

Il Set di Istruzioni RISC-V

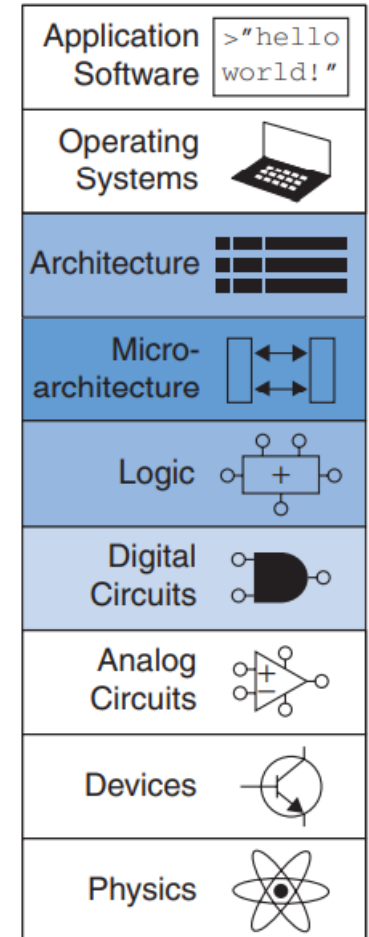
Prima parte

Autore principale

Introduzione

I progettisti di calcolatori vogliono definire un linguaggio che renda semplice

- costruire l'hardware
- scrivere i compilatori (o i programmi)
- modificare l'hardware (i.e. estenderne le funzionalità)
- massimizzare le prestazioni
- minimizzare i costi (anche energetici)



Architettura dei Calcolatori

Ricordiamo che:

Computer Architecture =
insieme delle istruzioni (linguaggio, ISA) +
organizzazione interna (registri, memorie...)

Un'architettura non definisce l'implementazione HW, ne
descrive le funzionalità.

Diversi microprocessori rappresentano l'implementazione
di una architettura e, tipicamente, propongono un
trade-off tra costo, prestazioni, consumo energetico.

Microarchitetture e ABI

Una microarchitettura definisce le connessioni tra registri e memorie, quante e quali ALU, e altri componenti (building block) che formano un microprocessore (e.g. taglia e numero di livelli di cache).

Un'architettura, diversi vendor.

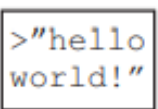


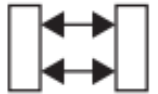
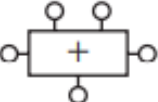

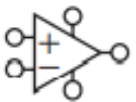


Application Binary Interface:
ISA + interfaccia del sistema operativo
https://wiki.gentoo.org/wiki/RISC-V_ABIs

ilp32

- int, long, pointers are 32bit
- long long is 64bit
- char is 8bit
- short is 16bit

lp64

- int is 32bit
- long and pointers are 64bit
- long long is 64bit
- char is 8bit
- short is 16bit

Application Software		Programs
Operating Systems		Device Drivers
Architecture		Instructions Registers
Micro-architecture		Datapaths Controllers
Logic		Adders Memories
Digital Circuits		AND Gates NOT Gates
Analog Circuits		Amplifiers Filters
Devices		Transistors Diodes
Physics		Electrons

RISC e CISC

- **Complex Instruction Set Computing (CISC)** e **Reduced Instruction Set Computing (RISC)** sono le due principali classi di architetture
- **RISC** ha un insieme semplificato di operazioni che mirano a raggiungere obiettivi semplici in pochi cicli.
 - La CPU non ha bisogno di circuiti complessi per eseguire queste istruzioni, quindi potenzialmente sono più economiche e semplici da implementare.
- **CISC** è l'esatto opposto
 - In passato i programmatori scrivevano in Assembler
- I processori moderni AMD ed INTEL incorporano le filosofie di progettazione di entrambe le architetture.
 - L'architettura x86 è principalmente CISC ma ha un **microcodice** per convertire istruzioni complesse in semplici istruzioni ridotte di tipo RISC.
 - Retrocompatibilità

- Il progetto nasce nel 2010 dall'Università della California, Berkeley
- Standard APERTO per un'ISA basata su RISC. I principi:
 1. La regolarità favorisce la semplicità
Ad esempio il numero degli operandi
 2. Il caso più frequente dev'essere anche il più veloce
 3. Più piccolo, più veloce
Pensiamo all'UTF-8 rispetto all'Unicode.
Qui pochi registri e poche istruzioni semplici
 4. Un buon progetto richiede compromessi astuti
Tutte le istruzioni hanno la stessa lunghezza, 32 bit.
Ma lo standard prevede possibili estensioni

5.

xxxxxxxxxxxxxxxxxxxx	xxxxxxxxxxxxbbb11	32-bit (bbb \neq 111)
----------------------	-------------------	-------------------------

...xxxx	xxxxxxxxxxxxxxxxxxxx	xxxxxxxxxxx011111	48-bit
---------	----------------------	-------------------	--------

...xxxx	xxxxxxxxxxxxxxxxxxxx	xxxxxxxxxxx011111	64-bit
---------	----------------------	-------------------	--------

Parola e doppia parola in RISC-V

- I gruppi di 32 bit sono così comuni in RISC-V che prendono il nome di **parola (word)**
- La **parola doppia (doubleword)** corrisponde quindi a 64 bit
- Sigle collegate:

Base ISAs

Name	Description
RV32I	32-bit integer instruction set
RV32E	32-bit integer instruction set for embedded microprocessors
RV64I	64-bit integer instruction set
RV128I	128-bit integer instruction set

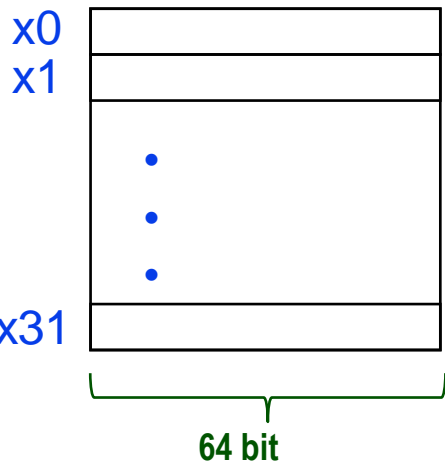
Sigle

Nome simbolico	Descrizione	Numero di istruzioni
I	Architettura di base	51
M	Moltiplicazione/divisione intera	13
A	Operazioni atomiche	22
F	Virgola mobile in precisione singola	30
D	Virgola mobile in doppia precisione	32
C	Istruzioni compresse	36
G	Shorthand for the base and above extensions	
Q	Standard extension for quad-precision floating-point	
L	Standard extension for decimal floating-point	
C	Standard extension for compressed instructions	
B	Standard extension for bit manipulation	
J	Standard extension for dynamically translated languages	
T	Standard extension for transactional memory	
P	Standard extension for packed-SIMD instructions	
V	Standard extension for vector operations	
N	Standard extension for user-level interrupts	
H	Standard extension for hypervisor	

