CALCOLATORI
Assembly RISC-V

Giovanni Iacca
giovanni.iacca@unitn.it

Luigi Palopoli
luigi.palopoli@unitn.it

UNIVERSITÀ DEGLI STUDI DI TRENTO
Dipartimento di Ingegneria
e Scienza dell'Informazione

Instruction Set

- Per impartire istruzioni al computer bisogna «parlare» il suo linguaggio
- Il linguaggio si compone di un *vocabolario di istruzioni* detto *instruction set (IS)*
- I vari tipi di processore hanno ciascuno il proprio *IS*.
- Tuttavia le differenze non sono eccessive
  - Un utile esempio è quello delle inflessioni regionali di un'unica radice linguistica

Instruction Set

- Come osservato da von Neumann:
  "certi [insiemi di istruzioni] in linea di principio si prestano a controllare l'hardware"
- Egli stesso osservava che le considerazioni davvero decisive sono di natura pratica:
  - semplicità dei dispositivi richiesti
  - chiarezza delle applicazioni
  - velocità di esecuzione
- Queste considerazioni scritte nel 1947 sono straordinariamente valide anche oggi

In queste lezioni...

- Nelle prossime lezioni, mostreremo diversi instruction sets
- Studieremo inoltre il concetto di *programma memorizzato:*

Istruzioni e dati sono memorizzati come numeri

- Considereremo tre IS:
  - RISC-V
  - Intel
  - ARM

Perché' il RISC-V

- Il motivo per cui mostreremo e faremo esercizi con Intel dovrebbe essere abbastanza chiaro…
  - Milioni di PC sono basati su architettura Intel o Intel compatibile
  - E' un esempio paradigmatico di architettura CISC
- Vedremo inoltre che ARM è una sorta di via di mezzo tra CISC e RISC (RISC «pragmatico»)
  - Usata in moltissimi sistemi embedded
- Partiremo però dall'architettura RISC-V, che è invece un'architettura appunto RISC, che ha origine all'Università di Berkeley nel 2010
  - Moderna, open-source
  - Piuttosto usata in alcune applicazioni specifiche

Incominciamo…

- Inizieremo dall'IS RISC-V
- Fin dai tempi di von Neumann, era convinzione diffusa che all'interno di un IS:

Devono necessariamente essere previste istruzioni per il calcolo delle operazioni aritmetiche fondamentali

Operazioni aritmetiche

- E' dunque naturale che l'architettura **RISC-V** supporti le operazioni aritmetiche
- Per progettare un IS come quello del **RISC-V** terremo conto di vari principi ispiratori

Principio di Progettazione n. 1:
La semplicità favorisce la regolarità

Istruzioni aritmetiche

- Il modo più semplice di immaginare una istruzione aritmetica è a tre operandi:

```
a = b + c
```

- Quindi l'architettura del RISC-V prevede *soltanto* istruzioni aritmetiche a tre operandi, ad esempio:

```
add a, b, c
```

Istruzioni più complesse

- Istruzioni più complesse si ottengono a partire dalla combinazione di istruzioni semplici.
- Esempio (in C):

```
a = b + c;
d = a - e;
```

- Diventa:

```
add a, b, c
sub d, a, e
```

Istruzioni più complesse

- Un altro esempio (in C):

```
a = b + c + d + e;
```

- Diventa:

```
add a, b, c
add a, a, d
add a, a, e
```

Esempio

- Nei linguaggi ad alto livello possiamo scrivere espressioni complesse a piacimento:

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

- Quando si traduce a basso livello bisogna per forza usare sequenze di istruzioni elementari

Esempio

- Ad esempio l'espressione precedente verrà tradotta in istruzioni elementari di questo tipo:

```
add t0, g, h    # la variabile temp.
                # t0 = g + h
add t1, i, j    # la variabile temp.
                # t1 = i + j
sub f, t0, t1   # f = t0 - t1
```

Alcune considerazioni

- Nell'assemblatore (assembler) **RISC-V**
  - I commenti iniziano con # e continuano fino alla fine della linea
  - L'operando "destinatario" dell'operazione è sempre il primo
- Questo non vale per tutti gli assemblatori
- Ad esempio se si usa gcc come assemblatore questo segue la sintassi AT&T per la quale:
  - I commenti si fanno *a la* C*:* /* commento */
  - L'operando destinatario dell'operazione è messo in fondo

Operandi

- Fino ad ora abbiamo usato gli operandi come se fossero “normali” variabili di un linguaggio ad alto livello…
- … in realtà, nel RISC-V gli operandi di operazioni aritmetiche sono *vincolati* ad essere (contenuti nei) *registri*
- Un registro è una particolare locazione di memoria che è *interna* al processore e quindi può essere reperita in maniera velocissima (un ciclo di clock)

Registri del RISC-V

- Il RISC-V contiene 32 registri a 64 bit
  - Gruppi di 64 bit detti «parola doppia» (double word)
  - Gruppi di 32 bit detti «parola» (word)
- Il vincolo di operare solo tra registri semplifica di molto il progetto dell'hardware
- Ma perché solo 32 registri?

