

# **Architettura Elaboratori Elettronici ESERCITAZIONI**

ISTRUZIONI MARS

**Franco Liberati**  
**[liberati@di.uniroma1.it](mailto:liberati@di.uniroma1.it)**



# Argomenti

- ❑ Istruzioni di spostamento
- ❑ Istruzioni logiche-aritmetiche
- ❑ Istruzioni di confronto
- ❑ Istruzioni di salto
  - ❖ Condizionato
  - ❖ Incondizionato
  - ❖ Istruzione di salto a funzione





# Suddivisione in classi

**Le istruzioni del linguaggio assembly possono essere divise nelle seguenti categorie:**

## **Istruzioni “Load and Store”**

Spostano dati tra la memoria e i registri generali del processore  
*(i valori non sono modificati, ma solo spostati)*

## **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  
*(il valore risultante modifica i condition code della ALU)*

## **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

## **Istruzioni di sistema o comando**

Influenzano lo svolgimento del programma (HALT, NOP, BREAK, TRAP ...)

# RAPPRESENTAZIONE DELLE PAROLE NEGLI ELABORATORI ELETTONICI





# Little endian / Big endian

❑ **Definizione di endianness:** disposizione di una parola (word) nei byte di memoria

❑ **Little Endian:** memorizza prima la “little end” (ovvero i bit meno significativi) della word nelle locazioni di memoria con indirizzo più basso

❑ **Big Endian:** memorizza prima la “big end” (ovvero i bit più significativi) della word nelle locazioni di memoria con indirizzo più basso

*Terminologia ripresa da “I viaggi di Gulliver” di Jonathan Swift, in cui due fazioni sono in lotta per decidere se le uova sode devono essere aperte a partire dalla “little end” o dal “big end” dell'uovo*

## ❑ **NB:**

❑ I processori x86 sono little-endian

❑ MIPS può essere little endian o 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

❑ Se si utilizzano operazioni uniformi, non vi sono problemi di sorta



# Little endian / Big endian

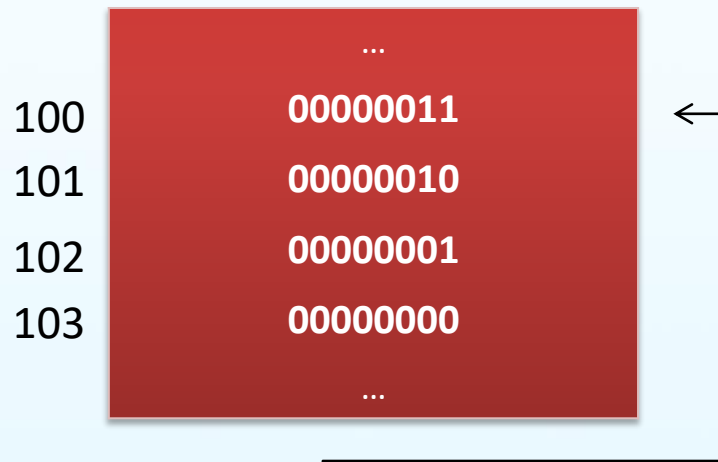
Esempio

Little Endian:



00000011 00000010 00000001 00000000

Big Endian:



00000011 00000010 00000001 00000000





# Little endian / Big endian

Esempio

**Little Endian:**

**lh \$t0,value**

#copia 16 bit a partire dalla locazione  
#con indirizzo value

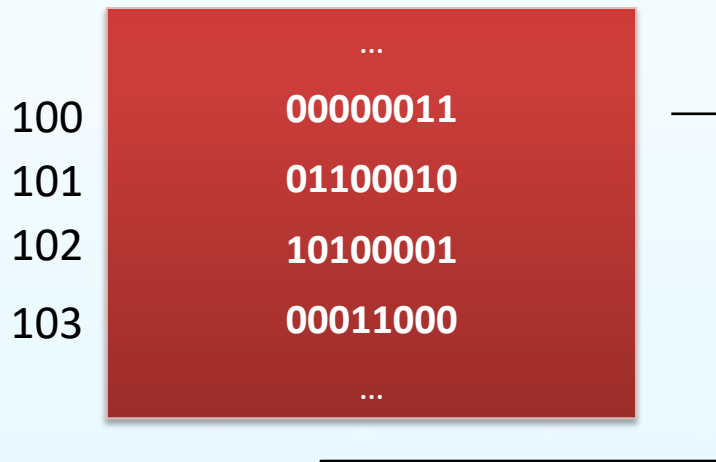


00000000 00000000 10100001 00001100

**Big Endian:**

**lh \$t0,value**

#copia 16 bit a partire dalla locazione  
#con indirizzo value



00000000 00000000 01100010 00000011

# ISTRUZIONI DI SPOSTAMENTO







# Generalità architettura Load-Store

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

In pratica, la maggioranza delle istruzioni di MIPS elaborano operandi siti nei registri interni al processore e non accedendo ad essi direttamente nelle celle di memoria nei quali risiedono

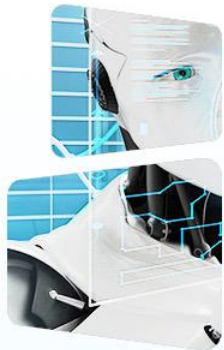
Esempio:

**add \$t0, \$t1, \$t2**      **#Somma \$t1+\$t2 e mette il risultato in \$t0**

Per questo motivo

**LOAD:** il dato è presente in una locazione di memoria è copiato in un registro

**STORE:** il valore di un registro è memorizzato in una locazione di memoria



# Istruzioni di spostamento

<b>lb rdest, address</b>	<i>Copia un byte sito all'indirizzo <b>address</b> nel registro <b>rdest</b></i>
<b>lbu rdest, address</b>	<i>Copia un unsigned-byte sito all'indirizzo <b>address</b> nel registro <b>rdest</b></i>
<b>lh rdest, address</b>	<i>Copia un halfword sito all'indirizzo <b>address</b> nel registro <b>rdest</b></i>
<b>lhu rdest, address</b>	<i>Copia un unsigned-halfword sito all'indirizzo <b>address</b> nel registro <b>rdest</b></i>
<b>lw rdest, address</b>	<i>Copia una word sita all'indirizzo <b>address</b> nel registro <b>rdest</b></i>
<b>la rdest, address</b>	<i>Copia un indirizzo <b>address</b> nel registro <b>rdest</b></i>
<b>sb rsource, address</b>	<i>Memorizza un byte all'indirizzo <b>address</b> prelevandolo dal registro <b>rsource</b></i>
<b>sh rsource, address</b>	<i>Memorizza un halfword all'indirizzo <b>address</b> prelevandolo dal registro <b>rsource</b></i>
<b>sw rsource, address</b>	<i>Memorizza una word all'indirizzo <b>address</b> prelevandola dal registro <b>rsource</b></i>



# Istruzione LOAD

Esempio

**Indirizzo in memoria della variabile valore  
(262160)**

**Contenuto  
(1.437.364.264)**

**00000000 00000100 00000000 00010000**

**01010101 10101100 01110000 00101000**

<b>lb rdest, address</b>	<b>lb \$t0, valore</b>	00000000 00000000 00000000 00101000
<b>lh rdest, address</b>	<b>lh \$t0, valore</b>	00000000 00000000 01110000 00101000
<b>lw rdest, address</b>	<b>lw \$t0, valore</b>	01010101 10101100 01110000 10101000
<b>la rdest, address</b>	<b>la \$t0, valore</b>	00000000 00000100 00000000 00010000



# Istruzione STORE

Esempio

Registro	Contenuto
\$t0	11010001 10000001 11010101 10101010

<b>sb rsource, address</b>	sb \$t0,variabile1
<b>sh rsource, address</b>	sh \$t0,variabile2
<b>sw rsource, address</b>	sw \$t0,variabile3

Indirizzo	Contenuto
variabile1	00000000 00000000 00000000 10101010
variabile2	00000000 00000000 11010101 10101010
variabile3	11010001 10000001 11010101 10101010