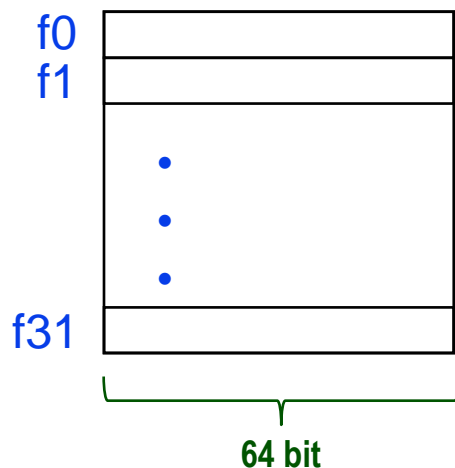
"Instruction Set Architecture" (es. RISC-V RV64IMDF)

Registri e memoria

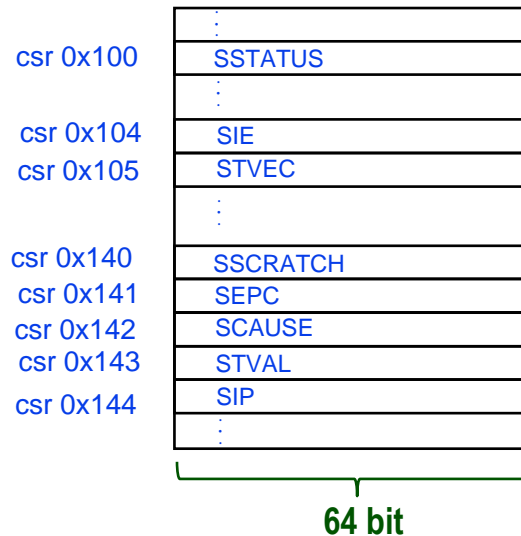
32 Registri
numeri interi (INT)



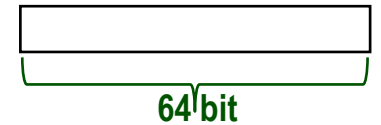
32 Registri
Floating Point (FP)



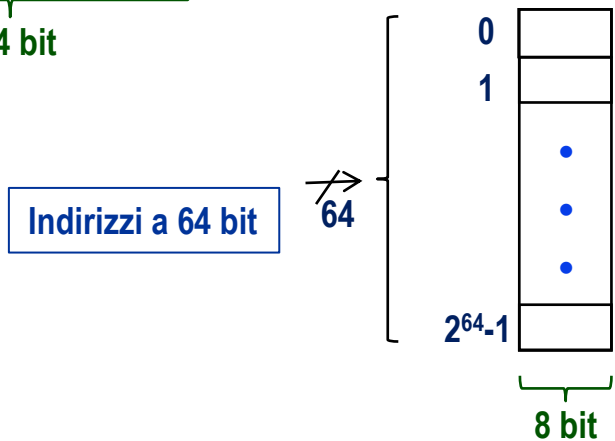
Vari Registri Speciali



Il Program Counter
PC



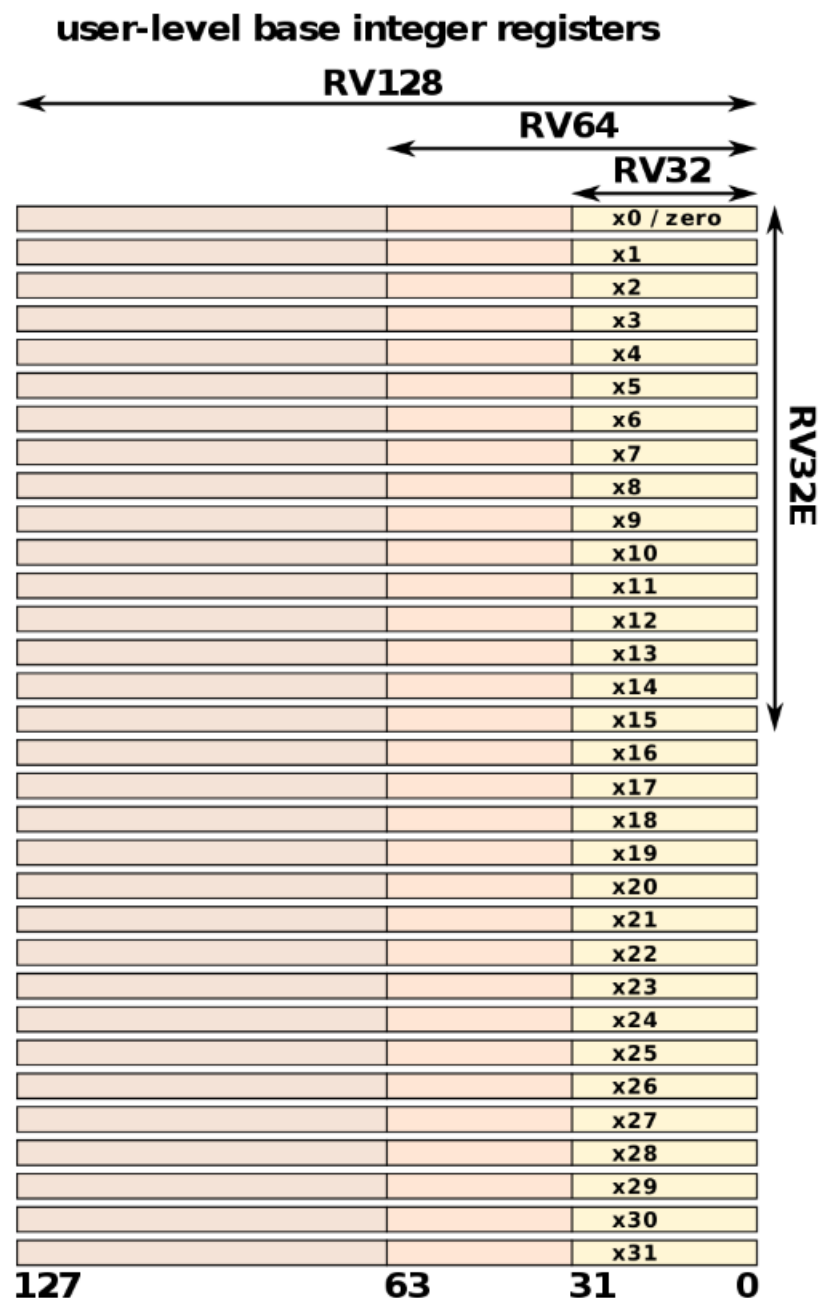
Memoria e I/O



I registri

Register	Alias	Description
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5-x7	t0-t2	Temporaries 0 to 2
x8	s0/fp	Saved register 0 / Frame pointer
x9	s1	Saved register 1
x10-17	a0-a7	Function arguments 0 to 7
x18-27	s2-s11	Saved registers 2 to 11
x28-31	t3-t6	Temporaries 3 to 6
pc	pc	Program counter

Più registri possono allungare il ciclo di clock, perché il segnali elettrici richiedono più tempo per compiere il percorso .
 Inoltre 32 = 5 bit. Su 32 bit prevederne un numero maggiore sarebbe un potenziale spreco/problema.
 NOTA: se RV64F o D ho anche i 32 registri FP



Perché i registri sono «disordinati»?

