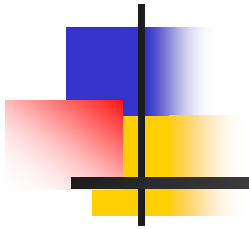
Università degli Studi di Roma "La Sapienza"

Architettura degli elaboratori II

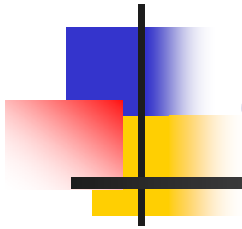
Istruzioni e programmi

Tratto da materiale di Solmi e Montresor – Univ. Di Bologna



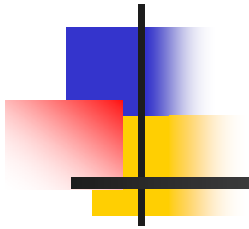
Indice degli argomenti

- **Le istruzioni assembly**
  - Categorie di istruzioni assembly
  - Istruzioni di caricamento e salvataggio
  - Gestione del segno
  - Ordine dei bytes (endianness)
  - Modi di indirizzamento
  - Istruzioni aritmetiche, logiche e di scorrimento
  - Istruzioni di confronto e salto condizionato
  - Istruzioni di salto non condizionato



Categorie istruzioni assembly

- Istruzioni “Load and Store”
  - Spostano dati tra la memoria e i registri generali del processore
- Istruzioni “Load Immediate”
  - Caricano nei registri valori costanti
- Istruzioni “Data Movement”
  - Spostano dati tra i registri del processore
- Istruzioni aritmetico/logiche
  - Effettuano operazioni aritmetico, logiche o di scorrimento sui registri del processore
- Istruzioni di confronto
  - Effettuano il confronto tra i valori contenuti nei registri
- Istruzioni di salto condizionato
  - Spostano l’esecuzione da un punto ad un altro di un programma in presenza di certe Condizioni
- Istruzioni di salto non condizionato
  - Spostano l’esecuzione da un punto ad un altro di un programma



Architettura LOAD-STORE

L'architettura di MIPS è di tipo Load-and-Store

In pratica, la maggioranza delle istruzioni di MIPS operano tramite i registri interni al processore

add $t0, $t1, $t2

Somma $t1+$t2 e mette il risultato in $t0

Per questo motivo

LOAD: i dati da elaborare devono essere prelevati dalla memoria

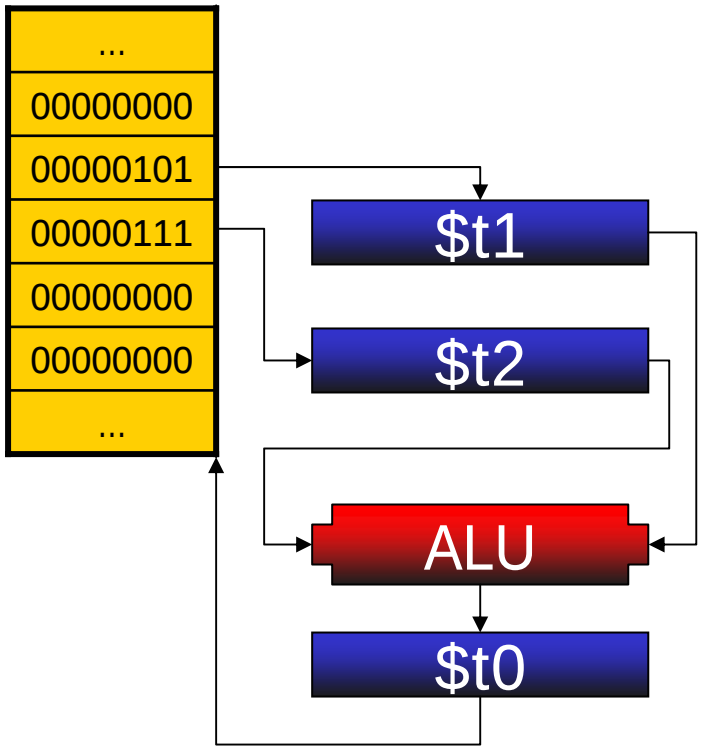
STORE: i risultati devono essere salvati in memoria

Istruzioni LOAD-STORE

lb rdest, address	*Carica un byte sito all'indirizzo address nel registo rdest*
lbu rdest, address	*Carica un unsigned-byte sito all'indirizzo address nel registo rdest*
lh rdest, address	*Carica un halfword sito all'indirizzo address nel registo rdest*
lhu rdest, address	*Carica un unsigned-halfword sito all'indirizzo address nel registo rdest*
lw rdest, address	*Carica una word sita all'indirizzo address nel registo rdest*
la rdest, address	*Carica un indirizzo address nel registo rdest*
sb rsource, address	*Memorizza un byte all'indirizzo address prelevandolo dal registro rsource*
sh rsource, address	*Memorizza un halfword all'indirizzo address prelevandolo dal registro rsource*
sw rsource, address	*Memorizza una word all'indirizzo address prelevandola dal registro rsource*

