	34
lw	35	reg	reg	n.a.	n.a.	n.a.
sw	43	reg	reg	n.a.	n.a.	n.a.

Esempio codifica da linguaggio assembly a linguaggio macchina

- Assembly:

```
lw $8, Astart($19)
```

```
add $8, $18, $8
```

```
sw $8, Astart($19)
```

es. $Astart = 1200_{10} = 0000\ 0100\ 1011\ 0000_2$

- Linguaggio macchina:

100011	10011	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	10011	01000	0000 0100 1011 0000		

Esempi di istruzioni MIPS ad indirizzamento misto e diretto

- Formato di: `j <address>`

op(6)	address(26)
-------	-------------

— Memoria indirizzabile: 2^{26} parole

- Formato di: `bne R1, R2, <address>`

op(6)	rs(5)	rt(5)	address(16)
-------	-------	-------	-------------

— è evidente che la memoria indirizzabile è ridotta

— poiché però le locazioni a cui saltare sono di solito allocate entro 2^{16} parole attorno all'istruzione corrente, per aumentare la memoria indirizzabile, si somma l'`address` al contenuto del registro PC

Istruzioni di salto a procedura

- Un opportuno sottoinsieme di istruzioni consente la chiamata a procedura
- Dopo che la procedura è stata eseguita, deve essere possibile riprendere l'esecuzione del programma chiamante
- Ciò richiede convenzioni circa:
 - il salvataggio del contesto
 - le modalità di passaggio dei parametri
 - il supporto dell'annidamento fra diverse chiamate

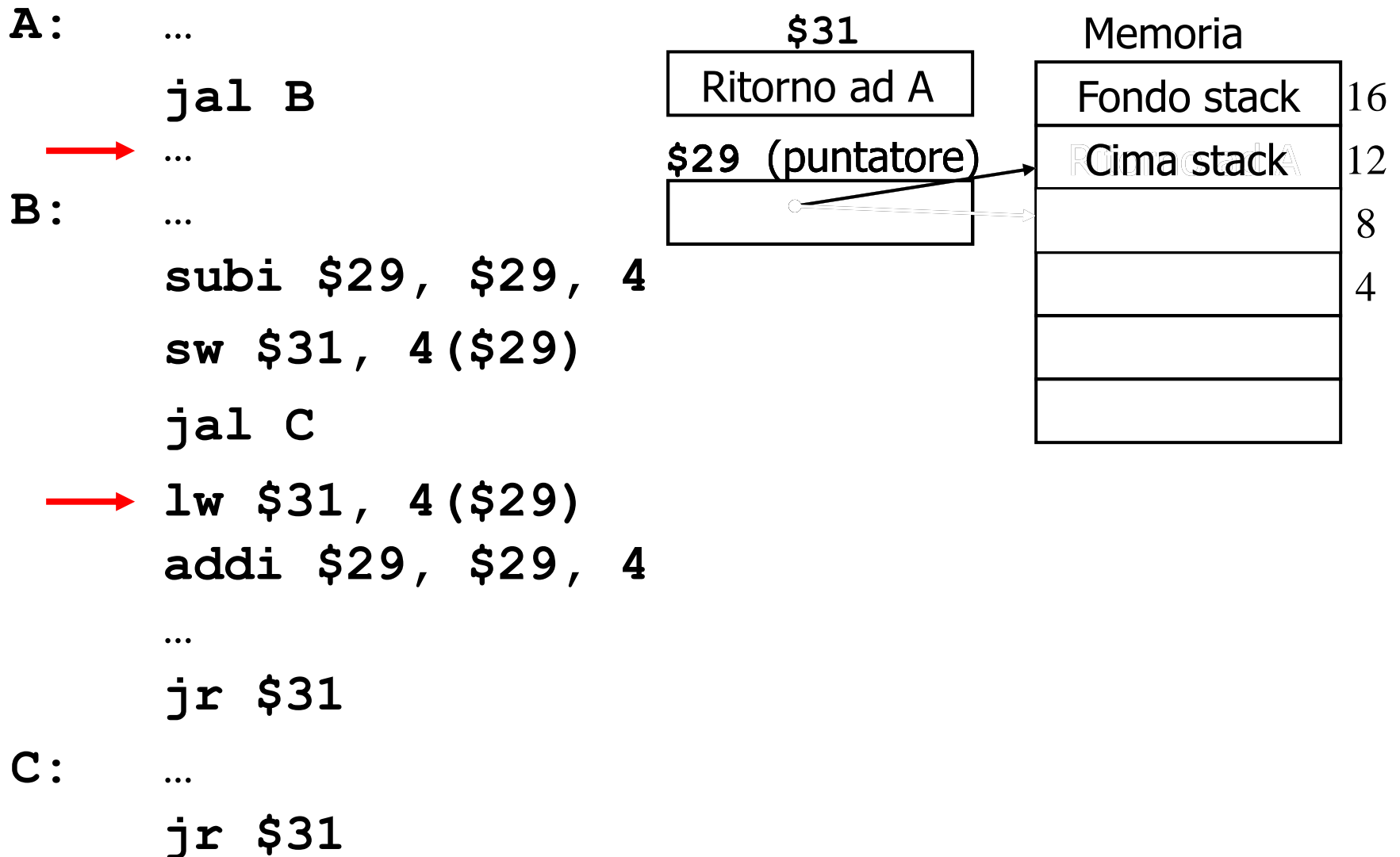
Istruzioni di base per la chiamata a procedura

