Lezione 7
Il Set di Istruzioni RISC-V

architettura OPEN HARDWARE chiunque può implementare il suo design

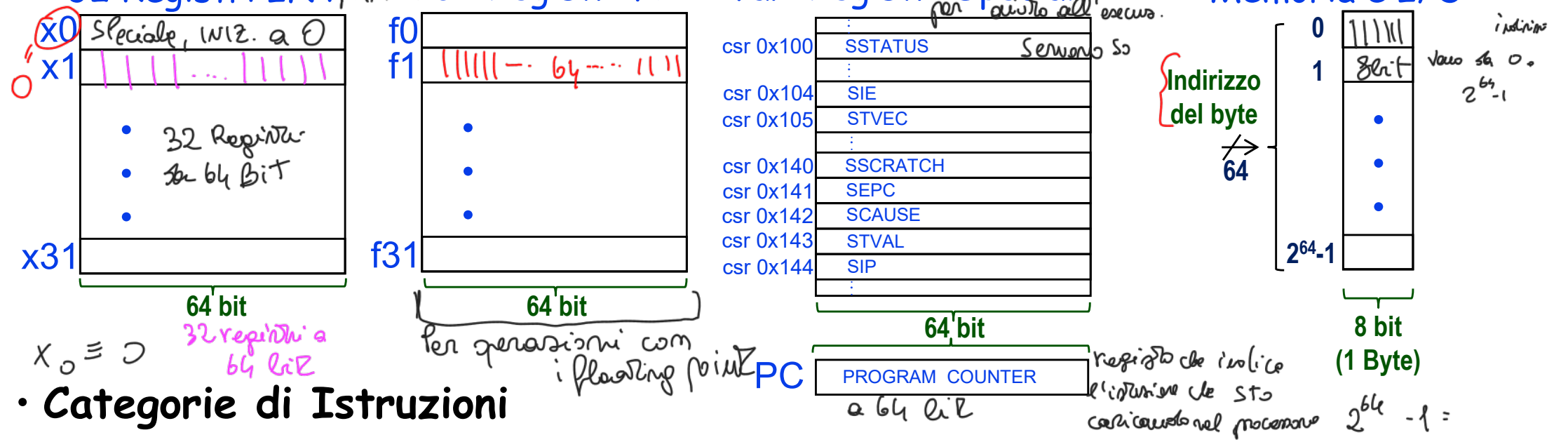
Definire Istruzioni = definire l'architettura stessa

RISC-V, istruzioni a 32 bit i quali operandi sono registri e locazioni di memoria (load/store)

Definire Istruzioni = definire l'architettura stessa, Sistema minimale per implementare a mano le

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

- **ISA = OPERANDI + ISTRUZIONI**
  - Le **istruzioni** sono codificate con un NUMERO a 32 bit o in altri termini la lunghezza delle istruzioni è FISSA a 32 bit
  - Gli **operandi** sono costituiti da «registri» o da «locazioni di memoria»
- **Noi esamineremo la specifica RV64IMDF* del processore RISC-V**



- **Categorie di Istruzioni**
  - **Load/Store, Calcolo (per Interi e per Frazionari), Jump/Branch, Istruzioni Speciali**

*Nota: "RV64IMDF" significa che il processore è un RISC-V (RV) a 64 bit, con il supporto per l'aritmetica sugli interi (I), la moltiplicazione e divisione (M) le operazioni floating point (singola precisione (F) e doppia precisione (D)

Memoria, enorme vettore in cui ogni elemento è a 8 bit, 1 byte

(Richiamo) Prefissi per i multipli di bit (b) e byte (B)

Valore decimale	Simbolo del Sistema Internazionale		
1000	10^3	k	kilo
1000^2	10^6	M	mega
1000^3	10^9	G	giga
1000^4	10^{12}	T	tera
1000^5	10^{15}	P	peta
1000^6	10^{18}	E	exa
1000^7	10^{21}	Z	zetta
1000^8	10^{24}	Y	yotta

Valore binario		Simbolo IEC	
1024	2^{10}	Ki	kibi
1024^2	2^{20}	Mi	mebi
1024^3	2^{30}	Gi	gibi
1024^4	2^{40}	Ti	tebi
1024^5	2^{50}	Pi	pebi
1024^6	2^{60}	Ei	exbi
1024^7	2^{70}	Zi	zebi
1024^8	2^{80}	Yi	yobi

Istruzioni Aritmetiche RISC-V (1)

Patterson 2.2

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

C code: `A = B + C`

RISC-V code: `add x5, x6, x7`

DESTINAZ. SORGENTI

operando destinazione

Sorgente 1, Sorgente 2

→ `x5≡A, x6≡B, x7≡C`

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;
```

→ x5≡A, x6≡B, x7≡C, x8=D, x9≡E, x10≡F

- RISC-V code:

```
add x11, x6, x7
add x5, x11, x8
sub x9, x10, x5
```

- Gli operandi delle istruzioni aritmetiche devono SEMPRE essere registri
- 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

Patterson 2.3

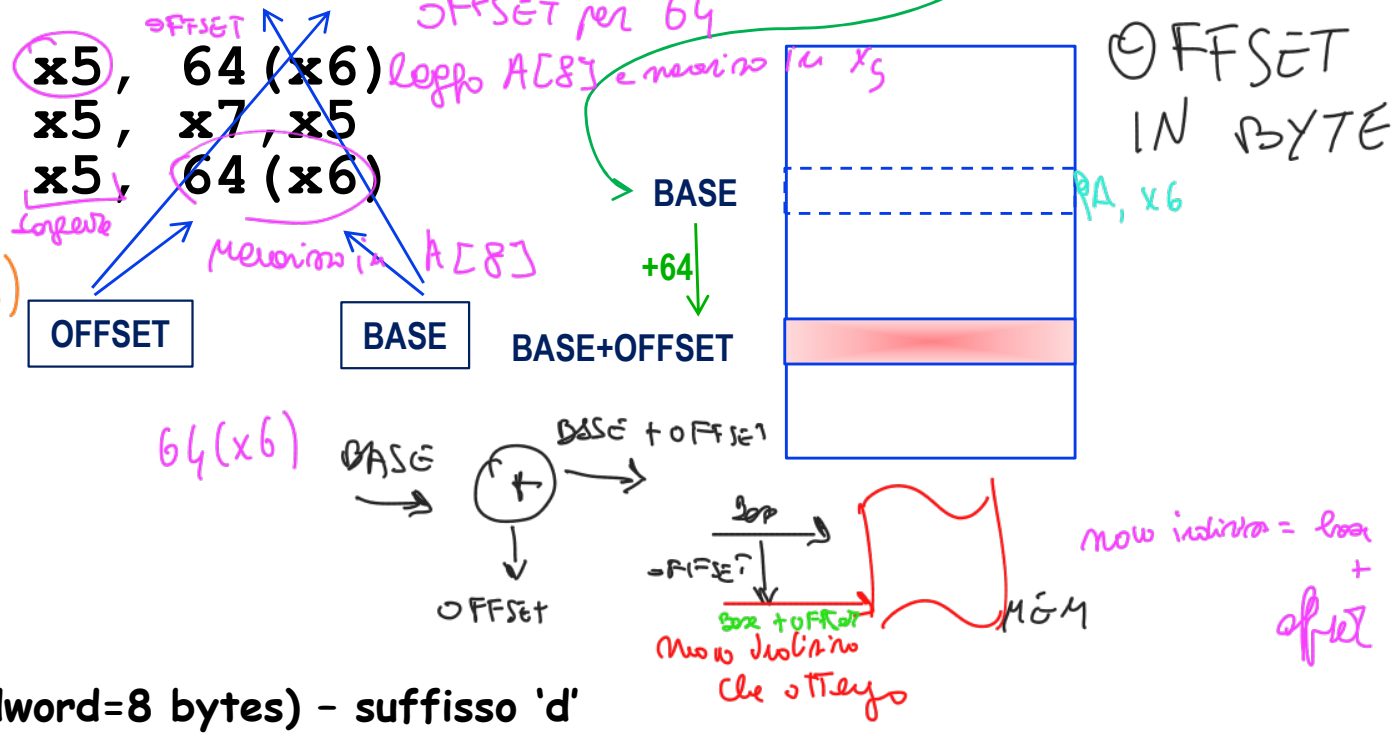
- **Istruzioni Load e Store**

Es.: x6≡A, x7≡h

C code: `A[8] = h + A[8];`

RISC-V code:

```
ld   x5, 64(x6)
add  x5, x7, x5
sd   x5, 64(x6)
```



- **NOTE:**
  - Il dato caricato è 64 bit (1 dword=8 bytes) – suffisso 'd'
  - Nella 'sd' (al contrario delle altre istruzioni con operandi), la destinazione (in memoria) è l'ultimo operando, anziché il primo
  - gli operandi di istruzioni Aritmetiche sono SEMPRE REGISTRI, mai celle di memoria! → NEL RISC-V, NON ESISTE add x5, x7, 64(x6)

Cosa fare se l'offset è dinamico?

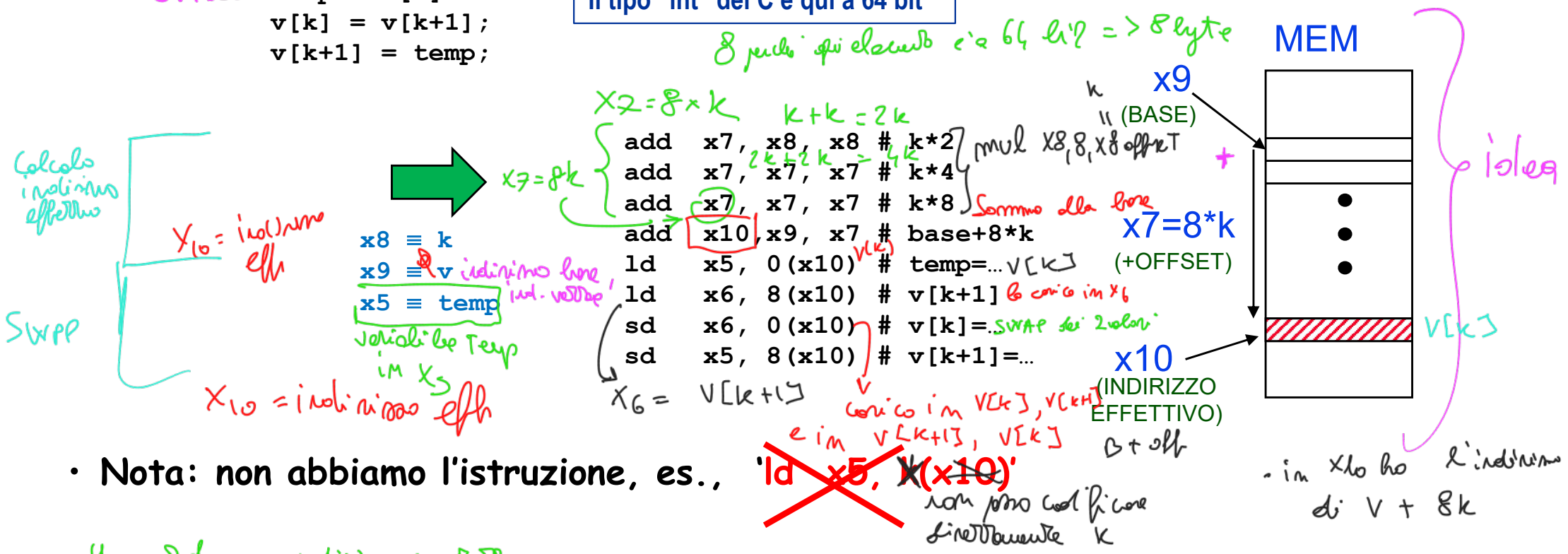
- **Esempio: scambio di due elementi di un vettore**

```
OFFSET  temp = v[k]
        v[k] = v[k+1];
        v[k+1] = temp;
```

Il tipo "int" del C è qui a 64 bit

```
        add  x7, x8, x8  # k*2
        add  x7, x7, x7  # k*4
        add  x7, x7, x7  # k*8
        add  x10,x9, x7  # base+8*k
        ld   x5, 0(x10)  # temp=...v[k]
        ld   x6, 8(x10)  # v[k+1]
        sd   x6, 0(x10)  # v[k]=...
        sd   x5, 8(x10)  # v[k+1]=...
```