Esempio istruzioni L-S

```
.text                   # Inizia il codice

lw $t1, operandA        # Load operandA

lw $t2, operandB        # Load operandB

add $t0, $t1, $t2       # operandA +
                        operandB
sw $t0, result          # Store result

.data                   # Iniziano i dati

operandA: .word 5

operandB: .word 7

result: .word 0
```

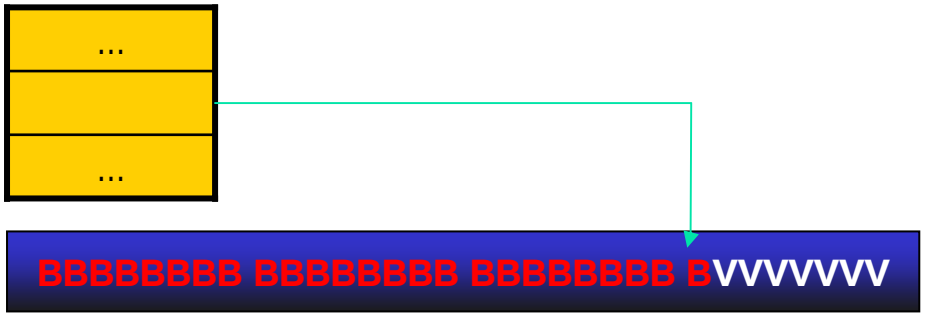


Load dati con segno e senza segno

- byte -

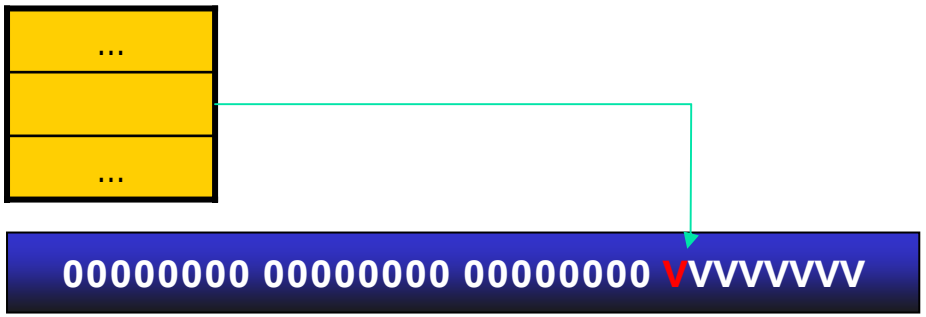
LB $t0, address

- Carica il byte indirizzato da *address* nel byte meno significativo del registro
- **Il bit di segno viene esteso**



LBU $t0, address

- Carica il byte indirizzato da address nel byte meno significativo del registro
- **Gli altri tre byte vengono posti a zero**



Load dati con segno e senza segno

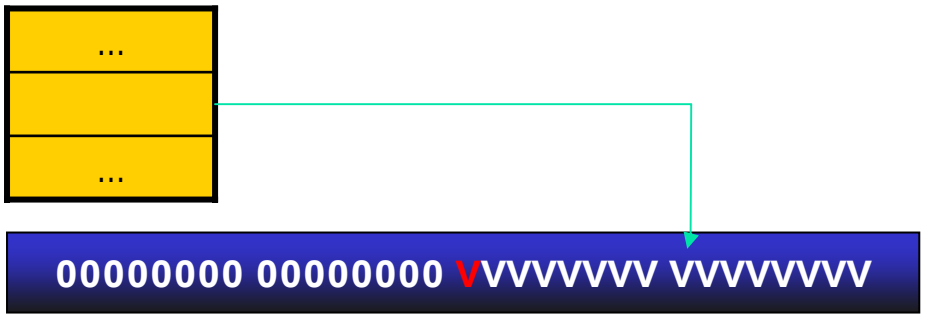
- Halfword -

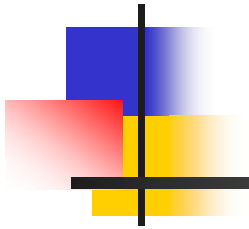
LH $t0, address

- Carica i 2 byte indirizzati da *address* nei 2 byte meno significativi del registro
- Il valore codifica un **numero relativo**
- **Il bit di segno viene esteso**

LHU $t0, address

- Carica i 2 byte indirizzati da address nei 2 byte meno significativi del registro
- Il valore codifica un **numero intero**
- **Gli altri due byte vengono posti a zero**