Perché RV32E ne ha solo 15:

- 3 temporanei
- 2 saved
- 6 per argomenti di funzione

...oltre al PC, IR e similari

Register	ABI Name	Description	Saved by Calle-
x0	zero	hardwired zero	-
x1	ra	return address	-R
x2	sp	stack pointer	-E
x3	gp	global pointer	-
x4	tp	thread pointer	-
x5	t0	temporary register 0	-R
x6	t1	temporary register 1	-R
x7	t2	temporary register 2	-R
x8	s0 / fp	saved register 0 / frame pointer	-E
x9	s1	saved register 1	-E
x10	a0	function argument 0 / return value 0	-R
x11	a1	function argument 1 / return value 1	-R
x12	a2	function argument 2	-R
x13	a3	function argument 3	-R
x14	a4	function argument 4	-R
x15	a5	function argument 5	-R

Tipi di dato

Una parola è un gruppo di bit di una determinata dimensione che sono gestiti come unità dall'ISA.

C datatype	RV32I native datatype	size in bytes
bool	byte	1
char	byte	1
unsigned char	unsigned byte	1
short	halfword	2
unsigned short	unsigned halfword	2
int	word	4
unsigned int	unsigned word	4
long	word	4
unsigned long	unsigned word	4
void*	unsigned word	4

“Instruction Set Architecture” (es. RISC-V RV64IMDF)

- Categorie di Istruzioni
 - Aritmetiche
 - Load/Store
 - Jump/Branch
 - Istruzioni speciali
- Formato delle Istruzioni RV64IMDF:
 - La lunghezza delle istruzioni e' FISSA a 32 bit

Tipo di istruzioni	Istruzioni	Esempio	Significato	Commenti
Aritmetiche	Somma	<code>add x5, x6, x7</code>	$x5 = x6 + x7$	Operandi in tre registri
	Sottrazione	<code>sub x5, x6, x7</code>	$x5 = x6 - x7$	Operandi in tre registri
	Somma immediata	<code>addi x5, x6, 20</code>	$x5 = x6 + 20$	Utilizzata per sommare delle costanti

- Tutte le istruzioni aritmetiche hanno 3 operandi
- L'ordine degli operandi è fisso
(il primo e' sempre l'operando destinazione)

Se A memorizzata nel registro s0,
B in s1 e C in s2

C code: $A = B + C$

RISC-V code: `add s0, s1, s2`

Istruzioni Aritmetiche RISC-V (2)

Principio di progetto: "la semplicità favorisce la regolarità". Naturalmente questo complica qualcosa...

C code:

```
A = B + C + D;  
E = F - A;
```

RISC-V code:

```
add t0, s1, s2  
add s0, t0, s3  
sub s4, s5, s0
```

Se s0:A, s1:B, s2:C, s3:D, s4:E, s5:F

Ricordiamoci che si hanno solo 32 registri (per gli interi) a disposizione

Cosa accade se il programma ha moltissime variabili
o fa accesso a strutture dati complesse (es. vettori)?

Istruzioni di accesso alla memoria

La memoria può essere vista come un vettore monodimensionale, con base + offset

Come scrivo l'offset?

La memoria è byte-addressable, cioè, posso indirizzare un byte alla volta

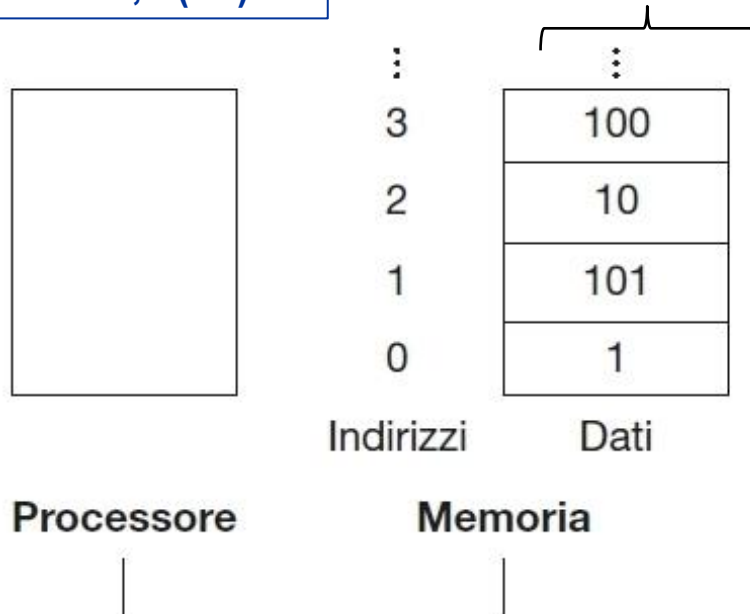
Trasferimento dati	Lettura parola	lw x5, 40(x6) <small>load word unsigned</small>	x5 = Memoria[x6 + 40]	Spostamento di una parola da memoria a registro
	Lettura parola, senza segno	lwu x5, 40(x6)	x5 = Memoria[x6+40]	Spostamento di una parola senza segno da memoria a registro
	Memorizzazione parola	sw x5, 40(x6)	Memoria[x6+40] = x5	Spostamento di una parola da registro a memoria

lwu: sposta una parola nel registro ed estende con 0 a sinistra
lw: estende con 0 se positivo, con 1 se negativo.

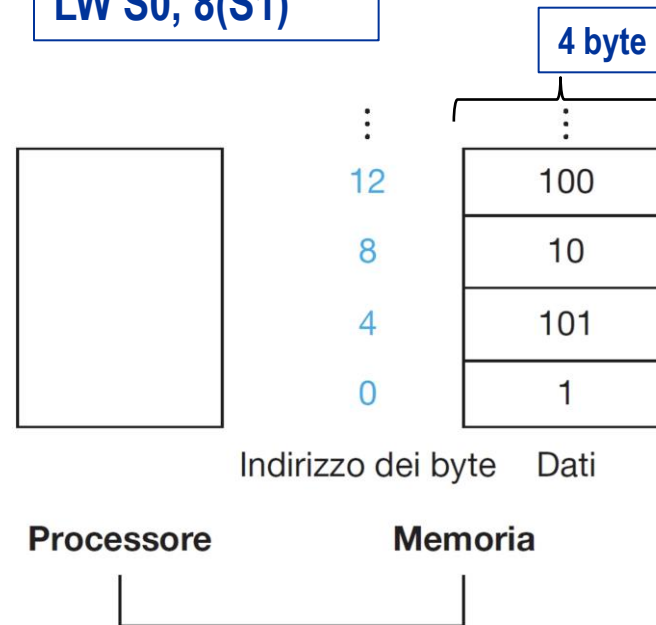
Istruzioni di accesso alla memoria

Se devo caricare $A[2]$, ovvero 10, nella variabile $S0$, avendo A in $S1$:

LW $S0, 2(S1)$



LW $S0, 8(S1)$



Indirizzamento per indirizzo di parola

Indirizzamento **EFFETTIVO** in RISC-V,
per byte (offset= $2 \times 4 \text{ byte} = 8 \text{ byte}$)