```
x8 ≡ k
x9 ≡ v
x5 ≡ temp
```



- **Nota: non abbiamo l'istruzione, es., 'ld x5, X(x10)'**

Nota: v è un vettore, quindi quando (in C) scrivo "v" intendo implicitamente "l'indirizzo base del vettore v"

Linguaggio Macchina

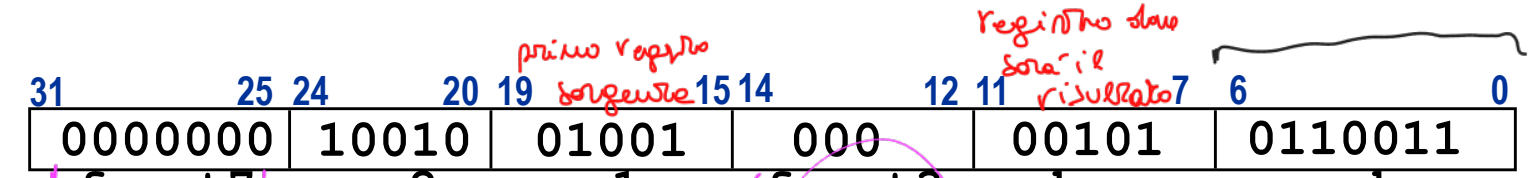
Codifico le istruzioni in vari formati, con numeri

Patterson 2.5

- Le istruzioni, ~~come i registri e le word~~ sono a 32 bit

Es.: `add x5, x9, x18`

- Formato dell'istruzione «add»:



QUESTO FORMATO è CHIAMATO "R"

- *Che cosa significano i nomi dei campi?*
  - *rs1 = first source register*
  - *rs2 = second source register*
  - *rd = destination register*
  - *Opcode = codice operativo*
  - *funct3,funct7 = codici funzione (estendono opcode)*

Per codificare il numero di registro ho bisogno di 5 bit

Opcode, determina il formato dell'istruzione e insieme a funct... (quando presente) cosa fa l'istruzione

- Il formato R è usato es. anche dalle istruzioni: `sub,and,or,xor,slt`

Accesso alla memoria

- Per fare accesso alla memoria, si usano solo due istruzioni: load e store
  - Es. di load: `ld x5, 32(x18)` (l sta per load e d sta per dword ovvero 64 bit)
  - Es. di store: `sd x5, 32(x18)` (s sta per store e d sta per dword ovvero 64 bit)
    - Qua x5 indica sorgente (load) o destinazione (store) in un registro
    - «32(x18)» indica l'indirizzo di memoria sommando «32» al contenuto del registro x18
- 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 x5, 32(x18)`

0000 0010 0000	10010	011	00101	0000011
imm12[11:0]	rs1	funct3	rd	opcode

QUESTO FORMATO è CHIAMATO "I"

- Per la store però, volendo rispettare il fatto che x5 stavolta è un registro sorgente (rs2), che si deve trovare nello stesso punto della codifica, occorre riferirsi ad un formato leggermente diverso:

Es.: `sd x5, 32(x18)`

0000 001	00101	10010	011	00000	0100011
imm12[11:5]	rs2	rs1	funct3	imm12[4:0]	opcode

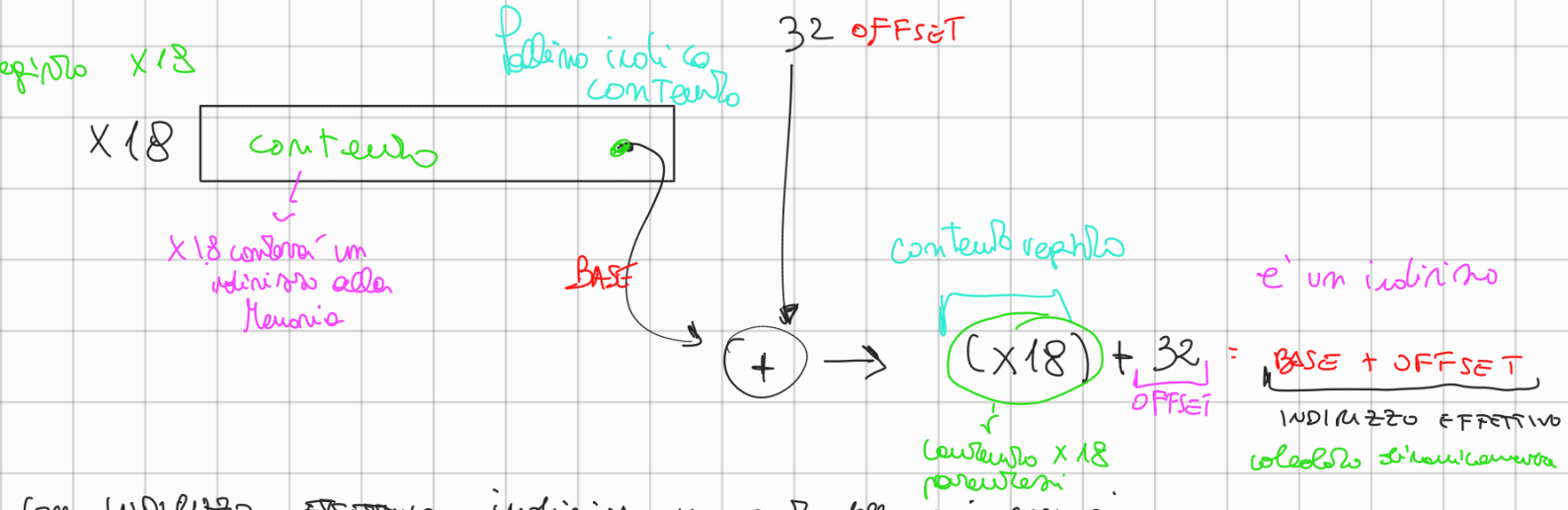
QUESTO FORMATO è CHIAMATO "S"

Che tipo di compromessi abbiamo fatto?

la destinazione oramai è in memoria, non ho rol, lo uso per IMM

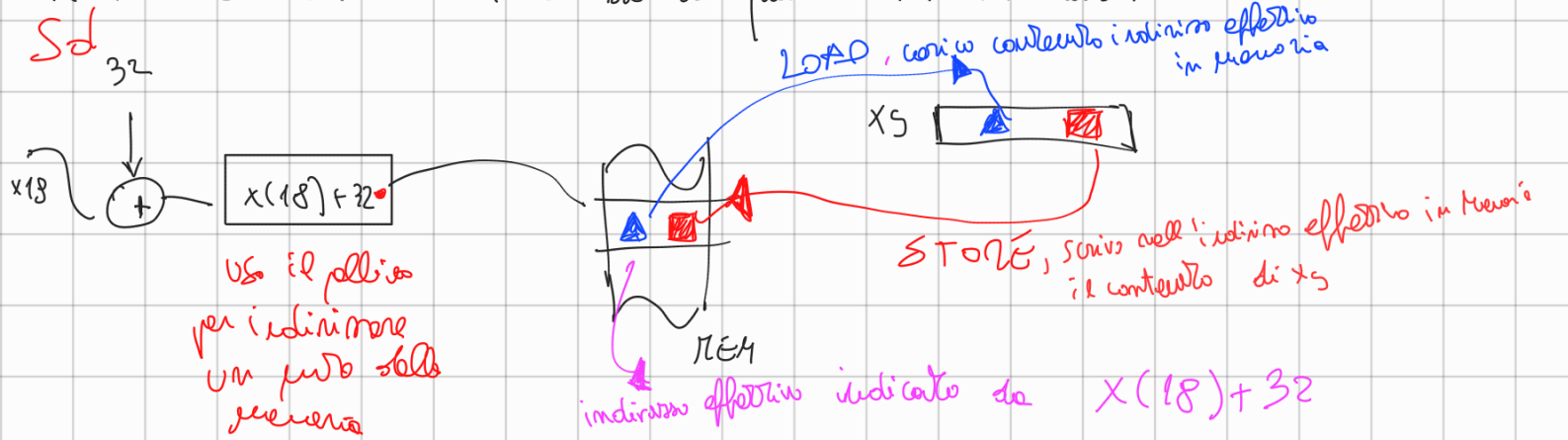
lol x_5, 32(x_{18}) cioè voglio caricare nel registro x_5 il valore che è in memoria all'indirizzo EFFETTIVO $x_{18} + 32$

allora meno 4 cose nella stessa posizione, ma allora sarebbero invertendo



Con INDIRIZZO ~~EFFETTIVO~~ indirizzo un punto della mia memoria

Sd 32



LE OPERAZIONI LOAD & STORE vengono fatte da un determinato INDIRIZZO ~~EFFETTIVO~~ calcolato dinamicamente facendo la somma tra il contenuto di X18 + 32

l'immediato va salvato da qualche parte nell'istruzione

Sd x_5 32(x_{18}) ora x_5 è l'operando sorgente voglio mettere ciò che è in x_5 all'indirizzo effettivo 32(x_{18}) in memoria

---

if(condizione) : uso una operazione

branch (diramazione | salto) Salto a LABEL non banale

Come cambio il flusso di controllo, p. es. gli if — BRENCH — Cambio FLUSSO programma con Branch — Patterson 2.7

Controllo del flusso del programma: salti condizionati e «formato B»

- L'architettura di Von Neumann segue il ciclo «fetch-and-execute» ovvero vengono prelevate in sequenza le istruzioni che si trovano in memoria.
- Il flusso sequenziale delle istruzioni può però essere alterato in conseguenza di una decisione del programma → la macchina supporta questo meccanismo con le istruzioni:

branch not equal — se falso, salto a punto del programma LABEL

```
bne x5, x6, Label
beq x5, x6, Label
```

branch equal — Se $x_5 = x_6$, salto a LABEL

Es.: `if (b==c) a = b + c;` ⇨

x7≡a, x5≡b, x6≡c

