

# Elenco (quasi completo) delle istruzioni RISC-V

## Operandi RISC-V

Nome	Esempio	Commenti
32 registri	x0-x31	Accesso veloce ai dati. Nel RISC-V gli operandi devono essere contenuti nei registri per potere eseguire delle operazioni. Il registro x0 contiene sempre il valore 0
$2^{61}$ parole di memoria	Memoria[0], Memoria[8], ... Memoria[18 446 744 073 709 551 608]	Alla memoria si accede solamente attraverso istruzioni di trasferimento dati. Il RISC-V utilizza l'indirizzamento al byte, perciò due variabili ampie due parole (double word) hanno indirizzi in memoria a distanza 8. La memoria consente di memorizzare strutture dati, vettori, o il contenuto dei registri

## Linguaggio assembler RISC-V

Tipo di istruzioni	Istruzioni	Esempio	Significato	Commenti
Aritmetiche	Somma	add x5, x6, x7	$x5 = x6 + x7$	Operandi in tre registri
	Sottrazione	sub x5, x6, x7	$x5 = x6 - x7$	Operandi in tre registri
	Somma immediata	addi x5, x6, 20	$x5 = x6 + 20$	Utilizzata per sommare delle costanti
Trasferimento dati	Lettura parola doppia	ld x5, 40(x6)	$x5 = \text{Memoria}[x6 + 40]$	Spostamento di una parola doppia da memoria a registro
	Memorizzazione parola doppia	sd x5, 40(x6)	$\text{Memoria}[x6 + 40] = x5$	Spostamento di una parola doppia da registro a memoria
	Lettura parola	lw x5, 40(x6)	$x5 = \text{Memoria}[x6 + 40]$	Spostamento di una parola da memoria a registro
	Lettura parola senza segno	lwu x5, 40(x6)	$x5 = \text{Memoria}[x6+40]$	Spostamento di una parola senza segno da memoria a registro
	Memorizzazione parola	sw x5, 40(x6)	$\text{Memoria}[x6+40] = x5$	Spostamento di una parola da registro a memoria
	Lettura mezza parola	lh x5, 40(x6)	$x5 = \text{Memoria}[x6+40]$	Spostamento di una mezza parola da memoria a registro
	Lettura mezza parola, senza segno	lhu x5, 40(x6)	$x5 = \text{Memoria}[x6+40]$	Spostamento di una mezza parola senza segno da memoria a registro
	Memorizzazione mezza parola	sh x5, 40(x6)	$\text{Memoria}[x6+40] = x5$	Spostamento di una mezza parola da registro a memoria
	Lettura byte	lb x5, 40(x6)	$x5 = \text{Memoria}[x6+40]$	Spostamento di un byte da memoria a registro
	Lettura byte, senza segno	lbu x5, 40(x6)	$x5 = \text{Memoria}[x6+40]$	Spostamento di un byte senza segno da memoria a registro
Operazioni atomiche	Memorizzazione byte	sb x5, 40(x6)	$\text{Memoria}[x6+40] = x5$	Spostamento di un byte da registro a memoria
	Lettura di una parola e blocco	lr.d x5, (x6)	$x5 = \text{Memoria}[x6]$	Caricamento di una parola come prima fase di un'operazione atomica sulla memoria
	Memorizzazione condizionata di una parola	sc.d x7, x5, (x6)	$\text{Memoria}[x6] = x5; x7 = 0/1$	Memorizzazione di una parola come seconda fase di un'operazione atomica sulla memoria
	Caricamento costante nella mezza parola superiore	lui x5, 0x12345	$x5 = 0x12345000$	Caricamento di una costante su 20 bit nei 12 bit più significativi di una parola