C code:

```
g = h + A[8];
```

RISC-V code:

x22 contiene l'indirizzo di base di A

x21 contiene il valore di h

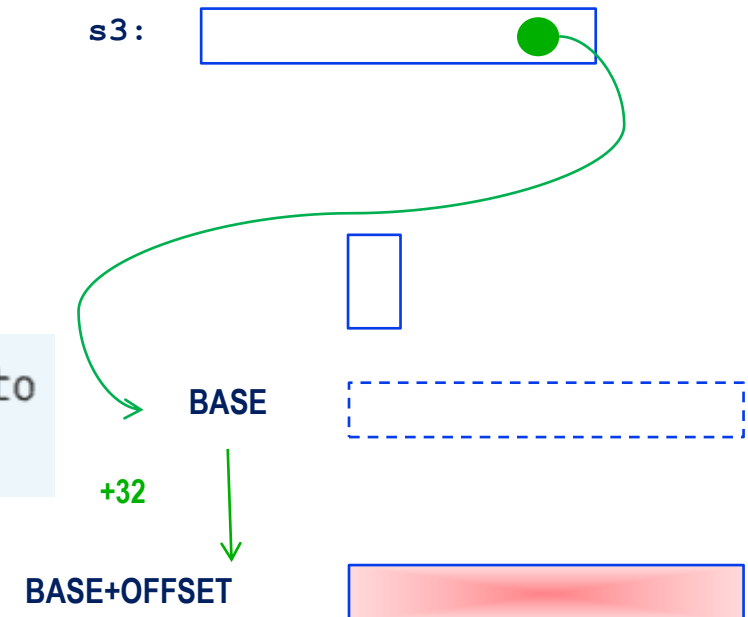
x30 contiene il valore di g

```
lw x9, 8(x22) // il valore di A[8] viene caricato
               // nel registro temporaneo x9
```

```
add x20, x21, x9 // g = h + A[8]
```

NOTE:

- Gli operandi di istruzioni Aritmetiche sono SEMPRE REGISTRI, mai celle di memoria! → **NEL RISC-V, NON ESISTE add t0, t1, 32(s3)**



Istruzioni di accesso alla memoria

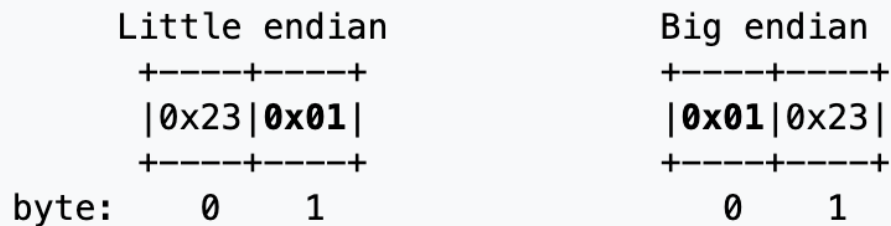
- In generale, dato un array $V[]$, per ottenere l'indirizzo di $V[x]$, cioè la cella in posizione x di V , devo considerare
 - $X*8$ se doubleword
 - $X*4$ se word
 - $X*2$ se halfword
 - X se byte

Istruzioni di accesso alla memoria

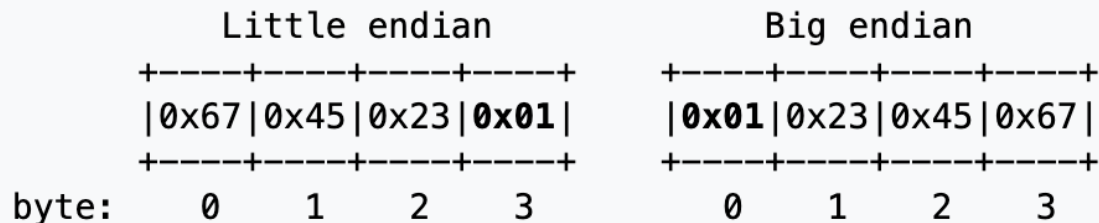
Uso byte, altrimenti non potrei indirizzare parole, mezze parole e byte.

Attenzione: Diversi modi di memorizzare i dati (es Little/Big Endian):

Nel caso di una WORD (16 bit), il numero **esadecimale** 0x0123 viene immagazzinato come:



Nel caso di una DWORD (32 bit), il numero **esadecimale** 0x01234567 verrà immagazzinato come:



Risc-V è Little Endian ma ad esempio PowerPc è Big Endian

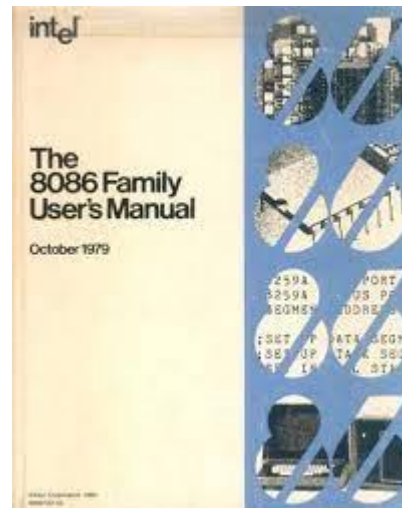
Ricordiamo

Codifica binaria (4 bit)	↔	Cifra esadecimale	↔	Valore (decimale)
0000	↔	0	↔	0
0001	↔	1	↔	1
0010	↔	2	↔	2
0011	↔	3	↔	3
0100	↔	4	↔	4
0101	↔	5	↔	5
0110	↔	6	↔	6
0111	↔	7	↔	7
1000	↔	8	↔	8
1001	↔	9	↔	9
1010	↔	A	↔	10
1011	↔	B	↔	11
1100	↔	C	↔	12
1101	↔	D	↔	13
1110	↔	E	↔	14
1111	↔	F	↔	15

Allocazioni

- In **Risc-V** le variabili devono essere allineate alla PAROLA (4 byte)
- In **x86** (purtroppo) no
- Es Char c, int i

7	6	5	4	3	2	1	0
			I				C
						I	C



Storage Organization

From a storage point of view, the 8086 and 8088 memory spaces are organized as identical arrays of 8-bit bytes (see figure 2-11). Instructions, byte data and word data may be freely stored at any byte address without regard for alignment thereby saving memory space by allowing code to be densely packed in memory (see figure 2-12). Odd-addressed (unaligned) word variables, however,

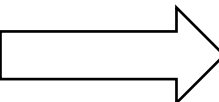
Load e Store

- Devo «riempire» i 64 bit del registro.
- Estendo il segno oppure uso 0 se unsigned