Principio di Progettazione n. 2:
minori sono le dimensioni, maggiore la velocità

- Avere molti registri obbligherebbe i segnali a «spostarsi» su distanze più lunghe all'interno del processore
- Quindi per effettuare uno spostamento all'interno di un ciclo di clock saremmo costretti a rallentare il clock

Esempio (ripreso)

- Torniamo al nostro esempio:

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

- Il codice con i registri diventa:

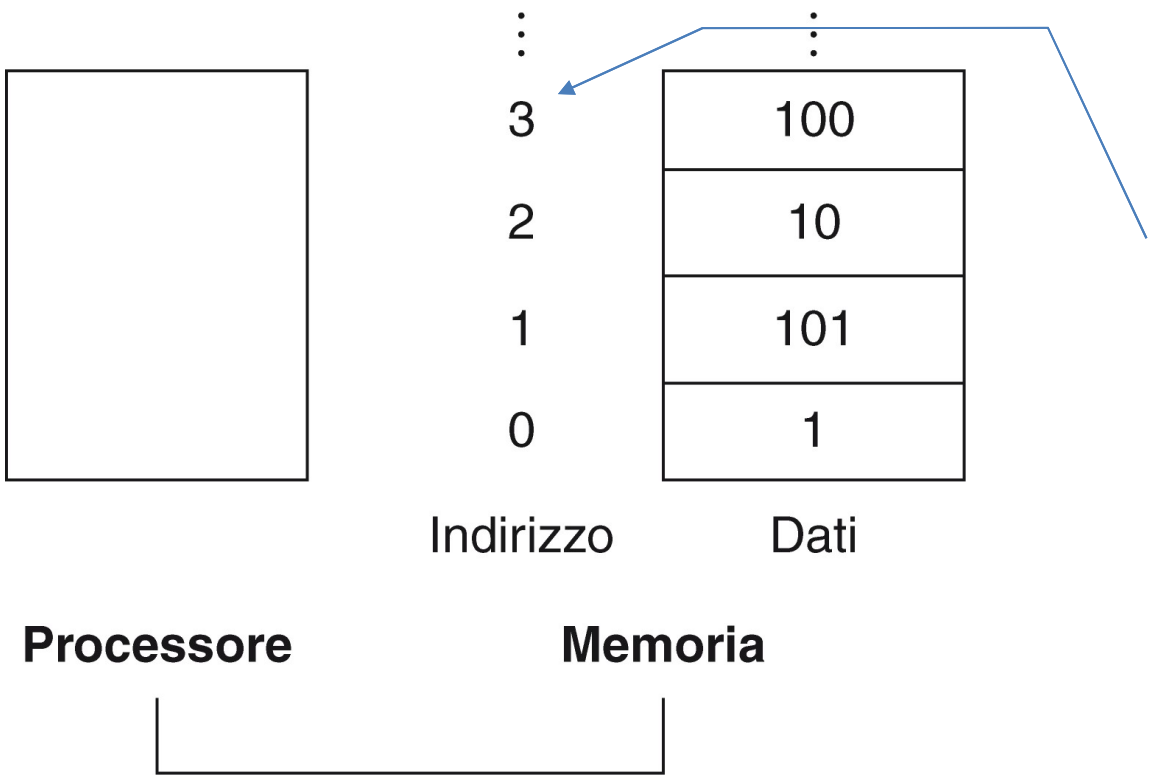
```
add x5, x20, x21      # Il registro temporaneo
                      # x5 viene settato alla
                      # somma dei registri:
                      # x20+x21 (g+h)
add x6, x22, x23      # Idem, x6 conterrà la
                      # somma x22+x23 (i+j)
sub x19, x5, x6       # f = x5 - x6
```

Osservazioni

- Come abbiamo detto, le operazioni logiche e aritmetiche e si effettuano solo tra registri
- Il problema è che i registri non bastano!
- Occorrono istruzioni di trasferimento che:
  - prelevino dei dati da locazioni di memoria, e li carichino nei registri (*load)*
  - salvino il contenuto dei registri in memoria (*store*)

