Corso di Architettura degli Elaboratori a.a. 2023/2024

L'Instruction Set Architecture (ISA) RISC-V

RISC-V Instruction set

https://riscv.org/

- Proposto e sviluppato all'Università di Berkeley dal 2010 da Andrew Waterman, Yunsup Lee, Krste Asanovic
- Obiettivi
  - Costruire un hardware semplice
  - Compilatori efficienti
  - Massimizzare le prestazioni
  - Minimizzare costi
  - Minimizzare il consumo energetico
- Standard aperto

https://riscv.org/

RISC-V ISA

- ISA semplice che può essere esteso
- License-free, royalty-free RISC ISA
- Standard mantenuto dalla RISC-V Foundation (fondata nel 2015)
- Specifiche adatte a dispositivi eterogenei (dai microcontrollori ai supercomputer)
- RISC-V non è
  - Una azienda
  - Una CPU

https://riscv.org/

RISC-V ISA

- ISA semplice che può essere esteso
- Esempio
  - RV64I: base integer 64bit instruction set
  - RV64M: aggiunta delle istruzioni per moltiplicazioni e divisioni intere
  - RV64F: aggiunta delle istruzioni per la gestione dei numeri in virgola mobile
  - RV64E / RV32E

Extension	Description
I	Integer
M	Integer Multiplication and Division
A	Atomics
F	Single-Precision Floating Point
D	Double-Precision Floating Point
G	General Purpose = IMAFD
C	16-bit Compressed Instructions
Non-Standard User-Level Extensions	
Xext	Non-standard extension “ext”

Common RISC-V Standard Extensions
***Not a complete list**

RISC-V Innovation Roadmap

2010 – 2016	2017	2018	2019	2020	2021	2022	2023	2024	2025
Test Chips Software tests Linux port	Proof of Concept SoCs Minion processors for power management, communications Bare metal software	IoT SoCs Microcontrollers RTOS, Firmware Development tools Technical Steering Committee, HPC SIG, GlobalPlatform partnership			AI SoCs, Application processors, Linux Drivers, AI Compilers Dev Board program Development Partners RISC-V Labs, Security response process, AI SIG, Graphics SIG, Android SIG, Communications SIG		**Industry Adoption** Proliferation of RISC-V CPUs across performance and application spectrum RISC-V dominant in universities Strategic and growing adoption in HPC, automotive, transportation, cloud, industrial, communications, IoT, enterprise, consumer, and other applications		
ISA Definition RISC-V Foundation	RV32	RV32I and RV64I Base instructions: Integer, floating point, multiply and divide, atomic, and compact instructions Priv modes, Interrupts, exceptions, memory model, protection, and virtual memory		Arch compatibility framework, Processor trace	Zfinx ZiHintPause BitManip Vector RISC-V Profiles & Platforms Crypto Scalar Virtual Memory Hypervisor & Advanced interrupt architecture Cache mgt ops Code size reduction* Trusted Execution Environment* P (Packed SIMD)*		RV32E and RV64E 64 bit and 128 bit addresses* Vector Atomic and quad-widening* Quad floating point in integer registers* Crypto Vector* Trusted Execution phase 2* Jit pointer masking & I/D synch* BitManip phase 2* Cache management phase 2* ... and more **Technical Deliverables**		

RISC-V®

** On track, subject to change*

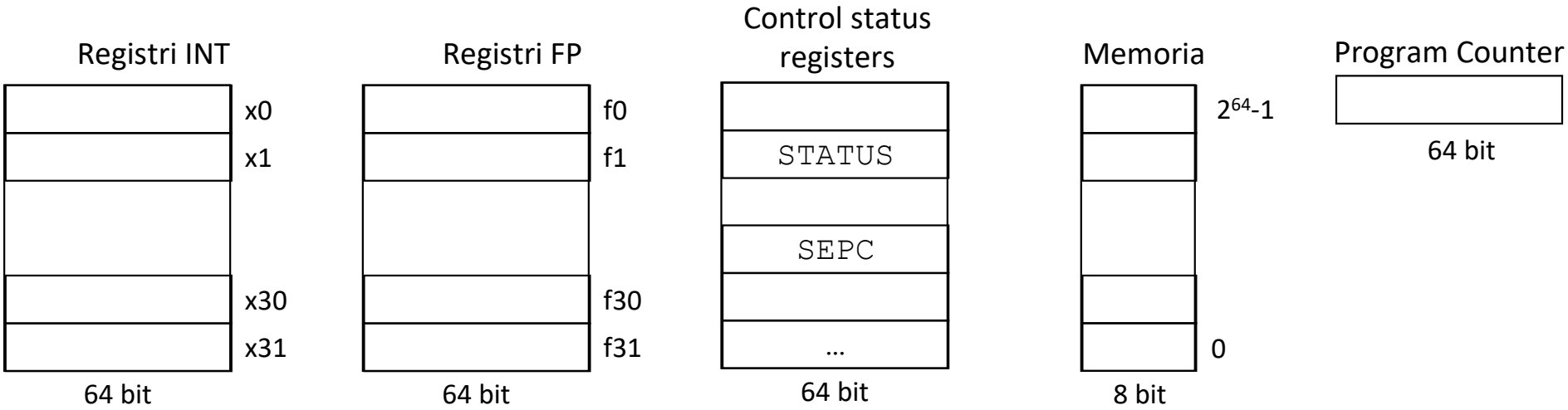
RISC-V Instruction set

- RISC
  - Reduced Instruction Set Computer
- Principi di progettazione
  1. La semplicità favorisce la regolarità
  2. Minori sono le dimensioni, maggiore è la velocità
  3. Un buon progetto richiede buoni compromessi

RISC-V Registri e memoria

Parola (word): 32 bit

Parola doppia (doubleword): 64 bit



- 32 registri per gli interi
- 32 registri per i numeri in virgola mobile
- Program Counter
- 4096 control status registers
- Memoria centrale

RISC-V Registri e memoria

Parola (word): 32 bit

Parola doppia (doubleword): 64 bit

- Registri per gli interi
  - Quantità: 32, indicati con x0 .. X31
  - Dimensione: 64 bit
  - ….

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

RISC-V Instruction set

- Tipologie di istruzione
  - Aritmetiche
  - Logiche
  - Accesso alla memoria
  - Condizionali

Istruzioni aritmetiche

- Notazione rigida
  - Tutte le istruzioni aritmetiche hanno esattamente 3 operandi
  - L'ordine degli operandi è fisso
- Addizione

```
a = b + c
```

Linguaggio C ⟹

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

RISC-V assembler

- Sottrazione

```
a = b - c
```

Linguaggio C ⟹

```
sub a, b, c
```

RISC-V assembler

Istruzioni aritmetiche

- Operandi
  - Gli operandi devono essere sempre tre registri scelti tra x0..x31
  - La ALU ha come input solo il contenuto di registri
  - I numeri interi sono rappresentati in complemento a due

```
a = b + c
```

Linguaggio C

⇨

a → x5
b → x20
c → x21

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

RISC-V assembler

Istruzioni aritmetiche

- A volte si ha la necessità di cambiare segno al valore di un registro
- L'istruzione `sub` può essere utilizzata, memorizzando il valore `0` nel secondo operando
- In RISC-V il registro **`x0`** contiene sempre il valore `0`

```
a = - a
```

Linguaggio C

$\xrightarrow{\texttt{a -> x19}}$

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

RISC-V assembler

Istruzioni aritmetiche

- Il processo di traduzione di codice ad alto livello in linguaggio assembler è svolto dal compilatore
- Un'unica istruzione in un linguaggio ad alto livello può corrispondere a diverse istruzioni assembler

```
f = a + b - c
```

Linguaggio C

```
f -> x19
a -> x20
b -> x21
c -> x22
```

```
add x19, x20, x21
sub x19, x19, x22
```

RISC-V assembler

Istruzioni aritmetiche

- Il processo di traduzione di codice ad alto livello in linguaggio assembler è svolto dal compilatore
- Un'unica istruzione in un linguaggio ad alto livello può corrispondere a diverse istruzioni assembler

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

Linguaggio C

⇒

```
?
```

RISC-V assembler

Istruzioni aritmetiche

- Il processo di traduzione di codice ad alto livello in linguaggio assembler è svolto dal compilatore
- Un'unica istruzione in un linguaggio ad alto livello può corrispondere a diverse istruzioni assembler

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

Linguaggio C

```
f -> x19
g -> x20
h -> x21
i -> x22
j -> x23
```

```
add x5, x20, x21
add x6, x22, x23
sub x19, x5, x6
```

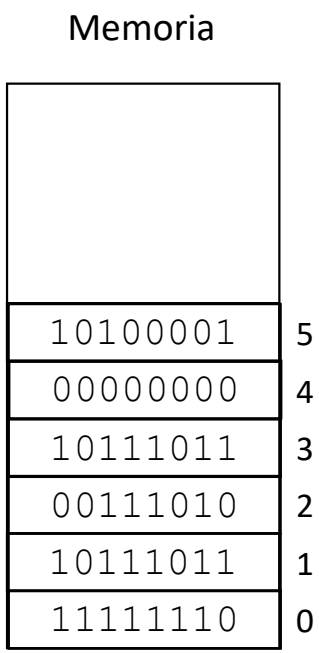
oppure

```
add x19, x20, x21
sub x19, x19, x22
sub x19, x19, x23
```

RISC-V assembler

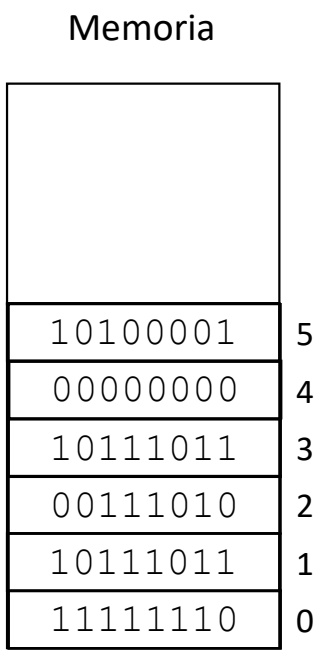
Istruzioni di accesso alla memoria

- 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 terza cella di memoria ha valore 00111010
  - M[2] = 00111010
- **FONDAMENTALE**: 2 è l'indirizzo del byte e 00111010 è il suo valore