# Elenco (quasi completo) delle istruzioni RISC-V

Tipo di istruzioni	Istruzioni	Esempio	Significato	Commenti
Logiche	And	and x5, x6, x7	$x5 = x6 \& x7$	Operandi in tre registri; AND bit a bit
	Or inclusivo	or x5, x6, x8	$x5 = x6   x8$	Operandi in tre registri; OR bit a bit
	Or esclusivo (Xor)	xor x5, x6, x9	$x5 = x6 ^ x9$	Operandi in tre registri; XOR bit a bit
	And immediato	andi x5, x6, 20	$x5 = x6 \& 20$	AND bit a bit tra un operando in registro e una costante
	Or immediato	ori x5, x6, 20	$x5 = x6   20$	OR bit a bit tra un operando in registro e una costante
	Xor immediato	xori x5, x6, 20	$x5 = x6 ^ 20$	XOR bit a bit tra un operando in registro e una costante
Scorrimento (shift)	Scorrimento logico a sinistra	sll x5, x6, x7	$x5 = x6 << x7$	Scorrimento a sinistra mediante registro
	Scorrimento logico a destra	srl x5, x6, x7	$x5 = x6 >> x7$	Scorrimento a destra mediante registro
	Scorrimento aritmetico a destra	sra x5, x6, x7	$x5 = x6 >> x7$	Scorrimento a destra aritmetico mediante registro
	Scorrimento logico a sinistra immediato	slli x5, x6, 3	$x5 = x6 << 3$	Scorrimento a sinistra mediante costante
	Scorrimento logico a destra immediato	srli x5, x6, 3	$x5 = x6 >> 3$	Scorrimento a destra mediante costante
	Scorrimento aritmetico a destra immediato	srai x5, x6, 3	$x5 = x6 >> 3$	Scorrimento a destra aritmetico mediante costante
Salvi condizionati	Salta se uguale	beq x5, x6, 100	Se ( $x5 == x6$ ) vai a PC+100	Test di uguaglianza; salto relativo al PC
	Salta se non è uguale	bne x5, x6, 100	Se ( $x5 != x6$ ) vai a PC+100	Test di disuguaglianza; salto relativo al PC
	Salta se minore di	blt x5, x6, 100	Se ( $x5 < x6$ ) vai a PC+100	Comparazione di minoranza; salto relativo al PC
	Salta se maggiore o uguale di	bge x5, x6, 100	Se ( $x5 >= x6$ ) vai a PC+100	Comparazione di maggioranza o uguaglianza; salto relativo al PC
	Salta se minore di senza segno	bltu x5, x6, 100	Se ( $x5 < x6$ ) vai a PC+100	Comparazione di minoranza senza segno; salto relativo al PC
	Salta se maggiore o uguale di senza segno	bgeu x5, x6, 100	Se ( $x5 >= x6$ ) vai a PC+100	Comparazione di maggioranza o uguaglianza senza segno; salto relativo al PC
Salvi incondizionati	Salta e collega	jal x1, 100	$x1 = PC+4; \text{ vai a PC+100}$	Chiamata a procedura con indirizzamento relativo al PC
	Salta e collega mediante registro	jalr x1, 100(x5)	$x1 = PC+4; \text{ vai a } x5+100$	Ritorno da procedura; chiamata indiretta

# Operazioni logiche

- I primi calcolatori operavano solo su parole intere
- Molto presto si manifestò la necessità di operare su porzioni di parole o addirittura sul singolo bit
- Il RISC-V fornisce alcune istruzioni che permettono di fare questo in maniera semplificata (rispetto ad altre architetture) ma efficace

# Shift logico

- Consideriamo per prima cosa lo shift logico a sinistra
- L'idea è di inserire degli zeri nella posizione meno significativa e traslare tutto a sinistra perdendo (nel caso di overflow) i bit più significativi
- Ad esempio, supponiamo che  $x_{19}$  contenga il valore 9:

```
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00001001
```

```
slli x11, x19, 4
```

```
00000000 00000000 00000000 00000000 00000000 00000000 00000000 10010000
```

- L'effetto è di memorizzare in  $x_{11}$  valore  $9 * 2^4 = 144$

# Il codice operativo

- Il codice macchina corrispondente all'istruzione:

```
slli x11, x19, 4
```

è:

funz6	immediato	rs1	funz3	rd	codop
0	4	19	1	11	19

Il numero di  
bit di cui  
effettuare lo  
shift

# Esempi

- Come abbiamo visto, l'effetto di questa istruzione è moltiplicare l'operando per  $2^k$ , dove k è il numero di elementi di cui si fa lo shift
- Funziona sempre? Dipende...
- Esempio, se  $x19$  contiene: (num. negativo, in complemento a 2)

```
10000000 00000000 00000000 00000000 00000000 00000000 00000000 00000001
```

- Se effettuiamo:

```
slli x11, x19, 4
```

- Otteniamo un num. positivo (16):

```
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00010000
```

