

# ARCHITETTURA DEL SET DI ISTRUZIONI

**Cosa differenzia un Computer  
da una Calcolatrice**



Michele Favalli

# Il computer è una calcolatrice?

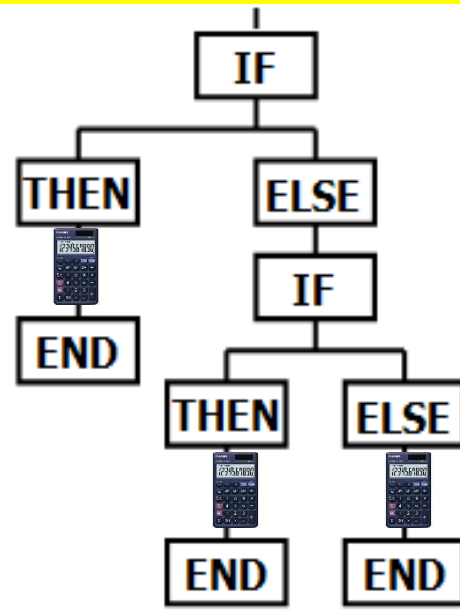


## Che differenza fondamentale c'è?

Il computer è in grado di prendere decisioni su quali istruzioni eseguire ad un certo punto nel tempo, in funzione degli ingressi e dello stato



**Non a caso in un linguaggio di programmazione di alto livello ci sono gli statement condizionali**



*Data-flow vs control flow*

# Istruzioni per Prendere Decisioni

MIPS ha due istruzioni per il SALTO CONDIZIONALE

## ▪ Branch-if-equal

**beq** register1, register2, L1

Salta al codice con  
«label» L1 se i  
contenuti di  
*register1* e  
*register2*  
coincidono



*Codice macchina*

10101000110

10100111111

10000010101

**beq** register1, register2, L1

10010101001

10000001010

L1: 00000010101

00010101010

## ▪ Branch-if-NOT-equal

**bne** register1, register2, L1



Grazie all'utilizzo di LABELS, né il programmatore né il compilatore devono calcolarsi gli indirizzi dei branch. Ci pensa la microarchitettura!

# Esempio

## Statement in C

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

***bne** risulta più  
efficiente in sede di  
codifica*

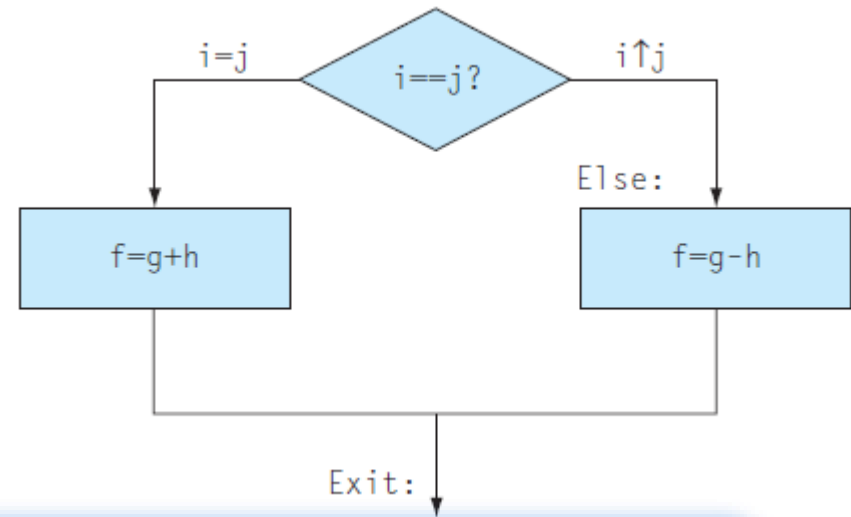
***Salto  
incondizionato***

*In assembler  
compaiono branch e  
labels non visibili dal C  
=> Programmare in C  
è più veloce!*