```
bne x5, x6, Label
add x7, x5, x6
....
Label:
```

branch if not equal — allineati. Sommo x_5 e x_6 — più avanti nel codice — seguiranno istruzioni — Label è una etichetta

Es. +7 istruzioni =28 Bytes

- Per quanto riguarda la codifica, l'etichetta («Label» nell'esempio) NON PUÒ essere memorizzata come tale: viene trasformata nel n. di byte (OFFSET) che separa l'istruzione di salto dalla posizione dell'etichetta stessa (es. +7 istr. x 4B ciascuna= 28B) e DIVIDO /2
- Tipicamente il salto avviene nelle vicinanze, quindi, tale OFFSET è spesso un numero piccolo che potrebbe essere memorizzato usando i 12 bit del campo immediato del formato I o S. Inoltre dato che il RISC-V usa un allineamento delle istruzioni sui 16 bit (non 32) l'ultima cifra binaria di tale numero è sempre zero quindi tali 12 bit «valgono 13»
- Volendo però conservare rs1 e rs2 nella stessa posizione viene introdotto un formato leggermente diverso (formato B):

ho diviso per 2, non mi interessa il bit meno significativo ⇒ 12 bit posso farli valere 13 in cui il 13° bit è sempre 0

Es.: `beq t0, t1, label` (x_5, x_6)

imm13[12]	imm13[10:5]	rs2 (x_7)	rs1 (x_5)	funct3	imm13[4:1]	imm13[11]	opcode
0	000000	00111	00101	000	11100	1	1100011

bit 12 — rimane qui — sorgenti — bit 11 — Uso Formato B per codificare in inviolo a 13

QUESTO FORMATO E' CHIAMATO "B" (o "SB")

Nota: I bit imm13[10:5] e imm13[4:1] non cambiano di posto rispetto al formato S, ma i bit 11 e 0 del formato S sono interpretati diversamente come bit 12 e 11 rispettivamente, dato che le istruzioni RISC-V stanno a multipli di 16 bit (bit 0 sempre 0)

Nell'esempio {imm13[12:1],0} codifica 0 0000 0001 1100 ovvero 28

Calcolo la distanza tra BRENCH e PUNTO di salto: es. 7

• il bit meno significativo dell'offset sarà sempre 0

Voglio codificare un immediato a 13 bit, non 12

NELL'IMMEDIATO a 12 BIT posso scrivere un IMM di 13

Composto da

$imm13 = \{ imm[12:1], 0 \}$ il primo è = 0, uso i primi 12

risultato della divisione per 2

$14 = 1110 = imm[4:1]$ per passaggio alla rete 0, offset positivo

dove il bit più significativo, I(12) è messo a SX

codifica LABEL prendo distanza tra istr. branch e il PC a cui devo saltare. Il risultato è un offset, e lo divido per due, e lo codifico

① Calcolo distanza tra branch e PC a cui devo saltare, e divido

distanza tra beq e LABEL

ogni istr. sono 32 bit, 4 byte

7 ISTRUZ. × 4 byte = 28 / 2 = 14

codifico LABEL, la andrò a scrivere nella codifica di branch

28 distanza in byte

• # Calcolo la DISTANZA NELLA MEMORIA ISTRUZ

es. 7 istruz. di distanza tra branch e etichetta. ogni istr., 4 byte

$7 \cdot 4 = 28$. Rimonino nell'IMMED.

$28 / 2 = 14$

/2 => shift right, non mi interessa bit meno significativo, e da IMPLICITAM. le istruz. sono allineate a 16 BIT

Ho allora a DISPOSIZ. 13 bit per l'immediato del salto

Bit più significativo rimane a SX dell'offset, così posso usare logica che controlla segno OFFSET

Gestione della condizione

- Abbiamo due sole istruzioni per gestire il salto su condizione (beq, bne)
  - come si fa un salto su "minore" (e <=, >, >=) ?
- Ci basta in realtà una sola istruzione aggiuntiva: Set on Less Than che mette in un registro l'esito del confronto con l'operatore "<"

```
slt x5, x17, x18
```

mette in x5 il valore «1» se x17<x18 altrimenti mette in x5 «0»

- Dopodiché la beq/bne può confrontare x5 con 1 o 0 e fare un salto
- Possiamo poi generalizzare:
  - per il confronto su ">=" basta invertire la condizione (bne)
  - per il confronto su ">" basta scambiare gli operandi della `slt`
  - per il confronto su "<=", inverto condizione e scambio operandi

Nota: la `slt` non usa un formato nuovo ☺ → usa il formato R dato che opera solo su registri

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 x9, x9,  4
slti x5, x6, 10
andi x5, x7, 0x001
ori  x6, x7, 0xFF0
xori x6, x8, 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)

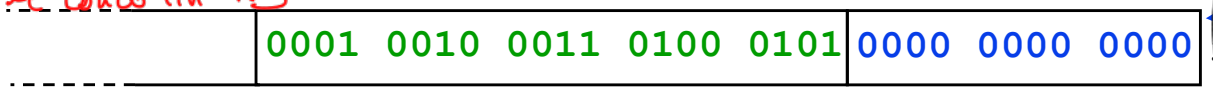
Nota2: la "subi" non esiste... si fa con la addi coll'immediato negato

Che si fa con le costanti "grandi"?

- Vorremo caricare ad es. costanti a 32 bit in un registro
  - Es. 0x1234'5678
  - La parte bassa (0x678) la potrei caricare con `ori x5, 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`)

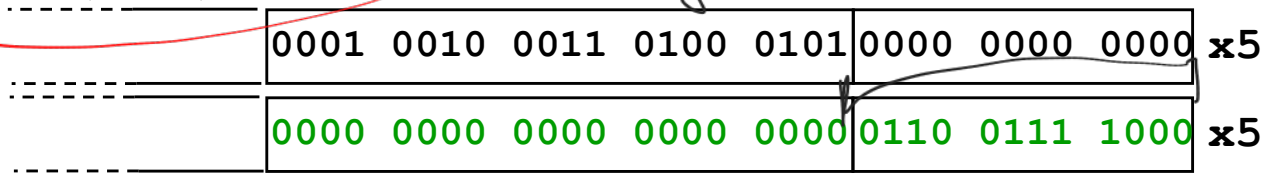
`lui x5, 0x12345` → ho però bisogno di un nuovo formato U (vedi slide successiva)



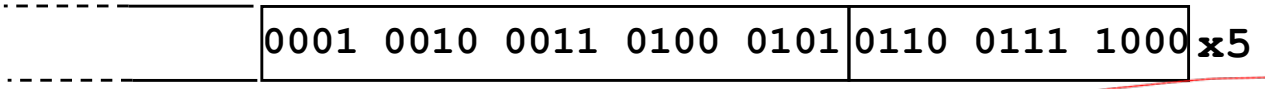
Riempita con zeri

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

`ori x5, x5, 0x678`



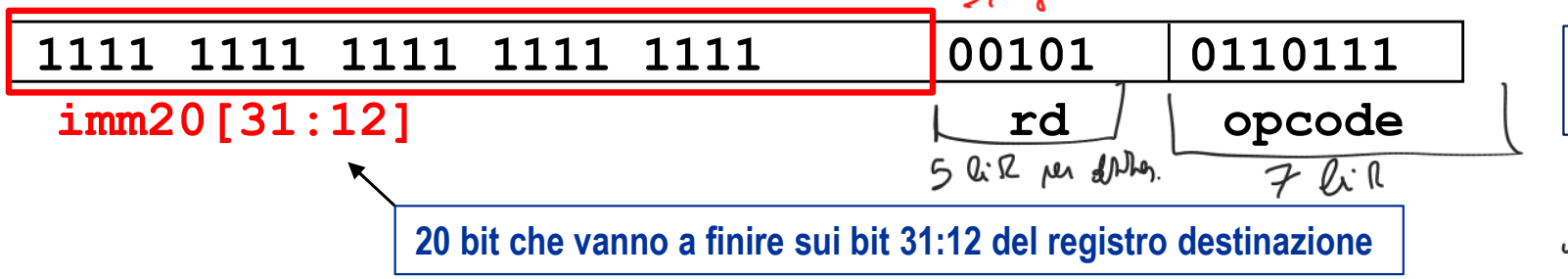
`ori`



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`)

Formato U per caricare questi 20 bit

Es.: `lui x5, 0xFFFFF`



x5, registro dest.

QUESTO FORMATO È CHIAMATO "U"

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

nella parte superiore di rd

load upper immediate usa formato U

Add, sub, beq ecc.

Finora: ad ogni codice mnemonico corrisponde una codifica a 32 bit

Pseudo-istruz. ha una implementaz. a più istruzioni

costante grande a 32 bit, tipicamente è un indirizzo

La pseudoistruzione «LOAD ADDRESS» (la)

- **La coppia di istruzioni**
  - lui x5, 20bit_alti
  - ori x5, x5, 12bit_bassi

⟺

la x5, costante_a_32_bit

è talmente frequente che a livello di assembler si può creare una 'pseudoistruzione' → in questo caso: la = "load address"

dove costante_a_32_bit:=(20bit_alti|12bit_bassi)
viene automaticamente spezzata all'interno del campo immediato delle istruzioni native lui e ori

- **Questa pseudoistruzione è molto utile per caricare in un registro «il nome» di una variabile (ovvero l'indirizzo di una variabile) – esempio:**

