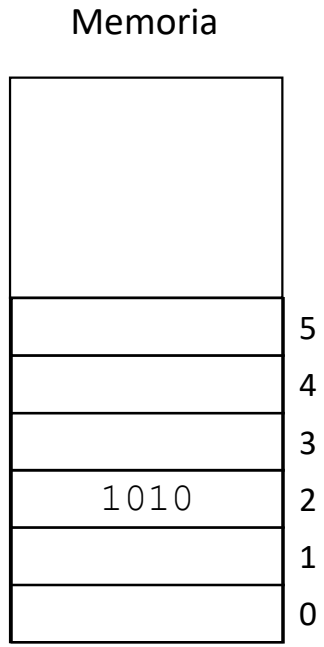
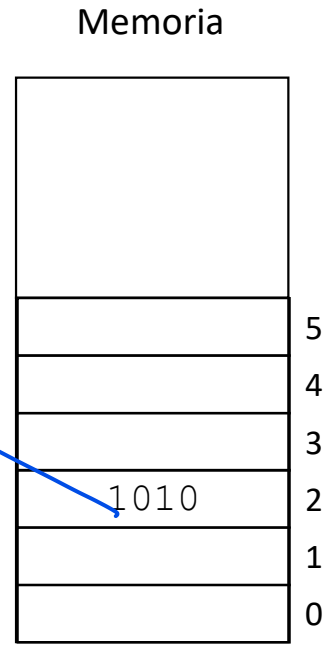
Istruzioni di accesso alla memoria

- Che cosa accade quando
  - Le variabili utilizzate in un programma sono maggiori del numero di registri a disposizione
  - Si utilizzano strutture dati complesse (vettori, liste, ecc)
- I dati sono salvati in memoria centrale
- La memoria centrale può essere astratta come un grande vettore monodimensionale
- Nell'esempio, la quarta cella di memoria ha valore 1010
  - M[2] = 1010



Istruzioni di accesso alla memoria

- La ALU può leggere e scrivere solo dai registri
- L'accesso alla memoria è più lento rispetto a quello dei registri
- Il compilatore si occupa di individuare la strategia più efficiente per le operazioni di caricamento e salvataggio dei dati tra registri e memoria
- Variabili utilizzate più di frequente devono rimanere il più possibile salvate nei registri



Istruzioni di accesso alla memoria

- L'istruzione `load` 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 **`ld`** (`load doubleword`) carica una **parola doppia** dalla memoria in un registro

```
long a;
long v[10];
…
a = v[3]
```

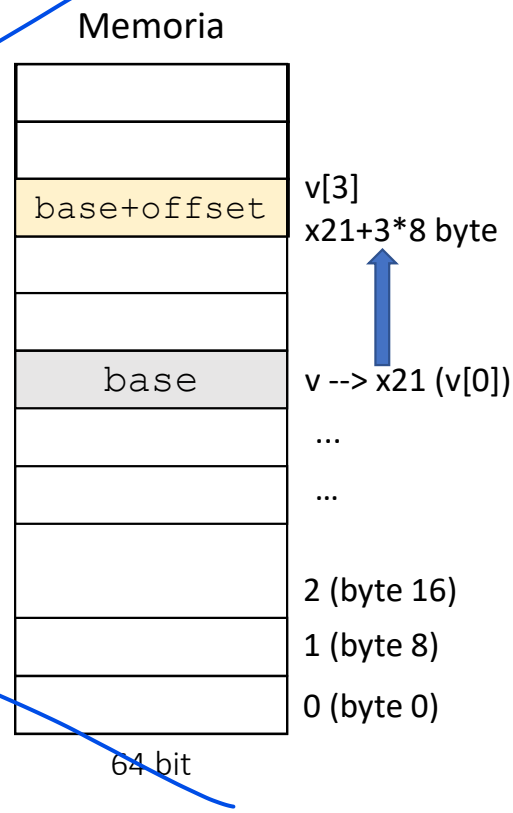
Linguaggio C

⇒

```
a -> x5
v -> x21
```

```
ld x5, 24(x21)
```

RISC-V assembler



Istruzioni di accesso alla memoria

- L'istruzione `ld` (`load doubleword`) carica una **parola doppia** dalla memoria in un registro
- Le celle dell'array possono essere memorizzate con differenti orientamenti

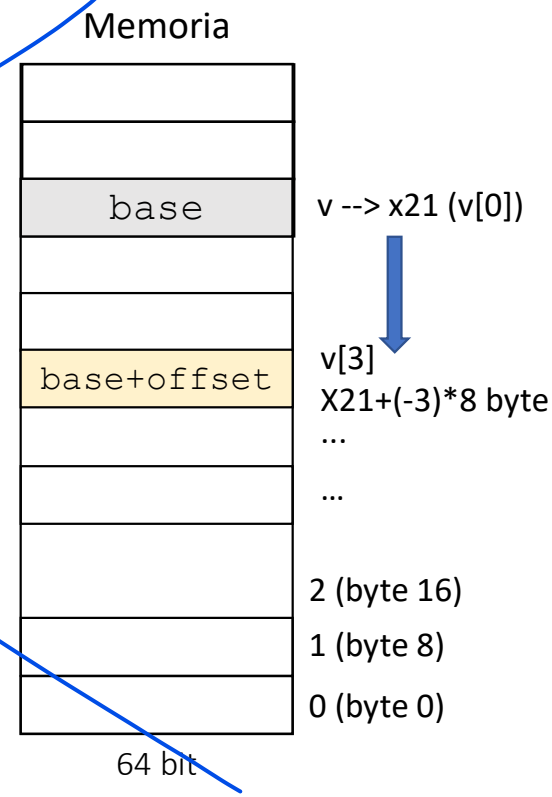
```
long a;
long v[10];
…
a = v[3]
```

Linguaggio C

```
a -> x5
v -> x21
```

`ld x5, -24(x21)`

RISC-V assembler



Istruzioni di accesso alla memoria

- L'istruzione `store` copia un dato da un registro alla memoria
- L'indirizzo di destinazione in memoria viene specificato da:
  - Indirizzo base (contenuto in un registro)
  - Scostamento o offset (compreso tra -2048 e +2047)
- L'istruzione **`sd`** (`store doubleword`) salva una **parola doppia** in memoria

