

Architettura degli elaboratori - lezione 3


Appunti di Davide Scarlata 2024/2025

 Prof: **Claudio Schifanella**

 Mail: claudio.schifanella@unito.it

 Corso: **C**

 [Moodle Unito](#)





 Data:** 25/02/2025

Architettura RISC-V

Memoria

L'architettura **RISC-V** prevede diversi tipi di registri e una RAM per la gestione dei dati:

Registri:

-  **32 registri interi** (`x0` - `x31`)
-  **32 registri floating-point** (`f0` - `f31`)
-  **Control Status Register** (gestione stato CPU)
-  **Program Counter (PC)** (indica l'istruzione successiva)
 - Registri per gli interi
 - Quantità: 32, indicati con `x0` .. `X31`
 - Dimensione: 64 bit se RV64I, 32 bit se RV32I
 -

x0	zero
x1	Return address (ra)
x2	Stack pointer (sp)
x3	Global pointer (gp)
x4	Thread pointer (tp)
x8	Frame pointer (fp)
x10-x17	Registri usati per il passaggio di parametri nelle procedure e valori di ritorno
x5-x7 , x28-x31	Registri temporanei, non salvati in caso di chiamata
x8-x9, x18-x27	Registri da salvare: il contenuto va preservato se utilizzati dalla procedura chiamata

Tipologie di istruzione:

- Aritmetiche
- Logiche
- Accesso alla memoria
- Condizionali

+ — Istruzioni Aritmetiche

📌 Operandi

- ✓ Le istruzioni aritmetiche utilizzano sempre **tre registri** tra `x0-x31`.
- Tutte le istruzioni aritmetiche hanno esattamente 3 operandi
- L'ordine degli operandi è fisso
- ✓ L'**ALU** (Arithmetic Logic Unit) **opera solo sui registri**.
- ✓ I numeri interi sono rappresentati in **complemento a 2**.

? Perché ogni operando deve essere associato a un registro?

💡 L'ALU può operare solo su **registri**, non direttamente sulla memoria, garantendo **maggiore velocità** nelle operazioni.

📌 ✎ Esempi di operazioni aritmetiche

somma:

```
int a = b + c;
```

In Assembly:

```
add x5, x20, x21
```

sottrazione:

```
a = b - c;
```

```
sub x5, x20, x21
```

📌 — Negare una variabile:

In C:

```
a = -a;
```

In Assembly:

```
sub x19, x0, x19
```

📌 ● Assegnare zero a una variabile:

In C:

```
a = 0;
```

In Assembly:

```
add x19, x0, x0 # x19 = 0 + 0
```

📌 + - Somma e sottrazione multipla:

In C:

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

In Assembly:

```
add x5, x20, x21 # x5 = g + h
add x6, x22, x23 # x6 = i + j
sub x19, x5, x6 # x19 = x5 - x6
```

Istruzioni di Accesso alla Memoria

📌 📂 Istruzione di Caricamento (Load)

a cosa fa? copia un dato dalla memoria ad un registro

- L'indirizzo del dato in memoria viene specificato da:
- Indirizzo base (contenuto in un registro)
- Scostamento o offset (compreso tra -2048 e +2047)

L'istruzione lw (load word)

cosa fa ? copia una parola dalla memoria ad un registro

In C:

```
int a = v[0];
```

In Assembly:

```
lw x5, 0(x21) # Carica v[0] in x5
```

📌 📁 Esempio con array:

In C:

```
int a = v[3];
```

In Assembly:

```
lw x5, 12(x21) # Carica v[3] in x5 (3*4 byte di offset)
```

📌 💻 Architettura a 64 bit:

In C:

```
long int a = v[3];
```

In Assembly:

```
ld x10, 24(x21) # Carica v[3] in x10
```

📌 📁 Istruzione di Memorizzazione (Store)

In C:

```
v[2] = a;
```

In Assembly:

```
sw x5, 8(x21) # Salva il valore di x5 in v[2]
```

Esempio completo:

In C:

```
g = h + v[3];  
v[6] = g - f;
```

In Assembly:

```
lw x10, 12(x21) # Carica v[3] in x10  
add x5, x9, x10 # g = h + v[3]  
sub x6, x5, x19 # g = g - f  
sw x6, 24(x21) # Salva g in v[6]
```

Architettura a 64 bit:

In C:

```
long int g = h + v[3];  
v[6] = g - f;
```

In Assembly:

```
ld x10, 24(x21) # Carica v[3] in x10  
add x5, x9, x10 # g = h + v[3]  
sub x6, x5, x19 # g = g - f  
sd x6, 48(x21) # Salva g in v[6]
```