```
int MAGIC;

int main() {
…
MAGIC = 1234;
…
}
```

Nota: questa è una variabile GLOBALE che quindi risiede nella memoria dati (sezione ".data" del codice assembly)

```
.data
MAGIC: dword(0)

.text
globl main
main:
…
addi x6, zero, 1234
la   x5, MAGIC
sd   x6, 0(x5)
…
```

Nota2: questo è il programma principale ("main") che risiede nella memoria istruzioni (sezione ".text" del codice assembly)

Esistono altre pseudoistruzioni, che vedremo all'occorrenza

Linguaggio Assembly e Linguaggio Macchina

- Il linguaggio Assembly fornisce una conveniente rappresentazione simbolica
  - È più comprensibile (a noi) che sequenze di numeri (binari)

LINGUAGGIO ASSEMBLY = CODIFICA MNEMONICA dell'istruzione

- Il linguaggio macchina è la "realtà sottostante"
  - La macchina comprende solo le istruzioni in formato binario...

- Il linguaggio Assembly fornisce "pseudoistruzioni"
  - es., MOVE: "mv x5, x6" esiste solo in Assembly
    - l'assemblatore traduce usando "add x5,x6,zero"
  - es., LOAD IMMEDIATE: "li x5, 5" esiste solo in Assembly
    - l'assemblatore traduce usando "addi x5,zero,5"

- Quando si effettua per es. il debugging è necessario ricordare quali siano le istruzioni reali

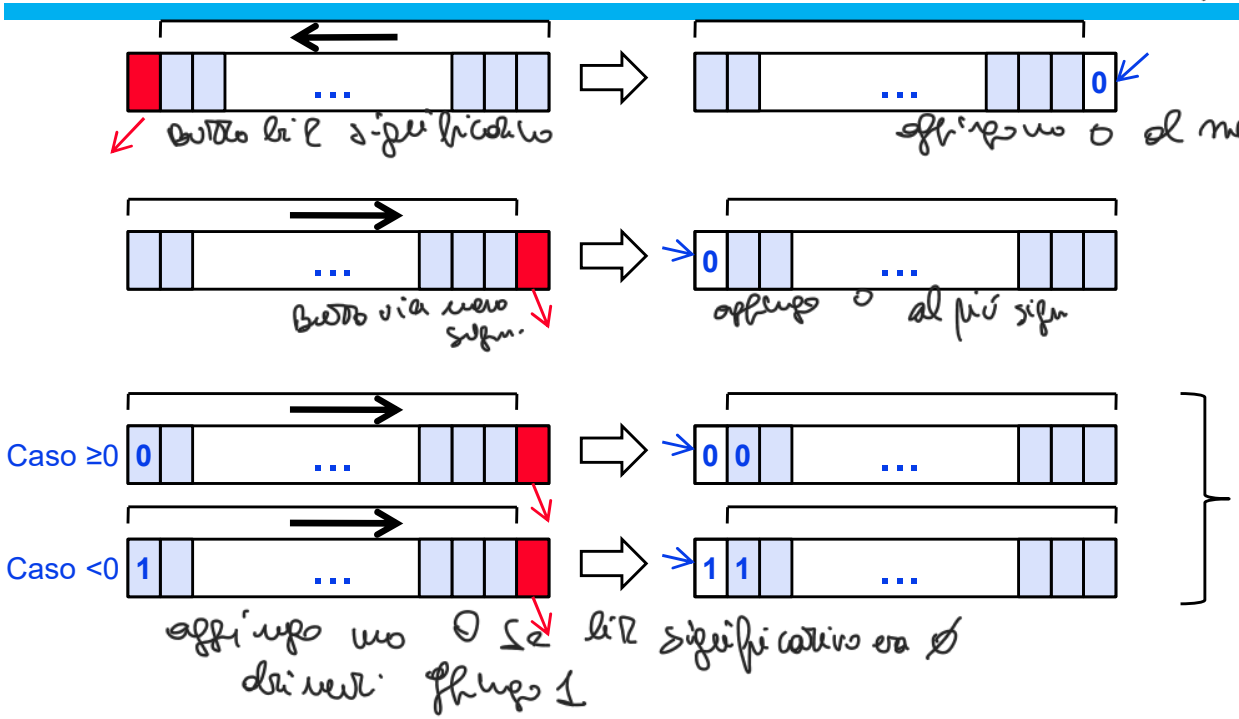
Aritmetica con segno e senza segno

- Le istruzioni aritmetiche fin qui esaminate operano su interi con segno
  - Gli operandi (a 64 bit, nei registri) sono rappresentati in complemento a due → i valori vanno da -2^{63} a $2^{63}-1$
  - Anche i byte-offset di `ld` e `sd` e il numero di istruzioni di cui saltare nella `bne`/`beq` sono interi con segno da -2^{11} a $2^{11}-1$
- È possibile utilizzare anche operandi senza segno
  - In tal caso i valori vanno da 0 a $2^{64}-1$
  - Per fare un semplice esempio, consideriamo operandi a soli 3 bit:
    - Effettuando un confronto tra 0b101 e 0b001 ottengo un risultato diverso a seconda che io usi l'aritmetica con o senza segno:
    - Arit. con segno: 0b101 è -3, mentre 0b001 è 1 → -3<1 è VERO
    - Arit. SENZA segno: 0b101 è 5, mentre 0b001 è 1 → 5<1 è FALSO
  - Necessito quindi di due istruzioni diverse per fare i confronti set-on-less-then (slt):
    - `slt`, `slti` si usano per l'aritmetica con segno
    - `sltu`, `sltiu` si usano per l'aritmetica SENZA segno

Nota: in RISC-V, né "`addu`" né "`addiu`" esistono... occorre ragionarci «a software»: è il software a interpretare gli interi «con segno» o «senza segno» quando si effettuano somme o sottrazioni (per divisioni e moltiplicazioni vedremo che è diverso)

RISC-V: istruzioni di scorrimento (shift)

Patterson 2.6



Shift Left Logical (SLL)

Shift Right Logical (SRL)

Shift Right Arithmetic (SRA)

Nota: SLA non esiste ...=SLL

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

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

Istruzione	*Esempio*	*Significato*	*Commento*
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 ⊕ 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 ⊕ 10	Logical XOR reg, constant

modo con una costante

- **La NOT è una pseudoistruzione: perché?** ho la xor, uso la xor per la not
  - **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
```

= NOT x5, x6

es. -1 ≡ 11111111

xor [-1] flippo i bit

Xor con -1 mi flippa i bit, realizza un not

  - **Notare che prima di fare l'operazione di xor logico, l'operando «-1» verrà esteso di segno su tutti i bit (diventa 0xFFF...FF)**

x6 = 100110 -1 = 11111111 => x5 = 011001

Istruzioni per la manipolazione di stringhe

Patterson 2.9

- 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" carica singolo byte in X5
sb	x5, 0(x6)	"store byte"