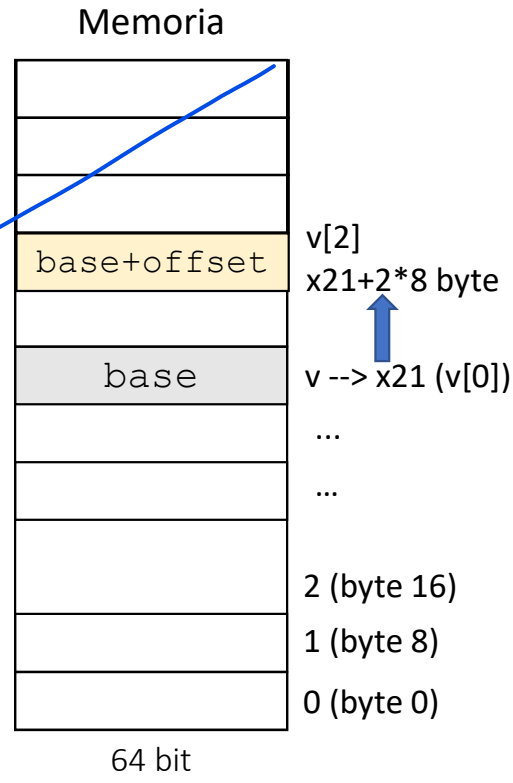
```
long a;
long v[10];
…
v[2] = a
```

Linguaggio C

```
a -> x5
v -> x21
```

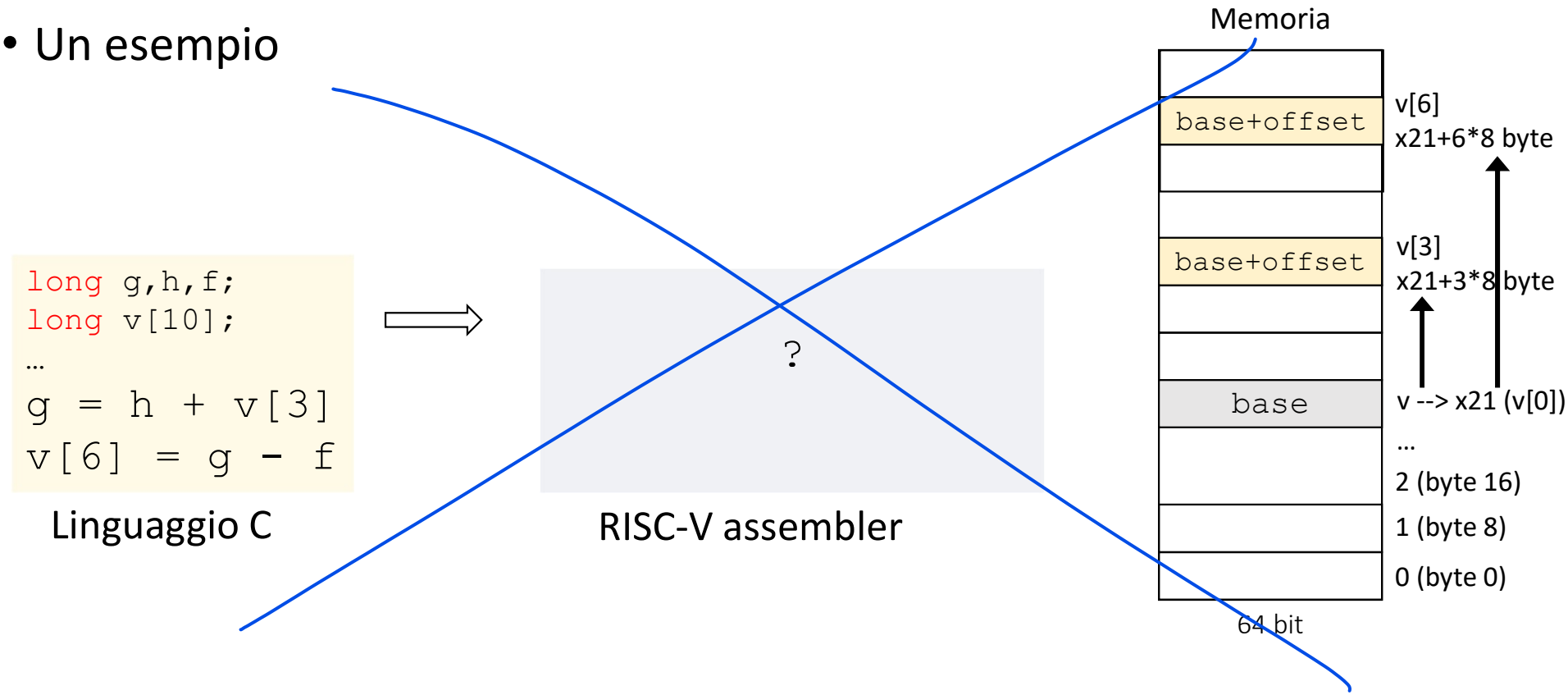
```
sd x5, 16(x21)
```

RISC-V assembler



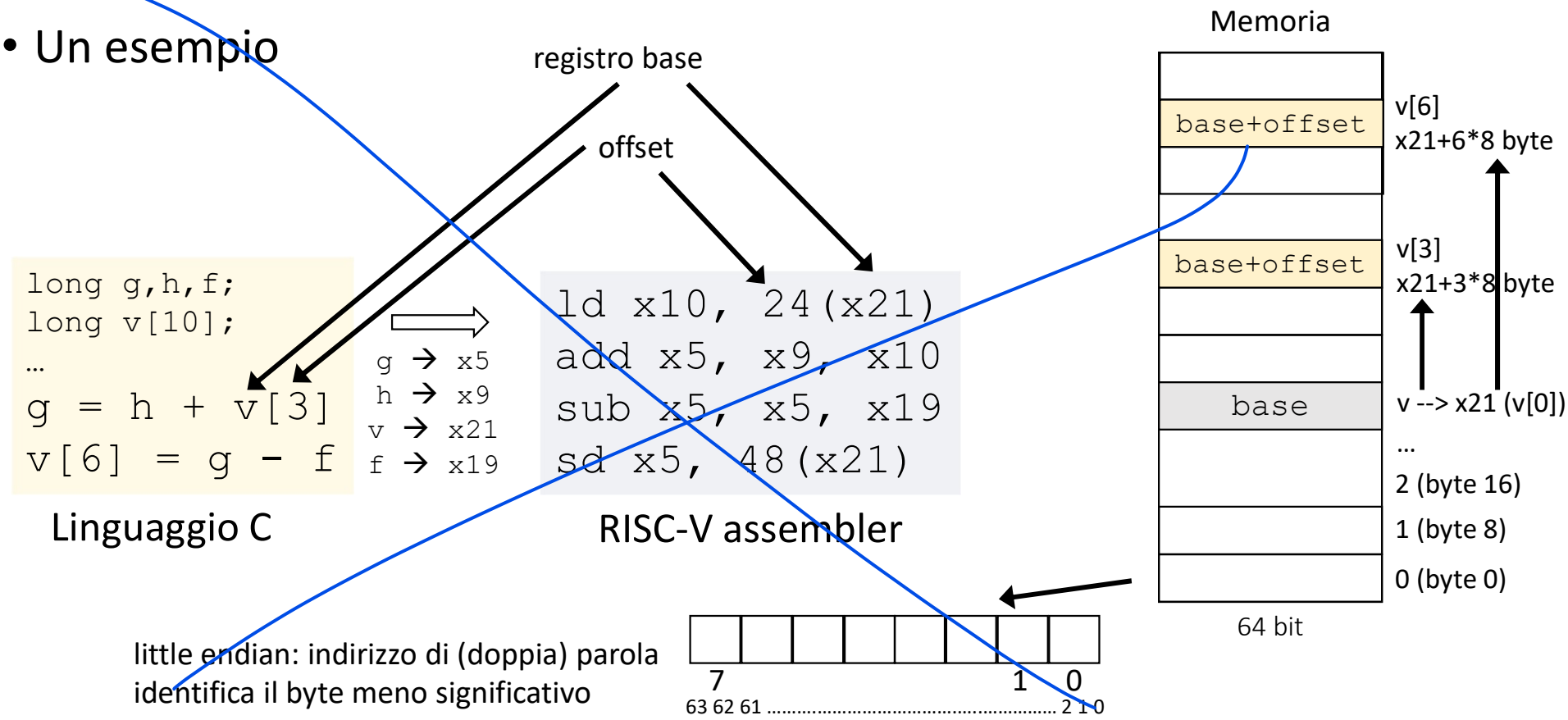
Istruzioni di accesso alla memoria

- Un esempio



Istruzioni di accesso alla memoria

- Un esempio



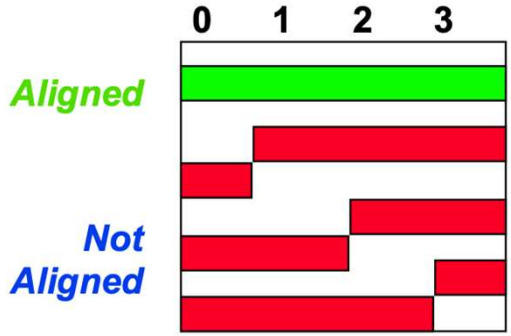
Istruzioni di accesso a byte, half-word e word

- Per accedere al singolo byte sono a disposizione
  - (Utile per le stringhe di caratteri ASCII)
  - `lb x5, 0(x6)` “load byte”
  - `sb x5, 0(x6)` “store byte”
- Per accedere alla half-word (16 bit) ci sono
  - (Utile per le stringhe di caratteri UNICODE, es. in Java)
  - `lh x5, 0(x6)` “load half-word”
  - `sh x5, 0(x6)` “store half-word”
- Per accedere alla word (32 bit) ci sono
  - `lw x5, 0(x6)` “load word”
  - `sw x5, 0(x6)` “store word”

Nota: in fase di caricamento (load), dovendo porre la quantità da 8/16/32 bit in 64 bit, viene automaticamente effettuata **l’estensione del segno**. Se ciò non si vuole, si devono usare `lbu` (al posto di `lb`) e `lhu` (al posto di `lh`) e `lwu` (al posto di `lw`) ed estensione con 0