# Istruzioni LOAD-STORE

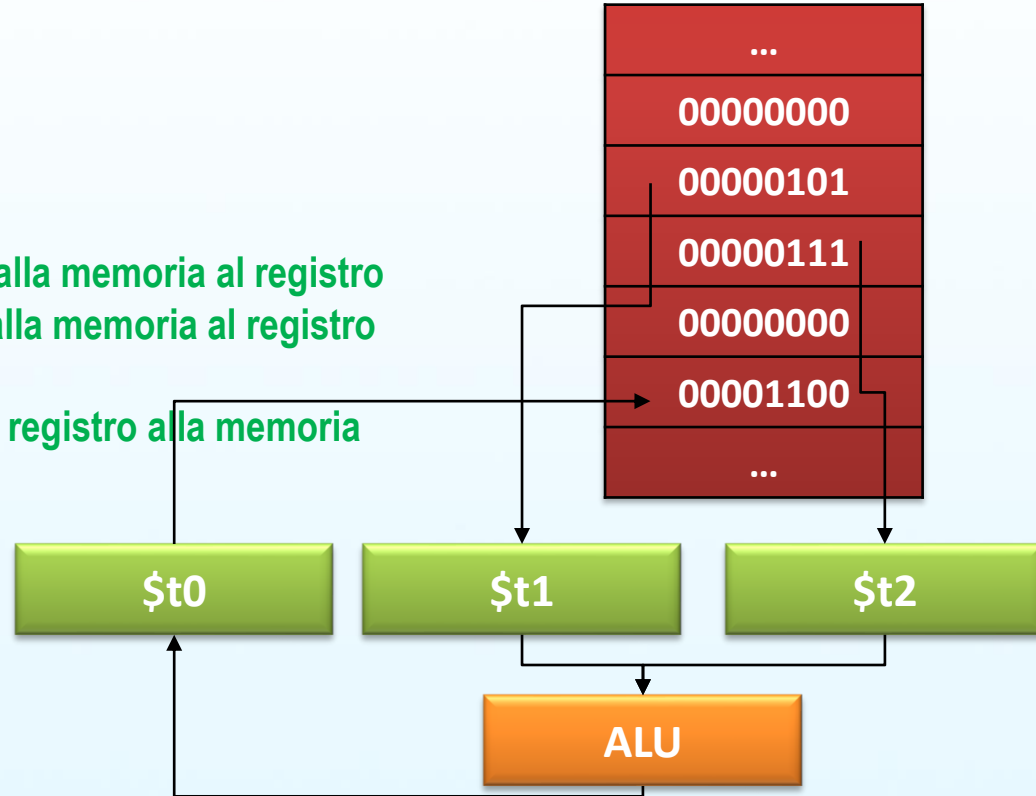
Esempio

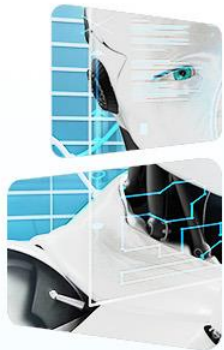
```
.text  
.globl main  
main:
```

```
    lb $t1, operandA #copia l'operando dalla memoria al registro  
    lb $t2, operandB #copia l'operando dalla memoria al registro  
    add $t0, $t1, $t2 #somma gli operandi  
    sb $t0, risultato #copia l'operando dal registro alla memoria
```

```
.end
```

```
.data  
operandA: .byte 5  
operandB: .byte 7  
risultato: .byte 0
```



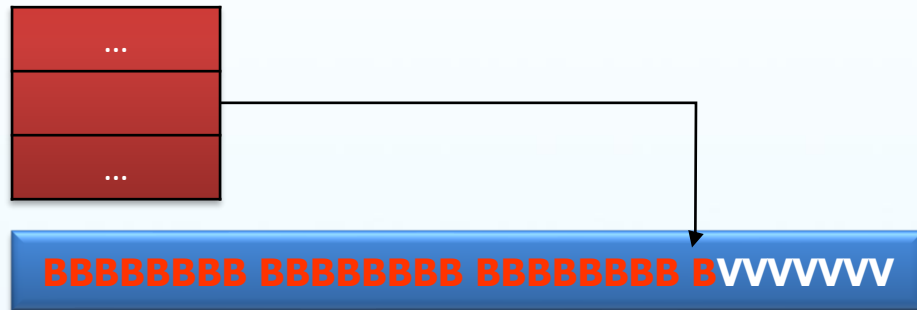


# Differenza tra SIGNED/UNSIGNED LB

## ❑ lb \$t0, address

*Copia l'operando di un byte presente nell'indirizzo address nel byte meno significativo del registro*

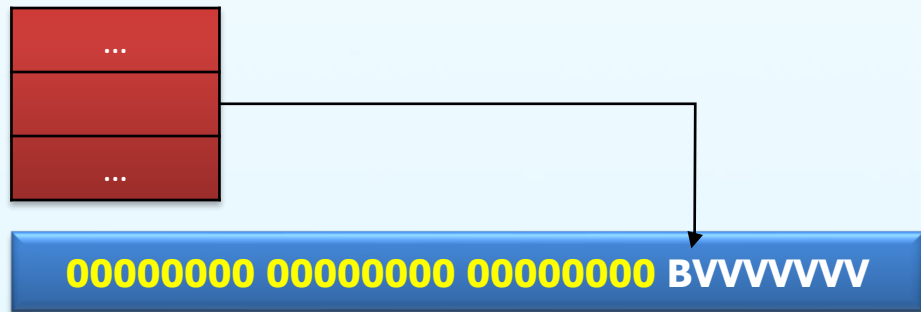
Il bit del segno viene esteso

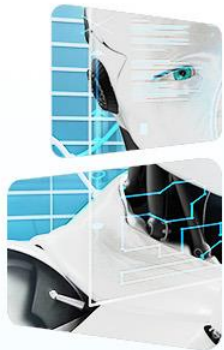


## ❑ lbu \$t0, address

*Copia l'operando di un byte presente nell'indirizzo address nel byte meno significativo del registro*

Gli altri tre byte sono posti a zero





# Differenza tra SIGNED/UNSIGNED LH

## ❑ lh \$t0, address

*Copia l'operando di due byte presenti a partire dall'indirizzo address nei due byte meno significativi del registro*

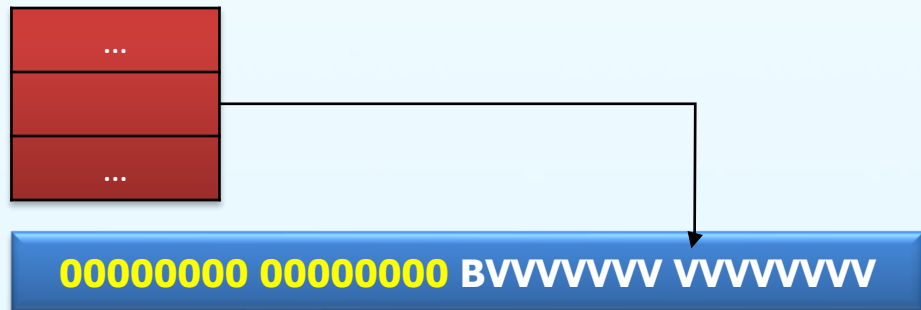
Il bit del segno viene esteso



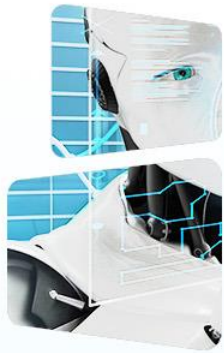
## ❑ lhu \$t0, address

*Copia l'operando di due byte presenti a partire dall'indirizzo address nei due byte meno significativi del registro*

Gli altri tre byte vengono posti a zero







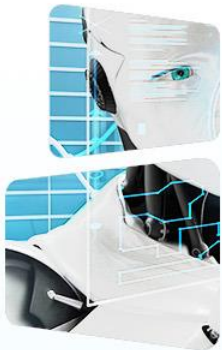
# Differenza tra SIGNED/UNSIGNED LW

## ❑ lw \$t0, address

*Copia l'operando di quattro byte presente a partire dall'indirizzo address nel registro*