## Codice Assembler

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

## Flow Chart dello statement



# Istruzione di branch

- Le istruzioni **bne** e la **j** sono istruzioni di trasferimento di controllo (condizionato e incondizionato)
- Per la macchina di Von Neumann questo significa che l'istruzione successiva a queste può non essere quella che si trova a PC+1 (PC+4 se usiamo l'indirizzamento a byte)
- Le label sono notazioni simboliche che aiutano il programmatore a non doversi calcolare gli indirizzi relativi del suo codice



# Nota Bene

- Nonostante il linguaggio assembler costringa il programmatore a pensare come la macchina, gli risparmia di:
  - calcolarsi gli indirizzi relativi di memoria dei salti condizionati o incondizionati (perchè relativi?)
  - calcolarsi gli indirizzi di memoria delle load e delle store

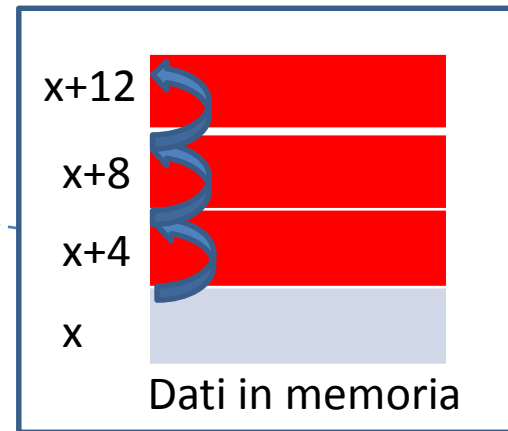
# Iterare una elaborazione

Un altro tipo di **decisione da prendere** riguarda l'opportunità o meno di **continuare ad iterare una elaborazione (loop statements)**

```
while (save[i] == k)
    i += 1;
```

Si noti ora l'«esplosione» del codice assembler:

```
# $s3 initial value of i
# $s6 base address of save
Loop: sll $t1,$s3,2    # $t1=4*i
      add $t1,$t1,$s6  # $t1 addr. of save[i]
      lw  $t0,0($t1)   # $t0 save[i]
      bne $t0,$s5,Exit # go to Exit if save[i]!=k
      addi $s3,$s3,1   # i=i+1
      j  Loop          # go to loop
Exit:
```



*Incremento  $i$  di 4 byte alla volta mediante una operazione di shift left di 2 posizioni ogni volta*

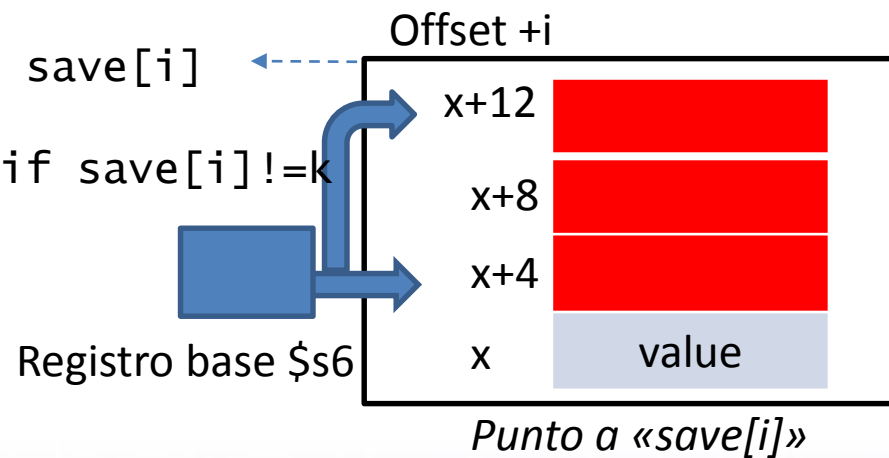
# Iterare una elaborazione

Un altro tipo di **decisione da prendere** riguarda l'opportunità o meno di **continuare ad iterare una elaborazione (loop statements)**

```
while (save[i] == k)
    i += 1;
```