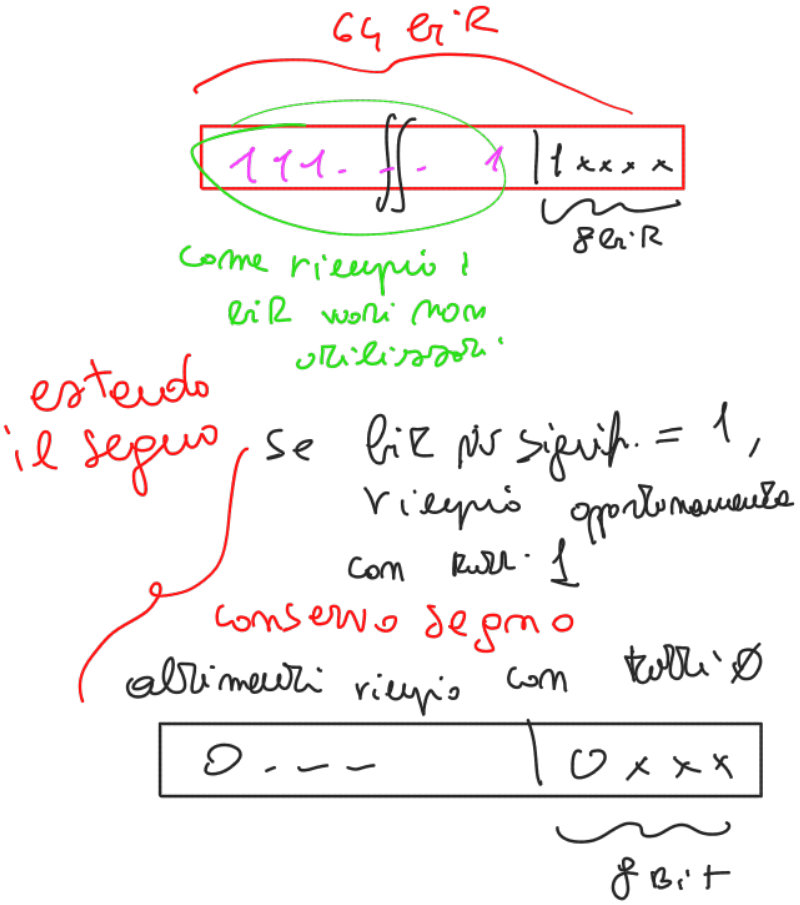
- Per accedere alla half-word (16 bit) ci sono

  (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

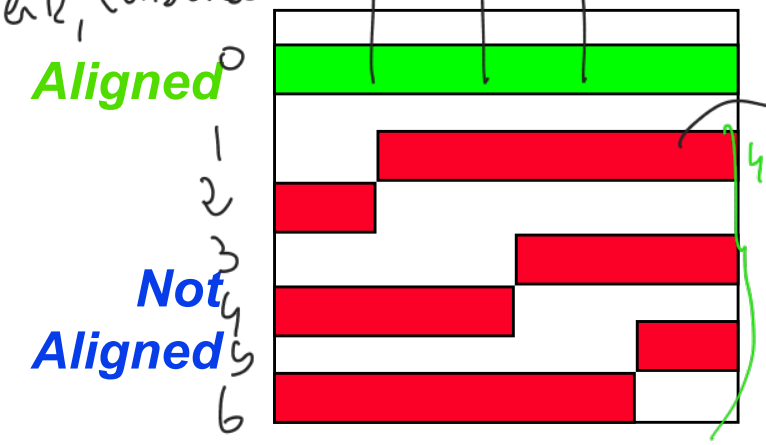
lw	x5, 0(x6)	"load word"
sw	x5, 0(x6)	"store half-word"



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

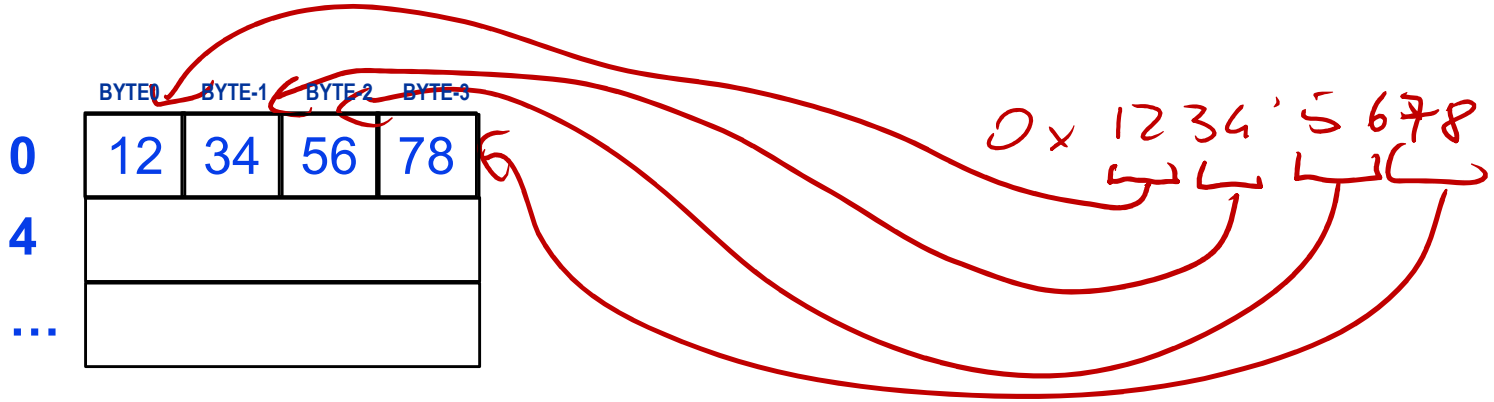
- La memoria è classicamente indirizzata "al byte"
  - Quindi, le istruzioni di load e store usano indirizzi al byte, però
    - `lw`, `lwu` e `sw` trasferiscono 32 bit
    - `lh`, `lhu` e `sh` trasferiscono 16 bit
    - solo `lb`, `lbu`, `sb` trasferiscono 8 bit
- È conveniente pertanto che l'indirizzo 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"



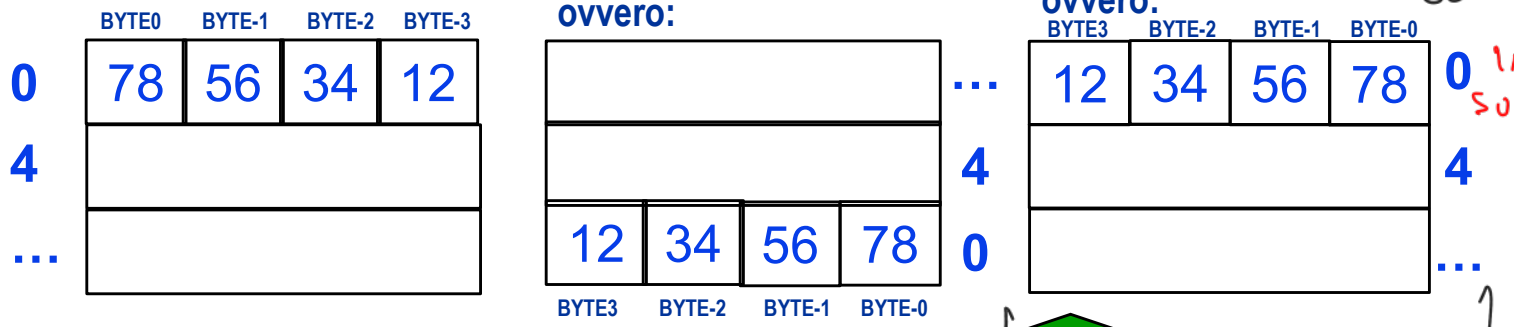
Nota: se si specifica un indirizzo non allineato rispetto a quanto l'istruzione desidera, il RISC-V genera uno warning (avvertendo che il tempo per l'accesso al dato risulterà 2 volte più lento)

Ordinamento dei byte (Endianess)

- **Big Endian: si inizia con la parte "più grossa" ovvero a indirizzi più bassi i byte più significativi:**
  IBM 360/370, Motorola 68k, MIPS R4000, Sparc(pre-v9)
  - **Se ad esempio devo memorizzare 0x12345678 in B.E.**



- **Little Endian: si inizia con la parte "più piccola" ovvero a indirizzi più bassi i byte meno significativi:**
  Intel 80x86, DEC Vax, DEC Alpha (Windows NT), RISC-V