La memoria

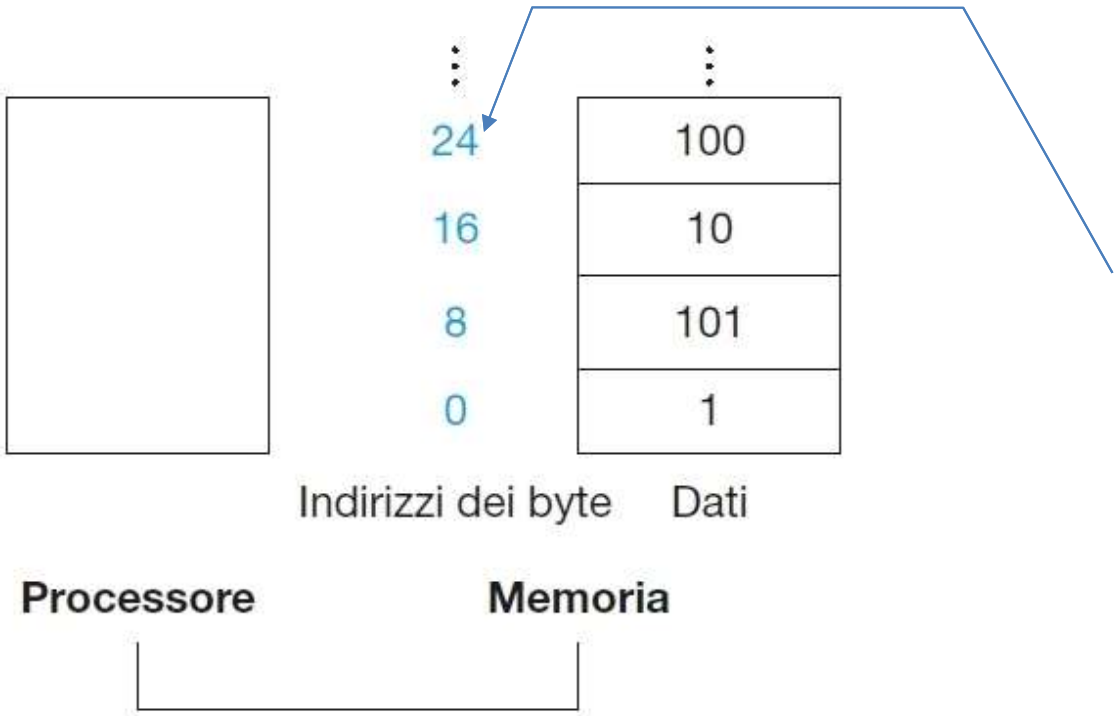
- La memoria è una sequenza di bit organizzati in gruppi di 8 (8 bit = 1 byte)



Ciascun byte è associato ad un indirizzo lineare progressivo tramite il quale è possibile prelevarlo

Word e double word

- Per quanto la maggior parte delle architetture permettano l'accesso a ciascun *byte,* la maggior parte delle volte si trasferiscono multipli di 4 o 8 byte, ovvero *parole* o *parole doppie*



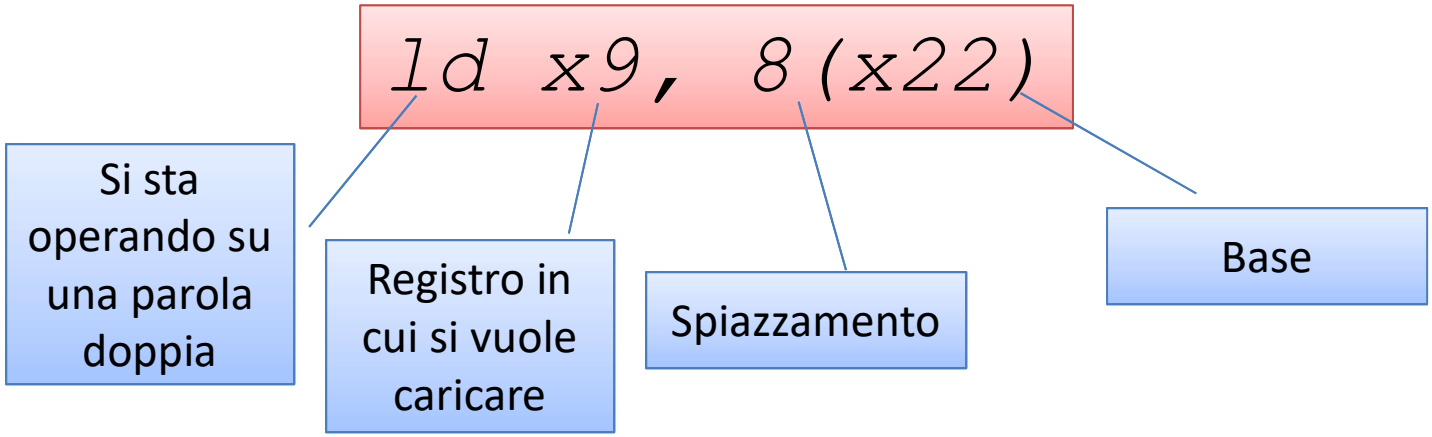
Vincolo di allineamento: è possibile accedere solo a parole poste a indirizzi multipli dell'ampiezza della parola (vincolo presente ad es. in Intel x86, ma non in RISC-V)