Si noti ora l'«esplosione» del codice assembler:

```
# $s3 initial value of i
# $s6 base address of save
Loop: sll $t1,$s3,2    # $t1=4*i
      add $t1,$t1,$s6 # $t1 addr. of save[i]
      lw  $t0,0($t1)  # $t0 save[i]
      bne $t0,$s5,Exit # go to Exit if save[i]!=k
      addi $s3,$s3,1  # i=i+1
      j   Loop        # go to loop
Exit:
```





# Iterare una elaborazione

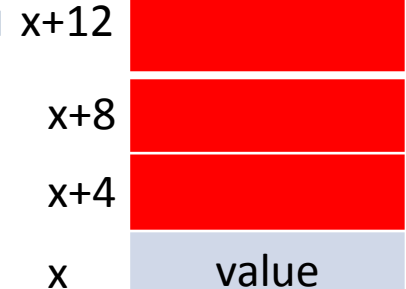
Un altro tipo di **decisione da prendere** riguarda l'opportunità o meno di **continuare ad iterare una elaborazione (loop statements)**

```
while (save[i] == k)
    i += 1;
```

Si noti ora l'«esplosione» del codice assembler:

```
# $s3 initial value of i
# $s6 base address of save
Loop: sll $t1,$s3,2    # $t1=4*i
      add $t1,$t1,$s6 # $t1 addr. of save[i]
      lw  $t0,0($t1)  # $t0 save[i]
      bne $t0,$s5,Exit # go to Exit if save[i]!=k
      addi $s3,$s3,1  # i=i+1
      j   Loop        # go to loop
Exit:
```

*Leggo «save[i]»*



# Iterare una elaborazione


Un altro tipo di **decisione da prendere** riguarda l'opportunità o meno di **continuare ad iterare una elaborazione (loop statements)**

```
while (save[i] == k)
    i += 1;
```

**Si noti ora l'«esplosione» del codice assembler:**

```
# $s3 initial value of i
# $s6 base address of save
Loop: sll $t1,$s3,2    # $t1=4*i
      add $t1,$t1,$s6 # $t1 addr. of save[i]
      lw  $t0,0($t1)  # $t0 save[i]
      bne $t0,$s5,Exit # go to Exit if save[i]!=k
      addi $s3,$s3,1  # i=i+1
      j  Loop         # go to loop
```

Exit:

 *Salto condizionale*

# Iterare una elaborazione


Un altro tipo di **decisione da prendere** riguarda l'opportunità o meno di **continuare ad iterare una elaborazione (loop statements)**

```
while (save[i] == k)
    i += 1;
```

**Si noti ora l'«esplosione» del codice assembler:**

```
# $s3 initial value of i
# $s6 base address of save
Loop: sll $t1,$s3,2    # $t1=4*i
      add $t1,$t1,$s6 # $t1 addr. of save[i]
      lw  $t0,0($t1)  # $t0 save[i]
      bne $t0,$s5,Exit # go to Exit if save[i]!=k
      addi $s3,$s3,1  # i=i+1
      j  Loop         # go to loop
```

Exit:



*Incremento indice  
+  
Salto  
incondizionato*

# Altri Tipi di Test

- Altri tipi di test sono possibili, in aggiunta a quello di «inequality».
- **set-on-less-than** `slt $t0, $s3, $s4`
  - ✓ Se il contenuto di \$s3 è minore di quello in \$s4, allora il registro \$t0 viene settato ad 1, altrimenti a 0.
  - ✓ Disponibile anche la versione *immediate*:  
`slti $t0, $s2, 10` # \$t0=1 se \$s2<10



Non esiste una versione di «**branch-if-less-than**» perché troppo complicata da realizzare in hardware. Meglio spezzare l'operazione in più istruzioni (es., combinando *slt* con *bne*), ma preservare la velocità di funzionamento.