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

Linguaggio C

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

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

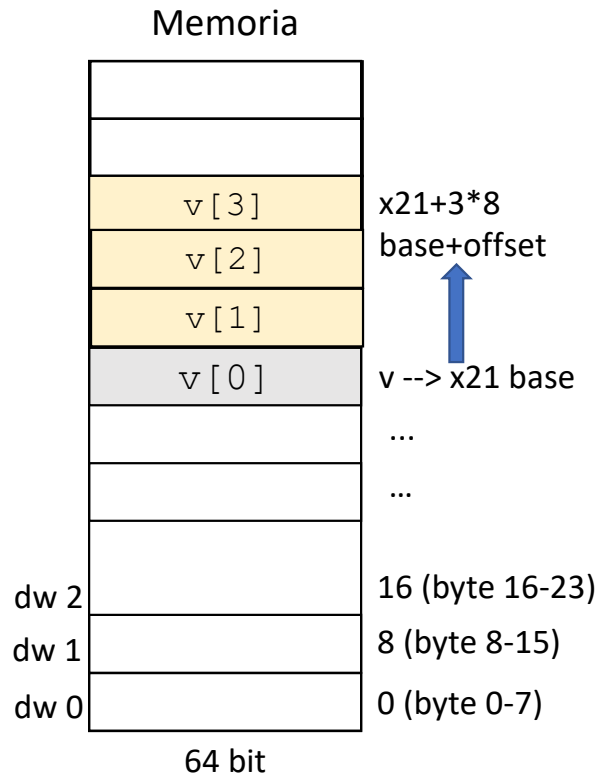
Destinazione della lettura

Offset

Indirizzo base

```
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

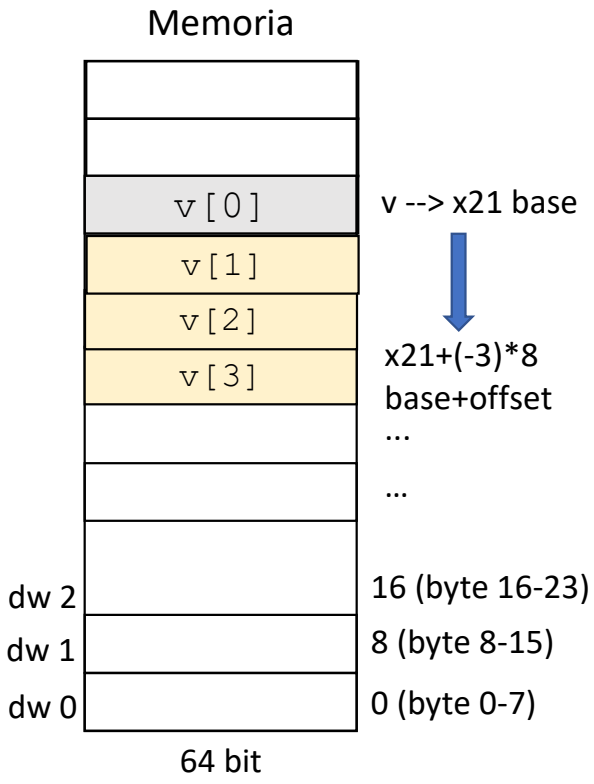
```
long a;
long v[10];
…
a = v[3]
```

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

Linguaggio C

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

a -> x5
v -> x21

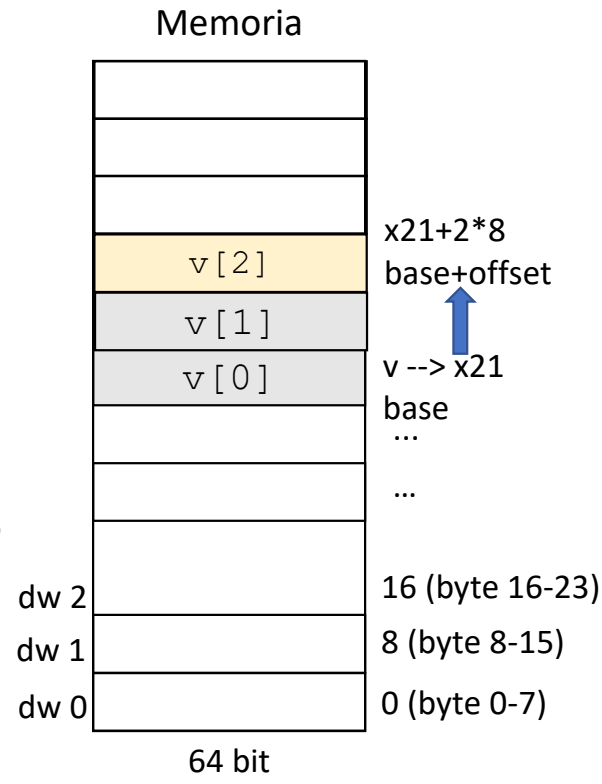
Dato da scrivere in memoria

Offset

Indirizzo base

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

RISC-V assembler

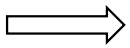


Istruzioni di accesso alla memoria

- Un esempio

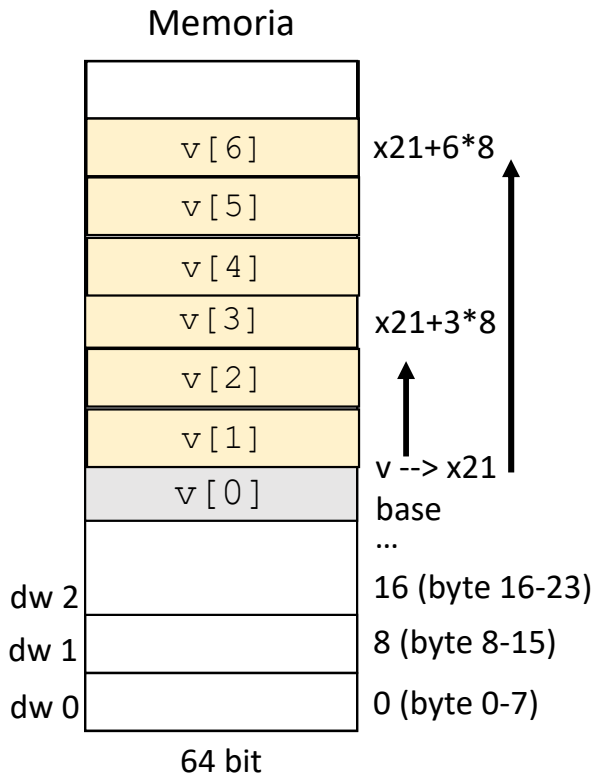
```
long g,h,f;
long v[10];
…
g = h + v[3]
v[6] = g - f
```

Linguaggio C



?

RISC-V assembler



Istruzioni di accesso alla memoria

- Un esempio

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

Linguaggio C

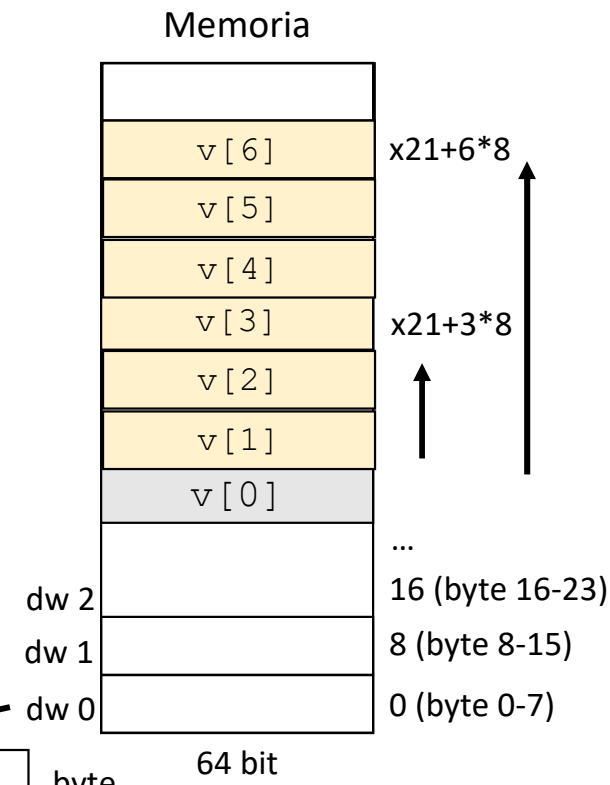
g → x5
h → x9
v → x21
f → x19

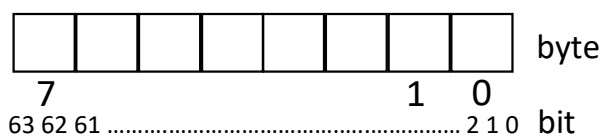
```
ld x10, 24(x21)
add x5, x9, x10
sub x6, x5, x19
sd x6, 48(x21)
```

RISC-V assembler

registro base

offset





little endian: indirizzo di (doppia) parola identifica il byte meno significativo

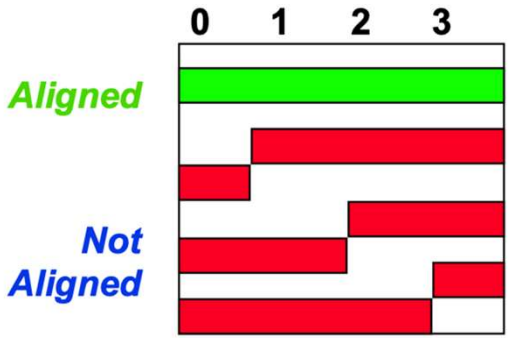
Istruzioni di accesso a byte, half-word e word

- 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"



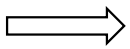
Nota: se si specifica un indirizzo non allineato rispetto a quanto l'istruzione desidera, il RISC-V impiegherà un tempo per l'accesso al dato maggiore

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
```

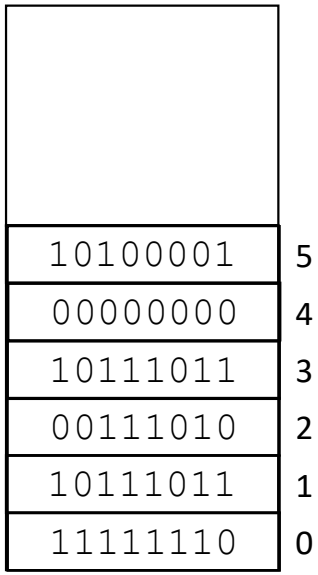
Linguaggio C



?

RISC-V assembler

Memoria



Istruzioni di accesso alla memoria

- Un esempio con variabili a 32 bit

registro base

offset

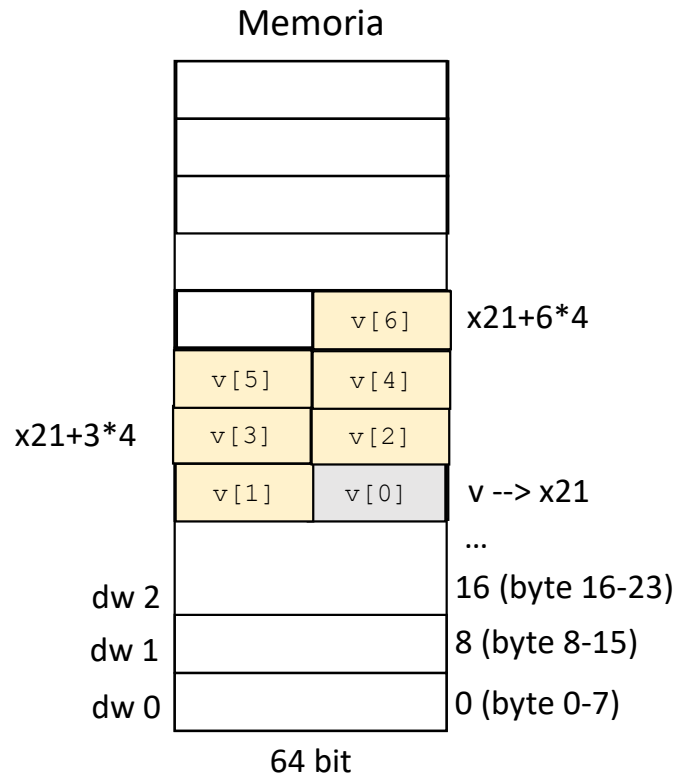
```
int g,h,f;
int v[10];
…
g = h + v[3]
v[6] = g - f
```

Linguaggio C

g → x5
h → x9
V → x21
f → x19

```
lw x10, 12(x21)
add x5, x9, x10
sub x6, x5, x19
sw x6, 24(x21)
```

RISC-V assembler



Istruzioni di accesso alla memoria

- Un esempio con variabili a 16 bit

registro base

offset

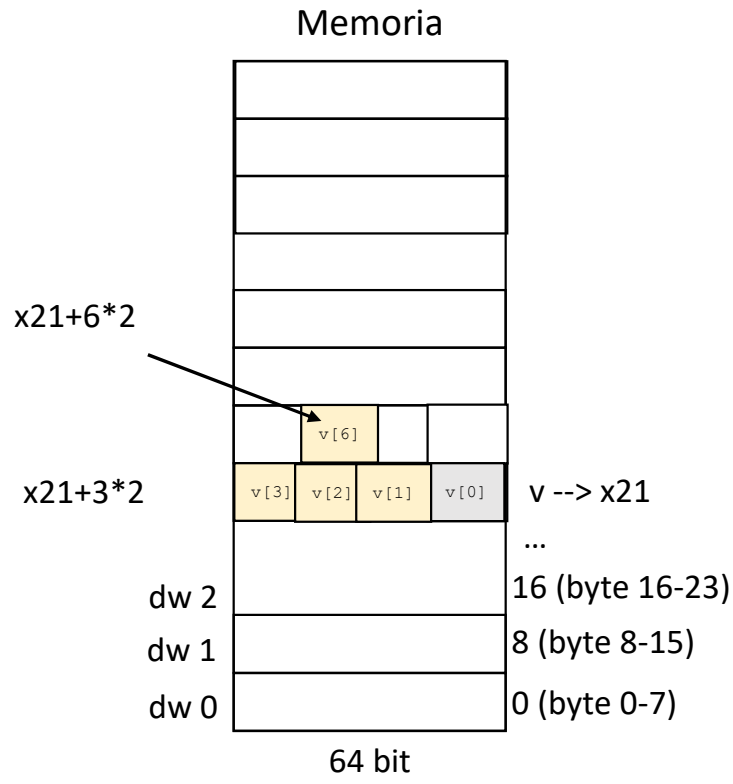
```
short g,h,f;
short v[10];
…
g = h + v[3]
v[6] = g - f
```

Linguaggio C

g → x5
h → x9
v → x21
f → x19

```
lh x10, 6(x21)
add x5, x9, x10
sub x6, x5, x19
sh x6, 12(x21)
```

RISC-V assembler



Istruzioni di accesso alla memoria

- Un esempio con variabili a 8 bit

registro base

offset

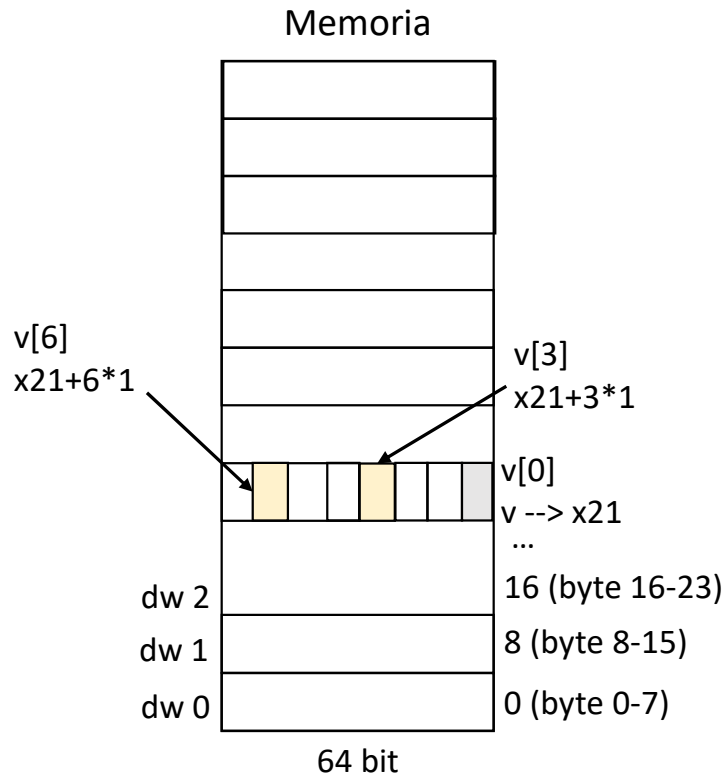
```
char g,h,f;
char v[10];
…
g = h + v[3]
v[6] = g - f
```

Linguaggio C

g → x5
h → x9
v → x21
f → x19

```
lb x10, 3(x21)
add x5, x9, x10
sub x6, x5, x19
sb x6, 6(x21)
```

RISC-V assembler



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

⇒ b → x5

```
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
```

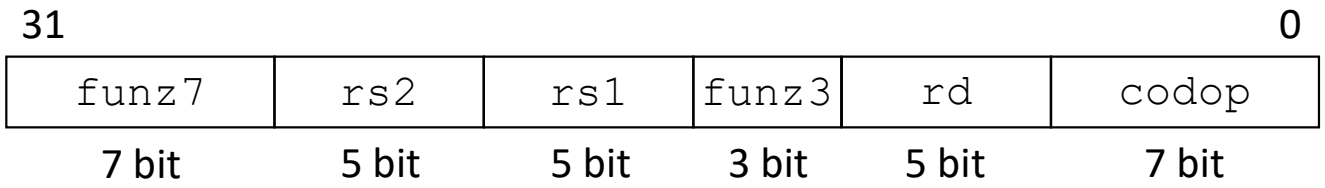
RISC-V assembler

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

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

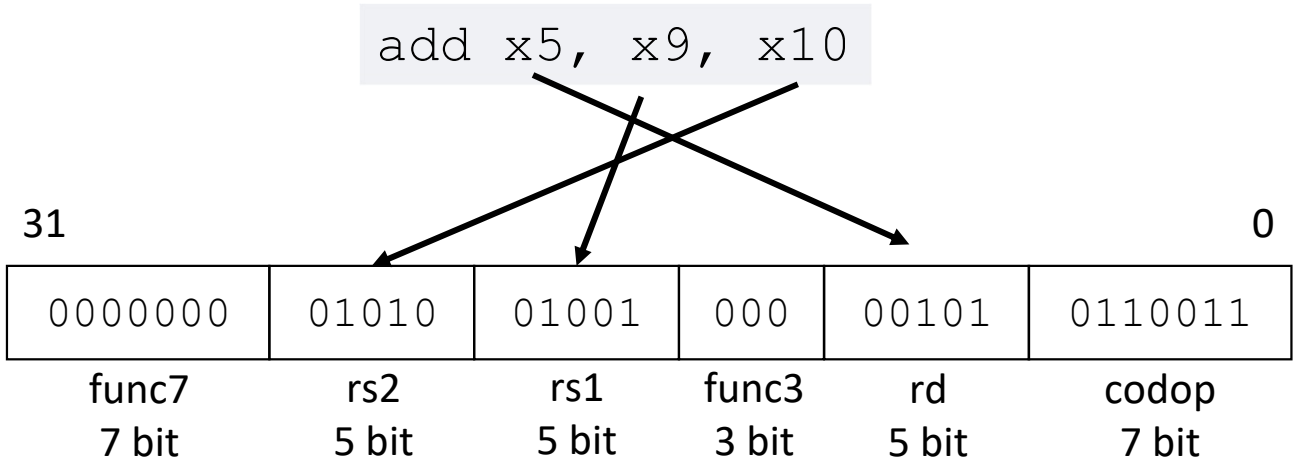
Formato di tipo R (registro)



- 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)