Endianness

Rappresentazione di una word (32 bit) in 4 byte di memoria

- **Little Endian**: memorizza prima la “little end” (ovvero i bit meno significativi) della word
- **Big Endian**: memorizza prima la “big end” (ovvero i bit più significativi) della word

Nota:

- I processori x86 sono little-endian
- Il processore MIPS può essere usato sia in modo little endian che big endian
- L’endianness di SPIM dipende dal sistema in cui viene eseguito, quindi in generale è little endian

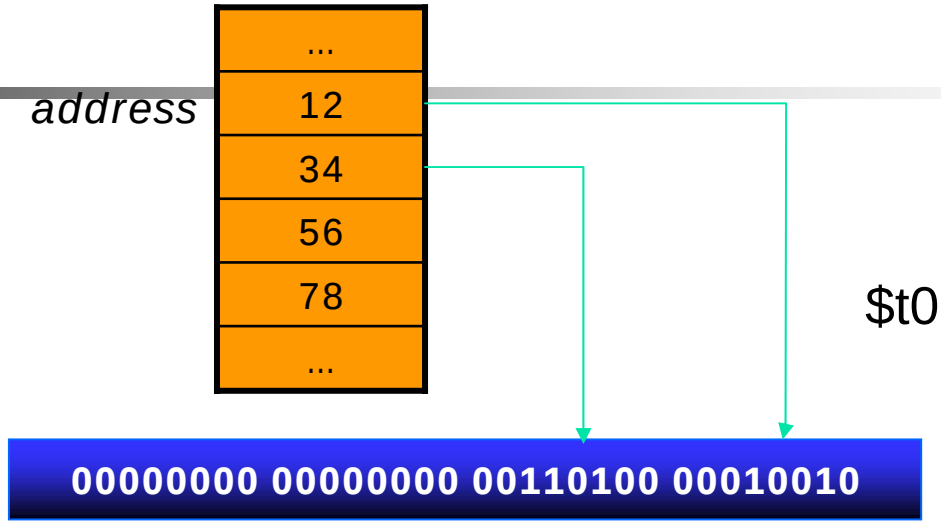
Quando bisogna affrontare i problemi di endianness?

- Quando si "mischiano" operazioni su 8, 16 e 32 bit.
- Quando invece si utilizzano operazioni uniformi, non vi sono problemi di sorta

Endianness:esempio

Little Endian:

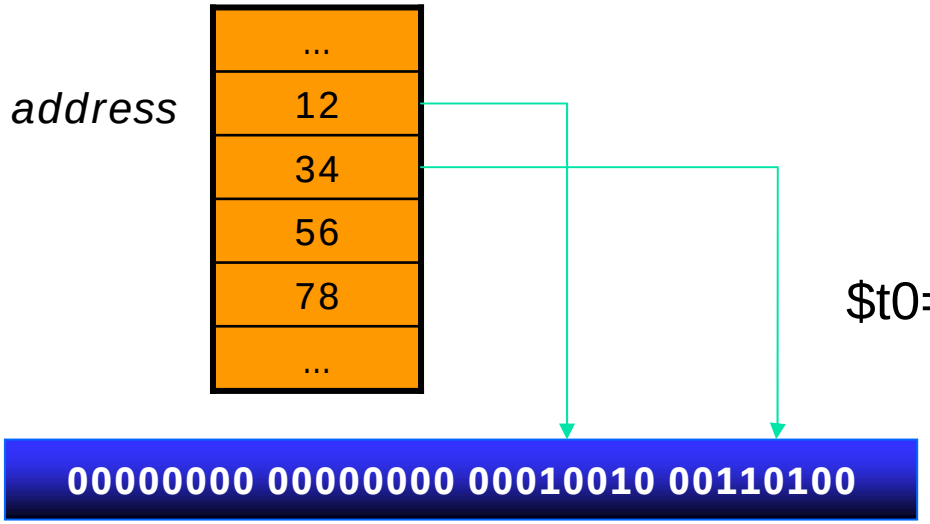
lh $t0,address



$t0=0x00003412

Big Endian:

lh $t0,address

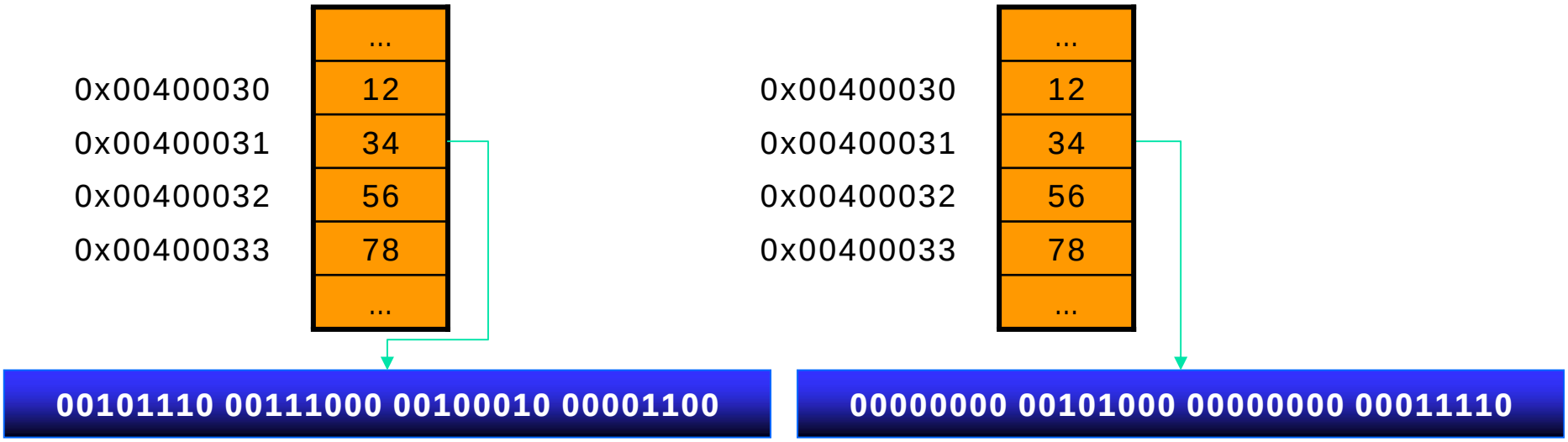


$t0=0x00001234

Load Adress vs Load Word

- L'istruzione **lw $t0, address** carica ***il contenuto*** della word indirizzata dall'etichetta *address* in memoria

- L'istruzione **la $t0, address** carica ***l'indirizzo*** della word indirizzata dall'etichetta *address* in memoria
(utile quando si vuole caricare in un registro l'indirizzo di una particolare zona di memoria)



Modi di Indirizzamento

Un modo per esprimere un indirizzo di memoria

NB: alla fine il risultato è sempre un indirizzo di memoria dove si andrà a leggere (load) o scrivere (store) un dato.

Nel linguaggio macchina MIPS, esiste un solo modo di indirizzamento

imm(register) dove l'indirizzo è dato dalla somma del ***valore immediato*** *imm* più il ***contenuto*** del registro *register*

Esempio: **lw $t0, 4 ($sp)** *(legge una word dall'indirizzo $sp + 4)*

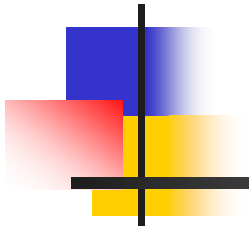
L'assembler fornisce per comodità più modi di indirizzamento

Un indirizzo si può esprimere come somma di:

una **etichetta** + una **espressione costante** + il contenuto di un **registro**

Esempio: **lw $t1, array + 0x100 ($t0)**
(legge una word dall'indirizzo 0x10010104 assumendo: array = 0x10010000 e $t0 = 4)

Modi di Indirizzamento



(base register)

- Specifica un indirizzo tramite registro base, assumendo un offset 0.
- L'indirizzo è dato dal contenuto del base register

Esempio: **lw $t0,($a0)**

expression

- Specifica un indirizzo assoluto con una espressione fatta solo di somme e sottrazioni di costanti espresse in decimale o esadecimale.
- L'indirizzo è dato dal risultato dell'espressione (costante).

Esempio: **lw $t0, 0x00 40 00 00 + 4**

- I numeri negativi vanno comunque preceduti anche dal simbolo +.

Esempio: **lw $t0, 0x10 01 00 0c + -4**

Modi di Indirizzamento

expression(base register)

- Specifica un indirizzo tramite registro base e un'offset tramite un'espressione.
- L'indirizzo è dato dal contenuto del base register + il risultato dell'espressione

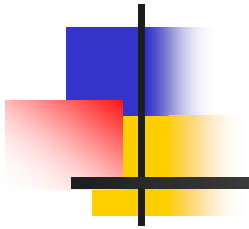
Esempio: **lw $t0, 4 ($sp)**

Relocatable-symbol

- Specifica un indirizzo (rilocabile) tramite un'etichetta.
- L'assemblatore genera le istruzioni necessarie per gestire l'etichetta e aggiunge al modulo oggetto generato le informazioni di rilocazione.

Esempio: **lw $t0, operandA**

Modi di Indirizzamento



Relocatable-symbol(index register)

- Specifica un indirizzo base tramite una etichetta, mentre l'offset viene specificato tramite un registro indice
- L'indirizzo è dato dalla somma dell'indirizzo associato all'etichetta con il contenuto del registro indice.