Complemento a 2

4 = 0..0100

-4 = 1..1100

	Trasferimento dati	Lettura parola doppia	ld x5, 40(x6)	x5 = Memoria[x6 + 40]	Spostamento di una parola doppia da memoria a registro
		Memorizzazione parola doppia	sd x5, 40(x6)	Memoria[x6 + 40] = x5	Spostamento di una parola doppia da registro a memoria
		Lettura parola	lw x5, 40(x6)	x5 = Memoria[x6 + 40]	Spostamento di una parola da memoria a registro
		Lettura parola senza segno	lwu x5, 40(x6)	x5 = Memoria[x6+40]	Spostamento di una parola senza segno da memoria a registro
		Memorizzazione parola	sw x5, 40(x6)	Memoria[x6+40] = x5	Spostamento di una parola da registro a memoria
		Lettura mezza parola	lh x5, 40(x6)	x5 = Memoria[x6+40]	Spostamento di una mezza parola da memoria a registro
		Lettura mezza parola, senza segno	lhu x5, 40(x6)	x5 = Memoria[x6+40]	Spostamento di una mezza parola senza segno da memoria a registro
		Memorizzazione mezza parola	sh x5, 40(x6)	Memoria[x6+40] = x5	Spostamento di una mezza parola da registro a memoria
		Lettura byte	lb x5, 40(x6)	x5 = Memoria[x6+40]	Spostamento di un byte da memoria a registro
		Lettura byte, senza segno	lbu x5, 40(x6)	x5 = Memoria[x6+40]	Spostamento di un byte senza segno da memoria a registro
		Memorizzazione byte	sb x5, 40(x6)	Memoria[x6+40] = x5	Spostamento di un byte da registro a memoria

Esempi

Consideriamo dati memorizzati in doubleword (8 byte)

1. Rappresentare in RISC-V la seguente istruzione
 $A[12] = h + A[8]$; con h in $x21$ e A in $x22$
2. Scambio di due elementi di un vettore, es $v[k] \leftrightarrow v[k+1]$
Con i in $x21$, j in $x22$ e v in $x23$

Esempi

Consideriamo dati memorizzati in doubleword (8 byte)

1) Rappresentare in RISC-V la seguente istruzione

$A[12] = h + A[8]$; con h in $x21$ e A in $x22$

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

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

```
sd  x9, 96(x22)
```

2) Scambio di due elementi di un vettore, es scambiare $A[k]$ e $A[k+1]$

Con A in $s1$, k in $s0$

```
add t2, s0, s0 // k*2
```

```
add t2, t2, t2 // k*4
```

```
add t2, t2, t2 // k*8
```

```
add t3, s1, t2 // base+8*k
```

```
ld  t4, 0(t3) // A[k]
```

```
ld  t5, 8(t3) // A[k+1]
```

```
sd  t5, 0(t3)
```

```
sd  t4, 8(t3)
```

Complicato, ma la load vuole come sorgente un registro ed una costante

- Le istruzioni, come i registri (se RV32) e le word sono a 32 bit

add x9 x20 x21

QUESTO FORMATO E' CHIAMATO "R"

- Formato dell'istruzione «add»:

La rappresentazione decimale è:

0	21	20	0	9	51
---	----	----	---	---	----

Ciascuna delle parti che costituiscono un'istruzione è chiamato *campo* (*field*). La combinazione del primo, del quarto e del sesto campo (che in questo caso contengono rispettivamente 0, 0 e 51) indica al processore RISC-V che questa istruzione esegue una somma. Il secondo campo indica il numero del registro che contiene il primo operando (21 = x21), il terzo campo il secondo operando della somma (20 = x20) e il quinto campo contiene il numero del registro in cui viene memorizzata la somma (9 = x9). Questa istruzione dunque somma il contenuto dei registri x21 e x20 e pone il risultato in x9.

Questa stessa istruzione può anche essere rappresentata come sequenza di campi contenenti numeri binari invece che decimali:

0000000	10101	10100	000	01001	0110011
7 bit	5 bit	5 bit	3 bit	5 bit	7 bit

funz7	rs2	rs1	funz3	rd	codop
7 bit	5 bit	5 bit	3 bit	5 bit	7 bit

Il significato del nome dato a ciascun campo è il seguente:

- *codop*: operazione base dell'istruzione, tradizionalmente chiamata **codice operativo** o **codop** (da *operating code*);
- *rd*: registro destinazione: riceve il risultato dell'operazione;
- *funz3*: un codice operativo aggiuntivo;
- *rs1*: registro contenente il primo operando sorgente;
- *rs2*: registro contenente il secondo operando sorgente;
- *funz7*: un codice operativo aggiuntivo.

• Il formato R è usato es. anche dalle istruzioni: **sub, and, or, xor**

Istruzioni

Nome (dimensione del campo)	Campi						Commenti
	7 bit	5 bit	5 bit	3 bit	5 bit	7 bit	
Tipo R	funz7	rs2	rs1	funz3	rd	codop	Istruzioni aritmetiche
Tipo I	Immediato[11:0]		rs1	funz3	rd	codop	Istruzioni di caricamento dalla memoria e aritmetica con costanti
Tipo S	immed[11:5]	rs2	rs1	funz3	immed[4:0]	codop	Istruzioni di trasferimento alla memoria (store)
Tipo SB	immed[12, 10:5]	rs2	rs1	funz3	immed[4:1,11]	codop	Istruzioni di salto condizionato
Tipo UJ	immediato[20, 10:1, 11, 19:12]				rd	codop	Istruzioni di salto incondizionato
Tipo U	immediato[31:12]				rd	codop	Formato caricamento stringhe di bit più significativi

Figura 2.19 Formati delle istruzioni RISC-V.

Formato	Istruzione	Codice Operativo	funz3	funz6/7
Tipo R	add	0110011	000	0000000
	sub	0110011	000	0100000
	sll	0110011	001	0000000
	xor	0110011	100	0000000
	srl	0110011	101	0000000
	sra	0110011	101	0000000
	or	0110011	110	0000000
	and	0110011	111	0000000
	lr.d	0110011	011	0001000
	sc.d	0110011	011	0001100

Guardate la differenza tra
ADD
SUB
XOR

ADDI
(non esiste SUBI)
XORI

Linguaggio Assembly e Linguaggio Macchina

- Il linguaggio Assembly fornisce una conveniente rappresentazione simbolica
 - È più comprensibile (a noi) che sequenze di numeri (binari)
- Il **linguaggio macchina** è la “realtà sottostante”
 - La macchina comprende solo le istruzioni in formato binario...
- Il linguaggio Assembly fornisce “**pseudoistruzioni**”
 - es., “mv t0, t1” esiste solo in Assembly
 - l'assemblatore traduce usando “add t0,t1,zero”
- Quando si effettua per es. il debugging è necessario ricordare quali siano le istruzioni reali
 - In genere si usa -g in fase di compilazione

Esempio

```
# hellocalc.cpp:6:      c = a+b;
    movl    -12(%rbp), %edx # a, tmp89
    movl    -8(%rbp), %eax  # b, tmp90
    addl    %edx, %eax      # tmp89, tmp88
    movl    %eax, -4(%rbp)  # tmp88, c
```

```
g++ -S -fverbose-asm hellocalc.cpp
vi hellocalc.s
```