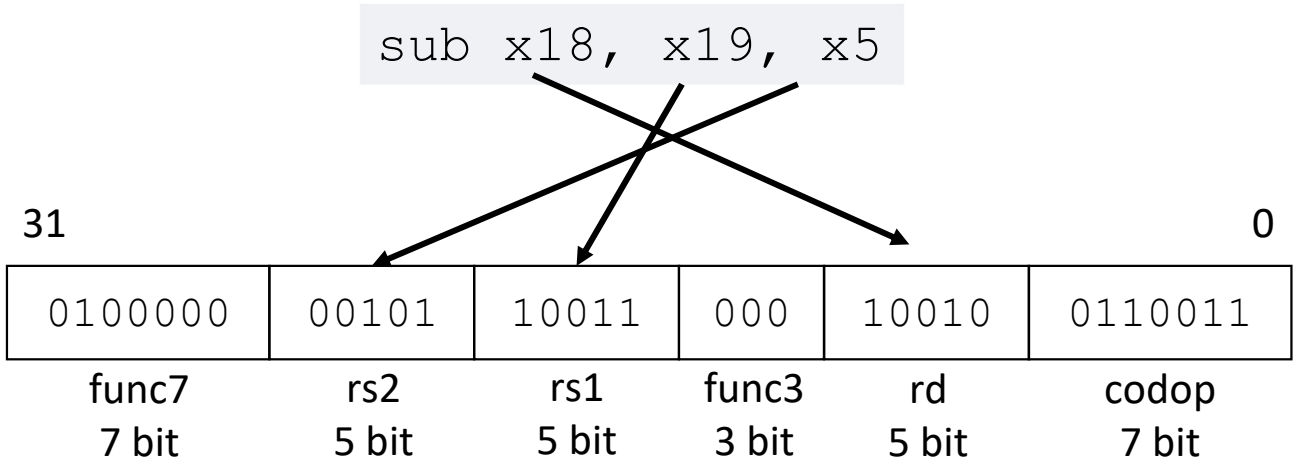
- Esempio



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

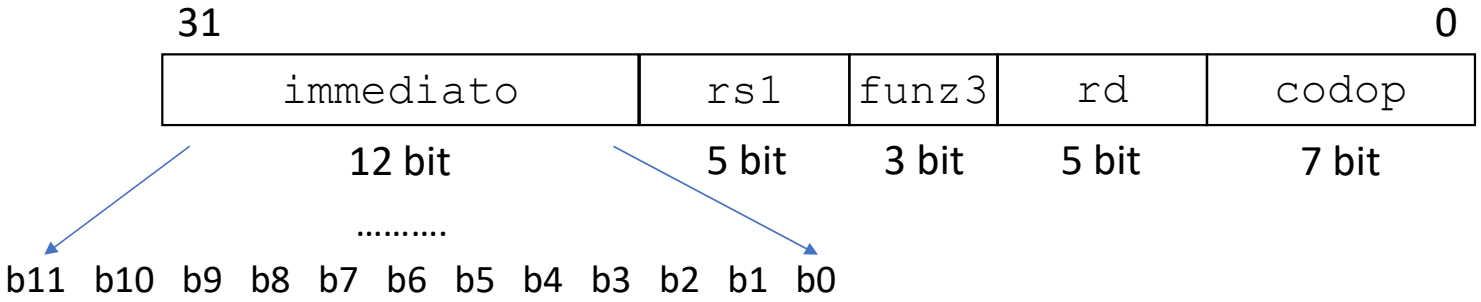
Formato di tipo R (registro)

- Esempio



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

Formato di tipo I (immediato)

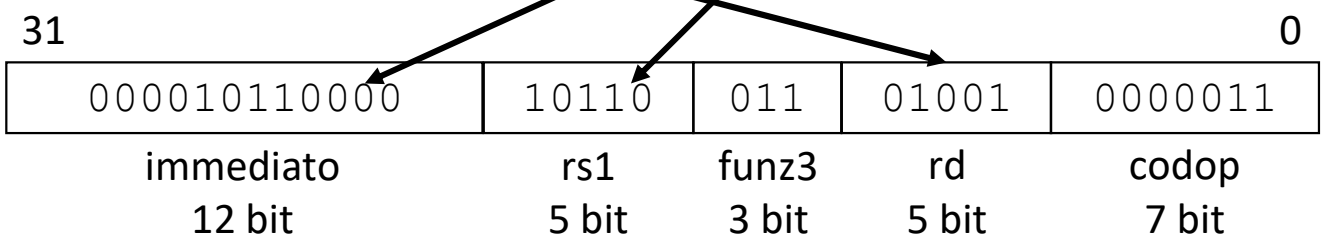


- Permette di codificare le istruzioni che richiedono il caricamento dalla memoria o una costante, come `load`, `addi`, `andi`, `ori`, …
- Sono presenti 12 bit perché con 5 bit l'intervallo di rappresentazione per costanti e (soprattutto) offset sarebbe stato troppo ridotto
- Il campo immediato
  - Rappresentato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

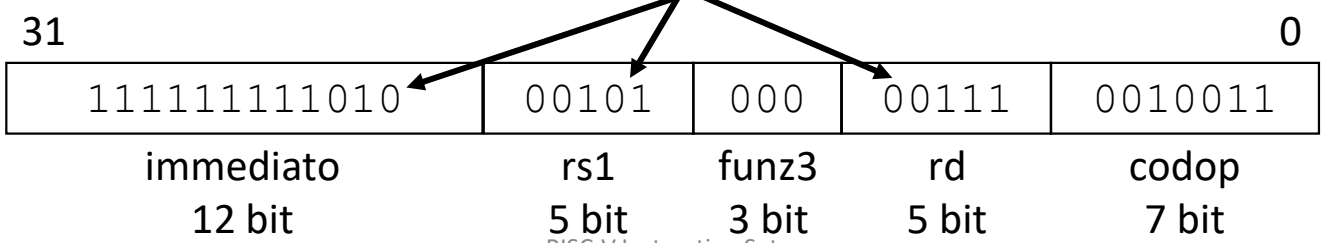
Formato di tipo I (immediato)

- Esempi

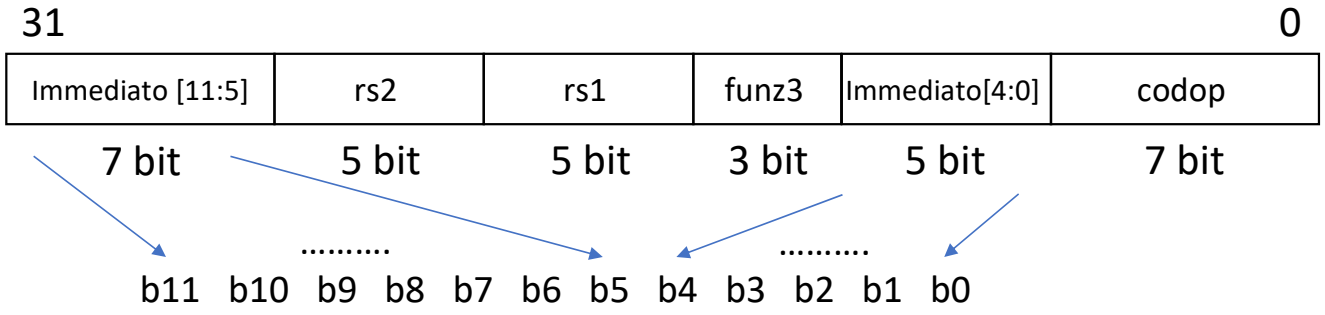
```
ld x9, 176(x22)
```




```
addi x7, x5, -6
```

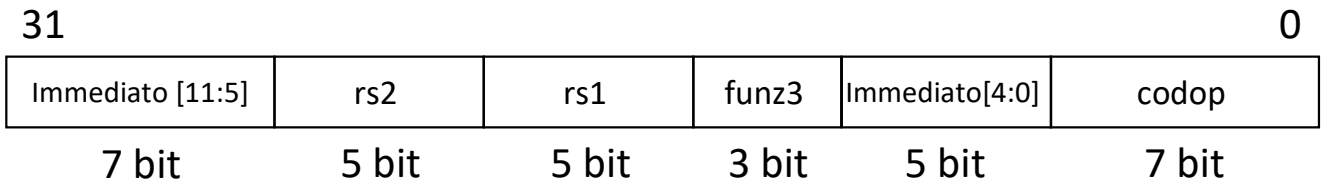


Formato di tipo S



- Permette di codificare le istruzioni che richiedono il salvataggio in memoria o una costante, come `store`
- Perché il campo immediato viene diviso in due parti?
- Il campo immediato
  - Rappresentato in complemento a due
  - Valori possibili: da `-2048` a `+2047`

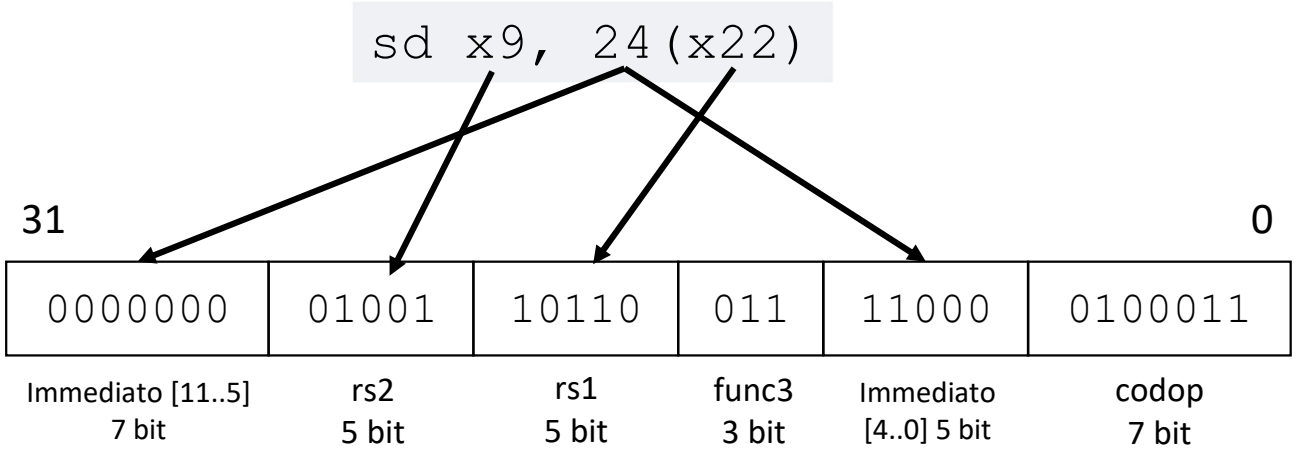
Formato di tipo S



- 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

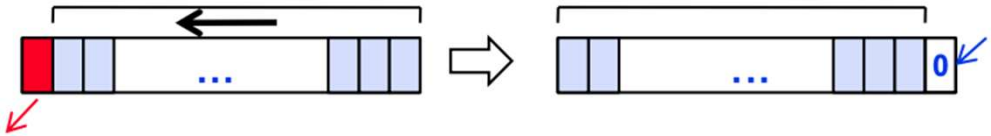
- Il libro di testo contiene tutti i dettagli sulla codifica in linguaggio macchina delle istruzioni assembler

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



Shift Left Logical

```
sll x9, x22, x19
```

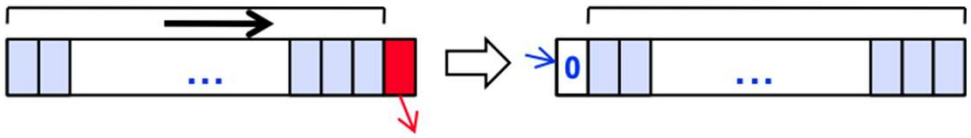
`x9 = x22 << x19`

```
slli x9, x22, 5
```

`x9 = x22 << 5`

Shift Left Logical Immediate

  - A destra



Shift Right Logical

```
srl x9, x22, x19
```

`x9 = x22 >> x19`

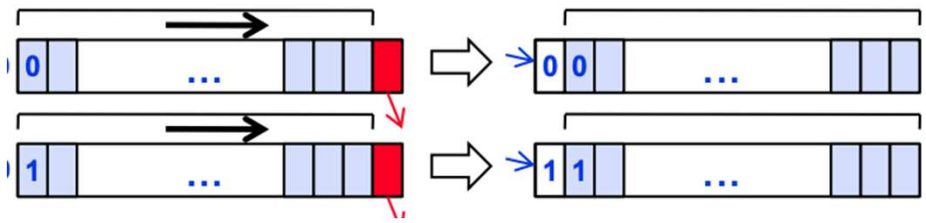
```
srli x9, x22, 5
```

`x9 = x22 >> 5`

Shift Right Logical Immediate

Operazioni logiche

- Shift aritmetico
  - A destra
    - Positivo
    - Negativo



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

Shift Right Arithmetic

```
sra x9,x22,x19
```

`x9 = x22 >> x19`

```
srai x9,x22,5
```

`x9 = x22 >> 5`

Shift Right Arithmetic Immediate

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}

Prima un esempio semplice…

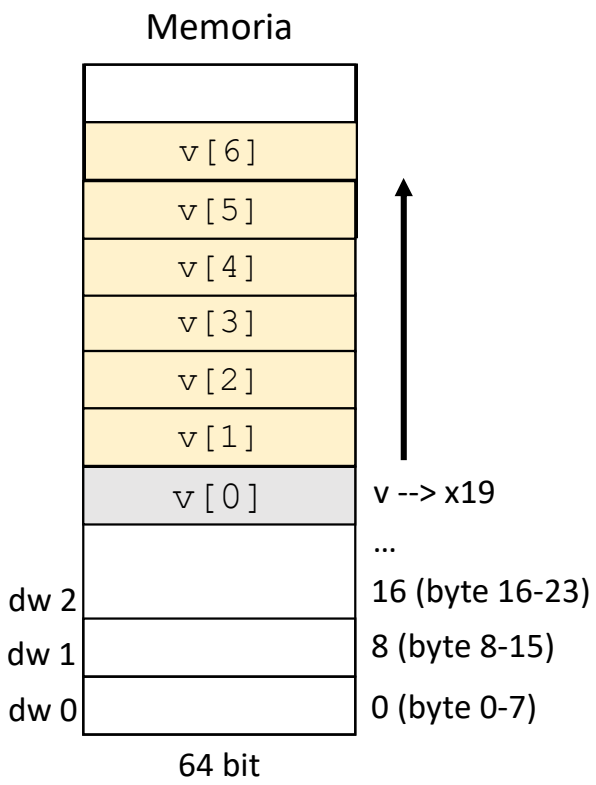
```
long i,j;
long v[10];

…
v[i]=v[j];
```

Linguaggio C

i → x9
j → x21
v → x19

?

RISC-V assembler



Prima un esempio semplice…

x6 contiene l'indirizzo del primo byte che rappresenta v[j]

```
long i,j;
long v[10];

…
v[i]=v[j];
```

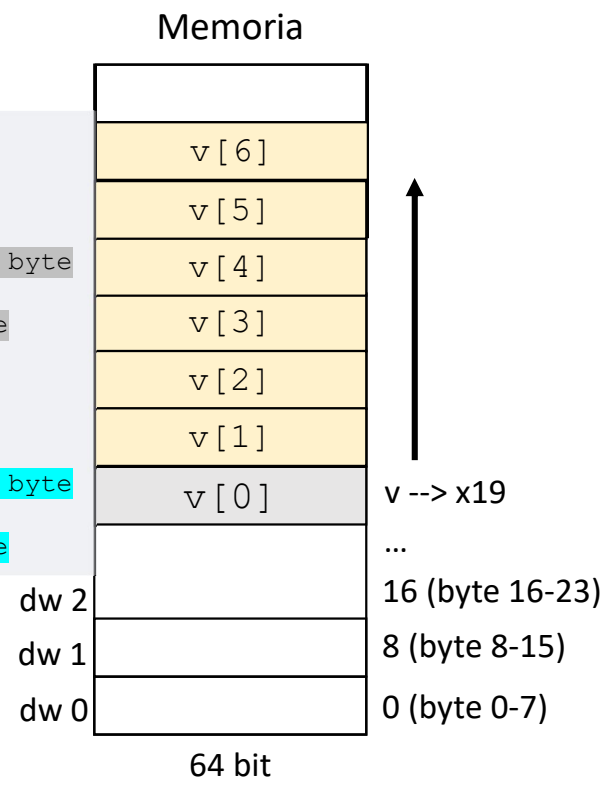
Linguaggio C

i → x9
j → x21
v → x19

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

```
…
Calcola indice in x6
Moltiplica per dim in byte
Somma l'indirizzo base

Calcola indice in x7
Moltiplica per dim in byte
Somma l'indirizzo base
```

RISC-V assembler



x7 contiene l'indirizzo del primo byte che rappresenta v[i]

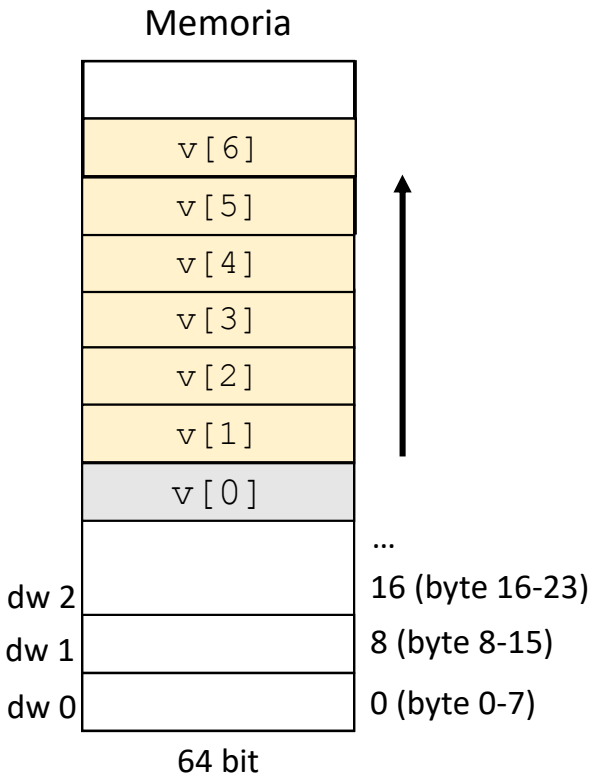
Poi un esempio più complesso…

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

Linguaggio C

?