Esempio: **lw $t0, array ($t1)**

Relocatable-symbol + expression

- Specifica un indirizzo base tramite una etichetta, mentre l'offset viene specificato tramite un'espressione
- L'indirizzo è dato dalla somma dell'indirizzo associato all'etichetta con il risultato dell'espressione

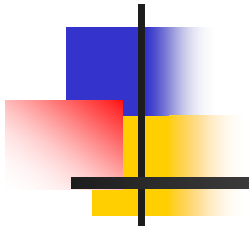
Esempio: **lw $t0, array + 4**

Modi di Indirizzamento

Relocatable-symbol + expression (*index register*)

- Specifica un indirizzo base tramite una etichetta, mentre l'offset viene specificato tramite un registro indice e un'espressione
- L'indirizzo è dato dalla somma dell'indirizzo associato all'etichetta con il contenuto del registro indice e il risultato dell'espressione.

Esempio: **lw $t0, array + 4($t1)**



Istruzioni Data Movement and Load Immediate

move rdest, rsource	*Muove il registro rsource nel registro rdest*
mfhi rdest	*Muove il registro hi nel registro rdest*
mflo rdest	*Muove il registro lo nel registro rdest*
mthi rsource	*Muove il registro rsource nel registro hi*
mtlo rsource	*Muove il registro rsource nel registro lo*
li rdest, imm	*Muove immediate imm nel registro rdest*
lui rdest, imm	*Muove la immediate halfword imm nella parte più alta dell'halfword del registro rdest*

Istruzioni Aritmetiche

add rd, rs, rt	*rd = rs + rt*	*(with overflow)*
addu rd, rs, rt	*rd = rs + rt*	*(without overflow)*
addi rd, rs, imm	*rd = rs + imm*	*(with overflow)*
addiu rd, rs, imm	*rd = rs + imm*	*(without overflow)*
sub rd, rs, rt	*rd = rs - rt*	*(with overflow)*
subu rd, rs, rt	*rd = rs – rt*	*(without overflow)*
neg rd, rs	*rd = - rs*	*(with overflow)*
negu rd, rs	*rd = - rs*	*(without overflow)*
abs rd, rs	*rd = |rs|*	.

Istruzioni Aritmetiche

mult rs, rt	*hi,lo = rs * rt (signed, overflow impossibile)*
multu rs, rt	*hi,lo = rs * rt (unsigned, overflow impossibile)*
mul rd, rs, rt	*rd = rs * rt (without overflow)*
mulo rd, rs, rt	*rd = rs * rt (with overflow)*
mulou rd, rs, rt	*rd = rs * rt (with overflow, unsigned))*
div rs, rt	*hi,lo = resto e quoz. di rs / rt (signed with overflow)*
divu rs, rt	*hi,lo = resto e quoz. di rs / rt (unsigned, no overflow)*
div rd, rs, rt	*rd = rs / rt (signed with overflow)*
divu rd, rs, rt	*rd = rs / rt (unsigned)*
rem rd, rs, rt	*rd = resto di rs / rt (signed)*
remu rd, rs, rt	*rd = resto di rs / rt (unsigned)*

Istruzioni Logiche

and rd, rs, rt	*rd = rs **AND** rt*
andi rd, rs, imm	*rd = rs **AND** imm*
or rd, rs, rt	*rd = rs **OR** rt*
ori rd, rs, imm	*rd = rs **OR** imm*
xor rd, rs, rt	*rd = rs **XOR** rt*
xori rd, rs, imm	*rd = rs **XOR** imm*
nor rd, rs, rt	*rd = **NOT** (rs **OR** rt)*
not rd, rs	*rd = **NOT** rs*

Istruzioni di Shift

sll rd, rs, rt	*rd = rs shifted left rt mod 32 bits*
srl rd, rs, rt	*rd = rs shifted right rt mod 32 bits*
sra rd, rs, rt	*rd = rs shifted right rt mod 32 bits (signed)*
rol rd, rs, rt	*rd = rs rotated left rt mod 32 bits*
ror rd, rs, rt	*rd = rs rotated right rt mod 32 bits*

Note:

- *rt* può anche essere una costante es. **sll $t1, $t2, 4**
- **Shift** perde bit da una parte, dall'altra entrano degli zero
- **Rotate** inserisce da una parte i bit che escono dall'altra
- Direzione: Left [right] verso quelli più [meno] significativi

Istruzioni Immediate ed Unsigned

Versioni immediate (i)

- Le versioni immediate (i) delle istruzioni precedenti utilizzano un **valore costante** (risultato di un'espressione) al posto di uno degli operandi
- Non sempre è necessario specificare la i; l'assemblatore è in grado di riconoscere le istruzioni immediate. Esempio: **add $t0, $t0, 1**