Non c'è alcuna estensione del bit del segno





# Acquisizione indirizzo etichetta

- ❑ L'istruzione **lw \$t0, address** copia **il contenuto della word** indirizzata dall'etichetta address nel registro t0
- ❑ L'istruzione **la \$t0, address** acquisisce **l'indirizzo indicato** dall'etichetta address nel registro t0 (utile per la gestione di strutture dati come i vettori e le stringhe)

**.text**

**lw \$t0, val**

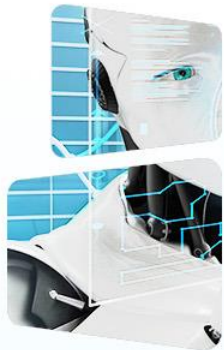
#In \$t0 si trova il dato numerico 250

**la \$t1, val**

#In \$t1 si trova l'indirizzo di memoria dove risiede il valore

**.data**

**val: .byte 250**



# Acquisizione indirizzo etichetta

## Esempio

- ❑ L'istruzione **lw \$t0, address** copia il **contenuto della word** indirizzata dall'etichetta address nel registro t0
- ❑ L'istruzione **la \$t0, address** acquisisce l'**indirizzo indicato** dall'etichetta address nel registro t0  
(utile per la gestione di strutture dati come i vettori e le stringhe)

### lw \$t0, address

0x00400030

0x00400031

0x00400032

0x00400033



\$t0=0x78563412

00101110 00111000 00100010 00001100

### la \$t0, address

0x00400030

0x00400031

0x00400032

0x00400033



\$t0=0x00400030

00000000 00101000 00000000 00011110



# Inizializzazione di un registro con un valore

(Indirizzamento immediato)

- ☐ Per copiare un operando direttamente in un registro si utilizza l'operazione **load immediate**
- ☐ Si evita di definire una valore in memoria il valore numerico è nella stessa istruzione
- ☐ Il valore numerico è anche detto **IMMEDIATE VALUE**
- ☐ Le istruzioni sono più lunghe

li <b>rdest, imm</b>	<i>Sposta il valore <b>imm</b> presente nell'istruzione nel registro <b>rdest</b></i>
lui <b>rdest, imm</b>	<i>Muove la halfword <b>imm</b> presente nell'istruzione nella parte più alta dell'halfword del registro <b>rdest</b></i>

Esempio

li \$t0, 234

#Mette nel registro \$t0 il valore 234



# Spostamento tra registri

## (indirizzamento a registro)

- ❑ La maggioranza delle istruzioni di MIPS operano tramite i registri interni al processore
- ❑ Per copiare i valori tra registri si utilizzano le operazioni di **data movement**

<b>move rdest, rsource</b>	Muove il registro <i>rsource</i> nel registro <i>rdest</i>
<b>mfhi rdest</b>	Muove il registro <i>hi</i> nel registro <i>rdest</i>
<b>mflo rdest</b>	Muove il registro <i>lo</i> nel registro <i>rdest</i>
<b>mthi rsource</b>	Muove il registro <i>rsource</i> nel registro <i>hi</i>
<b>mtlo rsource</b>	Muove il registro <i>rsource</i> nel registro <i>lo</i>

Esempio

**move \$t0, \$t1**

**#Copia il contenuto del registro \$t1 in \$t0**



# Spostamento tra registri

Esempio

Scambio dei valori contenuti in due registri

Prima: \$t0=5 e \$t1=3    Dopo: \$t0=3 e \$t1=5

```
.text  
.globl main
```

```
main:
```

li \$t0,5	#carico il valore 5 in \$t0
li \$t1,3	#carico il valore 3 in \$t1
move \$t2,\$t1	#c=y
move \$t1,\$t0	#y=x
move \$t0,\$t2	#x=c
xor \$t2,\$t2	#pulisco \$t2

```
li $v0,10  
syscall
```



# Spostamento tra locazioni di memoria

Esempio

Definiti due valori in memoria, Diabolik e Ginko, scambiare la loro posizione

```
text
.globl main
```

```
main:
```

```
lw $t0,Diabolik
lw $t1,Ginko
move $t2,$t1
move $t1,$t0
move $t0,$t2
xor $t2,$t2,$t2
sw $t0,Ginko
sw $t1,Diabolik
li $v0,10
syscall
```

```
#copia di Diabolik in $t0
#copia di Ginko in $t1
#scambio
#
#
#pulisco $t2
#copia dei scambiati
#
```

```
.data
Diabolik :.word 5
Ginko :.word 10
```





# Spostamento tra locazioni di memoria

Esempio (variante)

Definiti due valori in memoria, Diabolik e Ginko, scambiare la loro posizione

```
text  
.globl main
```

```
main:  
    lw $t0,Diabolik      #copia del valore di Diabolik in $t0  
    lw $t1,Ginko         #copia del valore di Ginko in $t1  
    sw $t0,Ginko         #scambio del valore di Diabolik nella locazione di Ginko  
    sw $t1,Diabolik      #scambio del valore di Ginko nella locazione di Diabolik  
    li $v0,10  
    syscall
```

```
.data  
Diabolik :.word 5  
Ginko :.word 10
```



# Istruzione di Spostamento

Relazione con i Condition Code

Codice	C	N	Z	W	P
LOAD	X	X	X	X	X
MOVE	X	X	X	X	X
STORE	X	X	X	X	X

# ISTRUZIONI LOGICHE-ARITMETICHE





# Istruzioni Logiche Aritmetiche

## Preludio

❑ Gli elaboratori elettronici hanno come principale motivo di esistenza la possibilità di svolgere un gran numero di operazioni in tempi rapidissimi

### *Esempio*

**add \$t2, \$t0, \$t1    #Somma il contenuto di \$t1 e \$t0 e lo mette in \$t2**  
**xor \$t2,\$t2,\$t2    #Inizializza il registro \$t2 a zero**

NB: Nel MARS le istruzioni aritmetiche operano su operandi siti nei registri.  
I registri sono usati per tre motivi principali:

- ❖ Costruiti con ottima tecnologia
- ❖ Sono interni alla CPU (non richiedono accessi in memoria)
- ❖ Consentono di trovare i dati su cui operare utilizzando pochi bit (RISC)



# Istruzioni Aritmetiche

Istruzioni: somma, sottrazione, opposto

<b>add rd, rs, rt</b>	<b><math>rd = rs + rt</math> (con overflow)</b>
<b>addu rd, rs, rt</b>	<b><math>rd = rs + rt</math> (senza overflow)</b>
<b>addi rd, rs, imm</b>	<b><math>rd = rs + imm</math> (con overflow)</b>
<b>addiu rd, rs, imm</b>	<b><math>rd = rs + imm</math> (senza overflow)</b>
<b>sub rd, rs, rt</b>	<b><math>rd = rs - rt</math> (con overflow)</b>
<b>subu rd, rs, rt</b>	<b><math>rd = rs - rt</math> (senza overflow)</b>
<b>neg rd, rs</b>	<b><math>rd = -rs</math> (con overflow)</b>
<b>negu rd, rs</b>	<b><math>rd = -rs</math> (senza overflow)</b>
<b>abs rd, rs</b>	<b><math>rd =  rs </math></b>



# Istruzioni Aritmetiche

## Istruzioni: moltiplicazione, divisione, resto

<b>mult rs, rt</b>	<i>hi,lo = rs * rt (signed, overflow impossibile)</i>
<b>multu rs, rt</b>	<i>hi,lo = rs * rt (unsigned, overflow impossibile)</i>
<b>mul rd, rs, rt</b>	<i>rd = rs * rt (senza overflow)</i>
<b>mulo rd, rs, rt</b>	<i>rd = rs * rt (con overflow)</i>
<b>mulou rd, rs, rt</b>	<i>rd = rs * rt (con overflow, unsigned))</i>
<b>div rs, rt</b>	<i>hi,lo = resto e quoz. di rs / rt (signed con overflow)</i>
<b>divu rs, rt</b>	<i>hi,lo = resto e quoz. di rs / rt (unsigned, no overflow)</i>
<b>div rd, rs, rt</b>	<i>rd = rs / rt (signed con overflow)</i>
<b>divu rd, rs, rt</b>	<i>rd = rs / rt (unsigned)</i>
<b>rem rd, rs, rt</b>	<i>rd = resto di rs / rt (signed)</i>
<b>remu rd, rs, rt</b>	<i>rd = resto di rs / rt (unsigned)</i>