Trasferimento

- Per caricare una parola doppia, una parola, o un byte, è necessario specificarne l'indirizzo
- Nell'assemblatore RISC-V l'indirizzo si specifica tramite una *base* (in un registro) e uno spiazzamento o *offset* (costante)
- Come vedremo, in altre architetture (es. Intel e ARM) c'è molta più flessibilità nello specificare l'indirizzo

Load double word

- Il caricamento di una parola doppia (contenuta in un certo indirizzo di memoria) in un registro avviene tramite la seguente istruzione:



- L'effetto di questa istruzione è di caricare in *x9* la parola doppia all'indirizzo dato da *x22* + 8

Esempio

- Supponiamo di volere effettuare l'istruzione:

```
A[12] = h + A[8]
```

- La traduzione in assembly RISC-V è:

```
ld x9, 64(x22)      # il puntatore di A è in x22
                    # e l'ottava parola doppia
                    # comincia all'indirizzo
                    # x22 + 8*8
add x9, x21, x9     # h è in x21
sd x9, 96(x22)      # memorizzo il contenuto
                    # del registro x9 in A[12],
                    # ovvero all'indirizzo
                    # x22 + 12*8
```

Register Spilling

- Tipicamente i programmi contengono più variabili che registri…
- Quello che si fa è caricare le variabili in *uso* in un dato momento nei registri e scaricare quelle che non si usano più in quel momento
- Questa operazione è chiamata *register spilling* ed è eseguita dal compilatore che stima il *working set* ed inserisce nel codice assembly le operazioni di load/unload appropriate
- Nell'esempio precedente, in base al codice potrebbe essere che ad es. l'indirizzo di A serve ancora in `x22` nelle istruzioni successive, mentre forse h si potrebbe scaricare da `x21` (fa tutto il compilatore!)

Operandi immediati o costanti

- Molto spesso nelle operazioni aritmetiche almeno uno dei due operandi è costante
- Un possibile approccio può essere di memorizzare la costante in un qualche indirizzo
- Esempio:

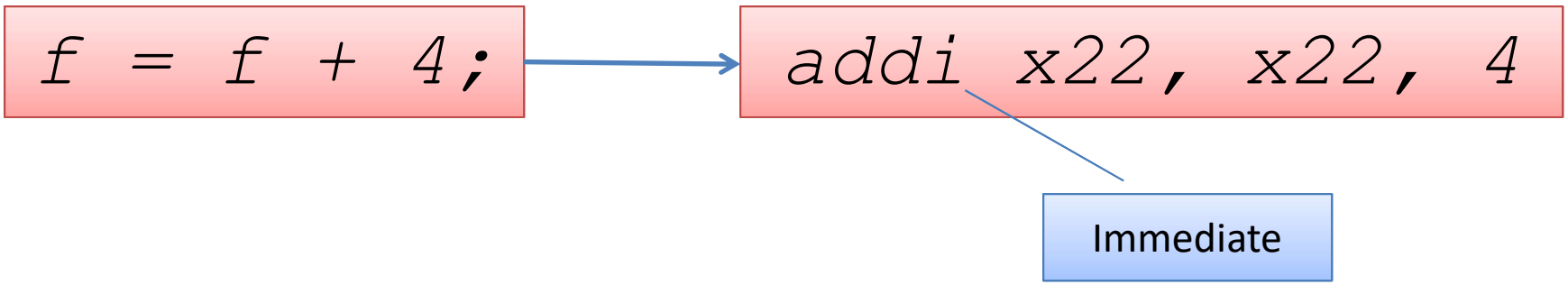
```
f = f + 4;
```



```
ld x9, IndCost4(x3)
# la costante 4 è all'indirizzo x3 + IndCost4
add x22, x22, x9
```