RISC-V assembler



x6 contiene l'indirizzo del primo byte che rappresenta v[j+2]

Poi un esempio più complesso…

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

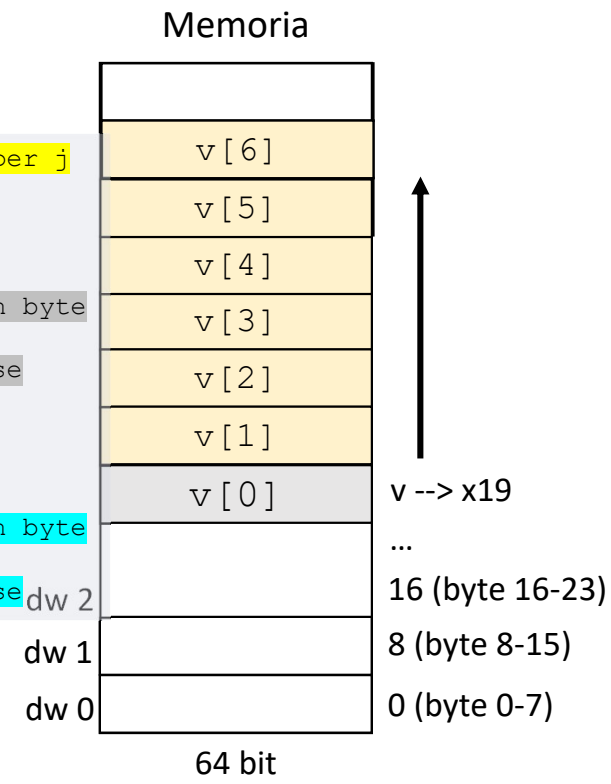
Linguaggio C

d → x5
i → x9
j → x21
v → x19

```
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

Calcola espressione per j
…
Calcola indice in x6
Moltiplica per dim in byte
Somma l'indirizzo base
Calcola indice in x7
Moltiplica per dim in byte
Somma l'indirizzo base



x7 contiene l'indirizzo del primo byte che rappresenta v[i+d]

Poi un esempio più complesso…

Nel simulatore…

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

Linguaggio C

Bkpt	Address	Code	Basic	Source
	0x00400000	0x00500a93	addi x21,x0,5	14: addi x21,x0,5
	0x00400004	0x002a8313	addi x6,x21,2	15: addi x6,x21,2
	0x00400008	0x00331313	slli x6,x6,3	16: slli x6,x6,3
	0x0040000c	0x01330333	add x6,x6,x19	17: add x6,x6,x19
	0x00400010	0x00033303	ld x6,0(x6)	18: ld x6,0(x6)
	0x00400014	0x005483b3	add x7,x9,x5	19: add x7,x9,x5
	0x00400018	0x00339393	slli x7,x7,3	20: slli x7,x7,3
	0x0040001c	0x013383b3	add x7,x7,x19	21: add x7,x7,x19
	0x00400020	0x0063b023	sd x6,0(x7)	22: sd x6,0(x7)

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 ⊕ x19

```
xori x9,x22,5
```

x9 = x22 ⊕ 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
```

x6	00100100 00010101 00001011 10100110	**Sorgente**
x7	00000100 00000110 00000010 00010010	**Maschera**
x5	00000100 00000100 00000010 00000010	**Risultato**

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
```

x6	00100100 00010101 00001011 10100110	**Sorgente**
x7	00000100 00000110 01000010 00010010	**Maschera**
x5	00100100 00010111 01001011 10110110	**Risultato**

Salti condizionati

- Permettono di variare il flusso del programma (variando il valore del PC) al verificarsi di una condizione

```
beq rs1,rs2,L1
```

Branch if EQual

- Il flusso di programma continua all'istruzione con etichetta `L1` se il valore del registro `rs1` è uguale a quello di `rs2`

```
bne rs1,rs2,L1
```

Branch if Not Equal

- Il flusso di programma continua all'istruzione con etichetta `L1` se il valore del registro `rs1` è diverso a quello di `rs2`

Salti condizionati

- Permettono di variare il flusso del programma (variando il valore del PC) al verificarsi di una condizione

```
blt rs1,rs2,L1
```

Branch if Less Than

- Il flusso di programma continua all'istruzione con etichetta `L1` se il valore del registro `rs1` è minore a quello di `rs2`

```
bge rs1,rs2,L1
```

Branch if Greater than or Equal

- Il flusso di programma continua all'istruzione con etichetta `L1` se il valore del registro `rs1` è maggiore o uguale a quello di `rs2`

Salti condizionati

- Esistono anche le operazioni di salto condizionato che confrontano i due registri `rs1` e `rs2` trattandoli come numeri senza segno

`bltu rs1,rs2,L1`	Branch if Less Than Unsigned
`bgeu rs1,rs2,L1`	Branch if Greater than or Equal Unsigned

Salti condizionati: costrutto if-then-else

- Attraverso le istruzioni `beq` e `bne` è possibile tradurre in assembler il costrutto `if` dei linguaggi di programmazione ad alto livello

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

Linguaggio C

f → x19
g → x20
h → x21
i → x22
j → x23

La scelta di test per not equal è più conveniente in questo caso

```
        bne x22,x23,ELSE
        add x19,x20,x21
        beq x0,x0,ENDIF
 ELSE:  sub x19,x20,x21
ENDIF:
```

Salto incondizionato

RISC-V assembler

Salti condizionati: ciclo for

- Una possibile implementazione

```
for (i=0;i<100;i++)
{
    …
}
```

⇨

?

Salti condizionati: ciclo for

- Una possibile implementazione

```
for (i=0;i<100;i++)
{
    …
}
```

i → x19

```
        add x19,x0,x0
        addi x20,x0,100
FOR:    bge x19,x20,ENDFOR
        …
        addi x19,x19,1
        beq x0,x0,FOR
ENDFOR:
```

Salti condizionati: ciclo while

```
long v[10], k, i;
while (v[i]==k)
{
    …
    i=i+1;
}
```

i → x22
k → x24
v → x25

?

Salti condizionati: ciclo while

```
long v[10], k, i;
while (v[i]==k)
{
    …
    i=i+1;
}
```

i → x22
k → x24
v → x25

```
LOOP:     slli x10,x22,3
          add x10,x10,x25

          ld x9,0(x10)

          bne x9,x24,ENDLOOP
          …
          addi x22,x22,1
          beq x0,x0,LOOP
ENDLOOP:
```

Salva in x10 l'indirizzo della doubleword v[i]

Carica dalla memoria il valore contenuto nella doubleword v[i]

Se v[i] != k, allora il ciclo termina

Incrementa i e torna alla valutazione della condizione del ciclo while

Salti condizionati

- L'istruzione `slt` (esiste anche `slti`) scrive 1 in `rd` se `rs1 < rs2` e scrive 0 altrimenti
- Permette di costruire strutture di controllo con istruzioni di salto generiche

```
if (i<j)
  k=1;
else
  k=0;
```

⟹

i → x19
j → x20
k → x21

```
slt x21,x19,x20
```

Tipo R in linguaggio macchina

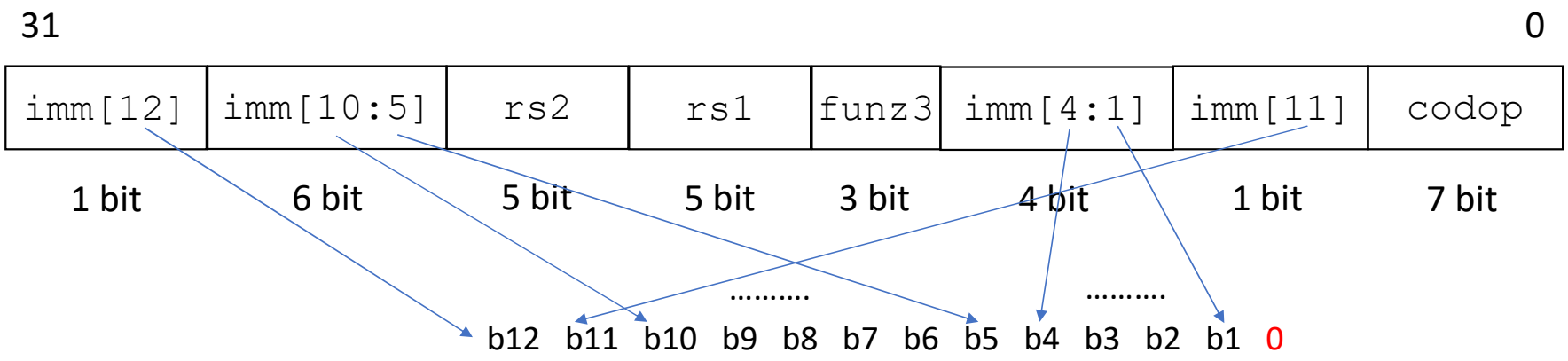
- Possiamo ad es, inserire dopo `slt` l'istruzione `beq rs1,x0,L1`
  - per il confronto su ''>='' basta invertire la condizione (`bne`)
  - per il confronto su '''>'' basta scambiare gli operandi della `slt`
  - per il confronto su ''<='', inverto condizione e scambio operandi

Salti condizionati

- 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 $2^{64}-1$
- Invece di `slt`, `slti` si useranno `sltu`, `sltiu`
  - `slt` e `sltu` forniranno un risultato diverso se un operando è negativo (es. -4 < 3 a 1 con `slt`, ma 0 con `sltu`)
  - Infatti il -4 verrebbe interpretato come un numero molto grande (es. 7 se gli operandi fossero solo di 3 bit), quindi 7 < 3 risulterebbe falso

Salti condizionati e linguaggio macchina

- Le istruzioni di salto condizionato utilizzano il **formato di tipo SB**
- Il formato può rappresentare indirizzi di salto da -4096 a 4094, in multipli di due
- Infatti, abbiamo 13 bit in complemento a 2 con solo i numeri pari



Salti condizionati e linguaggio macchina

- Si utilizza l'indirizzamento relativo al program counter (PC-relative addressing)
- Il campo immediato di 12+1 bit contiene l'offset rispetto al valore di PC (espresso in complemento a due)
- Per ottenere il prossimo valore di PC in caso di salto, viene eseguito il calcolo

```
PC = PC + immediato
```

Salti condizionati e linguaggio macchina

- Le istruzioni macchina RISC-V hanno una dimensione di 32 bit
- I possibili valori di scostamento sono compresi tra -4096 e + 4094 ([-2048*2,+2047*2])
- Non c'è un vincolo di allineamento a 32 bit, ma questo sistema consente solo di saltare ad indirizzi di memoria pari (perché b0 è implicitamente uguale a 0)
- Perché il moltiplicatore non è 4? I progettisti hanno voluto prevedere la possibilità di rappresentare le istruzioni macchina anche su 16 bit
- Noi, però, continueremo sempre a considerare istruzioni su 32 bit. Il compilatore genererà campi immediati di istruzioni di salto sempre multipli di 4, ovvero con b1 = 0 e con b0 implicitamente a 0.

Salti condizionati e linguaggio macchina

31				0
immediato	rs1	funz3	rd	codop
12 bit	5 bit	3 bit	5 bit	7 bit

Formato I

```
LOOP:     slli x10,x22,3
          add x10,x10,x25
          ld x9,0(x10)
          bne x9,x24,ENDLOOP
          addi x22,x22,1
          beq x0,x0,LOOP
ENDLOOP:
```

PC = 80000

Indirizzo	Istruzione					
80000	0000000	00011	10110	001	01010	0010011
80004	0000000	11001	01010	000	01010	0110011
80008	0000000	00000	01010	011	01001	0000011
80012	0000000	11000	01001	001	01100	1100011
80016	0000000	00001	10110	000	10110	0010011
80020	1111111	00000	00000	000	01101	1100011

Salti condizionati e linguaggio macchina

31					0
funz7	rs2	rs1	funz3	rd	codop
7 bit	5 bit	5 bit	3 bit	5 bit	7 bit

Formato R

```
LOOP:     slli x10,x22,3
          add x10,x10,x25
          ld x9,0(x10)
          bne x9,x24,ENDLOOP
          addi x22,x22,1
          beq x0,x0,LOOP
ENDLOOP:
```

PC = 80004

Indirizzo	Istruzione					
80000	0000000	00011	10110	001	01010	0010011
80004	0000000	11001	01010	000	01010	0110011
80008	0000000	00000	01010	011	01001	0000011
80012	0000000	11000	01001	001	01100	1100011
80016	0000000	00001	10110	000	10110	0010011
80020	1111111	00000	00000	000	01101	1100011

Salti condizionati e linguaggio macchina

31				0
immediato	rs1	funz3	rd	codop
12 bit	5 bit	3 bit	5 bit	7 bit

Formato I

```
LOOP:     slli x10,x22,3
          add x10,x10,x25
          ld x9,0(x10)
          bne x9,x24,ENDLOOP
          addi x22,x22,1
          beq x0,x0,LOOP
ENDLOOP:
```

PC = 80008

Indirizzo	Istruzione					
80000	0000000	00011	10110	001	01010	0010011
80004	0000000	11001	01010	000	01010	0110011
80008	0000000	00000	01010	011	01001	0000011
80012	0000000	11000	01001	001	01100	1100011
80016	0000000	00001	10110	000	10110	0010011
80020	1111111	00000	00000	000	01101	1100011

Salti condizionati e linguaggio macchina

31 0

imm[12]	imm[10:5]	rs2	rs1	funz3	imm[4:1]	imm[11]	codop
1 bit	6 bit	5 bit	5 bit	3 bit	4 bit	1 bit	7 bit

```
LOOP:     slli x10,x22,3
          add x10,x10,x25
          ld x9,0(x10)
          bne x9,x24,ENDLOOP
          addi x22,x22,1
          beq x0,x0,LOOP
ENDLOOP:
```

PC = 80012

Indirizzo	Istruzione					
80000	0000000	00011	10110	001	01010	0010011
80004	0000000	11001	01010	000	01010	0110011
80008	0000000	00000	01010	011	01001	0000011
80012	0000000	11000	01001	001	01100	1100011
80016	0000000	00001	10110	000	10110	0010011
80020	1111111	00000	00000	000	01101	1100011

Salti condizionati e linguaggio macchina

31 0

imm[12]	imm[10:5]	rs2	rs1	funz3	imm[4:1]	imm[11]	codop
1 bit	6 bit	5 bit	5 bit	3 bit	4 bit	1 bit	7 bit