Restrizioni sull'allineamento degli indirizzi

- La memoria è classicamente indirizzata "al byte"
- Quindi, le istruzioni di load e store usano indirizzi al byte, però
  - `lw`, `lwu` e `sw` trasferiscono 32 bit
  - `lh`, `lhu` e `sh` trasferiscono 16 bit
  - solo `lb`, `lbu`, `sb` trasferiscono 8 bit
- È conveniente pertanto che l'indirizzo sia opportunamente allineato...
  - per `lw`, `lwu`, `sw` dovrebbe essere allineato ad un multiplo di 4
  - per `lh`, `lhu`, `sh` dovrebbe essere allineato ad un multiplo di 2
- Esempi di dati ALLINEATI e NON ALLINEATI "alla word"



Nota: se si specifica un indirizzo non allineato rispetto a quanto l'istruzione desidera, il RISC-V impiegherà un tempo per l'accesso al dato maggiore

Istruzioni di accesso alla memoria

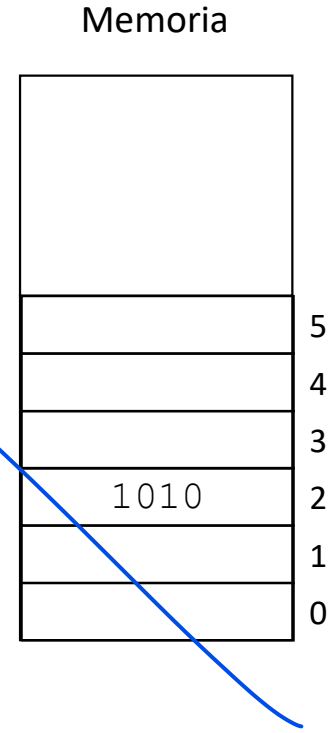
- Un esempio con variabili a 32 bit

```
int g,h,f;
int v[10];
…
g = h + v[3]
v[6] = g - f
```

Linguaggio C

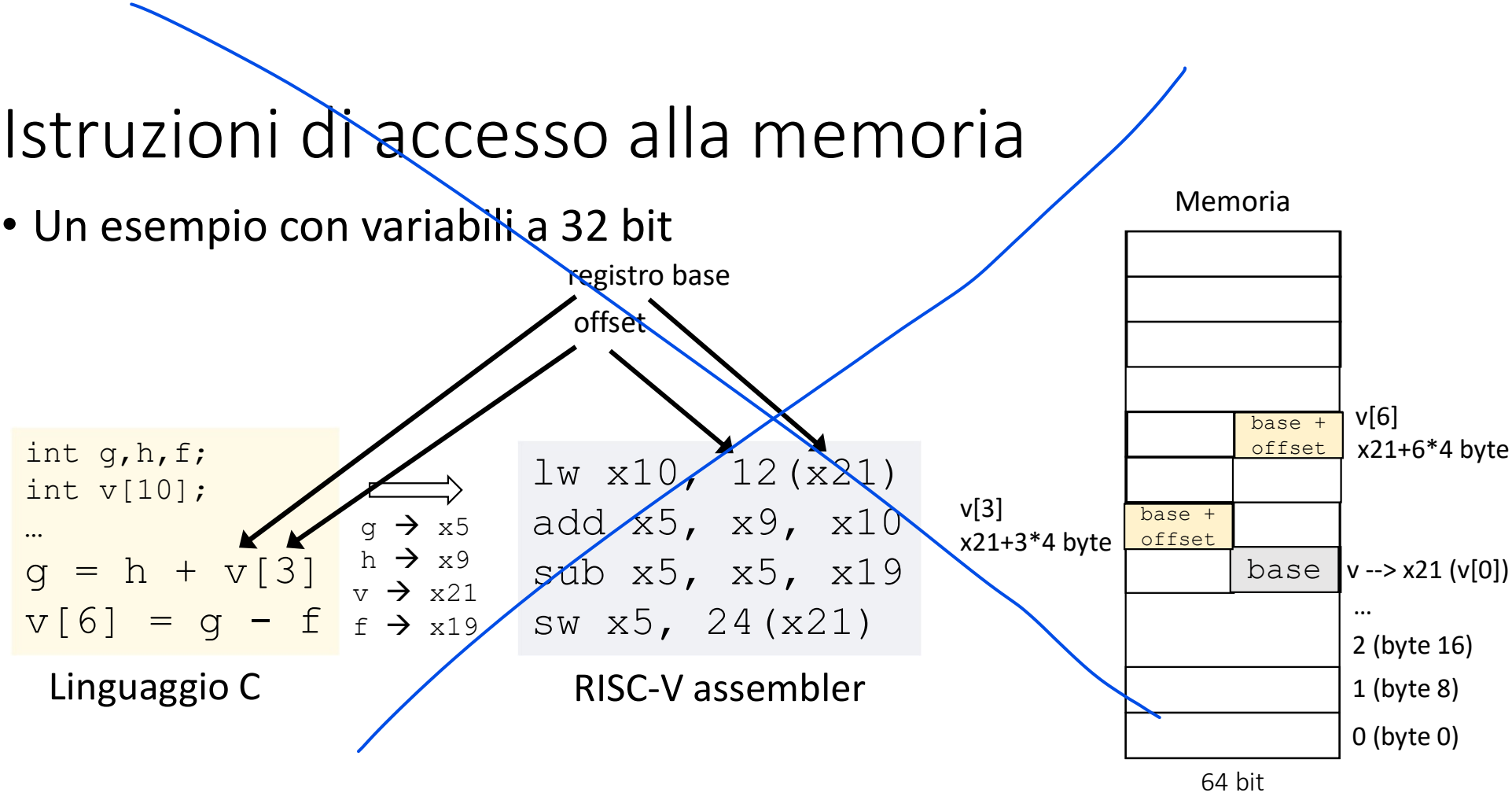
?