- `jal <Indirizzo della procedura>`
 - salva l'indirizzo della istruzione successiva nel registro `$31`
 - salta all'indirizzo indicato
 - `jal = jump and link`
- Ricordiamo che l'indirizzo dell'istruzione successiva è contenuto nel registro PC (Program Counter)
- Il collegamento con il chiamante è realizzato attraverso il registro `$31`
- All'istruzione "puntata" da `$31` si ritorna alla fine della procedura con l'istruzione `jr $31` che copia nel PC il contenuto di `$31`

Procedure nidificate

- Il semplice salvataggio di PC in $\$31$ non è sufficiente se entro una procedura avviene una chiamata a un'altra procedura
- Il contenuto di $\$31$ (quindi del PC) viene salvato, ad ogni chiamata a procedura, in un'area di memoria dedicata, detta "stack"
 - richiede un puntatore alla cima dello stack, per indicare dove la successiva procedura dovrebbe mettere i registri o recuperare i vecchi valori
 - le operazioni di inserimento ed estrazione dati in uno stack vengono chiamate rispettivamente "push" e "pop"

Esempio di procedure annidate con gestione dello stack



Convenzioni per il passaggio dei parametri

- Nel MIPS i parametri di ogni procedura vengono inseriti nei registri \$4 - \$7
- I parametri eccedenti vengono inseriti in cima allo stack
- In caso di chiamata ad un'altra procedura, il contenuto di tali registri viene salvato con la stessa modalità del PC (cioé nello stack)
- Se una procedura modifica i registri usati dalla procedura corrente:
 - modalità "caller save": la procedura chiamante è responsabile del salvataggio e del ripristino di tutti i registri da conservare dopo la chiamata; il chiamato può modificare qualunque registro
 - modalità "callee save": il chiamato è responsabile del salvataggio e ripristino di ogni registro che potrebbe utilizzare; il chiamante può quindi usare i registri senza preoccuparsi del loro ripristino dopo la chiamata

Un esempio completo

- Si scriva una funzione Assembly MIPS tale che, dato **l'indirizzo iniziale** di un vettore v **passato nel registro \$4** e **un valore k passato nel registro \$5**, permuti fra loro il k -esimo ed il $k+1$ -esimo valore del vettore.
 - In altre parole, si traduca in Assembly il seguente codice C:

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

Prima fase: allocazione dei registri

- Ricordiamo che la convenzione adottata da MIPS è di salvare i parametri nei registri $\$4 - \7
- In questo caso ci sono solo due parametri, l'indirizzo iniziale di v ed il valore k , che vengono passati attraverso $\$4$ e $\$5$
- Decisioni di progetto:
 - In $\$2$ salviamo l'indirizzo di $v[k]$
 - $v[k]$ viene salvato in $\$15$
 - $v[k+1]$ viene salvato in $\$16$
- Ricordiamo che ogni parola è di 4 byte, per cui $v[k+1]$ si trova **4 byte dopo** $v[k]$

Seconda fase: corpo della procedura

mul \$2, \$5, 4

add \$2, \$4, \$2

lw \$15, 0(\$2)

lw \$16, 4(\$2)

sw \$16, 0(\$2)

sw \$15, 4(\$2)

Terza fase: salvataggio dei registri durante la chiamata a procedura

- Utilizziamo la convenzione "callee save"
- Vengono modificati \$2, \$15 e \$16, quindi è necessario prima salvarli nello stack:

```
addi $29, $29, -12  
sw $2, 0($29)  
sw $15, 4($29)  
sw $16, 8($29)
```
- Il ripristino avviene con l'istruzione `lwr` prima di ritornare alla procedura chiamante

Il codice Assembly della swap

swap: `addi $29, $29, -12`

(1) `sw $2, 0($29)`
`sw $15, 4($29)`
`sw $16, 8($29)`

(2) `mulh $2, $5, 4`
`add $2, $4, $2`
`lw $15, 0($2)`
`lw $16, 4($2)`
`sw $16, 0($2)`
`sw $15, 4($2)`

(3)

`lw $2, 0($29)`
`lw $15, 4($29)`
`lw $16, 8($29)`
`addi $29, $29, 12`

`jr $31` (4)

Un esempio complesso

- Si ordinino gli elementi di un vettore di indirizzo iniziale v di n interi in ordine crescente. v ed n sono passati rispettivamente attraverso i registri \$4 ed \$5.

—In altre parole, si traduca in Assembly il seguente codice C:

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

Algoritmo «Bubble-Sort»

- Prevede l'ordinamento di un vettore confrontando ad ogni iterazione l'elemento $v[j]$ con il suo adiacente $v[j+1]$
- Conseguentemente:
 - «Muove» verso il fondo del vettore (verso la posizione $N-1$) l'elemento «maggiore»
 - L'elemento più «piccolo» è fatto «risalire» verso la cima del vettore (verso la posizione 0)
- Il processo simula le bollicine di gas nell'acqua
 - Essendo più leggere vanno verso l'alto, mentre i corpi pesanti finiscono in profondità
 - Da ciò il nome dell'algoritmo

Prima fase: allocazione dei registri

- Come da testo, l'indirizzo iniziale di v ed n sono **già** memorizzati nei registri $\$4$ e $\$5$
- I registri $\$19$ e $\$17$ contengono i e j
- Il registro $\$8$ servirà per la verifica delle condizioni di terminazione ciclo
- I registri $\$15$ e $\$16$ conterranno l'offset rispetto a v , dato da $j * 4$, e l'indirizzo di $v[j]$
- I registri $\$24$ e $\$25$ conterranno $v[j]$ e $v[j+1]$

Seconda fase: corpo della procedura

- Ciclo `for` esterno

```

                                move $18, $5          #$18<-n
                                move $19, $0          #i=0
for_i:                          slt $8, $19, $18      #se i<n, $8<-1
                                beq $8,$0,exit        #se $8<-0, esci
                                ...
                                (corpo del primo ciclo)
                                ...
                                addi $19, $19, 1      #i++
                                j for_i
exit:
```