Versioni unsigned (u)

- Le versioni unsigned delle istruzioni non gestiscono le problematiche relative all'overflow
- Dettagli in seguito quando parleremo di ***eccezioni***

Istruzioni di confronto

slt rd, rs, rt	*rd* = ***1*** *se rs* **<** *rt*, ***0*** *altrimenti (con overflow)*
sltu rd, rs, rt	*rd* = ***1*** *se rs* **<** *rt*, ***0*** *altrimenti (no overflow)*
slti rd, rs, imm	*rd* = ***1*** *se rs* **<** *imm*, ***0*** *altrimenti (con overflow)*
sltiu rd, rs, imm	*rd* = ***1*** *se rs* **<** *imm*, ***0*** *altrimenti (no overflow)*
sle, sgt, sge, seq, sne	*Analoghe per testare <=, >, >=, = e <>*

NB: *Di solito queste istruzioni vengono usate solo dall'assembler.*
I programmatori usano direttamente le istruzioni di ***salto condizionato***

Istruzioni di salto condizionato

beq rs, rt, target	*Salta all'istruzione puntata da target se rs = rt*
bne rs, rt, target	*Salta all'istruzione puntata da target se rs != rt*
bgez rs, target	*Salta all'istruzione puntata da target se rs >= 0*
bgtz rs, target	*Salta all'istruzione puntata da target se rs > 0*
blez rs, target	*Salta all'istruzione puntata da target se rs <= 0*
bltz rs, target	*Salta all'istruzione puntata da target se rs < 0*
beqz rs, target	*Salta all'istruzione puntata da target se rs = 0*
bnez rs, target	*Salta all'istruzione puntata da target se rs!= 0*
bge rs, rt, target **bgeu rs, rt, target**	*Salta all'istruzione puntata da target se rs >= rt*
bgt rs, rt, target **bgtu rs, rt, target**	*Salta all'istruzione puntata da target se rs > rt*
ble rs, rt, target **bleu rs, rt, target**	*Salta all'istruzione puntata da target se rs <= rt*
blt rs, rt, target **bltu rs, rt, target**	*Salta all'istruzione puntata da target se rs < rt*

Branch: osservazione

Nel linguaggio macchina, l'indirizzo dell'istruzione target in un branch viene espresso come *instruction offset* rispetto all'istruzione corrente

È possibile saltare 2^{15} -1 istruzioni in avanti o 2^{15} istruzioni indietro

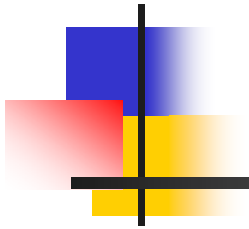
2^{15} = circa 32000

In linguaggio assembly, l'indirizzo dell'istruzione target è dato dall'etichetta associata

Il calcolo viene fatto automaticamente dall'assembler :

Esempio di istruzione di salto condizionato in SPIM:

0x00400030 beq $5, $4, 8 [stor-0x00400030] beq $a1, $a0, stor



Istruzioni di salto incondizionato

j target	*Salto incondizionato all'istruzione target*
jal target	*Salto incondizionato all'istruzione target*
	e salvataggio dell'indirizzo della prossima istruzione in $ra
jr rsource	*Salto incondizionato all'istruzione che ha indirizzo memorizzato nel registro rsource*
jalr rsource, rdest	*Salto incondizionato all'istruzione che ha indirizzo memorizzato nel registro rsource*
	e salvataggio dell'indirizzo della prossima istruzione in rdest

Sequenza di istruzioni

Le istruzioni formano sempre una sequenza che corrisponde all'ordine in cui vengono eseguite

- Il microprocessore non ha una visione di insieme del programma: "vede" solo l'istruzione che deve eseguire e il contenuto dei registri.
- Esegue una istruzione alla volta nell'ordine scritto in memoria

Fanno eccezione le istruzioni di salto che fanno proseguire l'esecuzione all'indirizzo specificato.

- L'istruzione che viene eseguita dopo un salto è quella che si trova all'indirizzo specificato come destinazione del salto.
- Se il salto è condizionato e la condizione è falsa si prosegue normalmente con l'istruzione successiva.