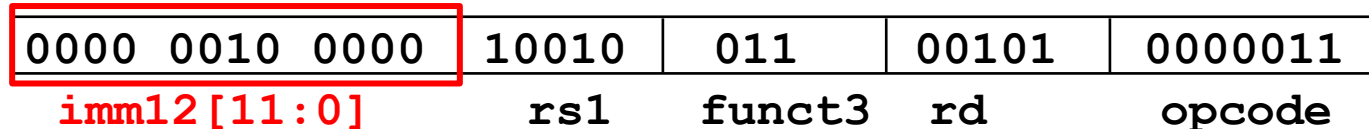
Altri compilatori <https://godbolt.org/z/hMdKPc68q>

```
dago@workstation:~$ cat hellocalc.cpp
#include <iostream>
int main(int argc, char *argv[])
{
    int a = 4, b, c;
    b = 5;
    c = a+b;
    std::cout << "Hello World! " << c << "\n";
    exit(0);
}
```

Accesso alla memoria e costanti

- Per fare accesso alla memoria, si usano solo due istruzioni: **load** e **store**
 - Es. di load: `ld t0, 32(s2)` (l sta per load e d sta per dword ovvero 64 bit)
 - Es. di store: `sd t0, 32(s2)` (s sta per store e d sta per dword ovvero 64 bit)
 - Qua t0 indica sorgente (load) o destinazione (store) in un registro
 - «32(s2)» indica l'indirizzo di memoria sommando «32» al contenuto del registro s2
- Devo quindi essere in grado di memorizzare da qualche parte il numero «32»
 - Cosa dovremmo fare in base al “principio di regolarità”?
 - Nuovo principio: “un progetto ottimo richiede compromessi”
- Introduciamo quindi un nuovo tipo di formato per le istruzioni (cambiando il meno possibile, ovvero la parte evidenziata in **rosso**)

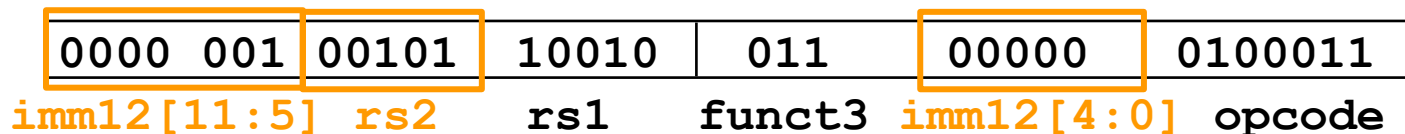
Es.: `ld t0, 32(s2)`



QUESTO FORMATO
E' CHIAMATO “I”

- Per la store pero', volendo rispettare il fatto che t0 stavolta e' un registro sorgente (rs2), che si deve trovare nello stesso punto della codifica, occorre riferirsi ad un formato leggermente **diverso**:

Es.: `sd t0, 32(s2)`



QUESTO FORMATO
E' CHIAMATO “S”

Costanti e istruzioni «con immediato»

- Le costanti “piccole” sono le più usate (50% dei casi), es.:

```
A = A + 5;  
B = B + 1;  
C = C - 18;
```

- Approccio RISC:

- Mettere le costanti “piccole” nell'istruzione
- Mettere le costanti meno frequenti in memoria
- Creare un registro hard-wired (x0) per la costante 0

- Istruzioni:

```
addi s0, s0, 4  
slti t1, t2, 10    (t1 = 1 se t2 < 10)  
andi t1, t3, 0x001  
ori  t1, t5, 0xFF0  
xori t1, t4, 0x0F0
```

- per codificare queste istruzioni si usa ancora il **formato I**
(I sta per 'immediato': uno degli operandi è dentro l'istruzione stessa)
QUINDI la costante piccola è un numero da -2048 a 2047 (intero con segno su 12 bit)

Nota2: la “subi” non esiste... si fa con la addi coll'immediato negato

Costanti

- Le costanti sono codificate su 12 bit in complemento a due con estensione del segno a 32 o 64 bit.
- Le costanti possono essere scritte come valori decimali, esadecimali (iniziano con 0x) o binari (iniziano con 0b)
- Utili per inizializzare i registri

High-Level Code

```
i = 0;  
x = 2032;  
y = -78;
```

RISC-V Assembly Code

```
# s4 = i, s5 = x, s6 = y  
addi s4, zero, 0      # i = 0  
addi s5, zero, 2032    # x = 2032  
addi s6, zero, -78     # y = -78
```

Che si fa con le costanti "grandi"?

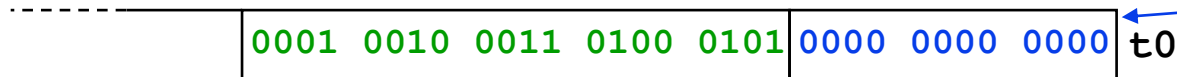
- Vorremo caricare ad es. costanti a 32 bit in un registro
 - Es. 0x12345678
 - La parte bassa (0x678) la potrei caricare con `ori t0, zero, 0x678`
 - Come carico però la parte alta di 20 bit (32 bit - 12 bit = 20 bit)?

1) Si introduce una nuova istruzione ("load upper immediate" = `lui`)

`lui t0, 0x12345`

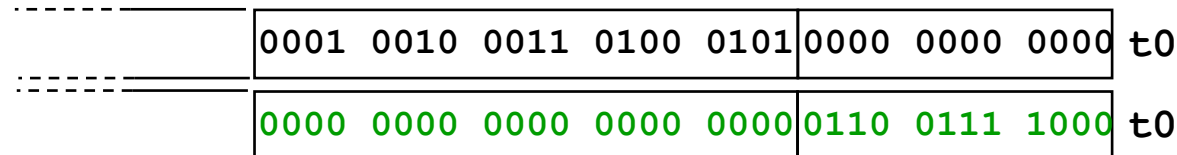
→ ho però bisogno di un nuovo formato U
(vedi slide successiva)

Riempita con zeri

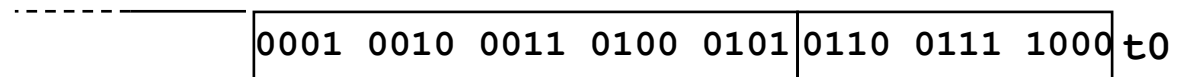


2) Si usa successivamente la `ori` per caricare la parte bassa (12 bit)

`ori t0, t0, 0x678`



`ori`



Formato U

Es.: lui t0, 0xFFFF



QUESTO FORMATO
È CHIAMATO "U"

20 bit che vanno a finire sui bit 31:12 del registro destinazione

Nota: per costanti più grandi (es. a 64 bit) si ripete due volte il ragionamento fatto per la costante a 32 bit, traslando verso sinistra (vedi più avanti l'istruzione per lo «shift» verso sinistra slli)

Altrimenti si estende i segno, i.e. il bit 32-esimo

Nome (dimensione del campo)	Campi						Commenti
	7 bit	5 bit	5 bit	3 bit	5 bit	7 bit	
Tipo R	funz7	rs2	rs1	funz3	rd	codop	Istruzioni aritmetiche
Tipo I	Immediato[11:0]		rs1	funz3	rd	codop	Istruzioni di caricamento dalla memoria e aritmetica con costanti
Tipo S	immed[11:5]	rs2	rs1	funz3	immed[4:0]	codop	Istruzioni di trasferimento alla memoria (store)
Tipo SB	immed[12, 10:5]	rs2	rs1	funz3	immed[4:1, 11]	codop	Istruzioni di salto condizionato
Tipo UJ	immediato[20, 10:1, 11, 19:12]				rd	codop	Istruzioni di salto incondizionato
Tipo U	immediato[31:12]				rd	codop	Formato caricamento stringhe di bit più significativi

Figura 2.19 Formati delle istruzioni RISC-V.