Operandi immediati

- La soluzione che abbiamo visto è piuttosto inefficiente
- Ciò segue l'idea generale di **rendere veloci le situazioni più comuni**
- Per questo motivo esistono istruzioni che permettono di operare con costanti



La costante 0 (zero)

- Esistono alcune costanti che possono essere di grande utilità per semplificare alcune operazioni
  - Es. per spostare un registro in un altro posso renderlo destinatario della somma del sorgente con 0
- Per questo motivo il RISC-V dedica un registro ad hoc ($x0$) alla costante 0

Numeri

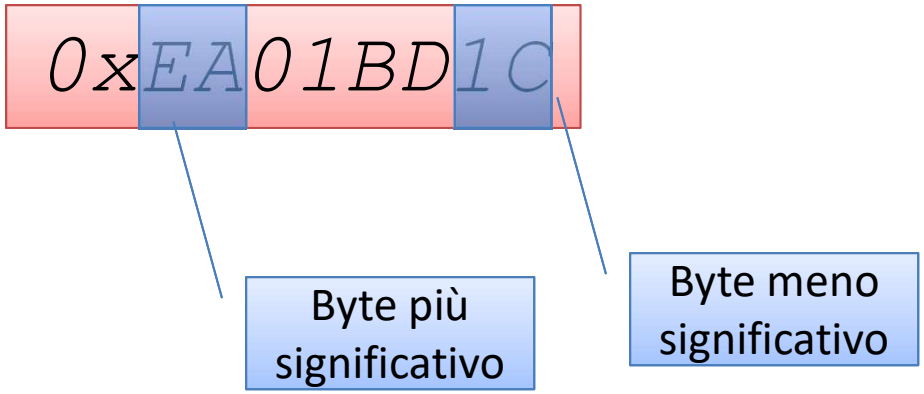
- Prima di parlare del modo in cui le istruzioni sono codificate attraverso numeri ricordiamo:
  - Nei calcolatori l'unità base di informazione è il bit
  - Un gruppo di 4 bit può essere associato ad un numero fino a 16 che rappresenta una cifra nella notazione *esadecimale*
  - Quindi un byte viene rappresentato da due cifre esadecimali (ciascuna corrispondente a 4 bit)
  - Ad esempio

Cifre esadecimali

Esadecimale	Binario	Esadecimale	Binario	Esadecimale	Binario	Esadecimale	Binario
0_{esa}	0000_{due}	4_{esa}	0100_{due}	8_{esa}	1000_{due}	c_{esa}	1100_{due}
1_{esa}	0001_{due}	5_{esa}	0101_{due}	9_{esa}	1001_{due}	d_{esa}	1101_{due}
2_{esa}	0010_{due}	6_{esa}	0110_{due}	a_{esa}	1010_{due}	e_{esa}	1110_{due}
3_{esa}	0011_{due}	7_{esa}	0111_{due}	b_{esa}	1011_{due}	f_{esa}	1111_{due}

Little e Big Endian

- Quando si memorizza una parola di quattro byte (o una doppia parola di 8) in una sequenza di byte posti a indirizzi progressivi, va capito dove va il byte più significativo e quello meno significativo
- Esempio (esadecimale)



Little e Big Endian

- Little Endian (utilizzato ad es. in architetture Intel e RISC-V)

`EA01BD1C` 

- Big Endian (utilizzato ad es. in architetture Motorola e protocolli Internet)

`EA01BD1C` 

Rappresentazione delle istruzioni

- Come i dati, anche un programma deve essere memorizzato in forma numerica
- Questo vuol dire che le istruzioni che abbiamo introdotto simbolicamente (scritte in *assembly*) prima di essere memorizzate (ed eseguite) devono essere convertite in una serie di numeri in formato binario (*codice macchina*)
- Cominciamo con un esempio