L'intero programma è una sequenza di istruzioni; con i salti si possono controllare l'ordine di esecuzione

Come realizzare l'istruzione IF

Si esegue del codice solo se una condizione (espressione) è vera

In assembler

1. Codice per valutare l'espressione e mettere il risultato in un registro (o due)

2. Salto condizionato per saltare il codice opzionale *se* la condizione è *falsa*

3. Codice opzionale da eseguire *solo* quando la condizione è vera

Esempio in assembly:

*Se x > 0 allora y = x*2 (altrimenti y resta invariato cioè uguale a 0)*

```
.text
.globl main
main:
lw $t0,x # Carica x
blez $t0,endif # Salto se condizione falsa
mul $t1,$t0,2 # codice opzionale
sw $t1,y # Salva $t1 in variabile y
endif: # Fine if
.data
x: .word 5 # valore
y: .word 0 # y = x*2 se x>0; y = 0
```

Come fare l' IF… ELSE

Si esegue del codice solo se una condizione (espressione) è vera altrimenti si esegue altro codice

In assembler

1. Codice per valutare l'espressione e mettere il risultato in un registro (o due)

2. Salto condizionato per saltare il "ramo then" *se* la condizione è *falsa*

3. Codice da eseguire *solo* quando la condizione è vera (ramo then)

4. Salto incondizionato per evitare di eseguire *anche* il ramo else

5. Codice da eseguire *solo* quando la condizione è falsa (ramo else)

NB. Posso scambiare i due rami (punti 3 e 5) se salto quando la condizione è vera

Esempio in assembly:

*Se x > 0 allora y = x*2 altrimenti y = -x.*

```
.text
.globl main
main:
lw $t0, x          # Carica x
blez $t0, else     # if $t0 <= 0
   mul $t1,$t0,2 # ramo THEN
   j end           # Salta all fine
else:
   neg $t1, $t0    # ramo ELSE
end:
sw $t1, y          # dopo l'IF…ELSE

.data
x: .word 5         # valore
y: .word 0         # risultato
```

Istruzione IF Annidati

Si esegue del codice solo se una condizione (espressione) è vera altrimenti si esegue altro codice

In assembler

Per farlo in assembly mi basta applicare uno dentro l'altro i due schemi di soluzione già visti nei due lucidi precedenti

NB: codice complicato e poco chiaro

Esempio in assembly:

Se x > 0 allora (ramo then 1 che contiene scelta annidata)

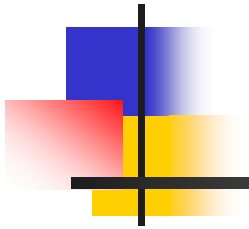
se y >= 0 allora y = x*2 (ramo then 2)

altrimenti y = -x (ramo else 2)

altrimenti incrementa y (ramo else 1)

```
lw $t0,x # Carica x
lw $t1,y # Carica y
blez $t0,else1 # Salta se la cond1 falsa
bltz $t1,else2 # Salta se la cond2 falsa
mul $t1,$t0,2 # Ramo then2: $t1 = x*2
j end # Salta fine
else2:neg $t1,$t0 # Ramo else2: $t1 =-x
j end # Salta fine
else1:addi $t1,$t1,1 # Ramo else1:$t1=y+1
end: sw $t1, y # Scrive $t1 in y
```

Istruzione SWITCH



Le condizioni davanti ad ogni ramo testano valori diversi di una stessa variabile

In assembler

Per farlo in assembly uso la variabile come indice per caricare dalla tabella l'indirizzo del codice dove saltare.

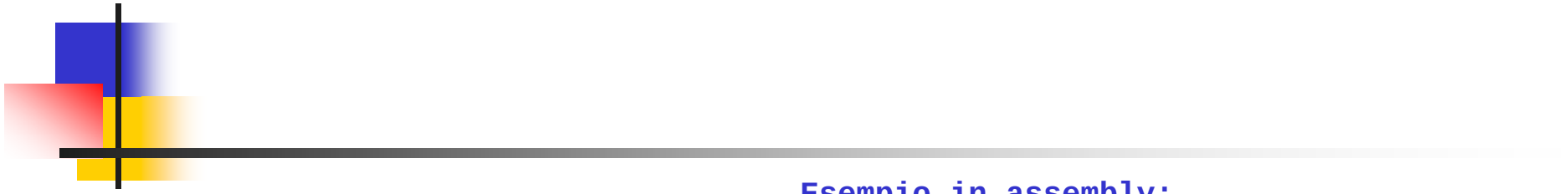
NB: viene fatto un solo test anziché uno per ogni ramo, riducendo la complessità da lineare a costante.