Seconda fase: corpo della procedura

- Ciclo `for` interno

```
        addi $17, $19, -1          #j=i-1
for_j:  slt $8, $17, $0            #se j<0, $8<-1 e esce da for_j
        bne $8, $0, exit_j        #(quindi se $8 diverso da 0 esce)
        muli $15, $17, 4          #j*4
        add $16, $4, $15          #&v[j]=&v[0]+j*4
        lw $24, 0($16)            #$24<-v[j]
        lw $25, 4($16)            #$25<-v[j+1]
        slt $8, $25, $24          #$8<-1 se v[j+1]<v[j]
        beq $8, $0, exit_j        #se $8<-0, non effettuare lo
                                   #swap (esci da for_j)

        move $5,$17              #passaggio dei par. a swap (fase 3)
                                   #solo $5 <- j, in $4 è già
                                   #presente &v[]
        jal swap                  #swap(v,j)
        subi $17,$17,1            #j--
        j for_j
```

`exit_j: ...`

Terza fase: il passaggio dei parametri

- \$4 e \$5 vengono usati sia da `sort` sia da `swap` come parametri.
 - `sort` ha come parametri `&v[0]` (\$4) e `n` (\$5)
 - `swap` ha come parametri `&v[0]` (\$4) e `j` (\$5)
- Prima di invocare la `swap`, dobbiamo assicurarci che in \$4 e in \$5 vi siano i parametri corretti: `&v[]` e `j`.
 - \$4 coincide con `&v[]` per entrambe le funzioni.
 - \$5 deve essere sovrascritto con il valore di `j`: **`move $5, $17`**
- Quindi, per evitare conflitti sul registro \$5, all'inizio del corpo della `sort`, copiamo il parametro \$5 sul registro \$18: **`move $18, $5`**
- Tutte le copie possono essere fatte con la pseudo-istruzione `move`

Quarta fase: salvataggio dei registri

- Utilizziamo la convenzione "callee save", e salviamo i registri necessari nello stack:

```
addi $29, $29, -36
```

```
sw $31, 0($29)
```

```
sw $18, 4($29)
```

```
sw $17, 8($29)
```

```
sw $19, 12($29)
```

```
sw $8, 16($29)
```

```
sw $15, 20($29)
```

```
sw $16, 24($29)
```

```
sw $24, 28($29)
```

```
sw $25, 32($29)
```

N.B.: è necessario salvare nello stack anche il registro \$31 perché dentro la procedura `sort` si invoca un'altra procedura, `swap`, mediante l'istruzione `jal` (che di fatto modifica \$31).

La procedura completa

sort:

```
    addi $29, $29, -36
    sw $31, 0($29)
    sw $18, 4($29)
    sw $17, 8($29)
    sw $19, 12($29)
    sw $8, 16($29)
    sw $15, 20($29)
    sw $16, 24($29)
    sw $24, 28($29)
    sw $25, 32($29)
```

```
    move $18, $5
    move $19, $0
```

for_i:

```
    slt $8, $19, $18
    beq $8,$0,exit
    addi $17, $19, -1
```

for_j:

```
    slt $8, $17, $0
    bne $8, $0, exit_j
```

```
    muli $15, $17, 4
    add $16, $4, $15
    lw $24, 0($16)
    lw $25, 4($16)
    slt $8, $25, $24
    beq $8, $0, exit_j
```

```
    move $5,$17
    jal swap
```

```
    subi $17,$17,1
    j for_j
```

exit_j:

```
    addi $19,$19,1
    j for_i
```

exit:

```
    lw $31, 0($29)
    lw $18, 4($29)
    lw $17, 8($29)
    lw $19, 12($29)
    lw $8, 16($29)
    lw $15, 20($29)
    lw $16, 24($29)
    lw $24, 28($29)
    lw $25, 32($29)
    addi $29, $29, 36
    jr $31
```

Per saperne di più...

- Patterson, A.D., Hennessy, J., *Struttura, organizzazione e progetto dei calcolatori elettronici*, Jackson Libri, Collana Università, 2000
- Stallinga, P., *Computer Architecture: with (MIPS) Assembly*, stallinga.org, 2020
- M.A.R.S. (MIPS Assembler and Runtime Simulator), <http://courses.missouristate.edu/kenvollmar/mars/>
- SPIM Simulator, <http://spimsimulator.sourceforge.net/>