# Istruzioni Logiche

Lista istruzioni: and, or, xor, not

<b>and rd, rs, rt</b>	<i><b>rd = rs AND rt</b></i>
<b>andi rd, rs, imm</b>	<i><b>rd = rs AND imm</b></i>
<b>or rd, rs, rt</b>	<i><b>rd = rs OR rt</b></i>
<b>ori rd, rs, imm</b>	<i><b>rd = rs OR imm</b></i>
<b>xor rd, rs, rt</b>	<i><b>rd = rs XOR rt</b></i>
<b>xori rd, rs, imm</b>	<i><b>rd = rs XOR imm</b></i>
<b>nor rd, rs, rt</b>	<i><b>rd = NOT (rs OR rt)</b></i>
<b>not rd, rs</b>	<i><b>rd = NOT rs</b></i>





# Istruzioni Logiche-Aritmetiche

## Shift e Rotate

- ❑ Lo **Shift** (spostamento logico-aritmetico) sposta  $n$  bit verso destra/sinistra e perde i bit dalla parte della movimentazione; mentre è colmato con 0 nella direzione opposta
- ❑ Il **Rotate** (rotazione o spostamento logico) sposta  $n$  bit verso destra/sinistra e mantiene i bit dalla parte della movimentazione riproponendo i bit spostati nella direzione opposta

<b>sll rd, rs, rt</b>	<i>rd = rs è shiftato verso sinistra di rt mod 32 bits</i>
<b>srl rd, rs, rt</b>	<i>rd = rs s è shiftato verso destra di rt mod 32 bits</i>
<b>rol rd, rs, rt</b>	<i>rd = rs è ruotato verso sinistra di rt mod 32 bits</i>
<b>ror rd, rs, rt</b>	<i>rd = rs è ruotato verso destra di rt mod 32 bits</i>

\$t2      **101**000000000000000000000000000010011      2684354579



# Istruzioni Logiche-Aritmetiche

## Immediate e Unsigned

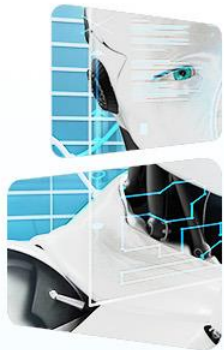
### Versioni immediate (i)

- ☐ Le versioni immediate (i) delle istruzioni precedenti utilizzano un valore immediato al posto di uno degli operandi
- ☐ Non sempre è necessario specificare la **i**; c'è un riconoscimento implicito da parte dell'assemblatore

Esempio: **add \$t0, \$t0, 1**

### Versioni unsigned (u)

- ☐ Le versioni unsigned delle istruzioni non gestiscono le problematiche relative all'overflow
- ☐ Dettagli in seguito quando si parlerà delle eccezioni



# Istruzioni Logiche-Aritmetiche

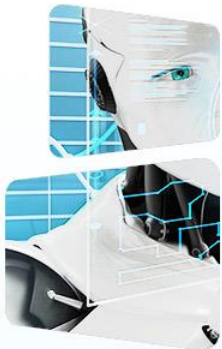
## Esempio I

Realizzare un programma che in linguaggio SPIM prenda un valore residente in memoria (di tipo byte) e metta 0 o 1 se il numero è rispettivamente pari o dispari

```
.text
.globl main
main:
    lb $t0,pippo
    li $t1,2
    rem $t2,$t0,$t1
    sb $t2,risc

li $v0,10
syscall
```

```
.data
pippo: .byte 100
risc: .byte 0
```



# Istruzioni Logiche-Aritmetiche

## Esempio II

Si scriva un programma in linguaggio assembly che definito un valore intero in memoria riporti in \$t0 l'intero precedente, in \$t1 il numero corrente e in \$t2 il successivo

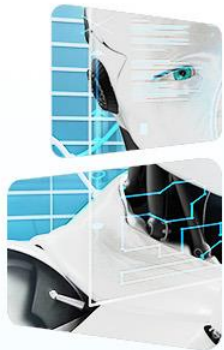
```
.text  
.globl main
```

```
main:
```

```
    lw $t1,pippo          #carico pippo  
    sub $t0,$t1,1         #pre=x-1  
    add $t2,$t1,1         #suc=x+1
```

```
li $v0,10  
syscall
```

```
.data  
pippo: .word 5
```



# Istruzioni Logiche-Aritmetiche

## Esempio III

Si scriva un programma in linguaggio assembly che definiti due numeri in memoria riporta la loro media (fra interi) in \$t2  
Esempio: Batman=5 Robin=8 **media**=(5+8)/2=6

```
.text  
.globl main
```

```
main:
```

```
    lw $t0,Batman    #carico Batman  
    lw $t1,Robin     #carico Robin  
    add $t2,$t0,$t1   #m=Bat+Rob  
    div $t2,$t2,2     #media=m/2
```

```
li $v0,10  
syscall
```

```
.data  
Batman: .word 5  
Robin: .word 6
```



# Istruzioni Logiche-Aritmetiche

## Esempio IV

Scrivere un programma che definito in memoria il prezzo al dettaglio e lo sconto e riporta in \$t2 il prezzo scontato (arrotondato all'intero)

```
.text  
.globl main
```

```
main:
```

```
lw $t0,prezzo  
lw $t1,sconto  
mul $t2,$t0,$t1  
div $t2,$t2,100  
li $v0,10  
syscall
```

```
#carico prezzo  
#carico sconto  
#prezzo=prezzo*25  
#sconto=prezzo/100
```

```
.data  
prezzo : .word 133  
sconto: .word 25
```





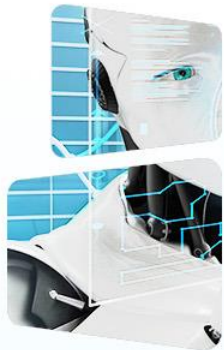
# Istruzioni Logiche-Aritmetiche

## Relazione con i Condition Code

Codice	C	N	Z	W	P
ADD	1/0	1/0	1/0	1/0	1/0
CMP	1/0	1/0	1/0	1/0	1/0
NEG	1/0	1/0	1/0	1/0	1/0
SUB	1/0	1/0	1/0	1/0	1/0
AND	0	1/0	1/0	0	1/0
OR	0	1/0	1/0	0	1/0
XOR	0	1/0	1/0	0	1/0
NOT	0	1/0	1/0	0	1/0
SL	1/0	1/0	1/0	1/0	1/0
SR	1/0	1/0	1/0	1/0	1/0
ROL	1/0	0	0	0	1/0
ROR	1/0	0	0	0	1/0

# ISTRUZIONI DI CONFRONTO





# Istruzioni di impostazione e confronto

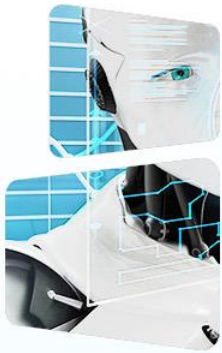
Set

❑ Le istruzioni di confronto sono di solito usate dall'assemblatore e sono usate per creare pseudo-istruzioni utili ai programmatori per i salti condizionati