- Il problema è semplicemente che il numero non è rappresentabile! (overflow)

# Esempi

- Consideriamo invece -2 (rappresentato in complemento a 2 su 64 bit):

```
11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111110
```

- Se effettuiamo:

```
slli x11, x19, 1
```

- Otteniamo, correttamente,  $-2 * 2^1 = -4$

```
11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111100
```

- Se rimaniamo nel **range di rappresentabilità** la moltiplicazione per potenze di 2 tramite shift funziona bene

# Shift a destra

- Analogamente allo shift logico a sinistra, si può avere uno shift logico a destra
- Esempio:

```
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00011000
```

- Se effettuiamo:

```
srl x11, x19, 4
```

- Otteniamo:

```
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000001
```

# Esempio

- Come abbiamo visto nelle lezioni sull'aritmetica dei calcolatori, lo shift a destra di  $k$  unità corrisponde a dividere per  $2^k$
- E' corretto?
- Nell'esempio fatto prima facendo lo shift a destra di 4 bit sul numero 24 abbiamo ottenuto 1 (che è il risultato della divisione intera di 24 per  $2^4$ )
- Consideriamo un altro esempio (decimale -2):

```
11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111110
```

```
srl i x11, x19, 4
```

```
00001111 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111
```

- Ciò che si ottiene è un numero positivo e questo nonostante il risultato dell'operazione sia perfettamente rappresentabile...

# Shift aritmetico

- Per risolvere il problema evidenziato si risolve con lo **shift aritmetico** a destra
- In sostanza quello che si inserisce a sinistra non sono bit uguali a 0 ma uguali al bit di segno
- Nel nostro esempio:

```
11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111110
```

*srai x11, x19, 4*

```
11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111
```

# Shift aritmetico

- Notare che lo shift aritmetico non ha nessun senso fatto a sinistra
  - In quel caso o il risultato è rappresentabile (e lo shift logico funziona come aritmetico), o non lo è (e non c'è molto che si possa fare!)
- Altre architetture offrono entrambi i tipi di shift
  - RISC-V essendo RISC fornisce solo l'essenziale!

# Altre operazioni logiche

- Il RISC-V offre in aggiunta alle operazioni logiche appena viste altre operazioni logiche
- Alcune di queste sono:

Operazioni logiche	Operatori C	Operatori Java	Istruzioni RISC-V
Shift a sinistra	<<	<<	sll, slli
Shift a destra	>>	>>>	srl, srli
Shift a destra aritmetico	>>	>>	sra, srai
AND bit a bit	&	&	and, andi
OR bit a bit			or, ori
XOR bit a bit	^	^	xor, xorri
NOT bit a bit	~	~	xori

# AND bit a bit

- Le operazioni logiche operano su ciascun bit
- Esempio, supponiamo che  $x10$  e  $x11$  contengano:

```
00000000 00000000 00000000 00000000 00000000 00000000 00010011 10000001
```

```
00000000 00000000 00000000 00000000 00000000 00000000 11111111 00000000
```

*and x9, x10, x11*

- Risultato:

```
00000000 00000000 00000000 00000000 00000000 00000000 00010011 00000000
```

- L'operazione AND forza alcuni bit a 0 usando come operando una maschera (i cui bit corrispondenti a quelli che si vogliono annullare sono settati a 0)

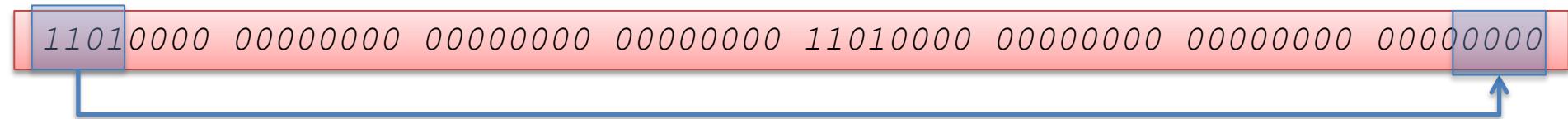
# OR bit a bit

- Analogamente è possibile settare a 1 alcuni bit facendo OR con un operando che abbia 1 nella posizione corrispondente (maschera)
- Esempio:
  - Supponiamo di volere impostare a 1 i 4 bit più significativi di una word contenuta in  $x9$

```
addi x10, x0, 0x000F  
slli x10, x10, 60  
or x9, x9, x10
```