Esempio

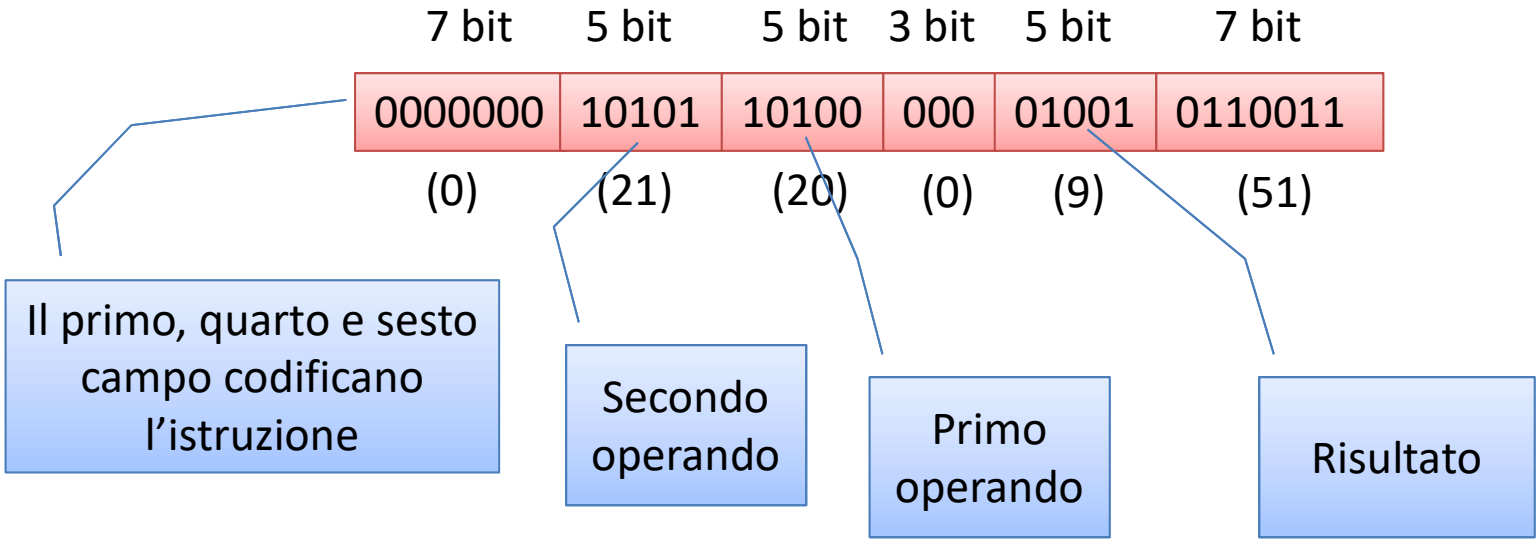
- Consideriamo l'istruzione:

```
add x9, x20, x21
```

- Per rappresentarla tramite un codice numerico *univoco* occorre:
  - Un codice numerico che ci dica che si tratta di una istruzione di somma (add)
  - Altri codici numerici che ci dicano quali sono gli operandi sorgente e destinazione
- In RISC-V, ogni istruzione viene codificata in una parola (ovvero, in 32 bit)

Risultato

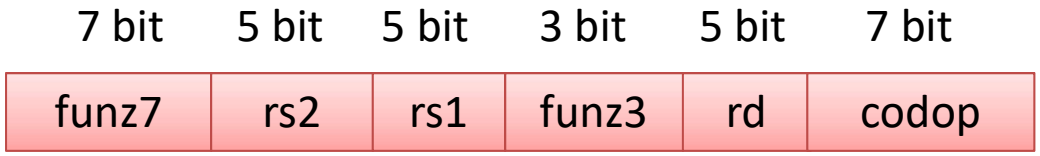
- Il risultato rappresentato in decimale è il seguente:



- In RISC-V i registri sono numerati da 0 a 31 e quindi possono essere specificati come operandi (5 bit) all'interno del codice dell'istruzione

Campi delle istruzioni RISC-V

- In generale, è utile dare un nome ai vari campi relativi al codice macchina di un'istruzione



- *codop:* codice operativo dell'istruzione
- *funz7* e *funz3*: codici operativi aggiuntivi
- *rs1:* primo operando sorgente
- *rs2:* secondo operando sorgente
- *rd:* operando destinazione

Trade-off

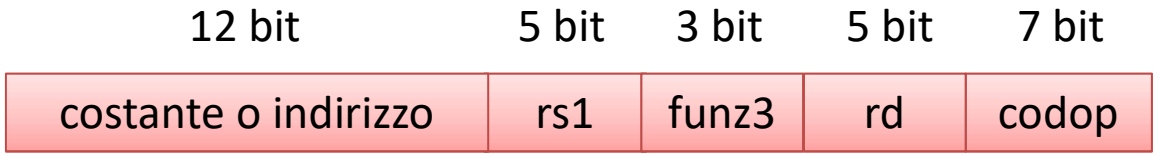
Principio di Progettazione n. 3: un buon progetto richiede buoni compromessi

- Nel nostro caso il *buon* compromesso è di codificare tutte le istruzioni in 32 bit
- Questa scelta ci costa in termini di:
  - limite al numero di istruzioni
  - limite al numero di registri
  - limite alle modalità di indirizzamento
- … ma ci permette di guadagnare molto in efficienza!