```
LOOP:     slli x10,x22,3
          add x10,x10,x25
          ld x9,0(x10)
          bne x9,x24,ENDLOOP
          addi x22,x22,1
          beq x0,x0,LOOP
ENDLOOP:
```

Indirizzo	Istruzione					
80000	0000000	00011	10110	001	01010	0010011
80004	0000000	11001	01010	000	01010	0110011
80008	0000000	00000	01010	011	01001	0000011
80012	0000000	11000	01001	001	01100	1100011
80016	0000000	00001	10110	000	10110	0010011
80020	1111111	00000	00000	000	01101	1100011

PC = 80012 + 12 = 80024

0 0 0 0 0 0 0 0 0 1 1 0 0 = 12

b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0

Se la condizione di `bne` è vera

Salti condizionati e linguaggio macchina

31				0
immediato	rs1	funz3	rd	codop
12 bit	5 bit	3 bit	5 bit	7 bit

Formato I

```
LOOP:     slli x10,x22,3
          add x10,x10,x25
          ld x9,0(x10)
          bne x9,x24,ENDLOOP
          addi x22,x22,1
          beq x0,x0,LOOP
ENDLOOP:
```

PC = 80016

Indirizzo	Istruzione					
80000	0000000	00011	10110	001	01010	0010011
80004	0000000	11001	01010	000	01010	0110011
80008	0000000	00000	01010	011	01001	0000011
80012	0000000	11000	01001	001	01100	1100011
80016	0000000	00001	10110	000	10110	0010011
80020	1111111	00000	00000	000	01101	1100011

Se la condizione di `bne` è falsa

Salti condizionati e linguaggio macchina

31 0

imm[12]	imm[10:5]	rs2	rs1	funz3	imm[4:1]	imm[11]	codop
1 bit	6 bit	5 bit	5 bit	3 bit	4 bit	1 bit	7 bit

```
LOOP:     slli x10,x22,3
          add x10,x10,x25
          ld x9,0(x10)
          bne x9,x24,ENDLOOP
          addi x22,x22,1
          beq x0,x0,LOOP
ENDLOOP:
```

PC = 80020

Indirizzo	Istruzione					
80000	0000000	00011	10110	001	01010	0010011
80004	0000000	11001	01010	000	01010	0110011
80008	0000000	00000	01010	011	01001	0000011
80012	0000000	11000	01001	001	01100	1100011
80016	0000000	00001	10110	000	10110	0010011
80020	1111111	00000	00000	000	01101	1100011

Se la condizione di `bne` è falsa

Salti condizionati e linguaggio macchina

31 0

imm[12]	imm[10:5]	rs2	rs1	funz3	imm[4:1]	imm[11]	codop
1 bit	6 bit	5 bit	5 bit	3 bit	4 bit	1 bit	7 bit

```
LOOP:     slli x10,x22,3
          add x10,x10,x25
          ld x9,0(x10)
          bne x9,x24,ENDLOOP
          addi x22,x22,1
          beq x0,x0,LOOP
ENDLOOP:
```

Indirizzo	Istruzione					
80000	0000000	00011	10110	001	01010	0010011
80004	0000000	11001	01010	000	01010	0110011
80008	0000000	00000	01010	011	01001	0000011
80012	0000000	11000	01001	001	01100	1100011
80016	0000000	00001	10110	000	10110	0010011
80020	1111111	00000	00000	000	01101	1100011

PC = 80020 - 20 = 80000

1 1 1 1 1 1 1 1 0 1 1 0 0 = -20

b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0

Se la condizione di `beq` è vera (in questo caso lo è sempre)

Salti incondizionati

- Jump and link register
  - jal e jalr
- Costrutto case/switch

Pseudoistruzioni

- Il linguaggio assembler fornisce “pseudoistruzioni”
  - Una pseudoistruzione esiste solo in linguaggio assembler e non in linguaggio macchina
  - l’assemblatore traduce le pseudoistruzioni nel linguaggio macchina delle corrispondenti istruzioni
  - Quando si effettua per es. il debugging è necessario ricordare quali siano le istruzioni reali
  - L’elenco delle pseudoistruzioni è presente nel manuale di riferimento
- Esempio

```
mv x5,x6  ──►  addi x5,x6,0

not x5,x6  ──►  xori x5,x6,-1
```

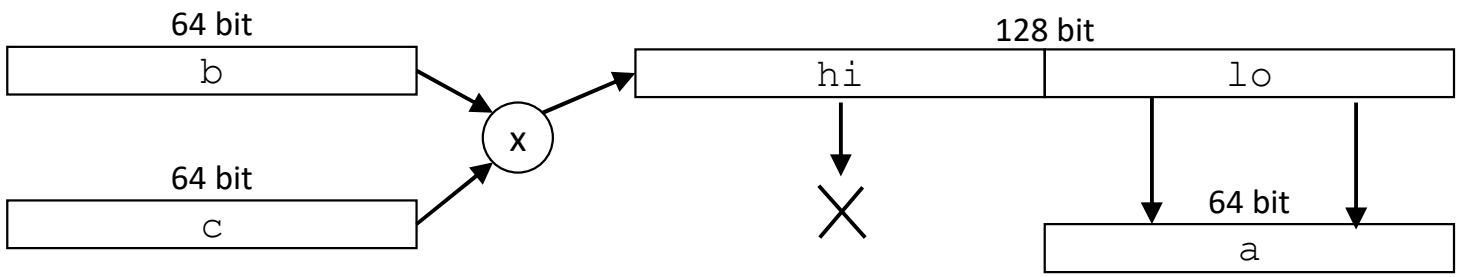
Istruzioni aritmetiche: moltiplicazione

```
a = b*c
```

Linguaggio C

```
mul a, b, c
```

RISC-V assembler



- Viene utilizzato un registro interno nascosto all'utente
- I 64 bit più significativi possono essere ottenuti con l'istruzione

```
mulh a, b, c
```

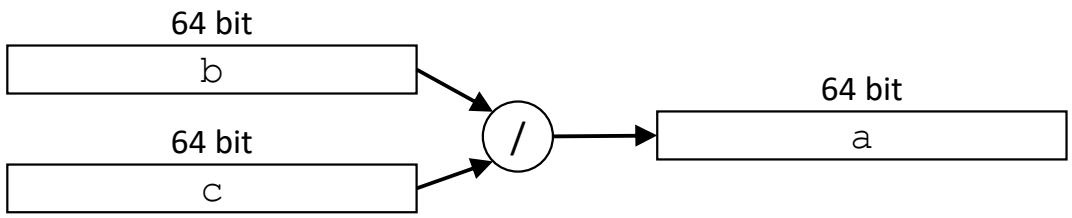
Istruzioni aritmetiche: divisione e resto

```
a = b/c
```

Linguaggio C

```
div a, b, c
```

RISC-V assembler

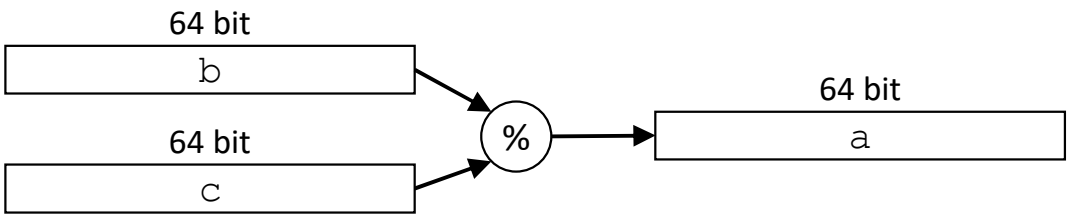



```
a = b%c
```

Linguaggio C

```
rem a, b, c
```

RISC-V assembler



Le procedure

- Porzioni di codice associate ad un **nome** che possono essere invocate più volte e che eseguono un compito specifico, avendo come input una lista di **parametri** e come output un **valore di ritorno**
- Vantaggi
  - Astrazione
  - Riusabilità del codice (librerie)
  - Maggiore organizzazione del codice
  - Testing più agevole

```
int somma(int x, int y)
{
  int rst;
  rst = x + y;
  return rst;
}
```

Procedure: un esempio

Programma (procedura)
chiamante

```
…
f=f+1;
risultato = somma(f,g);
…
```

Procedura
chiamata

```
int somma(int x, int y)
{
  int rst;
  rst = x + y + 2;
  return rst;
}
```

Le procedure: passi da seguire

- Chiamante
  - Mettere i parametri in un luogo accessibile alla procedura
  - Trasferire il controllo alla procedura
- Chiamato
  - Acquisire le risorse necessarie per l'esecuzione della procedura
  - Eseguire il compito richiesto
  - Mettere il risultato in un luogo accessibile al programma chiamante
  - Restituire il controllo al punto di origine

  (la stessa procedura può essere chiamata in differenti punti di un programma)

Modifica del flusso di programma: invocazione

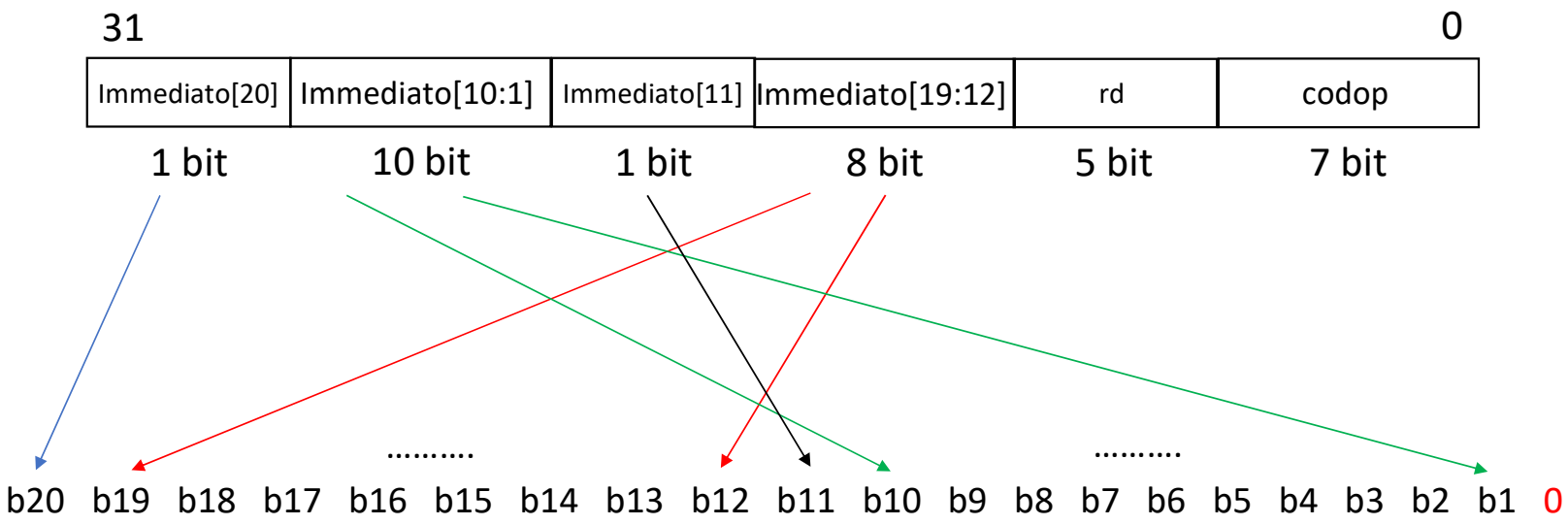
`jal IndirizzoProcedura` Jump-and-link

- Salta all'indirizzo (offset) con etichetta `IndirizzoProcedura`
- Memorizza il valore dell'istruzione successiva `PC+4` nel registro `x1` (**return address, `ra`**)
- Pseudo-istruzione, abbreviazione di

  `jal x1, IndirizzoProcedura`

JAL e linguaggio macchina

- Viene introdotto un nuovo tipo: J



Modifica del flusso di programma: ritorno al chiamante

Jump-and-link register

```
jalr rd, offset(rs1)
```

- Salta ad un indirizzo qualsiasi
  - `x[rd]=PC+4; PC=x[rs1]+sext(offset)&~1`

Segno esteso a 64bit

Bit 0 azzerato

- La procedura chiamata come ultima istruzione esegue

Pseudoistruzione

```
jr ra
```

```
jalr x0, 0(x1)
```

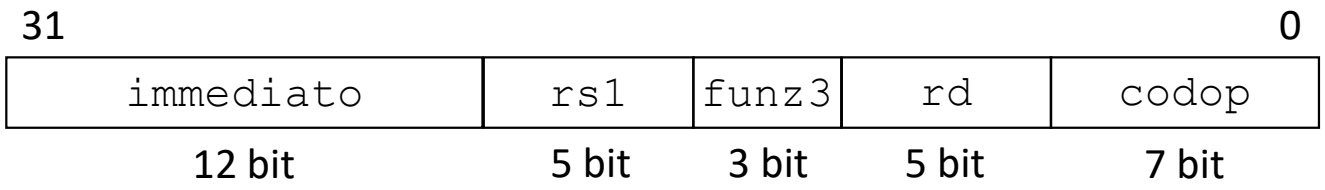
Tipo I in linguaggio macchina

`x0=PC+4; PC=x1+0`

Operazione nulla

Return Address

Formato di tipo I (immediato)



- Permette di codificare anche `jalr`
- Il campo immediato
  - Rappresentato in complemento a due
  - Valori possibili: da –`2048` a `+2047`

Side effect – sovrascrittura dei registri (1)

```
int somma(int x, int y) {
  int rst;
  rst = x + y + 2;
  return rst;
}

….
f=f+1
risultato=somma(f,g)
…
```

x → x10
y → x11
rst → x20
f → x6

⇨

```
SOMMA: add  x5,x10,x11
       addi x20,x5,2
       jalr x0,0(x1)
…
…
       addi x6,x6,1
…
       jal SOMMA
```

- **Problema**: che cosa succede se il registro x5 contiene un valore usato dalla procedura chiamante?
- **Soluzione**:

Side effect – sovrascrittura dei registri (1)

```
int somma(int x, int y) {
  int rst;
  rst = x + y + 2;
  return rst;
}

….
f=f+1
risultato=somma(f,g)
…
```

x → x10
y → x11
rst → x20
f → x6

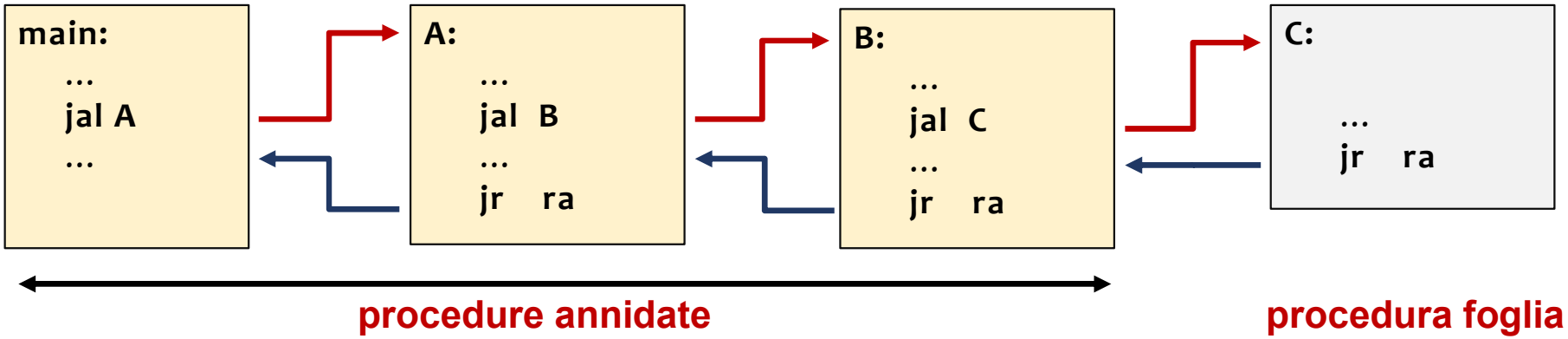
⇒