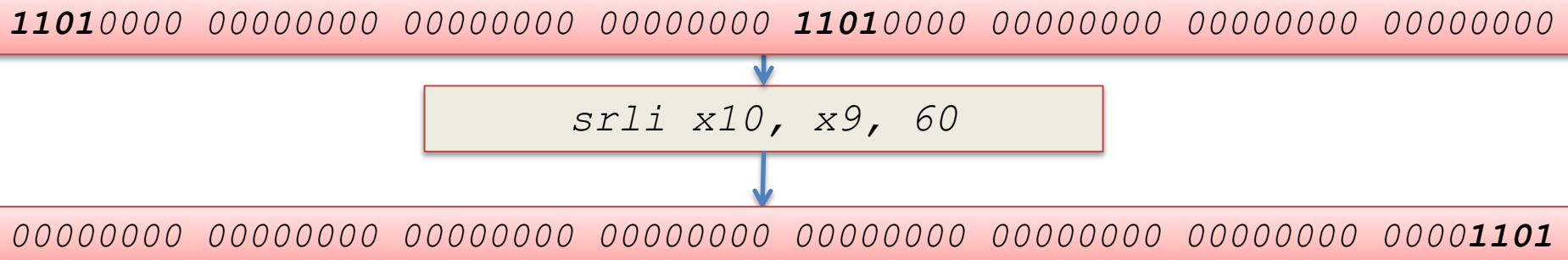
# Rotazione

- Esempio:
  - Supponiamo di volere «girare» i quattro bit più significativi del registro  $x9$  sui suoi bit meno significativi

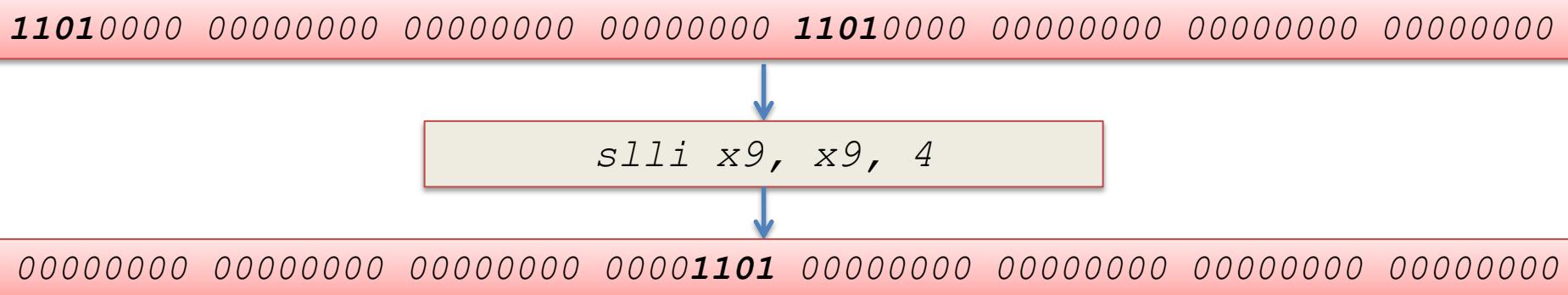


# Esempio

- Primo passo



- Secondo passo



# Esempio

- Terzo passo

```
srl x9, x9, 4
```



```
00000000 00000000 00000000 00000000 11010000 00000000 00000000 00000000
```

- Quarto passo

```
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00001101
```

```
00000000 00000000 00000000 00000000 11010000 00000000 00000000 00000000
```



```
or x9, x9, x10
```



```
00000000 00000000 00000000 00000000 11010000 00000000 00000000 00001101
```

# OR esclusivo (XOR)

- Oltre a AND e OR abbiamo anche un supporto per l'OR esclusivo (XOR)
- Lo XOR produce 1 se e solo se i due bit operandi sono diversi (e zero se sono uguali)

```
00000000 00000000 00000000 00000000 00000000 01100011 01110000 11001101
```

```
00000000 00000000 00000000 00000000 00000000 10011011 01110000 00111101
```

```
xor x11, x9, x10
```

- Risultato:

```
00000000 00000000 00000000 00000000 00000000 11111000 00000000 11110000
```