RISC-V assembler



Istruzioni di accesso alla memoria

- Un esempio con variabili a 32 bit



Istruzioni di accesso alla memoria

- Un esempio con variabili a 16 bit

```
short g,h,f;
short v[10];
…
g = h + v[3]
v[6] = g - f
```

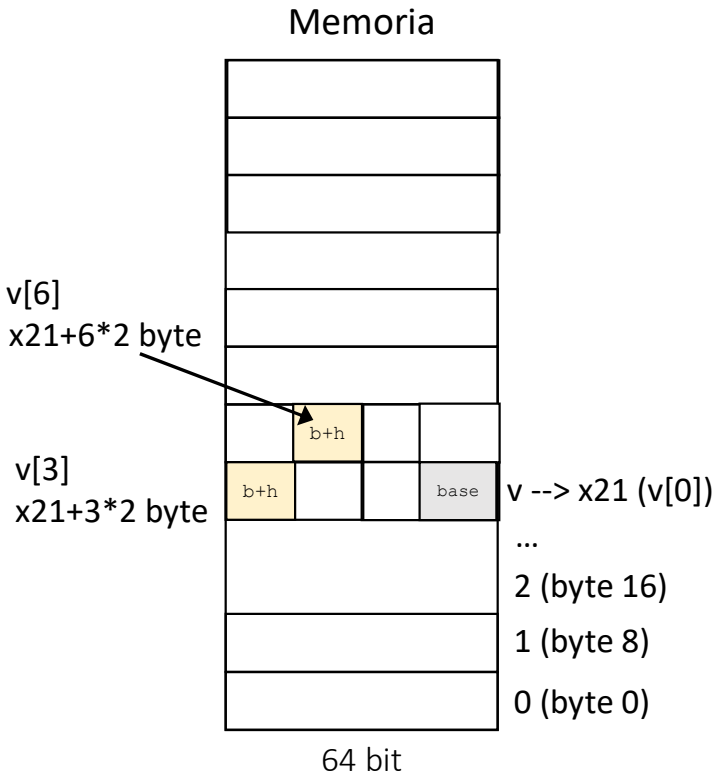
Linguaggio C

g → x5
h → x9
v → x21
f → x19

registro base
offset

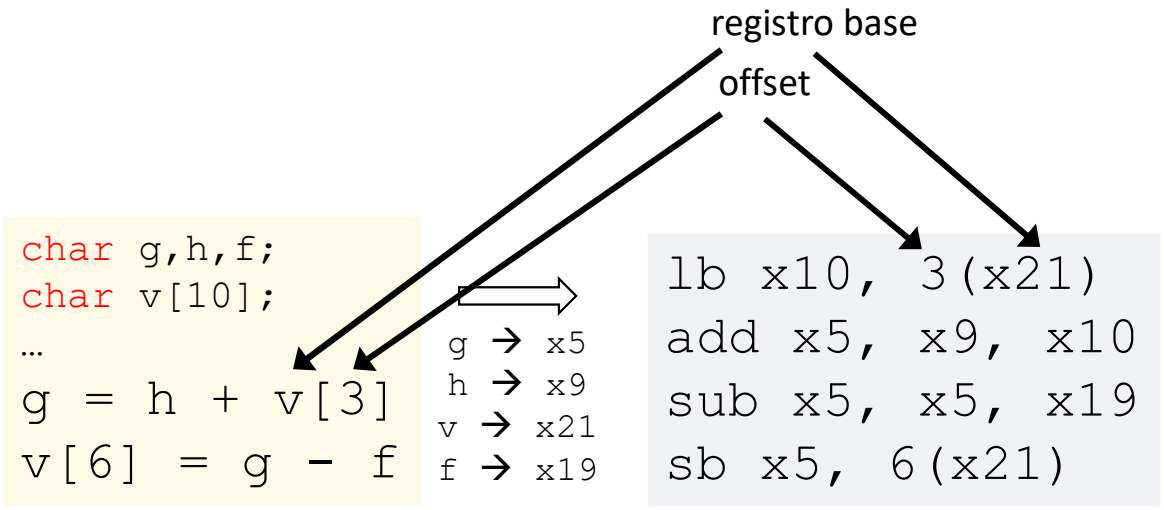
```
lh x10, 6(x21)
add x5, x9, x10
sub x5, x5, x19
sh x5, 12(x21)
```

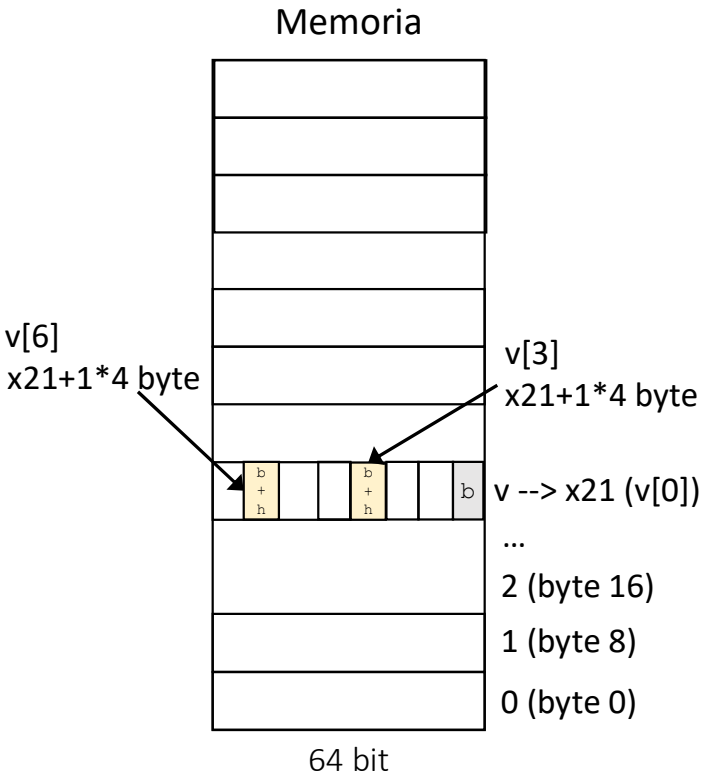
RISC-V assembler



Istruzioni di accesso alla memoria

- Un esempio con variabili a 8 bit



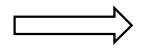


Operandi immediati e costanti

- In più della metà delle operazioni aritmetiche, uno degli operandi è una costante (benchmark SPEC CPU2006)
- I valori delle costanti solitamente sono molto piccoli
  - `a = a + 1`
  - `b = b + 5`
- Es: l'operazione `b = b + 5` può essere rappresentata con due istruzioni assembler

```
b = b + 5
```