```
SOMMA: add  x5,x10,x11
       addi x20,x5,2
       jalr x0,0(x1)
…
…
       addi x6,x6,1
…
       jal SOMMA
```

- **Problema**: che cosa succede se il registro x5 contiene un valore usato dalla procedura chiamante?
- **Soluzione**: **salvare** il valore di x5 **in memoria** prima di utilizzarlo (e ripristinarlo prima del ritorno al chiamante)

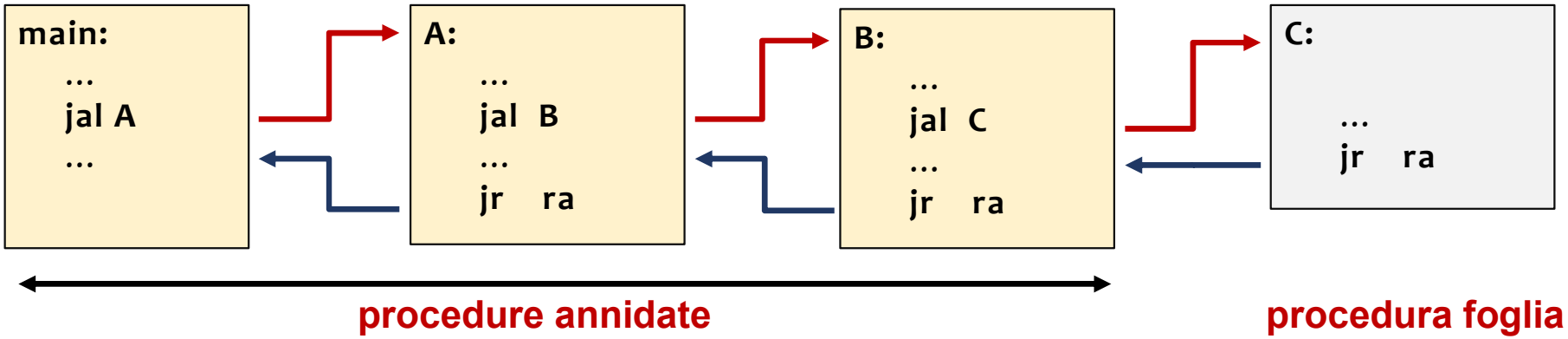
Side effect – procedure annidate (2)

- Problema
  - Nel caso di procedure annidate, il return address (`x1`) viene sovrascritto
- Soluzione:



Side effect – procedure annidate (2)

- Problema
  - Nel caso di procedure annidate, il return address (`x1 o ra`) viene sovrascritto
- Soluzione:
  - La procedura chiamata, deve **salvare in memoria** il valore di `x1` **prima** di chiamare la procedura annidata con l'istruzione `jal`



Side effect – parametri numerosi (3)

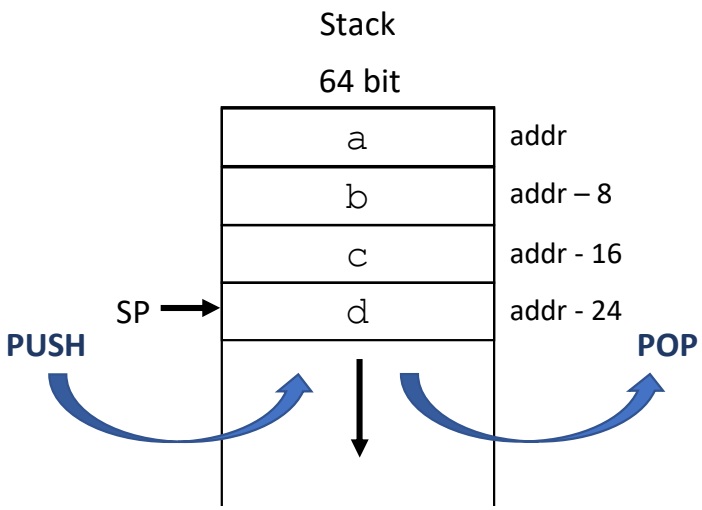
- Problema:
  - che cosa succede se i parametri e le variabili di una procedura superano il numero di registri disponibili?
- Soluzione:

Side effect – parametri numerosi (3)

- Problema:
  - che cosa succede se i parametri e le variabili di una procedura superano il numero di registri disponibili?
- Soluzione:
  - **Salvare** temporaneamente i dati **in memoria** per caricarli nei registri **prima** del loro utilizzo all'interno della procedura

Lo stack

- In quale area di memoria è possibile salvare i registri per evitarne la perdita del valore?
- In memoria viene definita una struttura dati dinamica, lo stack
- Accesso: Last In First Out (LIFO)
- Operazioni:
  - PUSH: aggiunge un elemento «in cima allo stack»
  - POP: rimuove un elemento «dalla cima dello stack»
  - La cima dello stack è identificata dallo Stack Pointer (SP)



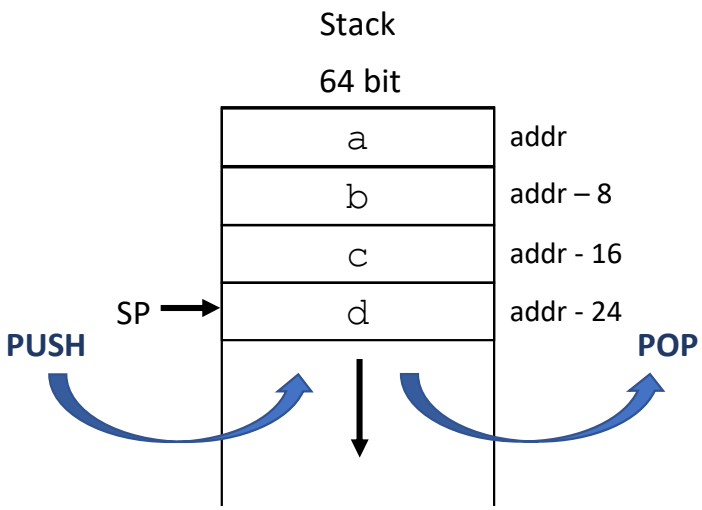
Lo stack

- In RISC V si usa la convenzione
  - **grow-down**: lo stack cresce da indirizzi di memoria alti verso indirizzi di memoria bassi
  - **last-full**: lo stack pointer (SP) contiene l'indirizzo dell'ultima cella di memoria occupata nello stack
  - Il valore di SP è salvato nel registro x2 (a.k.a. `sp`)
- PUSH
  - Decrementa SP
  - Scrive in M[SP]

```
addi sp, sp, -8
sd   x20, 0(sp)
```

- POP
  - Legge da M[SP]
  - Incrementa SP

```
ld   x20, 0(sp)
addi sp, sp, 8
```

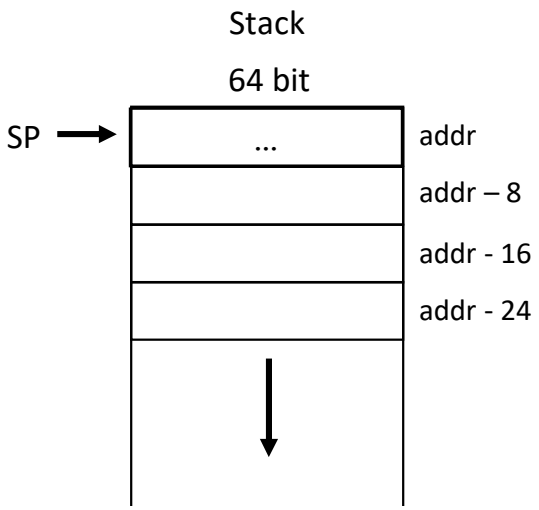


Nell’esempio di prima:

```
int somma(int x, int y) {
  int rst;
  rst = x + y + 2;
  return rst;
}

….
f=f+1
risultato=somma(f,g)
…
```

x → x10
y → x11
rst → x20
f → x6

```
SOMMA: addi sp,sp,-16
       sd   x5,0(sp)
       sd   x20,8(sp)
       add  x5,x10,x11
       addi x20,x5,2
       addi x10,x20,0
       ld   x5,0(sp)
       ld   x20,8(sp)
       addi sp,sp,16
       jalr x0,0(x1)
…
…
       addi x6,x6,1
…
       jal SOMMA
```

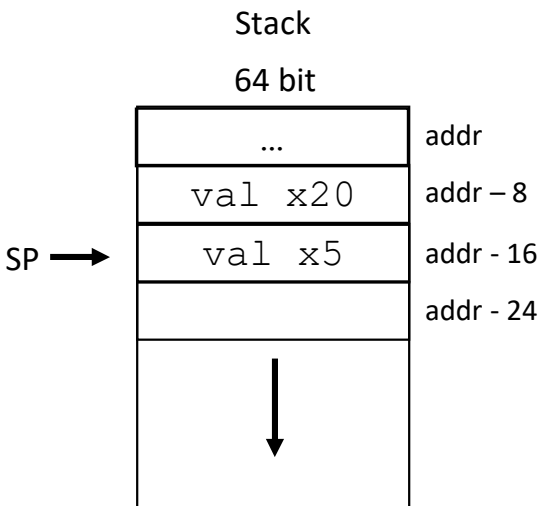


Nell’esempio di prima:

```
int somma(int x, int y) {
  int rst;
  rst = x + y + 2;
  return rst;
}

….
f=f+1
risultato=somma(f,g)
…
```

x → x10
y → x11
rst → x20
f → x6

```
SOMMA:  addi  sp,sp,-16
        sd    x5,0(sp)
        sd    x20,8(sp)
        add   x5,x10,x11
        addi  x20,x5,2
        addi  x10,x20,0
        ld    x5,0(sp)
        ld    x20,8(sp)
        addi  sp,sp,16
        jalr  x0,0(x1)
…
…
        addi  x6,x6,1
…
        jal   SOMMA
```



PUSH: salvataggio del valore di x5 e x20

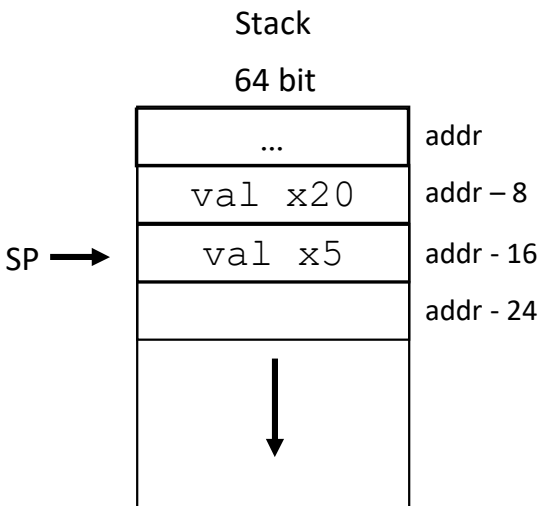
Nell’esempio di prima:

```
int somma(int x, int y) {
  int rst;
  rst = x + y + 2;
  return rst;
}

….
f=f+1
risultato=somma(f,g)
…
```

x → x10
y → x11
rst → x20
f → x6

⇨

```
SOMMA: addi sp,sp,-16
       sd   x5,0(sp)
       sd   x20,8(sp)
       add  x5,x10,x11
       addi x20,x5,2
       addi x10,x20,0
       ld   x5,0(sp)
       ld   x20,8(sp)
       addi sp,sp,16
       jalr x0,0(x1)
…
…
       addi x6,x6,1
…
       jal SOMMA
```



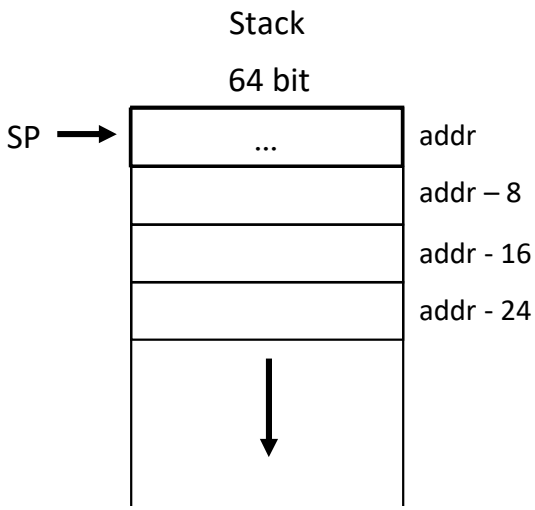
Corpo della procedura

Nell’esempio di prima:

```
int somma(int x, int y) {
  int rst;
  rst = x + y + 2;
  return rst;
}

….
f=f+1
risultato=somma(f,g)
…
```

x → x10
y → x11
rst → x20
f → x6

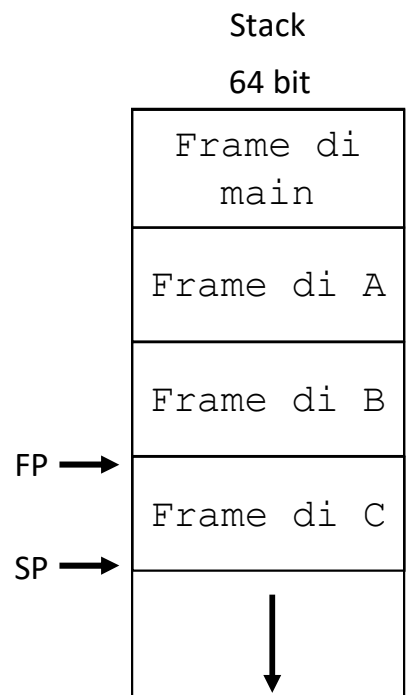
```
SOMMA: addi sp,sp,-16
       sd   x5,0(sp)
       sd   x20,8(sp)
       add  x5,x10,x11
       addi x20,x5,2
       addi x10,x20,0
       ld   x5,0(sp)
       ld   x20,8(sp)
       addi sp,sp,16
       jalr x0,0(x1)
…
…
       addi x6,x6,1
…
       jal SOMMA
```

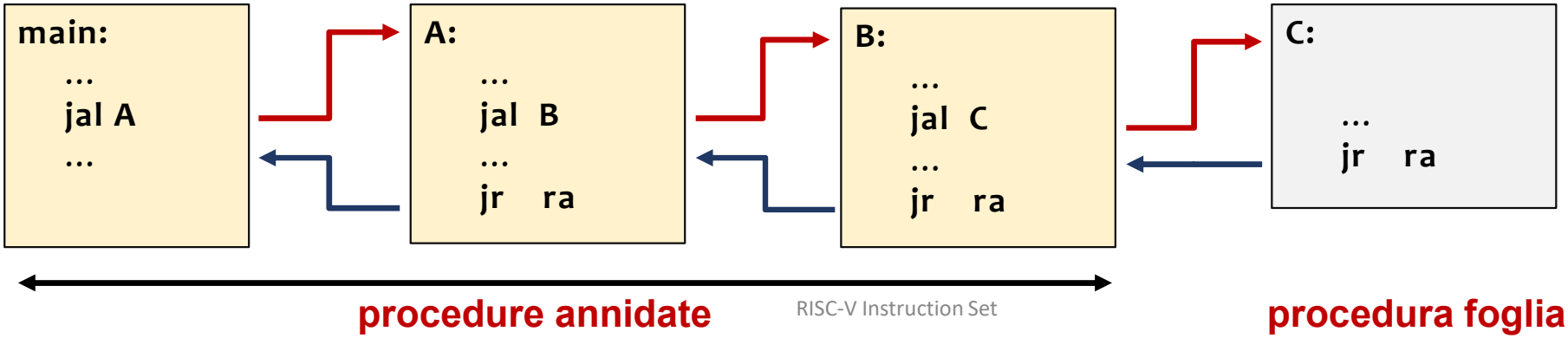


POP: ripristino del valore di x5 e x20

Record di attivazione

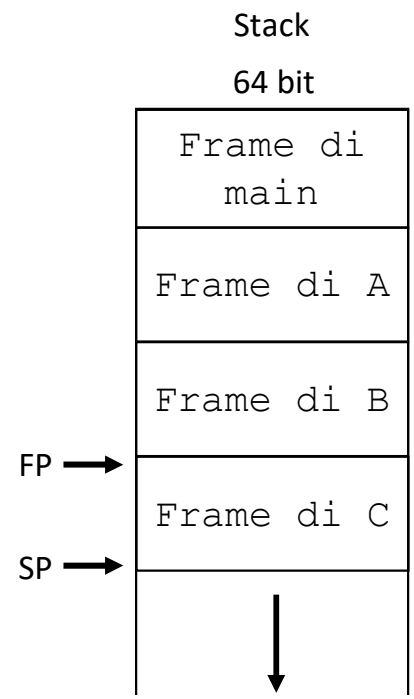
- Ad ogni esecuzione di procedura corrisponde un blocco di memoria all'interno dello stack che viene chiamato **record di attivazione o frame**
- Quando viene richiamata la funzione F, viene aggiunto un record di attivazione allo stack
- Quando la funzione F termina, il corrispondente record di attivazione viene cancellato

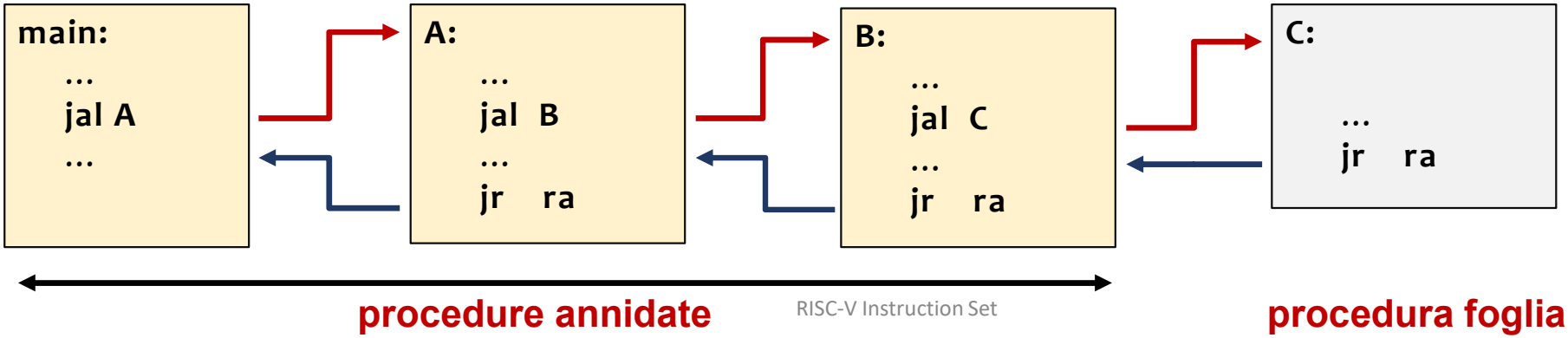




Record di attivazione

- Il frame in cima allo stack rappresenta il frame della procedura in esecuzione
- I confini del frame sono identificati da
  - Base: il Frame Pointer (registro x8, fp)
  - Cima: lo Stack Pointer (registro x2, sp)





I registri e convenzioni sul loro uso

Registro	Nome	Utilizzo
x0	zero	Costante zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Puntatore a thread
x8	s0 / fp	Frame pointer (il contenuto va preservato se utilizzato dalla procedura chiamata)
x10-x11	a0-a1	Passaggio di parametri nelle procedure e valori di ritorno
x12-x17	a2-a7	Passaggio di parametri nelle procedure
x5-x7 x28-x31	t0-t2 t3-t6	Registri temporanei, non salvati in caso di chiamata
x9	s1	Registri da salvare: il contenuto va preservato se utilizzati dalla procedura chiamata
x18-x27	s2-s11	

Convenzione nell'uso e salvataggio dei registri

- Chi è responsabile di salvare i registri quando si effettuano chiamate di funzioni?
  - La funzione chiamante conosce quali registri sono importanti per sé e che dovrebbero essere salvati
  - La funzione chiamata conosce quali registri userà e che dovrebbero essere salvati prima di modificarli
- Bisogna evitare le inefficienze → Minimo salvataggio dei registri:
  - La funzione chiamante potrebbe salvare tutti i registri che sono importanti per sé, anche se la procedura chiamata non li modificherà
  - La funzione chiamata potrebbe salvare tutti i registri che si appresta a modificare, anche quelli che non verranno poi utilizzati dalla procedura chiamante una volta che la procedura chiamata le avrà restituito il controllo

Convenzione nell'uso e salvataggio dei registri

- Necessità di stabilire delle convenzioni
- I registri `x10-x17 (a0-a7), x5-x7` e `x28-x31 (t0-t6)`
  - possono essere modificati dal chiamato senza nessun meccanismo di ripristino
  - Il chiamante se necessario dovrà salvare i valori dei registri prima dell'invocazione della procedura
- I registri `x1(ra), x2(sp), x3(gp), x4 (tp), x8(fp/s0), x9` e `x18-x27 (s1-s11)`
  - Se modificati dal chiamato devono essere salvati e poi ripristinati prima del ritorno al chiamante
  - Il chiamante non è tenuto al loro salvataggio e ripristino

Convenzioni nell'uso e salvataggio dei registri: perchè?

- Si supponga che
  - La funzione chiamante abbia bisogno del valore dei registri `s1,s2,s3` dopo l'invocazione di una funzione `f`
  - La funzione chiamata `f` abbia bisogno di modificare i valori dei registri `t0` e `t1`
- Quale convenzione è la più vantaggiosa?

Convenzioni nell'uso e salvataggio dei registri: perchè?

1. Nessuna convenzione: chiamante e chiamato si devono preoccupare di salvare e ripristinare i registri che gli interessano
   - Chiamante: 3 salvataggi e ripristini (`s1, s2, s3`)
   - Chiamato: 2 salvataggi e ripristini (`t0, t1`)
2. Il chiamante si deve preoccupare di salvare i registri che intende preservare dopo l'invocazione di `f`
   - Chiamante: 3 salvataggi e ripristini (`s1, s2, s3`)
   - Chiamato: Nessun salvataggio

Convenzioni nell'uso e salvataggio dei registri: perchè?

1. Il chiamato si deve preoccupare di salvare i registri che intende modificare durante l'esecuzione di `f`
   - Chiamante: nessun salvataggio e ripristino
   - Chiamato: 2 salvataggi e ripristini (`t0, t1`)
2. Convenzione RISC-V
   - Chiamante: nessun salvataggio e ripristino
   - Chiamato: nessun salvataggio e ripristino

Le fasi di una invocazione di procedura

Possiamo dividere l'invocazione di una procedura in 7 fasi:

1. Pre-chiamata
2. Invocazione della procedura

Chiamante

3. Prologo del chiamato
4. Corpo della procedura
5. Epilogo lato chiamato
6. Ritorno al chiamante

Chiamato

7. Post-chiamata

Chiamante

Le fasi di una invocazione di procedura

1. Pre-chiamata
2. Invocazione della procedura
3. Prologo del chiamato
4. Corpo della procedura
5. Epilogo lato chiamato
6. Ritorno al chiamante
7. Post-chiamata

Chiamante: 1–2; Chiamato: 3–6; Chiamante: 7

Fase 1 – Pre-chiamata del chiamante

- Eventuale salvataggio registri da preservare nel chiamante
  - Si assume che `x10-x17 (a0-a7)`, `x5-x7` e `x28-x31 (t0-t6)`, possano essere sovrascritti dal chiamato
  - se li si vuole preservare vanno salvati nello stack (dal chiamante) – vedi caso 1
  - il caso 2 mostra un caso in cui non è necessario salvare il contenuto del registro associato alla variabile `f`

- Preparazione degli argomenti della funzione
  - I primi 8 argomenti vengono posti in `x10-x17`, ovvero `a0-a7` (nuovi valori)
  - Gli eventuali altri argomenti oltre l'ottavo vanno salvati nello stack (EXTRA_ARGS), così che si trovino subito sopra il frame della funzione chiamata

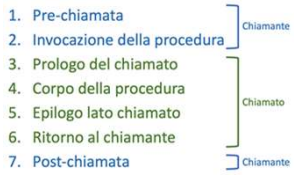
```
int somma(int x, int y) {
  x=x+y;
  return x;
}
….
f=f+1;
risultato=somma(f,g);
printf("%d", f);
return;
```

1

```
int somma(int x, int y) {
  x=x+y;
  return x;
}
….
f=f+1;
risultato=somma(f,g);
return;
```

2

Le fasi di una invocazione di procedura



Fase 2 – Invocazione della procedura

- Istruzione `jal NOME_PROCEDURA`

Fase 3 – Prologo lato chiamato

- Eventuale allocazione del call-frame sullo stack (aggiornare `sp`)
  - Salvataggio di `x1(ra)` nel caso in cui la procedura non sia foglia
  - Salvataggio di `x8(fp)`, solo se utilizzato all'interno della procedura
  - Salvataggio di `x9` e `x18-x27 (s1-s11)` se utilizzati all'interno della procedura (il chiamante si aspetta di trovarli intatti)
  - Salvataggio degli argomenti `x10-x17` (`a0-a7`) solo se la funzione li riuserà successivamente a ulteriori chiamate a funzione (procedure annidate) (nota: `x10-x17` sono trattati come i registri temporanei – possono essere sovrascritti)
- Eventuale inizializzazione di `fp`: punta al nuovo call-frame

Le fasi di una invocazione di procedura

1. Pre-chiamata
2. Invocazione della procedura
3. Prologo del chiamato
4. Corpo della procedura
5. Epilogo lato chiamato
6. Ritorno al chiamante
7. Post-chiamata

Chiamante

Chiamato

Chiamante

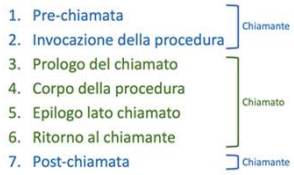
Fase 4 – Corpo della procedura

- Istruzioni che implementano il corpo della procedura

Fase 5 – Epilogo lato chiamato

- Se deve essere restituito un valore dalla funzione
  - Tale valore viene posto in `x10` (e `x11`) ovvero `a0-a1`
- I registri (se salvati) devono essere ripristinati
  - `x9` e `x18-x27 (s1-s11)`
  - `x1(ra)`
  - `x8(fp)`
- Notare che `sp` deve solo essere aumentato di opportuno offset (lo stesso sottratto nella Fase 3)

Le fasi di una invocazione di procedura



Fase 6 – Ritorno al chiamante

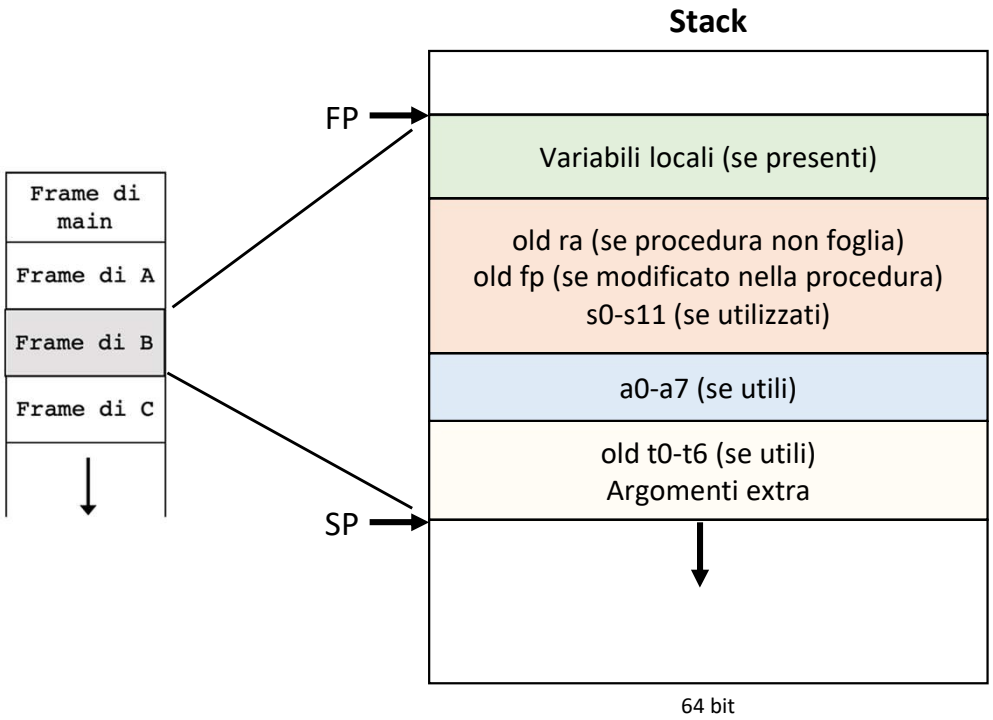
- Istruzione `jalr x0, 0(x1)` (o pseudo-istruzione `jr ra`)

Fase 7 – Post-chiamata lato chiamante

- Eventuale uso del risultato della funzione (in `x10` e `x11`, cioè `a0-a1`)
- Ripristino dei valori `x5-x7` e `x28-x31 (t0-t6)`, `x10-x17 (a0-a7)` vecchi, eventualmente salvati

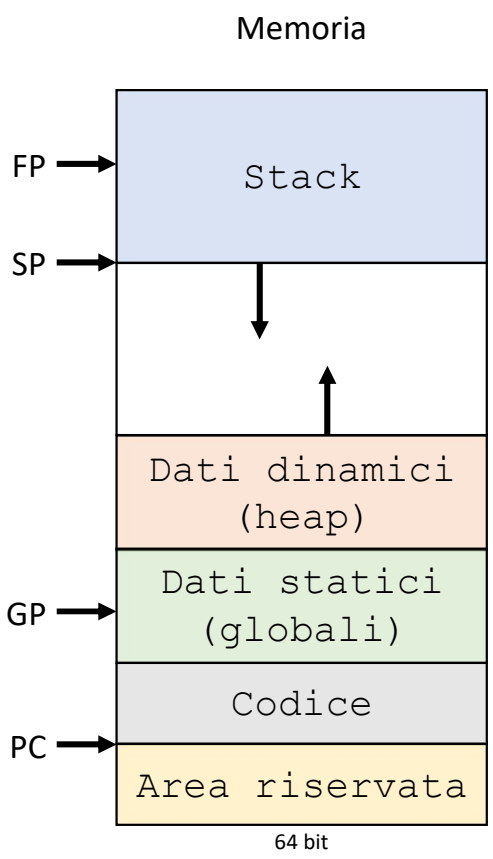
Record di attivazione: struttura

- Se utilizzato, il registro `fp` (frame pointer) viene inizializzato al valore di `sp` all'inizio della chiamata
- `fp` consente di avere un riferimento alle variabili locali che non muta con l'esecuzione della procedura
- Se lo stack non contiene variabili locali alla procedura, il compilatore risparmia tempo di esecuzione evitando di impostare e ripristinare il frame.



Organizzazione della memoria

- In memoria, oltre allo stack, trovano posto
  - Dati allocati dinamicamente
    (ad esempio, attraverso la malloc() del C)
  - Dati statici e variabili globali
  - Codice del programma
- Lo stack e i dati dinamici crescono in direzioni differenti, in modo da ottimizzarne la gestione



Esempio: area di un triangolo

Invocazione della funzione `area` con parametri `base=20 ,altezza=23`