Esempio

- Carichiamo la costante 3.998.976 = 0...0 00111101 00000101 00000000 nel registro x19

Esempio

- Carichiamo la costante 00000000 00111101 00000101 00000000
- Passo 1: lui x19, 976
- Passo 2: addi x19, x19, 1280

In esadecimale è più chiaro: 3D0500

E se avessi avuto 4001024?

Esempio

- Con 4001024 ho 00000000 00111101 00001101 00000000
- Passo 1: lui x19, 976
- Passo 2: addi x19, x19, -1280

Devo sommare 1 alla costante in lui, quindi

$$3D0D00 = 0x3D1000 + 0xFFFD00$$

001111010001 000000000000 +

111111111111 110100000000 =

001111010000 1101 00000000

Pseudoistruzioni

- La coppia di istruzioni

```
lui t0, 20bit_alti  
ori t0, t0, 12bit_bassi
```

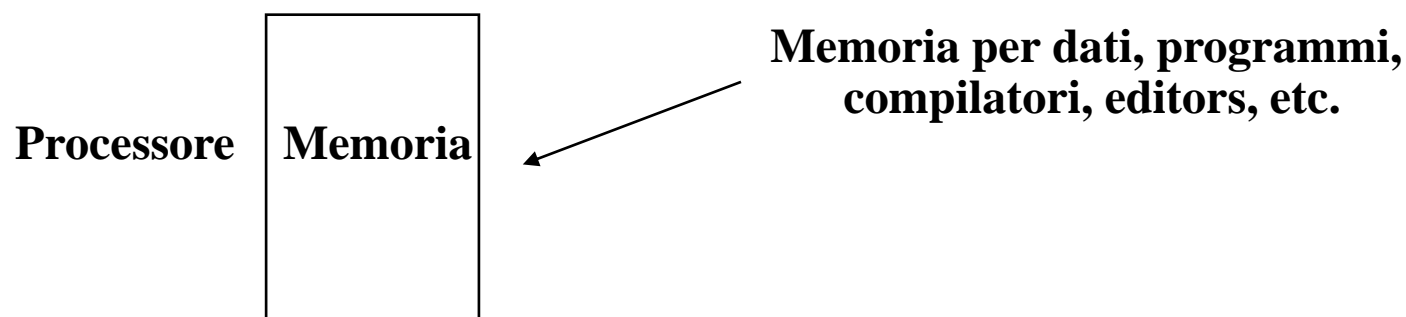
è talmente frequente che a livello di assembler si può creare una **'pseudoistruzione'** `la = "load address"`

- `la t0, costante_a_32_bit`

- dove `costante_a_32_bit := (20bit_alti | 12bit_bassi)`
- viene automaticamente tradotta nella coppia di istruzioni native `lui+ori`

- Nota: se la costante da caricare è a 64-bit sarà necessario effettuare `lui+ori+ sll t0, t0, 32 +lui+ori`
- Esistono altre pseudoistruzioni, che vedremo all'occorrenza

- Le istruzioni sono sequenze di bit (come i dati!)
- I programmi sono caricati in memoria
 - devono essere letti così come vanno letti i dati



- **Fetch & Execute Cycle**
 - Le istruzioni sono prelevate (fetched) e messe in un registro interno (non visibile al programmatore)
 - I bit in tale registro "controllano" le azioni successive del processore
 - Si preleva l'istruzione successiva e si continua

Spoiler

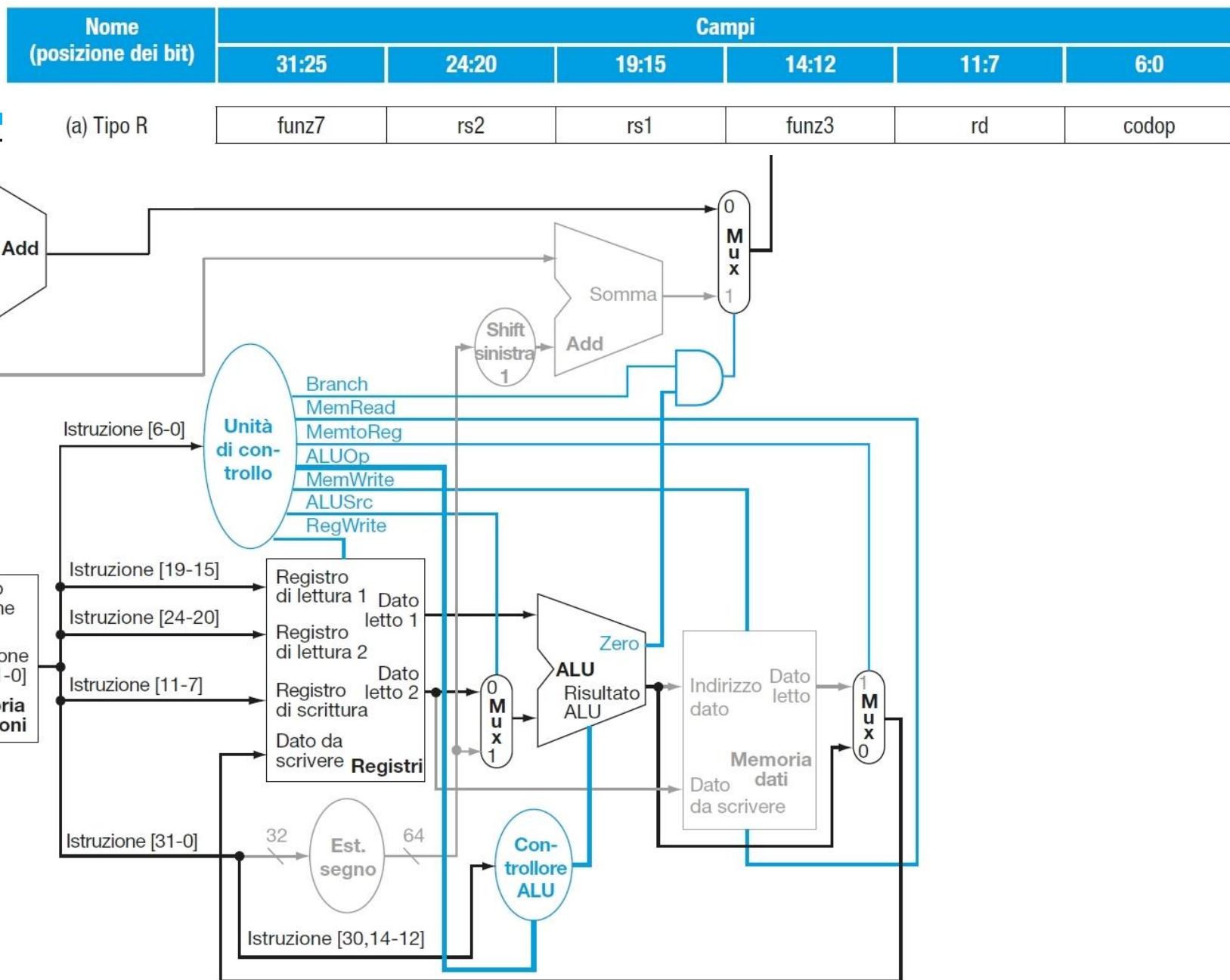
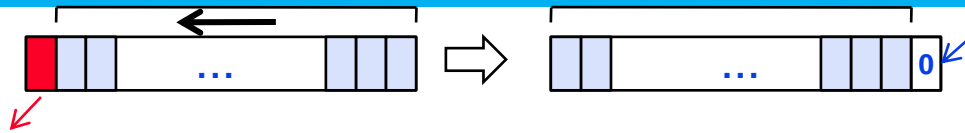
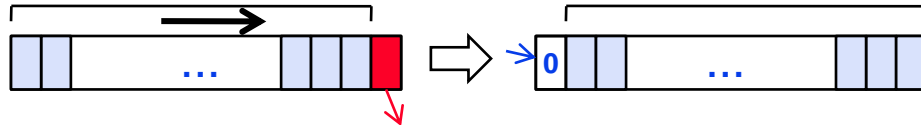