Linguaggio C




```
ld x9, indirizzoCostante5(x3)
add x5, x5, x9
```

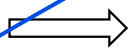
RISC-V assembler

Operandi immediati e costanti

- Alternativa: istruzioni aritmetiche in cui uno degli operandi è una costante
- L'istruzione di somma immediata è chiamata `addi` (`add immediate`)

```
b = b + 5
```

Linguaggio C



b → x5

```
addi x5, x5, 5
```

RISC-V assembler

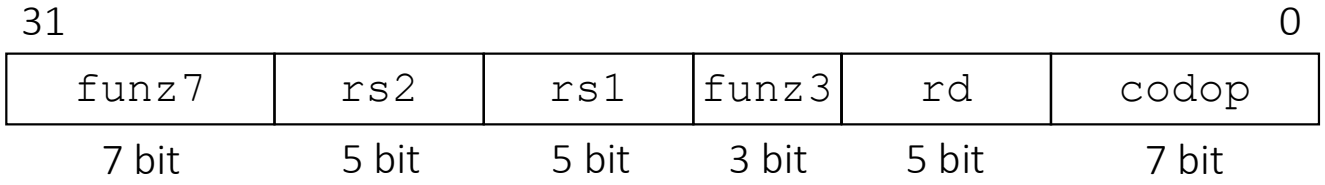
- La costante può assumere valori tra `-2048` e `+2047`
- La sottrazione immediata non esiste: si usano le costanti con valore negativo

RIP

Il linguaggio macchina

- Il linguaggio assembler fornisce una rappresentazione human readable delle istruzioni RISC-V
- Il calcolatore può eseguire solo istruzioni rappresentate come sequenze di bit (formato binario)
- RISC-V definisce diversi formati di istruzioni che consentono di codificare in binario ogni istruzione assembler
- Ogni istruzione RISC-V richiede esattamente 32 bit per la sua rappresentazione in linguaggio macchina
  - La semplicità favorisce la regolarità
- Una sequenza di istruzioni in linguaggio macchina viene chiamata codice macchina

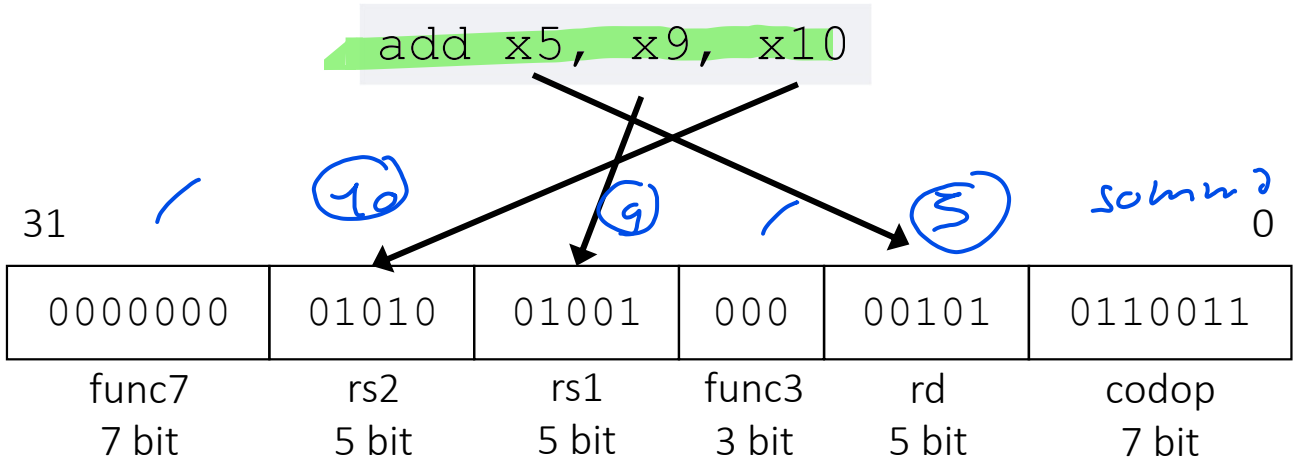
Formato di tipo R (registro)



- Permette di codificare le istruzioni `add`, `sub`, `and`, `or`, `xor`, …
  - `codop`: codice operativo dell'istruzione
  - `rd`: registro di destinazione
  - `rs1`: registro che contiene il primo operando sorgente
  - `rs2`: registro che contiene il secondo operando sorgente
  - `funz3`, funz7: codici operativi aggiuntivi

Formato di tipo R (registro)

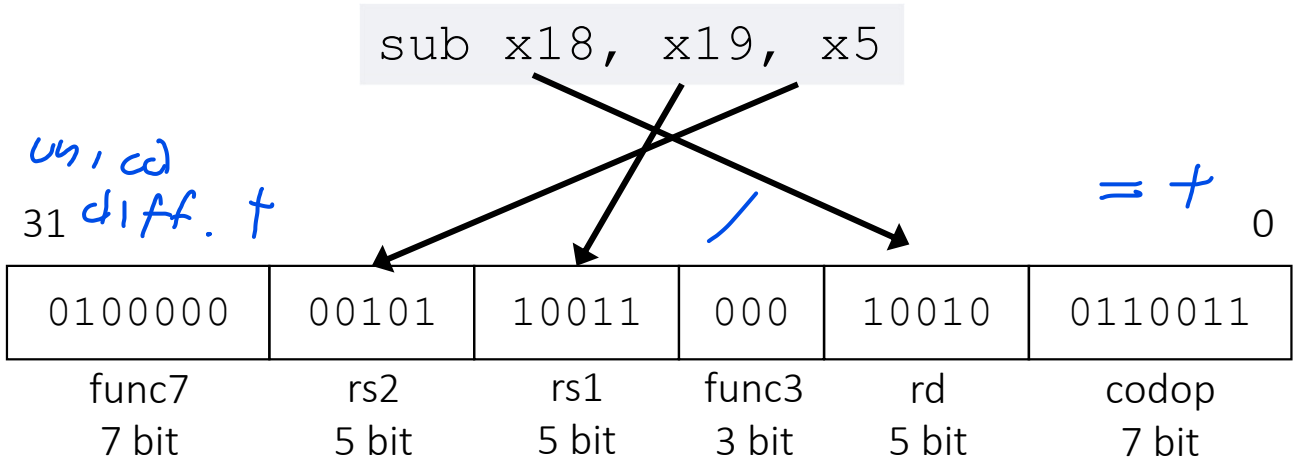
- Esempio



- Per specificare uno dei 32 registri sono necessari 5 bit
- `codop + func7 + func3` indicano l'istruzione rappresentata