Istruzioni immediate

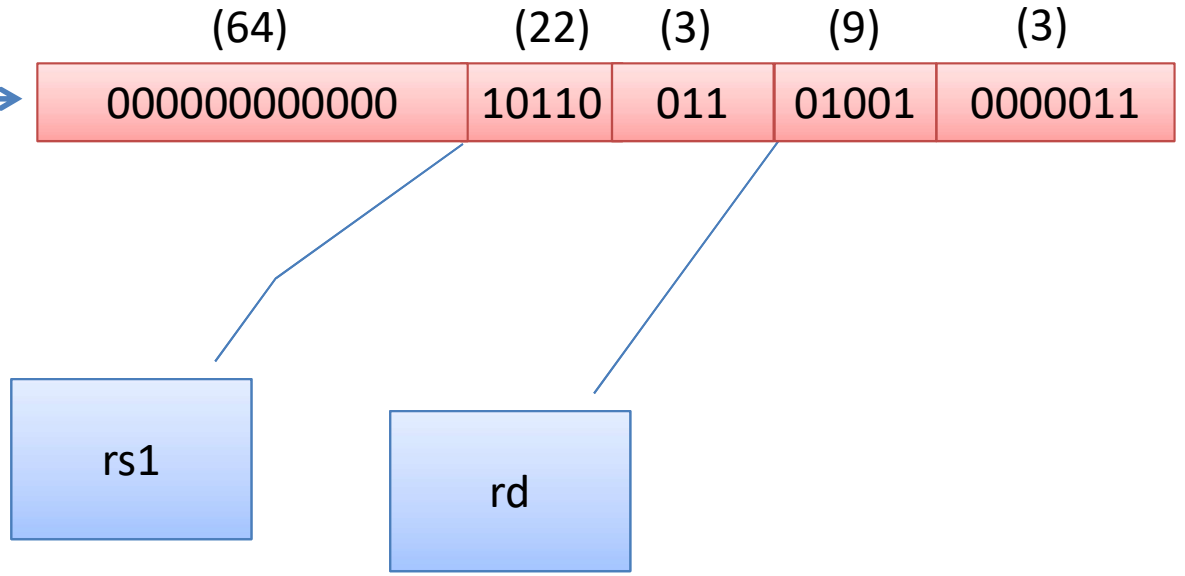
- Abbiamo visto che ci sono istruzioni di caricamento/salvataggio, e di somma con costanti, che sarebbero eccessivamente limitate se usassimo sempre il formato che abbiamo appena visto (detto **R**, da registro)
- Per questo motivo esiste anche un secondo formato (detto **I**, da immediato), utilizzato nei casi di indirizzamento immediato e di istruzioni che fanno uso di costanti

Istruzioni immediate



- Ad esempio:

```
ld x9, 64(x22)
```



Esempio

- Supponiamo di voler tradurre in linguaggio macchina la seguente istruzione:

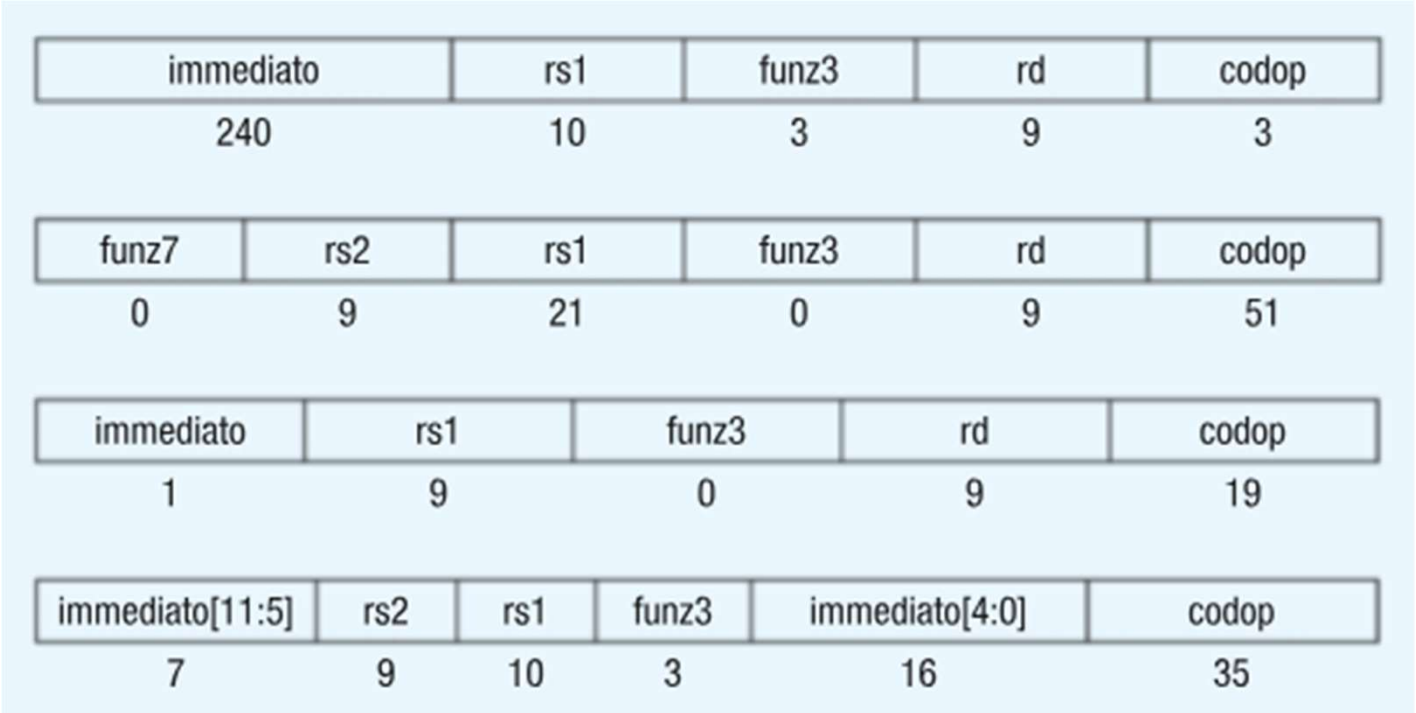
```
A[30] = h + A[30] + 1;
```

- La traduzione è la seguente:

```
ld x9, 240(x10)
add x9, x21, x9
addi x9, x9, 1
sd x9, 240(x10)
```

in codice macchina...

- Cominciamo con il guardare i codici decimali:



immediato	rs1	funz3	rd	codop
240	10	3	9	3

funz7	rs2	rs1	funz3	rd	codop
0	9	21	0	9	51

immediato	rs1	funz3	rd	codop
1	9	0	9	19

immediato[11:5]	rs2	rs1	funz3	immediato[4:0]	codop
7	9	10	3	16	35

in codice macchina...

- In binario:

immediato	rs1	funz3	Rd	codop
000011110000	01010	011	01001	0000011

funz7	rs2	rs1	funz3	rd	codop
0000000	01001	10101	000	01001	0110011

immediato	rs1	funz3	rd	codop
000000000001	01001	000	01001	0010011

immediato[11:5]	rs2	rs1	funz3	Immediato[4:0]	codop
0000111	01001	01010	011	10000	0100011

Riassumendo

- Ciascuna istruzione viene espressa come un numero binario di 32 bit
- Un programma consiste dunque in una sequenza di numeri binari
- Tale sequenza viene scritta in locazioni consecutive di RAM
- In momenti diversi, nella stessa RAM, possiamo rappresentare programmi diversi

Codici delle istruzioni viste fino ad ora

Istruzione (R)	funz7	rs2	rs1	funz3	rd	codop	Esempio
`add`	0000000	00011	00010	000	00001	0110011	add x1, x2, x3
`sub` (sottrazione)	0100000	00011	00010	000	00001	0110011	sub x1, x2, x3
Istruzione (I)	**immediato**		**rs1**	**funz3**	**rd**	**codop**	**Esempio**
`addi` (addizione immediata)	001111101000		00010	000	00001	0010011	addi x1,x2,1000
`ld` (caricamento di parola doppia)	001111101000		00010	011	00001	0000011	ld x1, 1000 (x2)
Istruzione (S)	**Immediato**	**rs2**	**rs1**	**funz3**	**immediato**	**codop**	**Esempio**
`sd` (memorizzazione di parola doppia)	0011111	00001	00010	011	01000	0100011	sd x1, 1000 (x2)