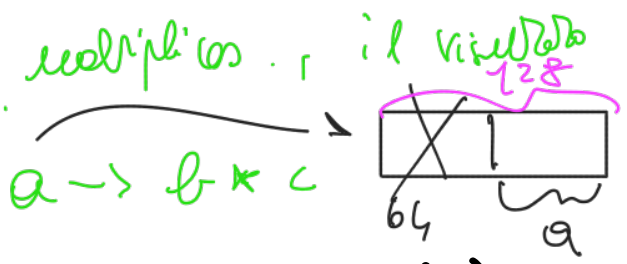


Nota: è importante sapere l'endianess di una macchina quando si scandiscono stringhe o dati byte-a-byte SPECIALMENTE dopo un TRASFERIMENTO DA RETE in cui la macchina remota può avere una endianess diversa

Operazione di moltiplicazione a 64 bit – cast implicito

- Si osservi il seguente frammento di codice C:

```
int a, b, c;
…
a = b * c;
…
```



- Se la macchina definisce l'intero 'int' come una quantità a 64 bit, lo statement di assegnamento 'a=b*c' contiene un cast implicito; ovvero è come se avessi scritto:

```
int a, b, c;
…
a = (int)(b * c);
…
```

- Quindi il risultato del prodotto di due interi (con segno) a 64 bit, che sta quindi in generale su 128 bit, viene troncato scartando IMPLICITAMENTE i 64 bit più significativi!

- In altri termini:

```
int a, b, c;
…
a = b * c;
…
```

⟺

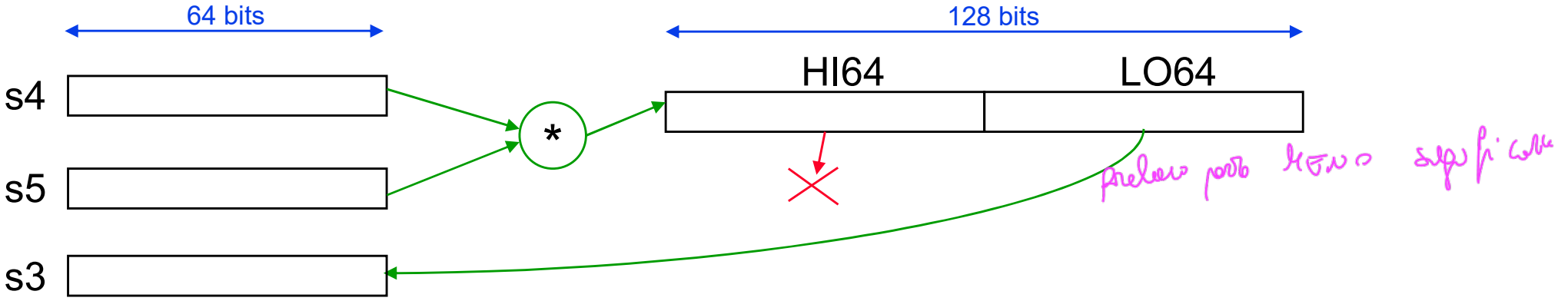
```
s3≡a, s4≡b, s5≡c

mul  s3, s4, s5
```

Moltiplicazione

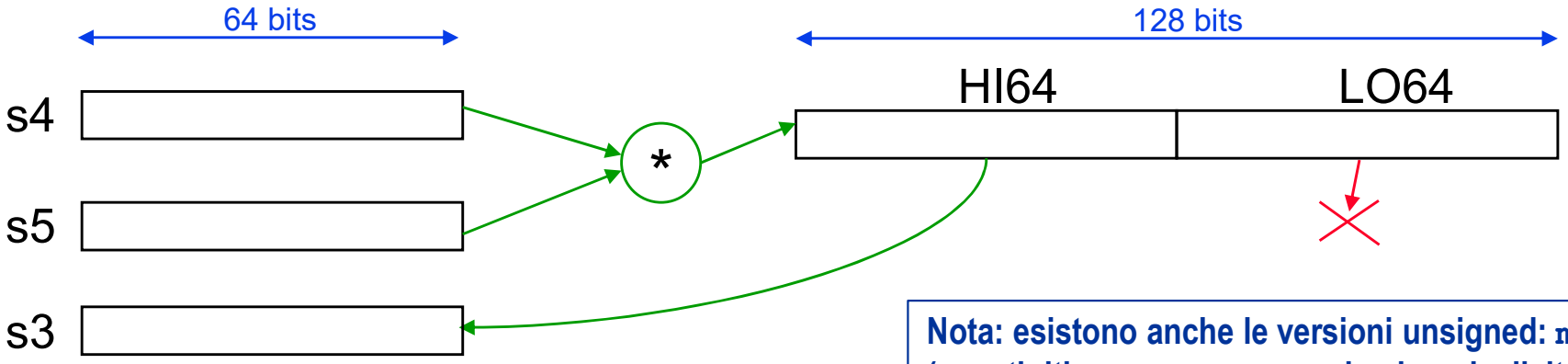
`mul s3, s4, s5` ⟺ **s3 = (int)(s4 * s5)**

- **Esiste quindi un registro interno (nascosto all'utente)**



- **Ma si possono ottenere i bit più significativi (HI64) con l'istruzione**

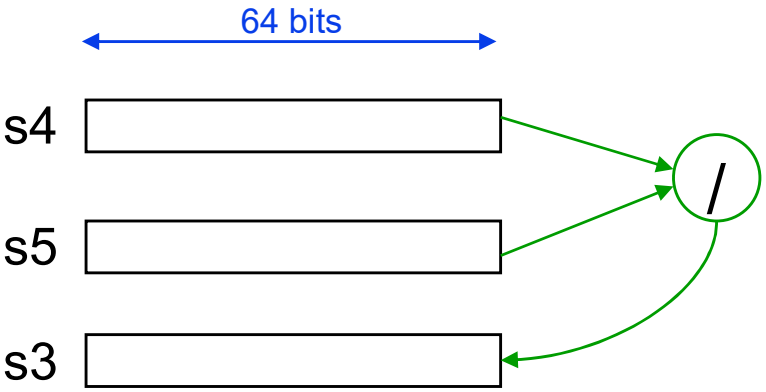
`mulh s3, s4, s5`



Nota: esistono anche le versioni unsigned: `mulhu` e `mulhsu`
(quest'ultima con un operando signed e l'altro unsigned)

Divisione e Resto

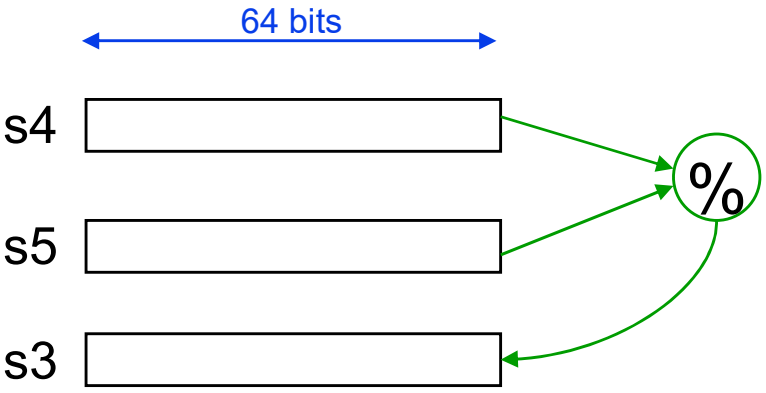
`div s3, s4, s5` ⟺ `s3 = s4 / s5`



`rem s3, s4, s5` ⟺ `s3 = s4 % s5`

rem = remainder Resto

numero resto divisione



REM = Remainder, resto divisione

Nota: esistono anche le versioni unsigned: `divu` e `remu`

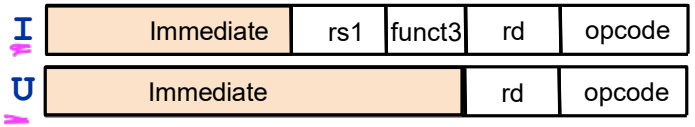
Ricapitolazione istruzioni RISC-V per categoria