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

# ISTRUZIONI DI SALTO CONDIZIONATO



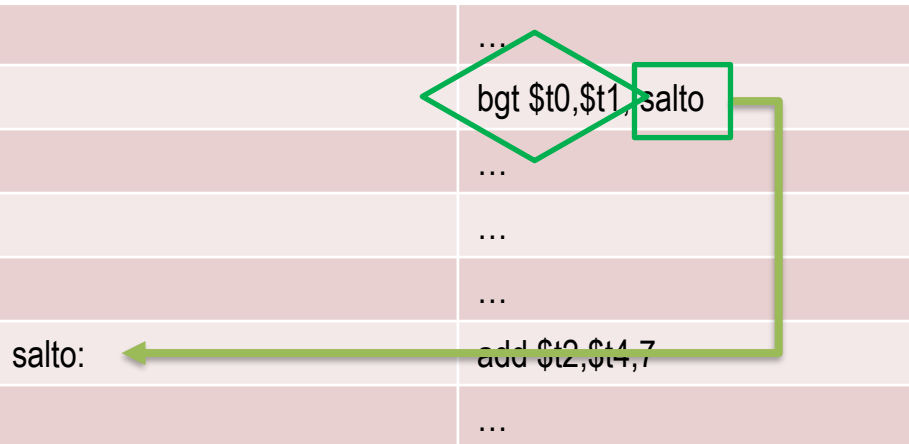


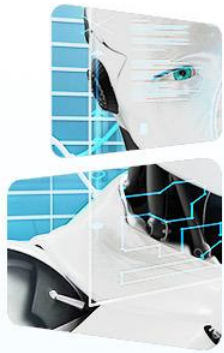
# Istruzioni di salto condizionato

## Introduzione

❑ Le **istruzioni di salto condizionato** considerano uno/due operandi e in relazione al verificarsi di una condizione di verità derivata da un confronto effettuano un salto ad un'altra locazione del programma

### MEMORIA ISTRUZIONI

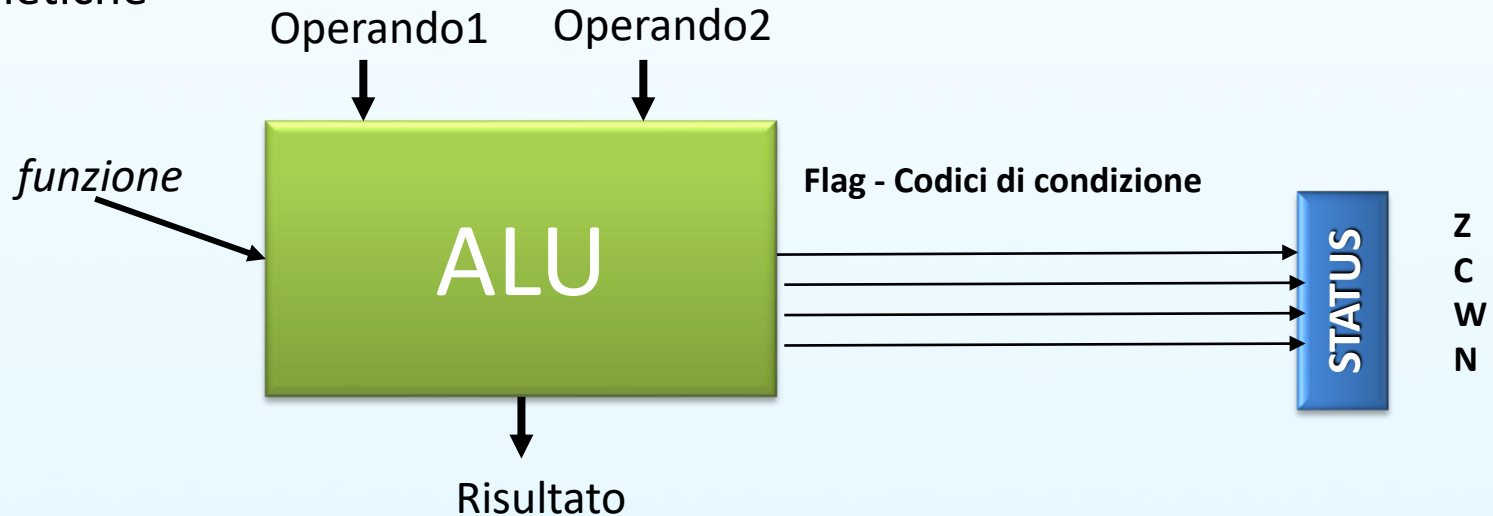


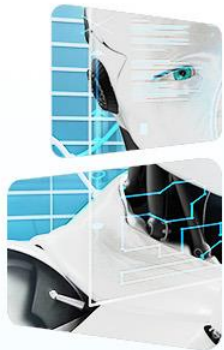


# Istruzioni di salto condizionato

Correlazione con ALU

❑ Le istruzioni di salto condizionato fanno riferimento ai *flag* o codici di condizione presenti nello STATUS REGISTER derivanti da operazione logico-aritmetiche





# Istruzioni di salto condizionato

## Andamento Program Counter

- ❑ Quando si verifica una condizione c'è un **salto** all'indirizzo interno al programma
- ❑ Tecnicamente il salto è una sovrascrittura del Program Counter con l'indirizzo mascherato dall'etichetta

100	li \$t0,56	PC: 101
101	li \$t1,122	PC: 102
102	bgt \$t1,\$t0,salto	PC: 104
103	mul \$t9,\$t0,\$t1	PC: 105
104	salto: li \$t9,0	



# Istruzioni di salto condizionato

Set

<b>beq</b> rs, rt, target	Salta all'istruzione puntata da <b>target</b> se <b>rs</b> = <b>rt</b>
<b>bne</b> rs, rt, target	Salta all'istruzione puntata da <b>target</b> se <b>rs</b> != <b>rt</b>
<b>bge</b> rs, rt, target <b>bgeu</b> rs, rt, target	Salta all'istruzione puntata da <b>target</b> se <b>rs</b> >= <b>rt</b> Comparazione senza considerare il segno
<b>bgt</b> rs, rt, target <b>bgtu</b> rs, rt, target	Salta all'istruzione puntata da <b>target</b> se <b>rs</b> > <b>rt</b> Comparazione senza considerare il segno
<b>ble</b> rs, rt, target <b>bleu</b> rs, rt, target	Salta all'istruzione puntata da <b>target</b> se <b>rs</b> <= <b>rt</b> Comparazione senza considerare il segno
<b>blt</b> rs, rt, target <b>bltu</b> rs, rt, target	Salta all'istruzione puntata da <b>target</b> se <b>rs</b> < <b>rt</b> Comparazione senza considerare il segno

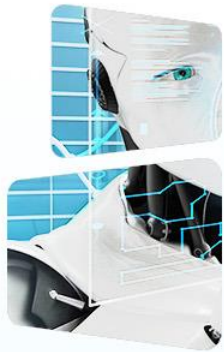




# Istruzioni di salto condizionato

Set con riferimento al registro \$ZERO

<b>bgez rs, target</b>	<i>Salta all'istruzione puntata da <b>target</b> se <b>rs</b> <math>\geq 0</math></i>
<b>bgtz rs, target</b>	<i>Salta all'istruzione puntata da <b>target</b> se <b>rs</b> <math>&gt; 0</math></i>
<b>blez rs, target</b>	<i>Salta all'istruzione puntata da <b>target</b> se <b>rs</b> <math>\leq 0</math></i>
<b>bltz rs, target</b>	<i>Salta all'istruzione puntata da <b>target</b> se <b>rs</b> <math>&lt; 0</math></i>
<b>beqz rs, target</b>	<i>Salta all'istruzione puntata da <b>target</b> se <b>rs</b> <math>= 0</math></i>
<b>bnez rs, target</b>	<i>Salta all'istruzione puntata da <b>target</b> se <b>rs</b> <math>\neq 0</math></i>



# Istruzioni di salto condizionato

## Analisi dei condition code

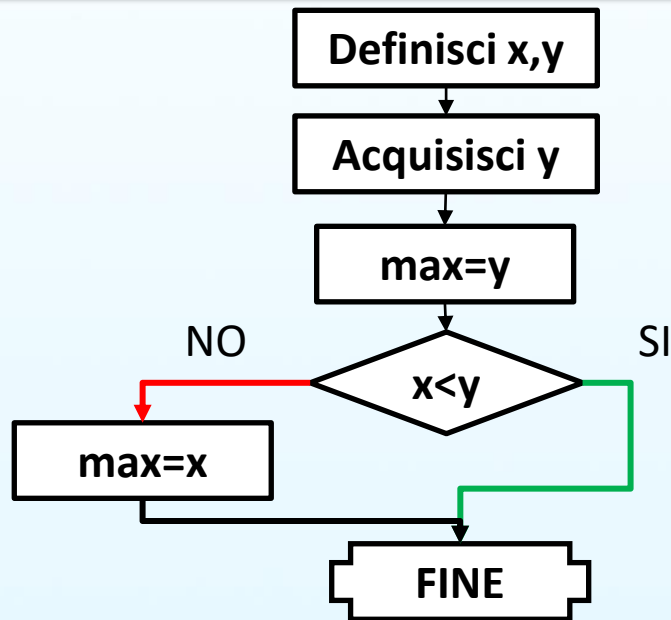
Mnemonico	Significato	Flag interessato
EQ	=	Z=1
NE	!=	Z=0
CS	> (unsigned)	C=1
	< (unsigned)	C=0
HI	>	C=1 e Z=0
LS	<	C=0 e Z=1
MI	negativo	N=1
PL	positivo	N=0