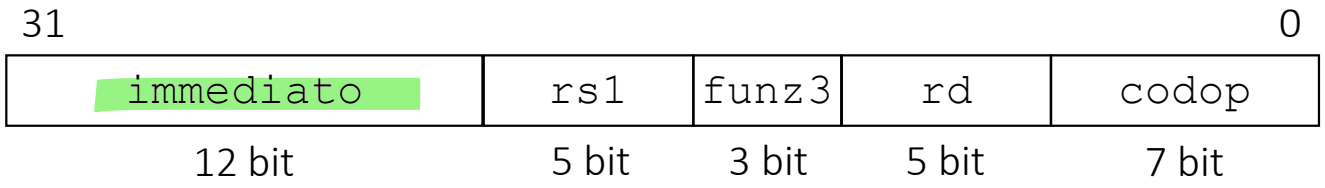
Formato di tipo R (registro)

- Esempio



- Per specificare uno dei 32 registri sono necessari 5 bit
- `codop + func7 + func3` indicano l'istruzione rappresentata

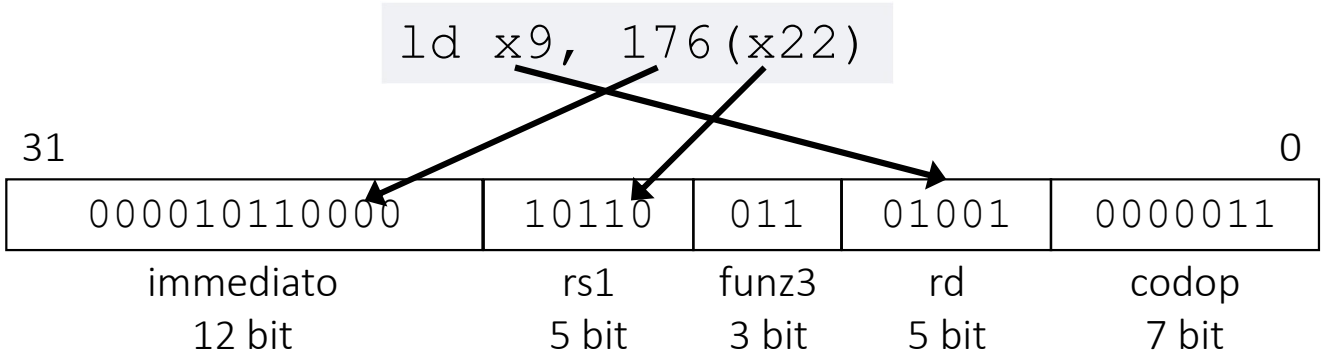
Formato di tipo I (immediato)

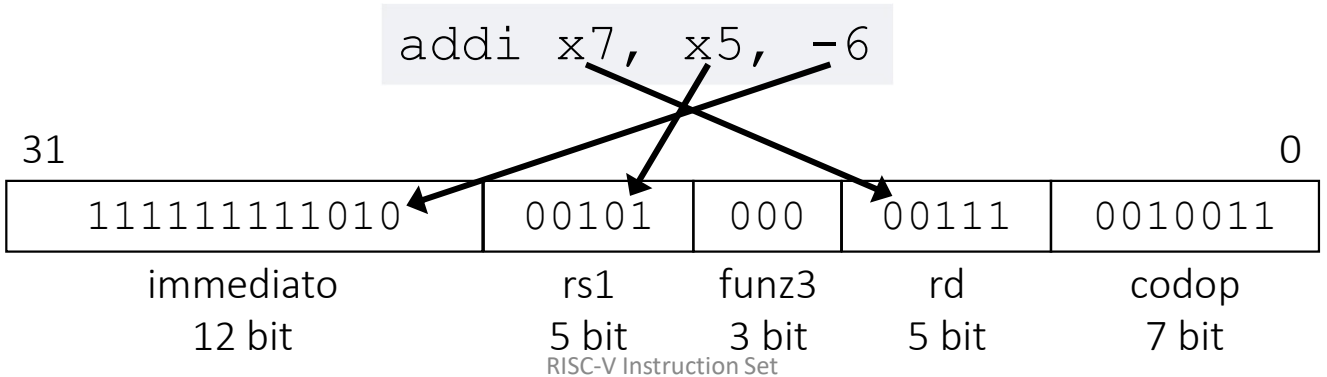


- Permette di codificare le istruzioni che richiedono il caricamento dalla memoria o una costante, come `load`, `addi`, `andi`, `ori`, …
- Sono presenti 12 bit perché con 5 bit l'intervallo di rappresentazione per costanti e (soprattutto) offset sarebbe stato troppo ridotto
- Il campo immediato
  - Rappresentato in complemento a due
  - Valori possibili: da `-2048` a `+2047`

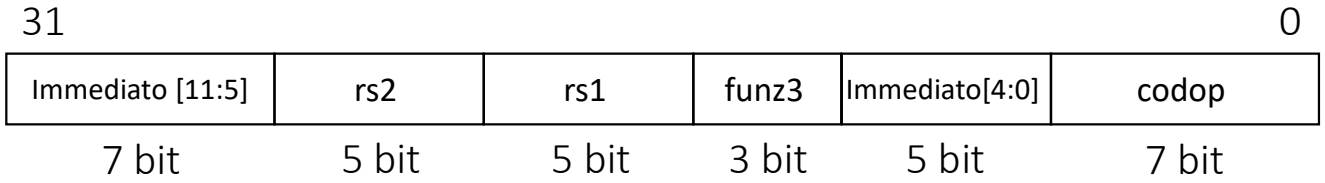
Formato di tipo I (immediato)