Esempio in assembly:

*se op = 0 allora y = x*3 se op = 1 allora y = x+5*
se op = 2 allora y = -x altrimenti y = x

```
lw $t1,x
lw $t0,op
bge $t0,3,default
sll $t0,$t0,2              # $t0 = op*4
lw $t0,stable($t0) # $t0 = (stable+op*4)
jr $t0   # salto alla routine giusta
case0:mul $t2,$t1,3        # $t2 = x*3
j end
case1:addi $t2,$t1,5       # $t2 = x+5
j end
case2:div $t2,$t1,2        # $t2 = x/2
j end
default: move $t2,$t1      # $t2 = x
end:sw $t2,y      # Scrive $t2 in y
.data
stable: .word case0,case1,case2
```

Istruzione WHILE



Voglio eseguire più volte del codice finché una espressione è vera. Non so se l'espressione è vera all'inizio (quindi la controllo prima di eseguire il codice)

In assembler

1. Codice per valutare l'espressione e mettere il risultato in un registro (o due)

2. Salto condizionato per saltare il codice da ripetere se la condizione è falsa

3. Codice da ripetere finché la condizione è vera

4. Salto al punto 1 in modo da poter eseguire il codice un'altra volta oppure

Salto condizionato al punto 3 per eseguire un'altra volta se la condizione è vera

Esempio in assembly:

Finché x > 0 lo sommo a y (y = y + x)
e lo decremento (x = x – 1)

```
lw $t0,x      # Carica x
lw $t1,y      # Carica y
blez $t0,end
while:        # (Inizio ciclo)
   add $t1,$t1,$t0
   sub $t0,$t0,1 # Sottrae uno
bgtz $t0,while
end:
sw $t1,y # Salva risultato in y
```

Istruzione LOOP

Voglio eseguire più volte del codice finché una espressione è vera. So che all'inizio l'espressione è vera (quindi eseguo il codice e poi controllo)

In assembler

Inizializzare i registri che mi servono prima di iniziare il ciclo

1. Codice da ripetere finché la condizione è vera

2. Codice per valutare l'espressione e mettere il risultato in un registro (o due)

3. Salto condizionato al punto 1 per eseguire un'altra volta se la condizione è *vera*

Esempio in assembly:

Finché x > 0 lo sommo a y (y = y + x) e

lo decremento (x = x – 1)

```
lw $t0,x            # Carica x
lw $t1,y            # Carica y
dowhile:            # (Inizio ciclo)
   add $t1,$t1 $t0  # Effettua sommatoria
   sub $t0,$t0,1    # Decrementa x
bgtz $t0,dowhile    # Riesegue ciclo finché cond. vera
sw $t1,y            # Salva risultato in y
```

Istruzione FOR

Voglio eseguire del codice per un certo numero di volte; voglio anche poter usare l'indice (numero di volte eseguite) nel codice.)

In assembler

Inizializzare i registri che mi servono prima di iniziare il ciclo (compreso l'indice)

1. Codice da ripetere il numero stabilito di volte

2. Incremento il registro usato come indice

3. Salto condizionato al punto 1 per eseguire un'altra volta se la condizione è *vera*

Esempio in assembly:

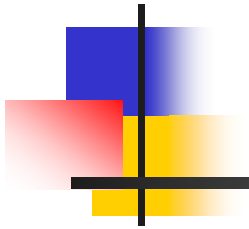
Per 5 volte (contando da 1 a 5) sommo l'indice a y (y = y + x)

```
li $t0,1 # Inizializza indice
lw $t1,y      # Carica y
loop:         # (Inizio ciclo)
   add $t1,$t1,$t0 # Fa la somma
   addi $t0,$t0,1 # Incr. indice
ble $t0,5,loop # Continua ciclo
se <= 5 volte
sw $t1,y # Salva risultato in y
```

Architettura degli elaboratori II

Chiamate a funzioni del Sistema Operativo

Chiamata a funzioni di SO



SPIM fornisce un piccolo insieme di servizi "operating-system-like" attraverso l'istruzione syscall

Per richiedere un servizio:

- Si carica il numero che identifica il servizio nel registro **$v0**
- Eventuali argomenti vengono caricati nei registri **$a0-$a1**
- Eventuali risultati vengono collocati dalla chiamata di sistema nel registro **$v0**

Esempio

```
li $v0,1 # Numero che identifica print_int
li $a0,5 # 5 è l'argomento da stampare
syscall # procedi alla stampa
```

Elenco System Call di SPIM

SERVIZI	CODICE	ARGOMENTI	RISULTATO
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer in $v0
read_float	6		float in $f0
read_double	7		double in $f0
read_string	8	$a0=buffer, $a1=length	
sbrk	9	$a0=amount	address in $v0
exit	10		

Esempio di uso di System Call

```
.text
.globl main
main:
li $v0,4      # print_string
la $a0,prompt
syscall       # stampa il prompt
li $v0,8      # read_string
la $a0,buffer
li $a1,256
syscall       # legge un nome
li $v0,4      # print_string
la $a0,msg
syscall # stampa “Ciao " + nome
li $v0, 10 # exit
syscall # termina programma


.data
prompt:
.asciiz "Come ti chiami: "
msg:
.ascii “Ciao "
buffer:
.space 256
```