# Istruzioni di salto condizionato

## Esempio I

Realizzare un programma che valuta il massimo fra due numeri  $x, y$  (di tipo byte) definiti in memoria. Riportare il risultato in memoria

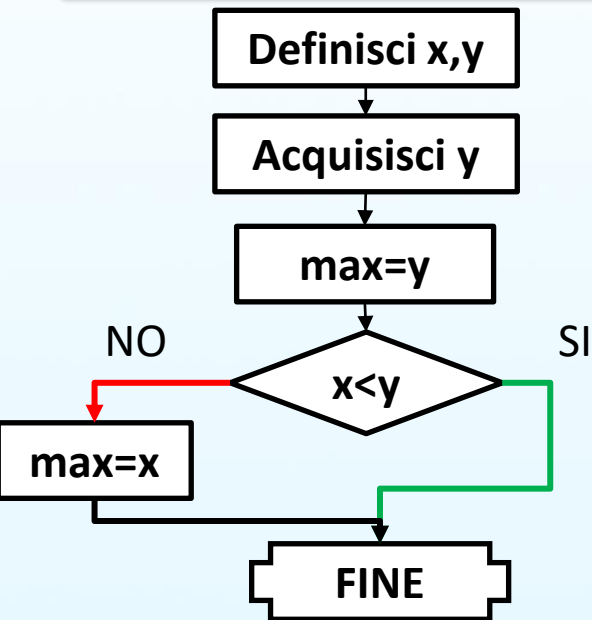




# Istruzioni di salto condizionato

## Esempio I

Realizzare un programma che valuta il massimo fra due numeri x,y (di tipo byte) definiti in memoria. Riportare il risultato in memoria



.text

```
lb $t0,x
lb $t1,y
move $t2,$t1
blt $t0,$t1,max_y
move $t2,$t0
```

max\_y:

```
sb $t2, max
```

.data

```
x: .byte 34
y: .byte 76
max: .byte 0
```

#prelievo di x

#prelievo di y

#In t2 conservo y come se fosse il massimo

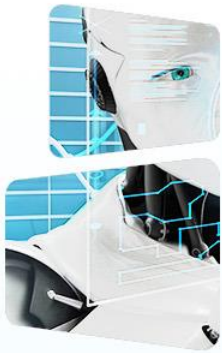
#salto perché y è il massimo

#aggiorno il valore massimo mettendo x in \$t2

#conservo il massimo in memoria

# ISTRUZIONI DI SALTO INCONDIZIONATO





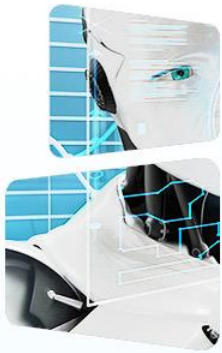
# Istruzioni di salto incondizionato

## Introduzione

- ❑ Le istruzioni di salto incondizionato effettuano un salto ad un'altra locazione del programma senza valutare la veridicità di alcuna condizione

### MEMORIA ISTRUZIONI

	...
	j salto
	...
	...
	...
salto: ←	add \$t2,\$t1,7
	...

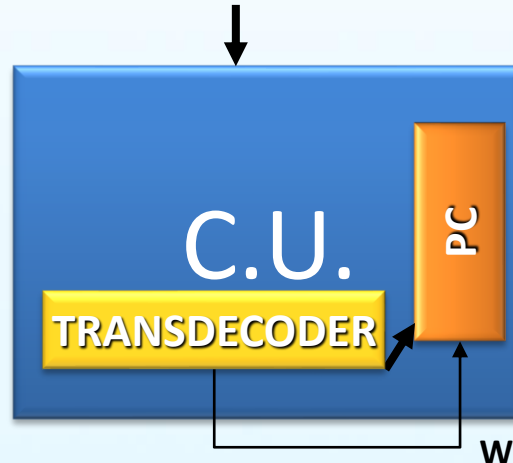


# Istruzioni di salto incondizionato

Relazione con C.U.

- ❑ Le istruzioni di **salto incondizionato** intervengono direttamente mutando il valore del PC

Istruzione di salto incondizionato





# Istruzioni di salto incondizionato

Sintassi MIPS/MARS

- ☐ Le istruzioni di **salto incondizionato** sono utili per realizzare cicli e strutture iterative

<b>j target</b>	<i>Salto incondizionato all'istruzione puntata da <b>target</b></i>
-----------------	---

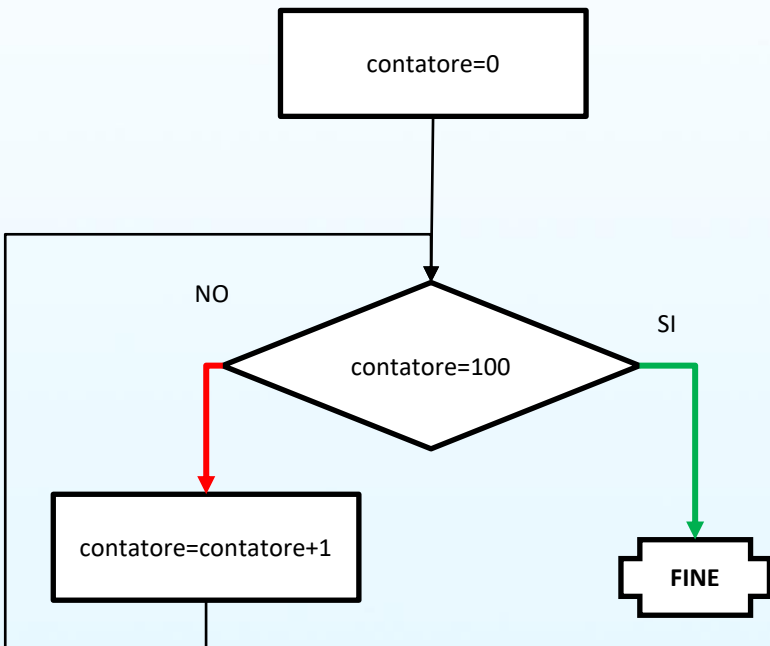




# Istruzione di salto incondizionato

## Esempio I

Realizzare un contatore da 0 a 100



```
.text  
.globl main  
main:
```

```
li $t0,0  
li $t1,100
```

**ciclo:**

```
beq $t0,$t1,fine  
add $t0,$t0,1  
j ciclo
```

**fine:**

```
li $v0,10  
syscall
```

#inizializzazione contatore  
#inizial. condizione di fine

#confronto se arrivo alla fine  
#incremento contatore  
#salto incondizionato fino...  
#...alla condizione di fine



# Istruzioni di salto

## Esempio II

Realizzare un programma che in linguaggio SPIM definisca un valore di tipo byte residente in memoria e metta in \$t2 il valore 1 o 2 se il numero è rispettivamente multiplo di 7 o no

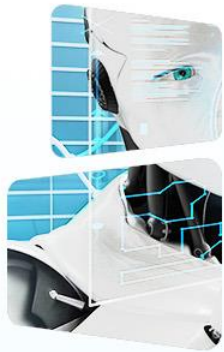
```
.text
.globl main
main:
    lb $t0,valore
    li $t1,7
    rem $t3,$t0,$t1
    li $t2,1
    beqz $t3,salta
    li $t2,2
```

```
salta:
    li $v0,10
    syscall
```

```
.data
valore: .byte 100
risb: .byte 0
```