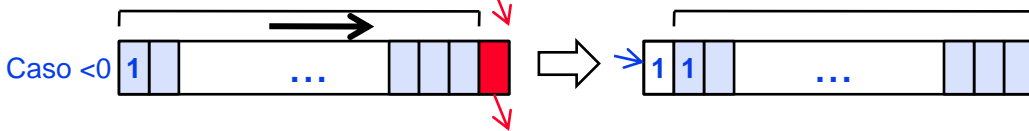
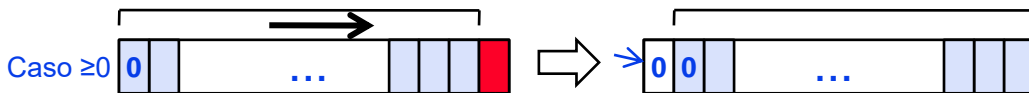
Figura 4.19 L'unità di elaborazione durante l'esecuzione di un'istruzione di tipo R, come `add x1, x2, x3`. Le linee di controllo, le unità funzionali dell'unità di elaborazione e le connessioni attive sono evidenziate in blu.



Shift Left Logical (SLL)



Shift Right Logical (SRL)



Shift Right Arithmetic (SRA)

Nota: SLA non esiste ...=SLL

	<i>Instruzione</i>	<i>Esempio</i>	<i>Significato</i>	<i>Commento</i>
Immediate	shift left logical	slli x5,x6,10	$x5 = x6 \ll 10$	Shift left by constant
	shift right logical	srli x5,x6,10	$x5 = x6 \gg 10$	Shift right by constant
	shift right arithm.	srai x5,x6,10	$x5 = x6 \gg 10$	Shift right (sign extend)
Register	shift left logical	sll x5,x6, x7	$x5 = x6 \ll x7$	Shift left by variable
	shift right logical	srl x5,x6, x7	$x5 = x6 \gg x7$	Shift right by variable
	shift right arithm.	sra x5,x6, x7	$x5 = x6 \gg x7$	Shift right arith. by variable

RISC-V: istruzioni logiche AND, OR, XOR – pseudoistruzione NOT

<i>Istruzione</i>	<i>Esempio</i>	<i>Significato</i>	<i>Commento</i>
and	and x5,x6,x7	$x5 = x6 \& x7$	3 reg. operands; Logical AND
or	or x5,x6,x7	$x5 = x6 x7$	3 reg. operands; Logical OR
xor	xor x5,x6,x7	$x5 = x6 \oplus x7$	3 reg. operands; Logical XOR
and immediate	andi x5,x6,10	$x5 = x6 \& 10$	Logical AND reg, constant
or immediate	ori x5,x6,10	$x5 = x6 10$	Logical OR reg, constant
xor immediate	xori x5, x6,10	$x5 = x6 \oplus 10$	Logical XOR reg, constant

- Con andi e 0 nei bit opportuni eseguo il mascheramento (nascondo)
- La NOT è una pseudoistruzione: perché?
 - Dalla teoria generale sull'algebra di Boole sappiamo che anche con la sola funzione NAND potremmo generare le altre funzioni quali NOT, AND, OR
 - Nel caso del processore RISC-V, seguendo l'approccio minimalista, si è deciso di non introdurre nuovi opcode per l'istruzione not
 - In particolare posso realizzare not x5, x6 usando invece:
xori x5, x6, -1 (il -1 è esteso, diventa 0xFFF...FF)

- Fino a qui abbiamo visto istruzioni che operano su quantità a 32 o 64 bit
- 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

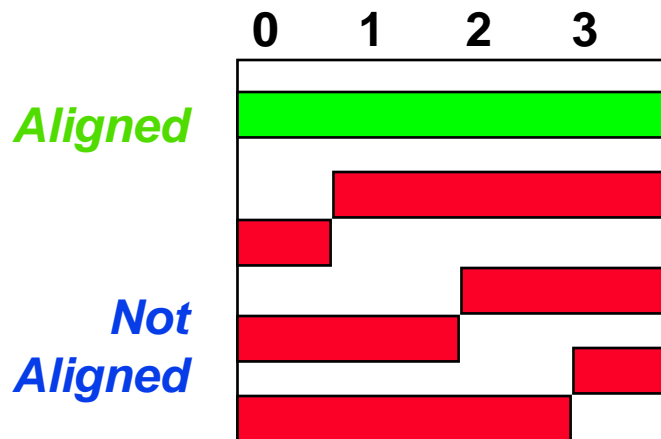
lw x5, 0(x6) "load word"

sw x5, 0(x6) "store half-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) → 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 genera un warning (avvertendo che il tempo per l'accesso al dato risulterà 2 volte più lento)

Ricapitolando

Vedi anche 3.5

Tipi del C	Tipi di Java	Trasferimento dati	Operazioni
long long int	long	ld, sd	add, sub, addi, mul, mulh, mulhu, mulhsu, div, divu, rem, remu, and, andi, or, ori, xor, xori
unsigned long long int	—	ld, sd	add, sub, addi, mul, mulh, mulhu, mulhsu, div, divu, rem, remu, and, andi, or, ori, xor, xori
char	—	lb, sb	add, sub, addi, mul, div, divu, rem, remu, and, andi, or, ori, xor, xori
short	char	lh, sh	add, sub, addi, mul, div, divu, rem, remu, and, andi, or, ori, xor, xori
float	float	flw, fsw	fadd.s, fsub.s, fmul.s, fdiv.s, feq.s, flt.s, fle.s
double	double	fld, fsd	fadd.d, fsub.d, fmul.d, fdiv.d, feq.d, flt.d, fle.d

Ricapitolando - Da Wikipedia

RV32IMAC