Tutti i tipi di test (compresi «less-than-or-equal», «greater-than», «greater-than-or-equal») vengono realizzati mediante combinazione dei test-base «slt», «slti», «beq», «bne»

# Cosa fa la ALU per eseguire le istruzioni di salto condizionato ?

- Lo vedremo meglio parlando di microarchitetture
- Vi anticipo che nel caso della `slt`, la ALU esegue una differenza e poi il bit di flag di segno viene messo nel risultato
- Problema: il risultato è un registro a 32 bit, il bit di flag è uno solo



# Pseudo-istruzione «bgt»

```
bgt $t0, $t1, Maggiore    # Salta a «Maggiore» se  
.....                  # $t0 > $t1
```

Maggiore:



```
slt $t2, $t1, $t0    # $t2 vale 1 se $t0 > $t1  
bne $t2, $zero, Maggiore # Salta a Maggiore se  
.....              # $t0 > $t1
```

Maggiore:

# Pseudo-istruzione «bge»

**bge** \$t0, \$t1, Maggiore # Salta a «Maggiore» se  
..... # **\$t0 >= \$t1**

Maggiore:



slt \$t2, \$t0, \$t1 # \$t2 vale 1 se \$t0 < \$t1  
beq \$t2, \$zero, Maggiore # Salta a Maggiore se  
..... # \$t0 >= \$t1

Maggiore:

# Pseudo-istruzione «blt»

blt \$t0, \$t1, Minore # Salta a «Minore» se  
..... # \$t0 < \$t1

Minore:



slt \$t2, \$t0, \$t1 # \$t2 vale 1 se \$t0 < \$t1  
bne \$t2, \$zero, Minore # Salta a Minore se  
..... # \$t0 < \$t1

Minore:

# Pseudo-istruzione «ble»

**ble** \$t0, \$t1, Minore # Salta a «Minore» se

..... # **\$t0 <= \$t1**

Minore:



slt \$t2, \$t1, \$t0 # \$t2 vale 1 se \$t1 < \$t0

beq \$t2, \$zero, Minore # Salta a Minore se

..... # \$t0 <= \$t1

Minore:

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Three operands; data in registers
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Three operands; data in registers
Data transfer	load word	lw \$s1,100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$	Data from memory to register
	store word	sw \$s1,100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$	Data from register to memory
Logical	and	and \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$	Three reg. operands; bit-by-bit AND
	or	or \$s1,\$s2,\$s3	$\$s1 = \$s2 \mid \$s3$	Three reg. operands; bit-by-bit OR
	nor	nor \$s1,\$s2,\$s3	$\$s1 = \sim (\$s2 \mid \$s3)$	Three reg. operands; bit-by-bit NOR
	and immediate	andi \$s1,\$s2,100	$\$s1 = \$s2 \& 100$	Bit-by-bit AND reg with constant
	or immediate	ori \$s1,\$s2,100	$\$s1 = \$s2 \mid 100$	Bit-by-bit OR reg with constant
	shift left logical	sll \$s1,\$s2,10	$\$s1 = \$s2 \ll 10$	Shift left by constant
	shift right logical	srl \$s1,\$s2,10	$\$s1 = \$s2 \gg 10$	Shift right by constant
Conditional branch	branch on equal	beq \$s1,\$s2,L	if ( $\$s1 == \$s2$ ) go to L	Equal test and branch
	branch on not equal	bne \$s1,\$s2,L	if ( $\$s1 != \$s2$ ) go to L	Not equal test and branch
	set on less than	slt \$s1,\$s2,\$s3	if ( $\$s2 < \$s3$ ) $\$s1 = 1$ ; else $\$s1 = 0$	Compare less than; used with beq, bne
	set on less than immediate	slti \$s1,\$s2,100	if ( $\$s2 < 100$ ) $\$s1 = 1$ ; else $\$s1 = 0$	Compare less than immediate; used with beq, bne
Unconditional jump	jump	j L	go to L	Jump to target address

Il set di istruzioni continua a crescere!



# Quiz

- Il linguaggio C ha molti statements per il supporto delle decisioni e dei loop, mentre il linguaggio assembler per il MIPS ne ha di meno (fondamentalmente, il salto condizionale). Perché?