# PROGETTAZIONE DI UN ESEGUIBILE





# Progettazione Applicativo Informatico

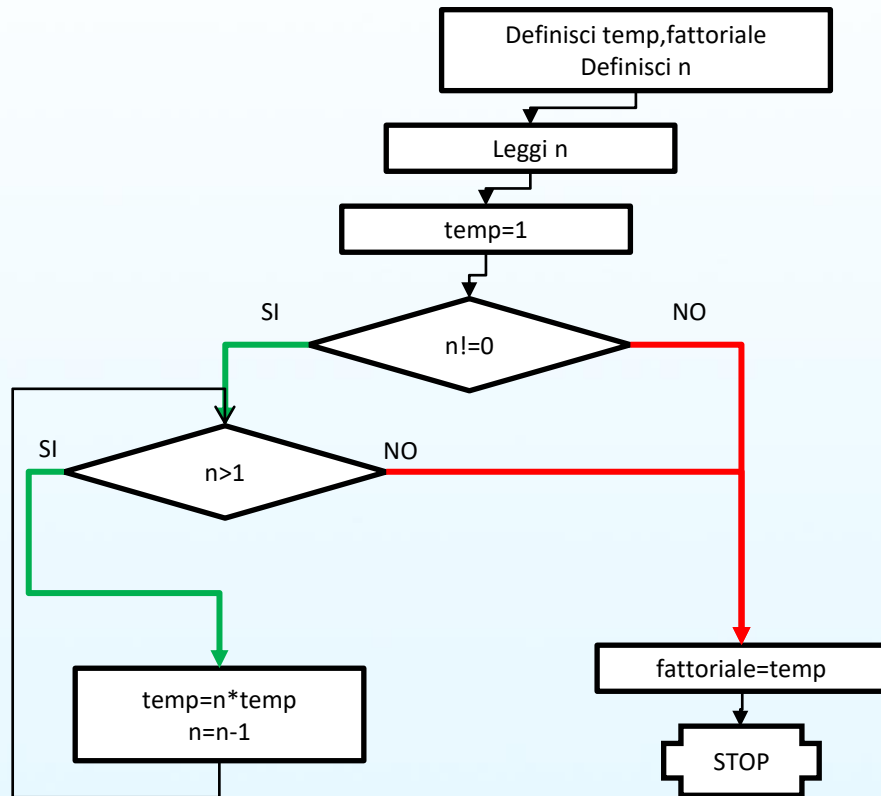
## Implementazione diagramma di flusso

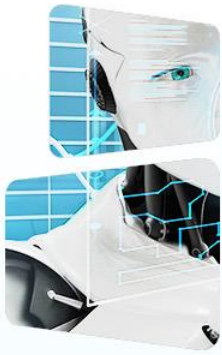
Effettuare il **fattoriale** di un numero  $n$  ( $n \geq 0$  e  $n \in \mathbb{N}$ )

*Il fattoriale di un numero  $N$  è definito come:*

$$N! = N \cdot N-1 \cdot N-2 \cdot \dots \cdot 3 \cdot 2 \cdot 1$$

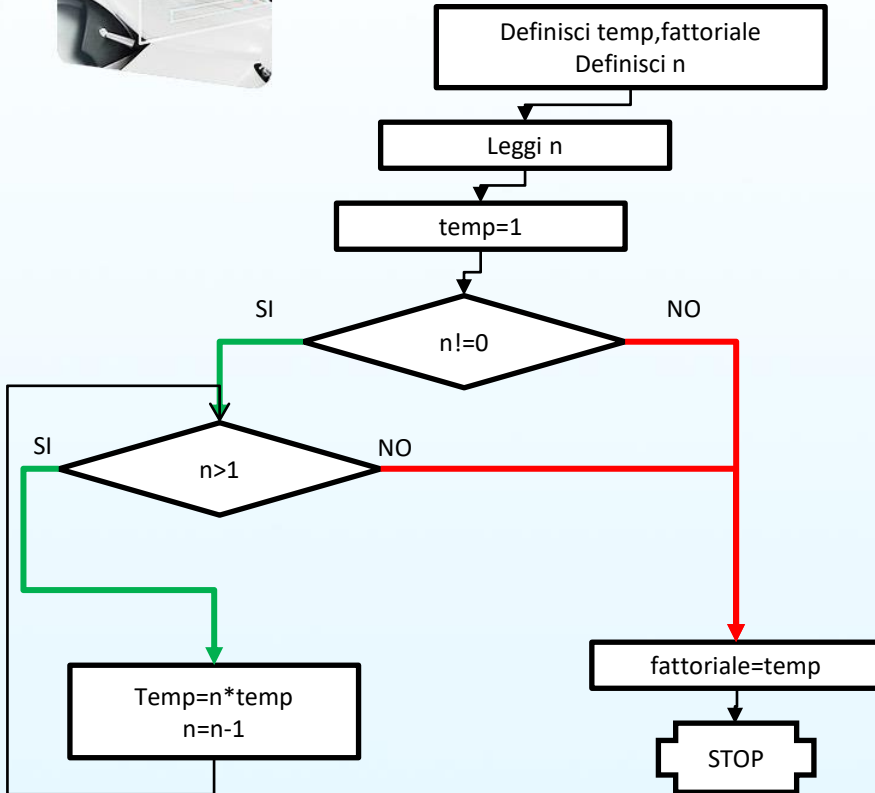
NB:  $0! = 1$



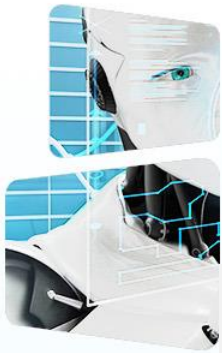


# Progettazione Applicativo Informatico

Scrittura linguaggio alto livello



```
main()
{
  int n,temp, fattoriale;
  temp=1;
  if (n!=0){
    while (n>1)
    {
      temp=temp*n;
      n=n-1;
    }
    fattoriale=temp;
  }
}
```



# Progettazione Applicativo Informatico

## Scrittura linguaggio assembly

```
main()
{
  int n,temp, fattoriale;
  temp=1;
  if (n!=0){
    while (n>1)
    {
      temp=temp*n;
      n=n-1;
    }
  }
  fattoriale=temp;
}
```

```
.text
    lw $t0,n
    lw $t1,temp
    li $t5,1
    beqz $t0, fine

ciclo:
    ble $t0,$t5, fine
    mul $t1,$t1,$t0
    sub $t0,$t0,1
    j ciclo

fine:
    sw $t1, fattoriale

.data
fattoriale: .word 0
temp: .word 1
n: .word 35
```



# Progettazione Applicativo Informatico

## Assemblatore

**.text**

lw \$t0,n  
lw \$t1,temp  
li \$t5,1  
beqz \$t0, fine

**ciclo:**

ble \$t0,\$t5, fine  
mul \$t1,\$t1,\$t0  
sub \$t0,\$t0,1  
j ciclo

**fine:**

sw \$t1, fattoriale

**.data**

fattoriale: .word 0  
temp: .word 1  
n: .word 35

Indirizzo	Etichetta	Istruzione
400		lw \$t0,n
404		lw \$t1,temp
408		li \$t5,1
412		beqz \$t0,fine
416	ciclo	ble \$t0,\$t5,fine
420		mul \$t1,\$t1,\$t0
424		sub \$t0, \$t0,1
428		j ciclo
432	fine:	sw \$t1,fattoriale
10000	fattoriale	0
10004	temp	1
10008	n	35



# Progettazione Applicativo Informatico

## Assemblatore: risoluzione etichette – tabella simboli

Indirizzo	Etichetta	Istruzione
400		lw \$t0,n
404		lw \$t1,temp
408		li \$t5,1
412		beqz \$t0,fine
416	ciclo:	ble \$t0,\$t5,fine
420		mul \$t1,\$t1,\$t0
424		sub \$t0, \$t0,1
428		j ciclo
432	fine:	sw \$t1,fattoriale
10000	fattoriale	0
10004	temp	1
10008	n	35