# Esempio

- Cosa succede se facciamo?

```
xor x9, x9, x9
```

- Il risultato è quello di annullare il valore di  $x9$
- E' una maniera molto veloce ed efficiente per annullare un registro
- Lo XOR si può rappresentare (forma SP) come:
  - $A \text{ XOR } B = (A \text{ AND } \text{NOT}(B)) \text{ OR } (\text{NOT}(A) \text{ AND } B)$

# Lo strano caso del NOT

- Il NOT normalmente è un operatore unario
- Per i progettisti RISC-V tutte le operazioni aritmetiche hanno tre operandi di tipo registro
- Per questo il NOT non è fornito nativamente dal RISC-V
- Tuttavia, si può ottenere dallo XOR come:  $\text{NOT}(A) = \text{XOR}(A, 1)$

```
00000000 00000000 00000000 00000000 00000000 01100011 01100011 11001101
```

```
11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111
```

```
xor x9, x9, x10
```

- Risultato:

```
11111111 11111111 11111111 11111111 11111111 10011100 10011100 00110010
```

# Istruzioni per prendere decisioni

- Una delle caratteristiche fondamentali dei calcolatori (che li distinguono dalle calcolatrici) è la possibilità di alterare il flusso del programma al verificarsi di certe condizioni
  - costrutto *if* nelle varie forme
- Il linguaggio macchina delle varie architetture supporta questa possibilità fornendo istruzioni di «salto condizionato»
- Tipicamente i salti avvengono sulla base di certe condizioni sui registri

# Salto su condizioni

- Le due principali istruzioni di salto condizionato sono:

`beq rs1, rs2, L1`

Se il registro *rs1* è uguale al registro *rs2* effettua un salto all'istruzione con etichetta L1

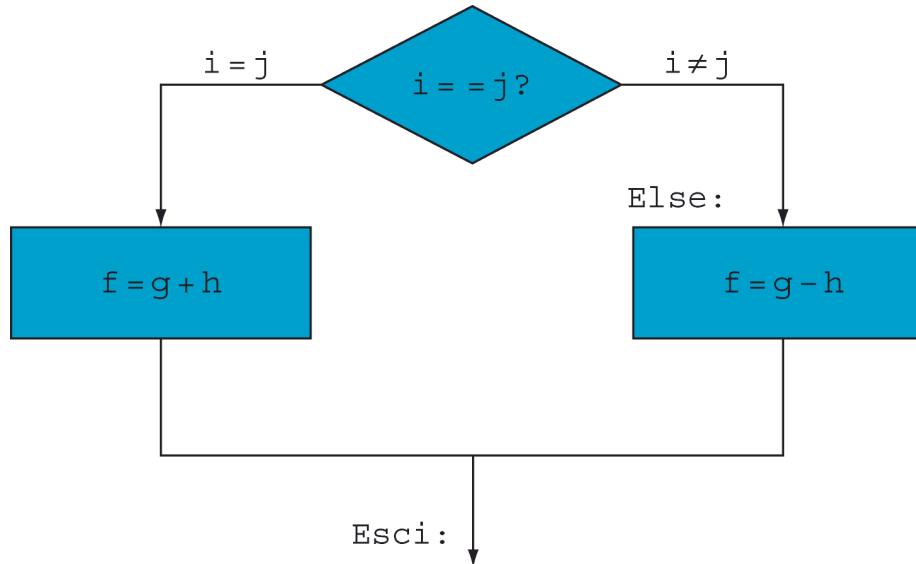
`bne rs1, rs2, L1`

Se il registro *rs1* è diverso dal registro *rs2* effettua un salto all'istruzione con etichetta L1

# Il costrutto if

- Supponiamo di avere il seguente codice C. Come viene tradotto?

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



# Il costrutto if

- Traduzione costrutto *if* precedente:

```
bne x22, x23, ELSE      # Salta a ELSE se x22 div. da x23
add x19, x20, x21       # f = g + h
beq x0, x0, ESCI        # Salto incondizionato a ESCI
ELSE: sub x19, x20, x21  # f = g - h
ESCI: ...
```

- Alcune osservazioni:
  - Le label vengono alla fine tradotte in indirizzi
  - Per fortuna questo lavoro noioso è opera del compilatore!