- Esempi





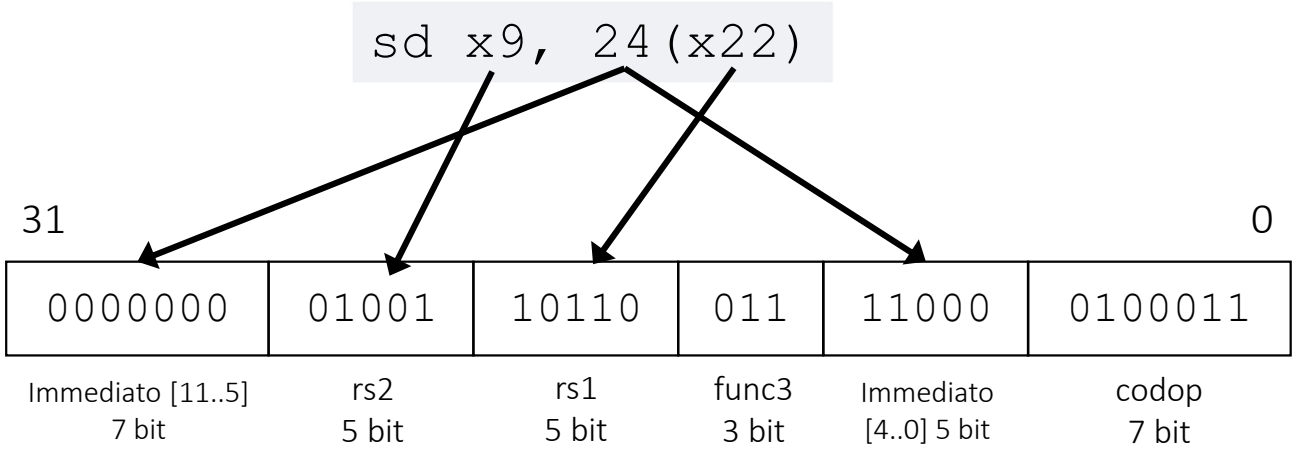
Formato di tipo S



- Permette di codificare le istruzioni che richiedono il salvataggio in memoria o una costante, come `store`
- Il campo immediato (offset) viene diviso in due parti per mantenere i due campi rs1 e rs2 nella stessa posizione rispetto alle istruzioni di tipo R
- Il campo immediato
  - Rappresentato in complemento a due
  - Valori possibili: da -2048 a $+2047$

Formato di tipo S

- Esempio



Manuale di riferimento RISC-V

- Il libro di testo contiene tutti i dettagli sulla codifica in linguaggio macchina delle istruzioni assembler

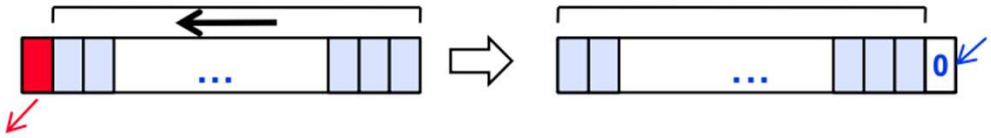
istruzione mnemonica

ISTRUZIONI ORDINATE NUMERICAMENTE PER CODICE OPERATIVO

Mnemonico	FMT	Codice operativo	Funz3	Funz7 o cost	Esadecimale
lb	I	0000011	000		03/0
lh	I	0000011	001		03/1
lw	I	0000011	010		03/2
ld	I	0000011	011		03/3
lbu	I	0000011	100		03/4
lhu	I	0000011	101		03/5
lwu	I	0000011	110		03/6
fence	I	0001111	000		0F/0
fence.i	I	0001111	001		0F/1
addi	I	0010011	000		13/0
silli	I	0010011	001	0000000	13/1/00
slti	I	0010011	010		13/2
sltiu	I	0010011	011		13/3
xori	I	0010011	100		13/4
srli	I	0010011	101	0000000	13/5/00
srai	I	0010011	101	0100000	13/5/20
ori	I	0010011	110		13/6
andi	I	0010011	111		13/7
auipc	U	0010111			17
addiw	I	0011011	000		1B/0
slliw	I	0011011	001	0000000	1B/1/00
srliw	I	0011011	101	0000000	1B/5/00
sraiw	I	0011011	101	0100000	1B/5/20
sb	S	0100011	000		23/0
sh	S	0100011	001		23/1
sw	S	0100011	010		23/2
sd	S	0100011	011		23/3
add	R	0110011	000	0000000	33/0/00
sub	R	0110011	000	0100000	33/0/20
sll	R	0110011	001	0000000	33/1/00
slt	R	0110011	010	0000000	33/2/00

Operazioni logiche

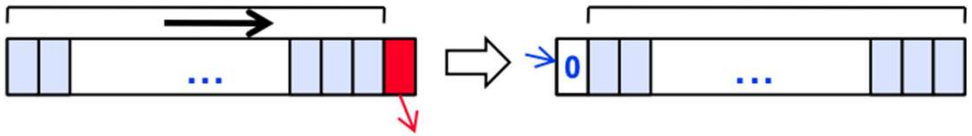
- Shift logico
  - A sinistra




```
sll x9, x22, x19
```
x9 = x22 << x19

```
slli x9, x22, 5
```
x9 = x22 << 5

  - A destra

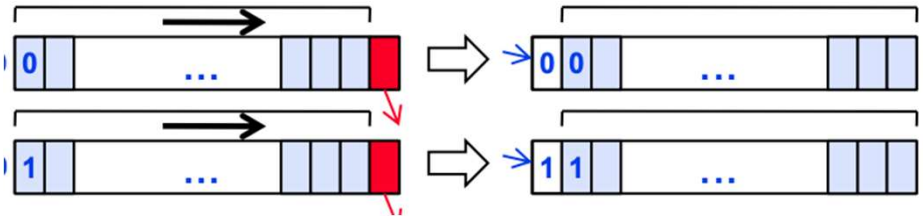



```
srl x9, x22, x19
```
x9 = x22 >> x19

```
srli x9, x22, 5
```
x9 = x22 >> 5

Operazioni logiche

- Shift aritmetico
  - A destra
    - Positivo
    - Negativo




```
sra x9,x22,x19
x9 = x22 >> x19
```

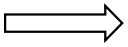
```
srai x9,x22,5
x9 = x22 >> 5
```

  - A sinistra
    - Non esiste perché non ha senso: identico a `sll`

Un esempio

```
long d,i,j;
long v[10];
…
j=5;
…
v[i+d]=v[j+2];
```

Linguaggio C

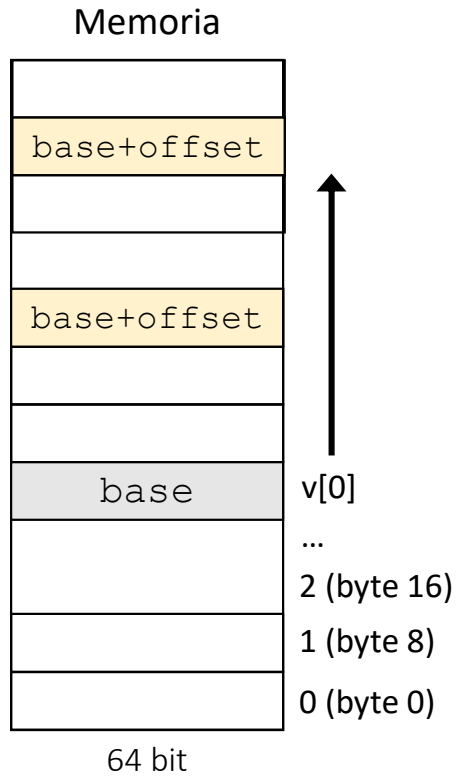


```
addi x18 x0 5

addi x6 ? x17 2
add x6 x6 x18
ld x7 0(x6)
```