Tempo	TABELLA DEI SIMBOLI	
	ETICHETTA	VALORE
T1	n	?
T2	temp	?
T4	fine	?
T5	ciclo	416
T9	fine	432
T9	fattoriale	?
T10	fattoriale	10000
T11	temp	10004
T12	n	10008

Indirizzo	Etichetta	Istruzione
400		lw \$t0,n
404		lw \$t1,temp
408		li \$t5,1
412		beqz \$t0,fine
416	ciclo:	ble \$t0,\$t5,fine
420		mul \$t1,\$t1,\$t0
424		sub \$t0, \$t0,1
428		j 416
432	fine:	sw \$t1,fattoriale
10000	fattoriale	0
10004	temp	1
10008	n	35





# Progettazione Applicativo Informatico

## Assemblatore: risoluzione etichette – tabella simboli

Indirizzo	Etichetta	Istruzione
400		lw \$t0,n
404		lw \$t1,temp
408		li \$t5,1
412		beqz \$t0,fine
416	ciclo:	ble \$t0,\$t5,fine
420		mul \$t1,\$t1,\$t0
424		sub \$t0, \$t0,1
428		j 416
432	fine:	sw \$t1,fattoriale
10000	fattoriale	0
10004	temp	1
10008	n	35

Tempo	TABELLA DEI SIMBOLI	
	ETICHETTA	VALORE
T1	n	10008
T2	temp	10004
T4	fine	432
T5	ciclo	416
T5	fine	432
T9	fattoriale	10000

Indirizzo	Etichetta	Istruzione
400		lw \$t0,10008
404		lw \$t1,10004
408		li \$t5,1
412		beqz \$t0,432
416	ciclo:	ble \$t0,\$t5,432
420		mul \$t1,\$t1,\$t0
424		sub \$t0, \$t0,1
428		j 416
432	fine:	sw \$t1,10000
10000	fattoriale	0
10004	temp	1
10008	n	35

# ESERCIZIO RIEPILOGATIVO





# Esercizio

## Individuazione numero primo

**Implementare un programma che letto un intero maggiore di 1 da tastiera stampa su videoterminale un messaggio che indica se il numero è primo (\$t2=1) o no (\$t2=2)**

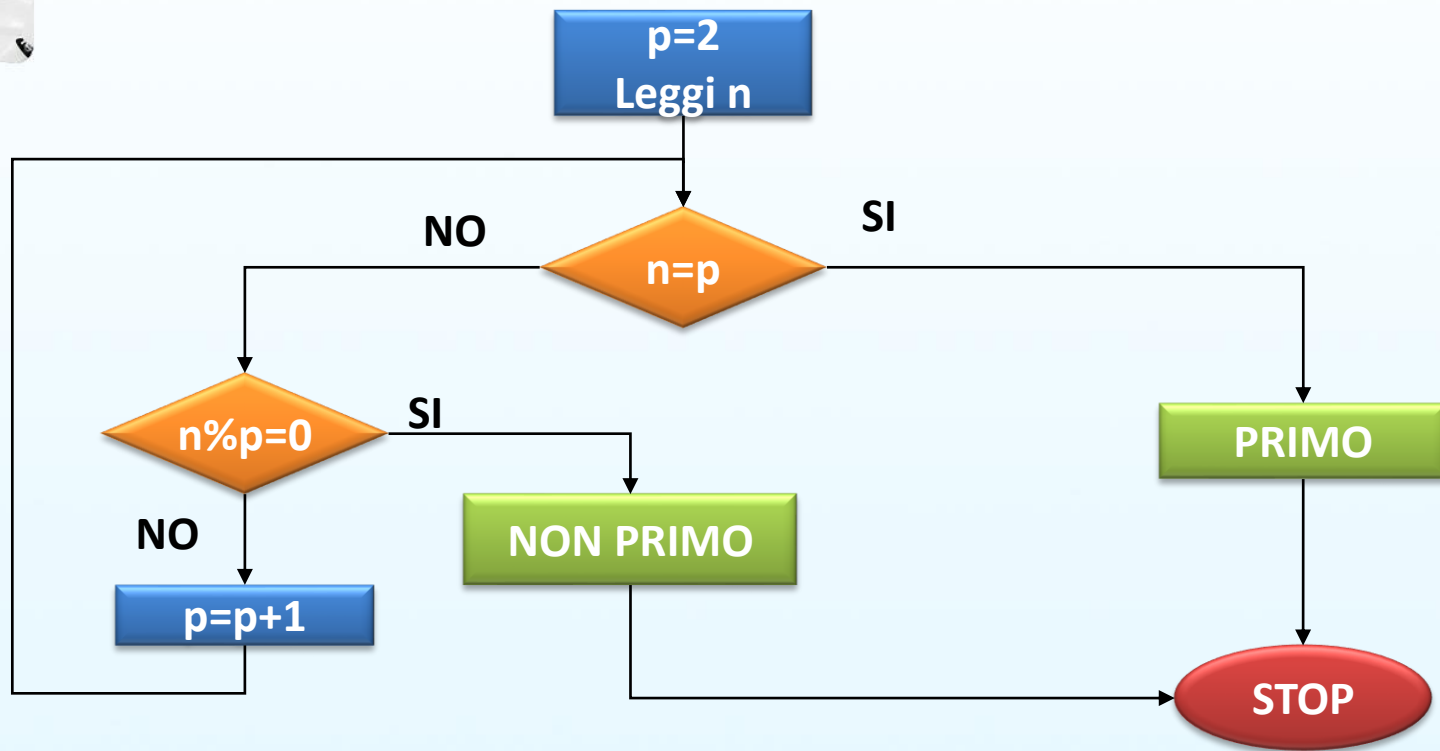
*NB: un numero è primo se, oltre ad essere divisibile per 1, è divisibile solamente per se stesso ovvero:*

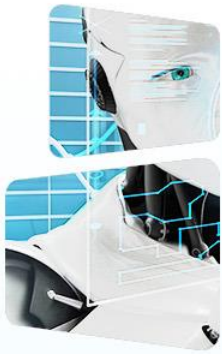
$(n,x)=1 \ \forall \ x \in \mathbb{N}-\{0\} \text{ con } x=1 \wedge x=n \text{ cioè } (n,1)=1 \wedge (n,n)=1$



# Esercizio

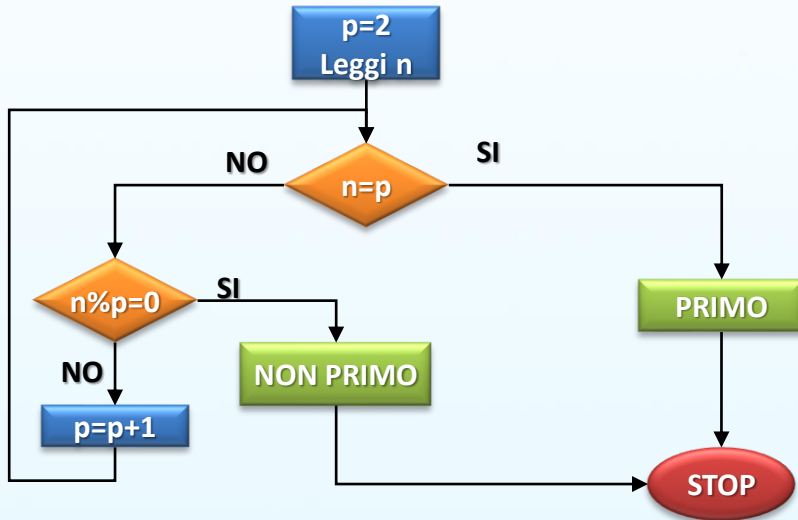
## Individuazione numero primo





# Esercizio

## Individuazione numero primo



```
.text  
.globl main  
main:
```

```
lw $t0,n  
lw $t1,p  
li $t2,1
```

```
inizio:
```

```
beq $t0,$t1, primo      # quis  
                        #sixjx
```

```
rem $t3,$t0,$t1  
beqz $t3,non_primo  
add $t1,$t1,1  
j inizio
```

```
non_primo:
```

```
li $t2,2
```

```
primo:
```

```
li $v0,10  
syscall
```

FINE