# Cicli

- La struttura che abbiamo presentato può essere usata anche per realizzare dei cicli
- Consideriamo l'esempio:

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

- Traduzione:

```
Ciclo: slli x10, x22, 3      # Registro temp. x10 = 8*i
       add x10, x10, x25      # Ind. di salva[i] in x10
       ld x9, 0(x10)          # Carica salva[i] in x9
       bne x9, x24, Esci      # Esci se raggiunto limite
       addi x22, x22, 1        # i = i+1
       beq x0, x0, Ciclo
Esci: ...
```

# Alcune considerazioni

- Le sequenze di istruzioni tra due salti condizionati (*conditional branch*) sono così importanti che viene dato loro un nome: **blocchi di base**
  - **Blocco di base:** seq. di istruzioni che non contiene né istruzioni di salto (con l'eccezione dell'ultima) né etichette di destinazione (con l'eccezione della prima)
- Una delle prime fasi della compilazione è di individuare i blocchi base
- Tutti i cicli in linguaggio ad alto livello vengono implementati con blocchi di base

# Altri confronti

- Oltre al salto su operandi uguali o diversi, è utile avere un salto anche su operandi minori/maggiori o minori o uguali
- Il RISC-V mette a disposizione diversi tipi di confronti, in particolare:

---

Salta se minore di      blt x5, x6, 100      Se  $(x5 < x6)$  vai a PC+100

---

Salta se maggiore o  
uguale di      bge x5, x6, 100      Se  $(x5 \geq x6)$  vai a PC+100

---

Salta se minore di      bltu x5, x6, 100      Se  $(x5 < x6)$  vai a PC+100  
senza segno

---

Salta se maggiore  
o uguale di senza  
segno      bgeu x5, x6, 100      Se  $(x5 \geq x6)$  vai a PC+100

# Esempio

- Consideriamo il codice

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

- Traduzione

```
bge x22, x23, ELSE    # Salta a ELSE se x22 >= x23
add x19, x20, x21      # f = g + h
beq x0, x0, ESCI       # Salto incondizionato a ESCI
ELSE: sub x19, x20, x21 # f = g - h
ESCI: ...
```

# Signed e unsigned

- L'esito del confronto è ovviamente diverso se si tiene conto del segno oppure no
- Per questo motivo troviamo due versioni di **blt** e **bge** (rispettivamente, **bltu** e **bgeu**) che operano su interi senza segno («u» sta per «unsigned»)

# Esempio

- Si supponga che  $x9$  e  $x10$  contengano:

```
11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111
```

```
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000001
```

- Cosa produce?

```
blt x9, x10, L1
```

→ salta a L1

- E cosa produce?

```
bltu x9, x10, L1
```

→ non salta a L1

# Un piccolo trucco

- Supponiamo di volere controllare se un indice ( $x20$ ) è fuori dal limite di un array ( $[0, x11]$ )
- Ce la possiamo cavare con un solo confronto!

```
bgeu x20, x11, FuoriLimite
```

- Perché?
- Se  $x20 \geq x11$ , avviene il salto a *FuoriLimite*
- Ma ciò avviene anche se  $x20$  è negativo: interpretando  $x20$  come unsigned, necessariamente sarà  $x20 \geq x11$  (si noti che  $x11$  è necessariamente un num. positivo, in quanto uguale alla dimensione dell'array-1)

# Il costrutto case/switch

- Una possibilità è quella di effettuare una sequenza in cascata di *if-then-else*
- In realtà esiste una tecnica diversa:
  - Memorizzare i vari indirizzi del codice da eseguire in una tabella (una sequenza di word)
  - Caricare in un registro l'indirizzo a cui saltare
  - Fare un salto all'indirizzo puntato dal registro tramite l'istruzione **jalr** («jump and link register», che vedremo più avanti)

# Il costrutto case/switch

- Esempio:

```
switch(a) {  
    case 1: <code 1>;  
    case 2: <code 2>;  
    ...  
}
```

```
slli x9, x10, 3 # moltiplica per 8  
add x9, x9, x11 # x11 ind. base TABLE  
ld x12, 0(x9)   # carica indirizzo  
jalr x1, x12
```

Tabella che  
memorizza gli indirizzi  
dei codici da eseguire

Salto all'indirizzo nel  
registro x12