- **Aritmetiche**
  - ADD/ADDI/SUB/MUL/DIV/REM/MULH
  - ADDW/ADDIW/SUBW/MULW/DIVW/REMW
  - SLT/SLTI/SLTU/SLTIU
  - MV/LI/LA (pseudoistruzioni, usano ADD/ADDI/LUI)
- **Logiche**
  - AND/OR/XOR/ANDI/ORI/XORI
  - SLL/SLLI/SRL/SRLI/SRA/SRAI
  - SLLW/SLLIW/SRLW/SRLIW/SRAW/SRAIW
- **Accesso alla memoria**
  - LB/LBU/LH/LHU/LW/LWU/LD
  - SB/SH/SW/SD
- **Diramazione, Salto, Eccezioni**
  - BEQ/BNE
  - B/JAL/RET (pseudoistruzioni, usano BEQ/JALR)
  - ECALL/SRET

Tabella riassuntiva modi indirizzamento del RISC-V:

Patterson 2.10

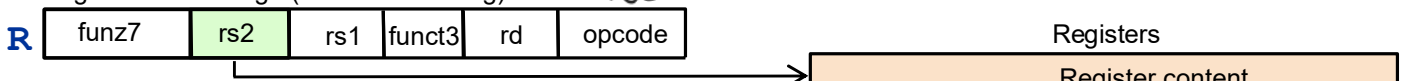
1. Immediate addressing



Usato da:
ADDI/ADDIW, SLTI, ANDI/ORI/XORI
SLLI/SLLIW/SRLI/SRLIW/SRAI/SRAIW, ECALL

Usato da: LUI

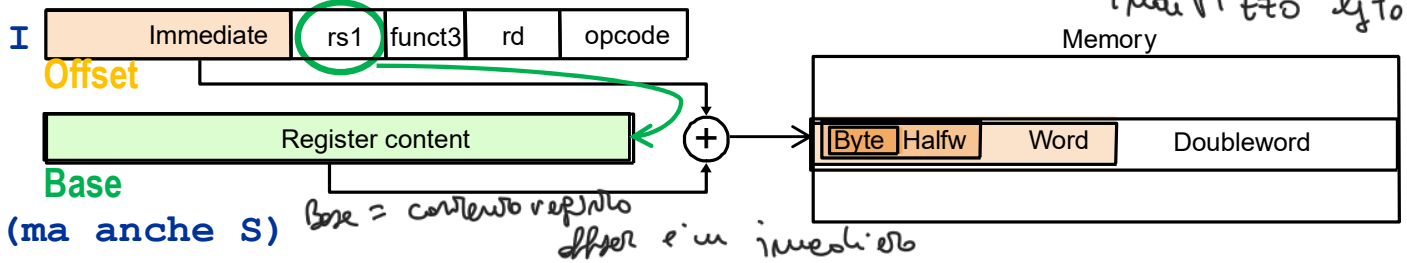
2. Register addressing: (direct addressing)



(ma anche I,U,S,B,J)

Usato in modo esclusivo da:
ADD/SUB/MUL/DIV/REM
ADDW/SUBW/MULW/DIVW/REMW
SLT/SLTU SRET
ma anche (parzialmente) da tutte le altre istruzioni e formati ove c'è almeno un registro

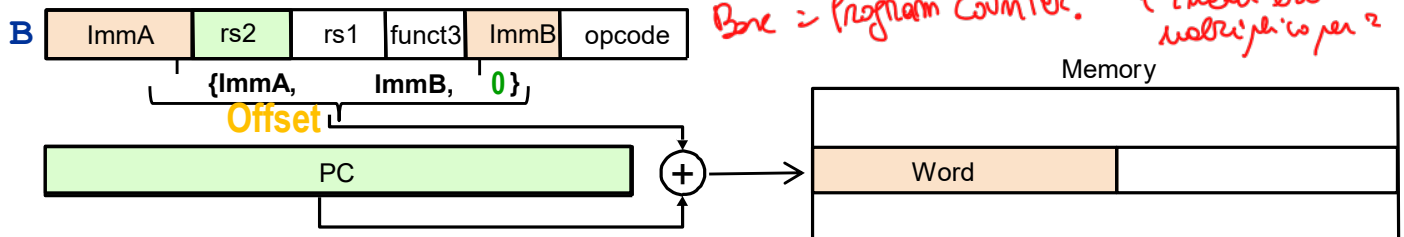
3. Base addressing (Base + Offset)



(ma anche S)

Usato da:
LB/LBU/LH/LHU/LW/LWU/LD
(formato I)
SB/SH/SW/SD
(formato S)

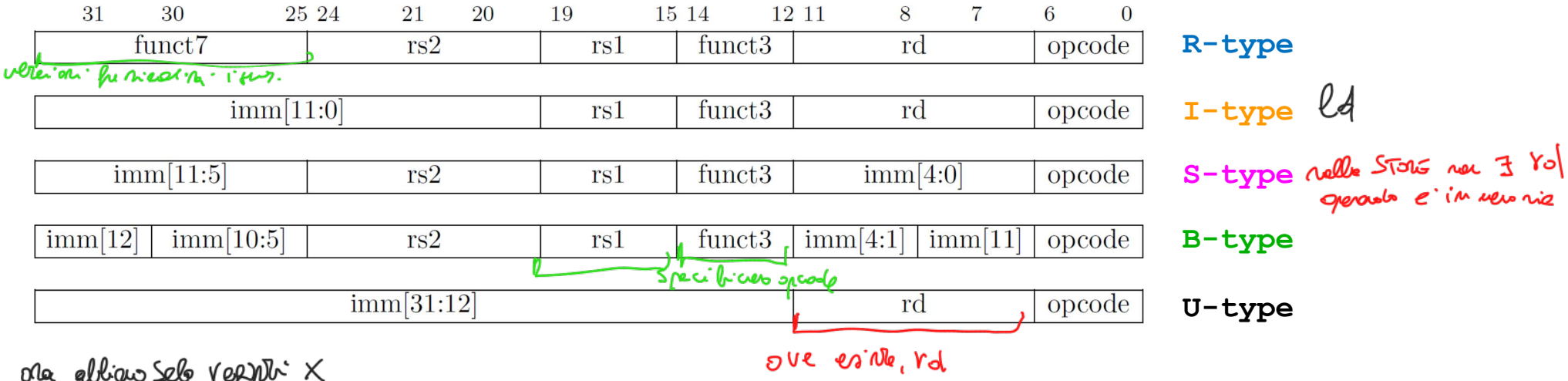
4. PC-relative addressing (Base + Offset) PC + 2*Immediate



(ma anche U,J) poi

Usato da:
BEQ/BNE/B
(formato B)
JAL/RET
(ovvero JALR - formato J)
AUIPC
(formato U)

Ricapitolazione dei formati e istruzioni rappresentative



- **Istruzione** — **Significato** — **Variante** — **Significato**

Istruzione	Significato	Variante	Significato
`add s1,s2,s3`	`s1 = s2 + s3`	`addi s1,s2,10`	`s1 = s2 + 10`
`sub s1,s2,s3`	`s1 = s2 - s3`		
`slt t0,s1,s2`	`t0 = (s1<s2) ? 1 : 0`	`slti s1,s2,10`	`t0 = (s1<10) ? 1 : 0`
`lw s1,100(s2)`	`s1 = Memory[s2+100]` (32 bit)	`ld s1,100(s2)`	legge 64 bit
`sw s1,100(s2)`	`Memory[s2+100] = s1` (32 bit)	`sd s1,100(s2)`	scrive 64 bit

`bne s4,s5,L` salta a L se s4!=s5 (L si trova a offset imm13=(&L-PC))

`beq s4,s5,L` salta a L se s4==s5 (L si trova a offset imm13=(&L-PC))

`lui x5,imm20` carica imm20 nei 20 bit alti di x_5