RISC-V assembler

```
add x6 x14 x15
sd x7 0(x6
```



Un esempio

```
long d,i,j;
long v[10];
…
j=5;
…
v[i+d]=v[j+2];
```

Linguaggio C

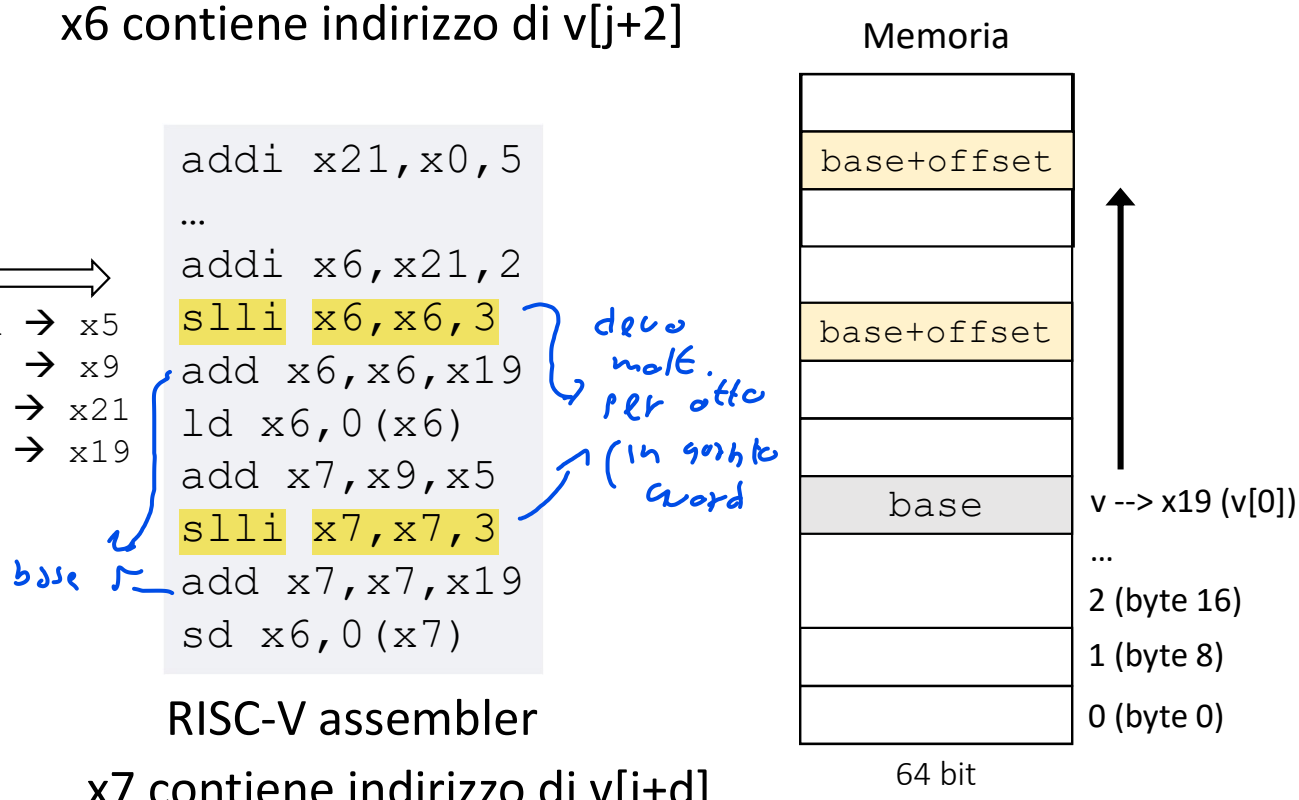
d → x5
i → x9
j → x21
v → x19

x6 contiene indirizzo di v[j+2]

```
addi x21,x0,5
…
addi x6,x21,2
slli x6,x6,3
add x6,x6,x19
ld x6,0(x6)
add x7,x9,x5
slli x7,x7,3
add x7,x7,x19
sd x6,0(x7)
```

RISC-V assembler

x7 contiene indirizzo di v[i+d]



Operazioni logiche

- Le istruzioni assembler `sll` e `srl` e `sra` si rappresentano in linguaggio macchina con il formato `R`
- Le istruzioni assembler `slli` e `srli` e `srai` si rappresentano in linguaggio macchina con il formato `I` (vengono utilizzati i 6 bit meno significativi del campo immediato per codificare la costante, gli altri sono posti a zero)
- Lo shift a sinistra di `i` posizioni calcola una moltiplicazione per 2^i
- Lo shift a destra aritmetico di `i` posizioni calcola una divisione intera per 2^i

Operazioni logiche

- AND

```
and x9,x22,x19
```
x9 = x22 & x19

```
andi x9,x22,5
```
x9 = x22 & 5

- OR

```
or x9,x22,x19
```
x9 = x22 | x19

```
ori x9,x22,5
```
x9 = x22 | 5

- XOR

```
xor x9,x22,x19
```
x9 = x22 $\oplus$ x19

```
xori x9,x22,5
```
x9 = x22 $\oplus$ 5

- NOT

Pseudoistruzione
```
not x5,x6
```
x5 = $\overline{x6}$

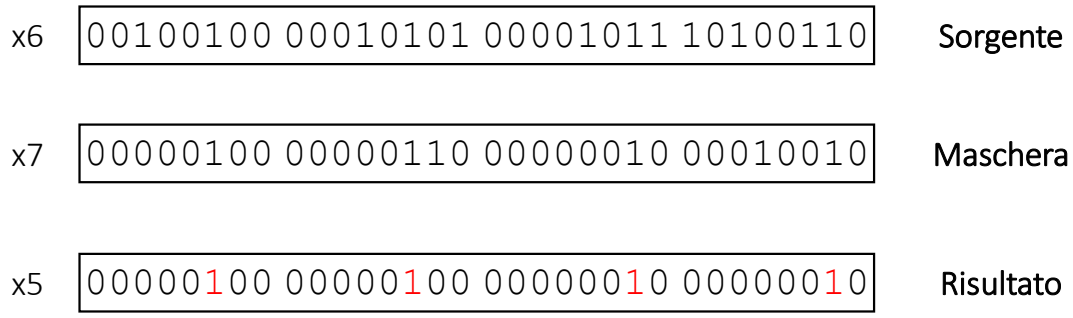
Operazioni logiche

- Le istruzioni assembler `and` e `or` e `xor` si rappresentano in linguaggio macchina con il formato `R`
- Le istruzioni assembler `andi` e `ori` e `xori` si rappresentano in linguaggio macchina con il formato `I`

Operazioni logiche

- L'istruzione `and` permette di selezionare alcuni bit del primo operando indicandoli all'interno di una maschera (secondo operando)
- Esempio (su 32 bit, per esigenze di spazio)

```
and x5,x6,x7
```



Operazioni logiche

- L'istruzione `or` permette di ricopiare il primo operando, settando ad uno anche i bit che sono specificati nella maschera indicata come secondo operando
- Esempio (su 32 bit, per esigenze di spazio)

```
or x5,x6,x7
```