```
long moltiplicazione(long a, long b) {
  long rst=a;
  for(long i=1;i<b;i++)
    rst = rst+a;
  return rst;
}

long area(long base, long altezza) {
  long rst = moltiplicazione(base, altezza)/2;
  return rst;
}
…
printf("Area = %li\n",area(20,23));
…
```

Si supponga di non avere una operazione di moltiplicazione

1. Pre-chiamata
2. Invocazione della procedura

Chiamante

3. Prologo del chiamato
4. Corpo della procedura
5. Epilogo lato chiamato
6. Ritorno al chiamante

Chiamato

7. Post-chiamata

Chiamante

```
_start:
  li a0, 20 # salvo la base in a0
  li a1, 23 # salvo l'altezza in a1

# chiama area(base,altezza)
  jal  ra, area       # altezza in a0, base in a1
  add  t0, a0, zero   # salva il risultato in t0

# stampa messaggio per il risultato
  la   a0, visris   # pseudo-istruzione
  addi a7, zero,  4
  ecall

# stampa il risultato
  add  a0, t0, zero
  addi a7, zero,  1
  ecall

# stampa \n
  la   a0, RTN       # pseudo-istruzione
  addi a7, zero,  4
  ecall

# exit
  addi a7, zero,  10
  ecall
```

Il codice può essere ottimizzato

Esempio: area di un triangolo

```
long moltiplicazione(long a, long b) {
  long rst=a;
  for(long i=1;i<b;i++)
    rst = rst+a;
  return rst;
}

long area(long base, long altezza) {
  long rst = moltiplicazione(base, altezza)/2;
  return rst;
}
…
printf("Area = %li\n",area(20,23));
…
```

Funzione `area` con invocazione della funzione `moltiplicazione`

```
area:
# crea il call frame sullo stack (24 byte=ra+fp+rst)
# lo stack cresce verso il basso
    addi sp, sp, -24     # allocazione del call frame nello stack
    sd   ra, 8(sp)       # salvataggio dell'indirizzo di ritorno
    sd   fp, 0(sp)       # salvataggio del precedente frame pointer
    addi  fp, sp, 16     # aggiornamento del frame pointer

# calcolo dell'area
    jal ra, moltiplicazione
    sd a0, 0(fp)         # salva il risultato della moltiplicazione
                         # nella variabile locale rst
    srai a0,a0,1

# uscita dalla funzione
    ld   fp, 0(sp)       # recupera il frame pointer
    ld   ra, 8(sp)       # recupera l'indirizzo di ritorno
    addi sp, sp, 24      # elimina il call frame dallo stack
    jr   ra              # ritorna al chiamante, pseudo-istruzione
```

Si supponga di non avere una operazione di moltiplicazione

1. Pre-chiamata
2. Invocazione della procedura
3. Prologo del chiamato
4. Corpo della procedura
5. Epilogo lato chiamato
6. Ritorno al chiamante
7. Post-chiamata

Chiamante: 1–2; Chiamato: 3–6; Chiamante: 7

Il codice può essere ottimizzato

Esempio: area di un triangolo

```
long moltiplicazione(long a, long b) {
  long rst=a;
  for(long i=1;i<b;i++)
    rst = rst+a;
  return rst;
}

long area(long base, long altezza) {
  long rst = moltiplicazione(base, altezza)/2;
  return rst;
}
…
printf("Area = %li\n",area(20,23));
…
```

Si supponga di non avere una operazione di moltiplicazione

1. Pre-chiamata
2. Invocazione della procedura

Chiamante

3. Prologo del chiamato
4. Corpo della procedura
5. Epilogo lato chiamato
6. Ritorno al chiamante

Chiamato

7. Post-chiamata

Chiamante

Funzione `moltiplicazione`

```
moltiplicazione:
  addi sp, sp, -16   # allocazione del call frame nello stack
  sd   ra, 8(sp)     # salvataggio dell'indirizzo di ritorno
  sd   fp, 0(sp)     # salvataggio del precedente frame pointer
  addi  fp, sp, 8    # aggiornamento del frame pointer
  sd a0, 0(fp)       # salvataggio variabile locale

  ld t2, 0(fp)
  li t0, 1
for:
  bge t0,a1,endfor
  add t2,a0,t2
  addi t0,t0,1
  j for                # pseudo-istruzione
endfor:
  ld   fp, 0(sp)       # recupera il frame pointer
  ld   ra, 8(sp)       # recupera l'indirizzo di ritorno
  addi sp, sp, 16      # elimina il call frame dallo stack
  add a0,t2,zero       # valore di ritorno in a0
  jr   ra              # ritorna al chiamante, pseudo-istruzione
```

Il codice può essere ottimizzato

Esempio: area di un triangolo

```
long moltiplicazione(long a, long b) {
  long rst=a;
  for(long i=1;i<b;i++)
    rst = rst+a;
  return rst;
}

long area(long base, long altezza) {
  long rst = moltiplicazione(base, altezza)/2;
  return rst;
}
…
printf("Area = %li\n",area(20,23));
…
```

Si supponga di non avere una operazione di moltiplicazione

1. Pre-chiamata
2. Invocazione della procedura

Chiamante

3. Prologo del chiamato
4. Corpo della procedura
5. Epilogo lato chiamato
6. Ritorno al chiamante

Chiamato

7. Post-chiamata

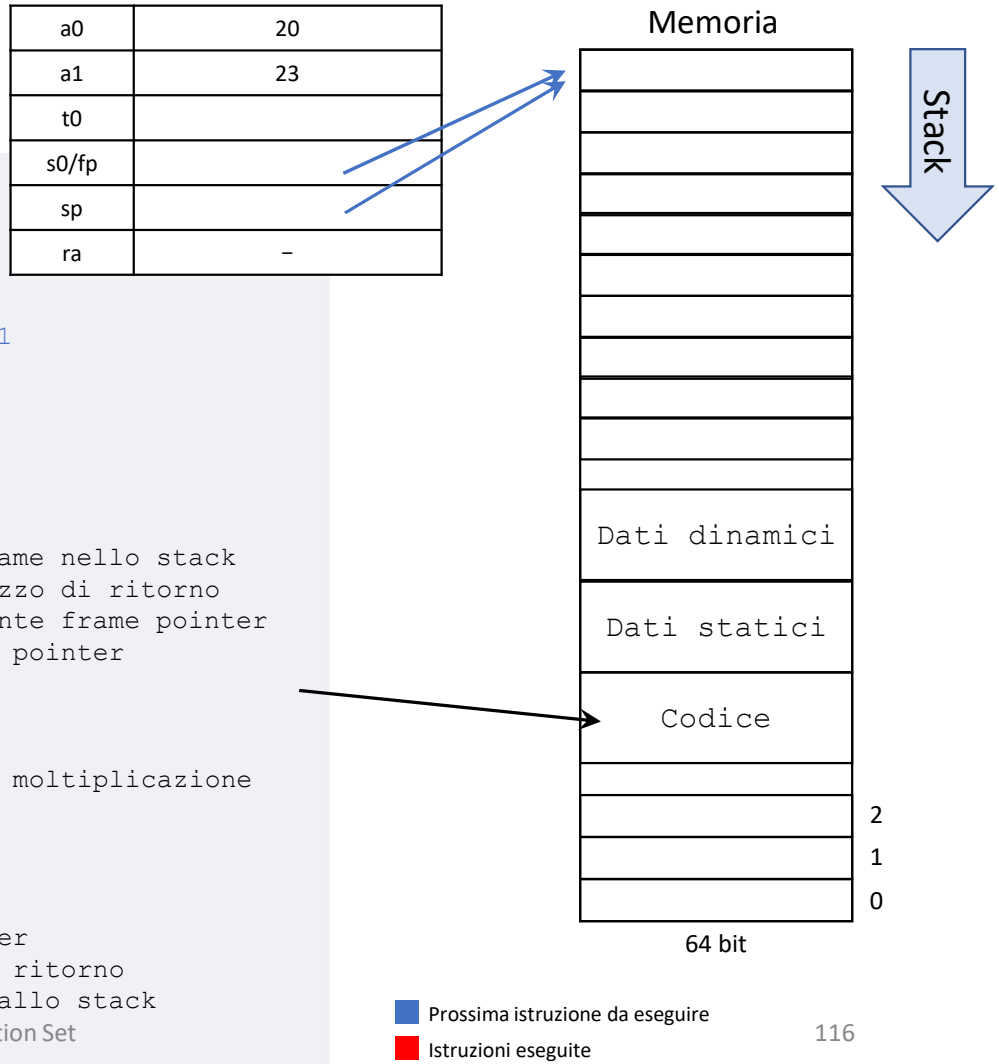
Chiamante

Il codice può essere ottimizzato

```
moltiplicazione:
  addi sp, sp, -16   # allocazione del call frame nello stack
  sd   ra, 8(sp)     # salvataggio dell'indirizzo di ritorno
  sd   fp, 0(sp)     # salvataggio del precedente frame pointer
  addi  fp, sp, 8    # aggiornamento del frame pointer
  sd a0, 0(fp)       # salvataggio variabile locale
  ld t2, 0(fp)
  li t0, 1
for:
  bge t0,a1,endfor
  add t2,a0,t2
  addi t0,t0,1
  j for                # pseudo-istruzione
endfor:
  ld   fp, 0(sp)       # recupera il frame pointer
  ld   ra, 8(sp)       # recupera l'indirizzo di ritorno
  addi sp, sp, 16      # elimina il call frame dallo stack
  add a0,t2,zero       # valore di ritorno in a0
  jr   ra              # ritorna al chiamante, pseudo-istruzione
area:
# crea il call frame sullo stack (24 byte=ra+fp+rst)
addi sp, sp, -24     # allocazione del call frame nello stack
    sd   ra, 8(sp)       # salvataggio dell'indirizzo di ritorno
    sd   fp, 0(sp)       # salvataggio del precedente frame pointer
    addi  fp, sp, 16     # aggiornamento del frame pointer
# calcolo dell'area
    jal ra, moltiplicazione
    sd a0, 0(fp)  # salva il risultato della moltiplicazione in rst
srai a0,a0,1
# uscita dalla funzione
    ld   fp, 0(sp)       # recupera il frame pointer
    ld   ra, 8(sp)       # recupera l'indirizzo di ritorno
    addi sp, sp, 24      # elimina il call frame dallo stack
    jr   ra              # ritorna al chiamante, pseudo-istruzione
_start:
  li a0, 20 # salvo la base in a0
  li a1, 23 # salvo l'altezza in a1
# chiama area(base,altezza)
  jal  ra, area       # altezza in a0, base in a1
  add  t0, a0, zero   # salva il risultato in t0
# stampa messaggio per il risultato
  la   a0, visris             # pseudo-istruzione
  addi a7, zero,  4
  ecall
# stampa il risultato
  add  a0, t0, zero
  addi a7, zero,  1
  ecall
# stampa \n
  la   a0, RTN  # pseudo-istruzione
  addi a7, zero,  4
  ecall
# exit
  addi a7, zero,  10
  ecall
```

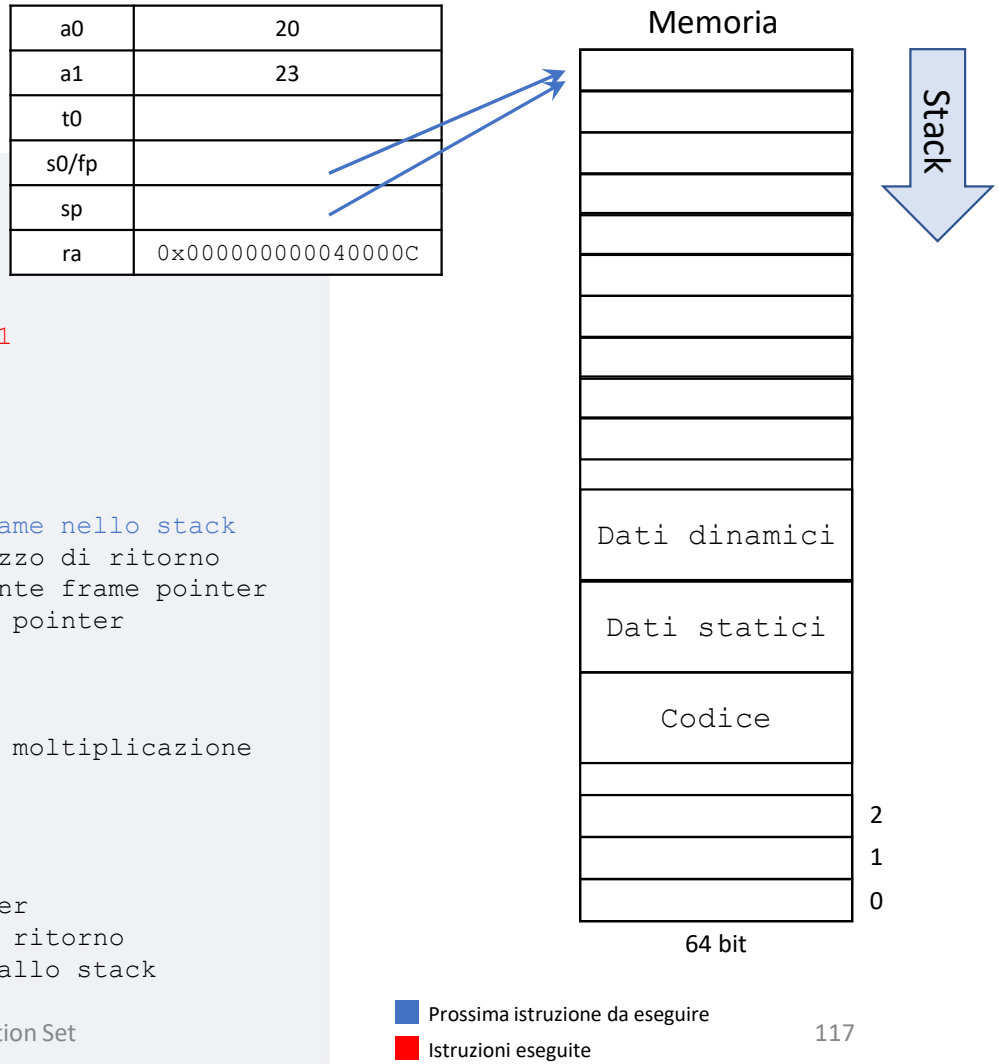
Un esempio

```
_start:
0x0000000000400000  li a0, 20 # salvo la base in a0
0x0000000000400004  li a1, 23 # salvo l'altezza in a1

# chiama triangolo(base,altezza)
0x0000000000400008  jal  ra, area       # altezza in a0, base in a1
0x000000000040000C  add  t0, a0, zero   # salva il risultato in t0
…

area:
# crea il call frame sullo stack (24 byte=ra+fp+rst)
# lo stack cresce verso il basso
0x0000000000400010  addi sp, sp, -24     # allocazione del call frame nello stack
0x0000000000400014  sd   ra, 8(sp)       # salvataggio dell'indirizzo di ritorno
0x0000000000400018  sd   fp, 0(sp)       # salvataggio del precedente frame pointer
0x000000000040001C  addi  fp, sp, 16     # aggiornamento del frame pointer

# calcolo dell'area
0x0000000000400020  jal ra, moltiplicazione
0x0000000000400024  sd a0, 0(fp)         #salva il risultato della moltiplicazione
                         #nella variabile locale rst
0x0000000000400028  srai a0,a0,1

# uscita dalla funzione
0x000000000040002C  ld   fp, 0(sp)       # recupera il frame pointer
0x0000000000400030  ld   ra, 8(sp)       # recupera l'indirizzo di ritorno
0x0000000000400034  addi sp, sp, 24      # elimina il call frame dallo stack
0x0000000000400038  jr   ra              # ritorna al chiamante
…
```



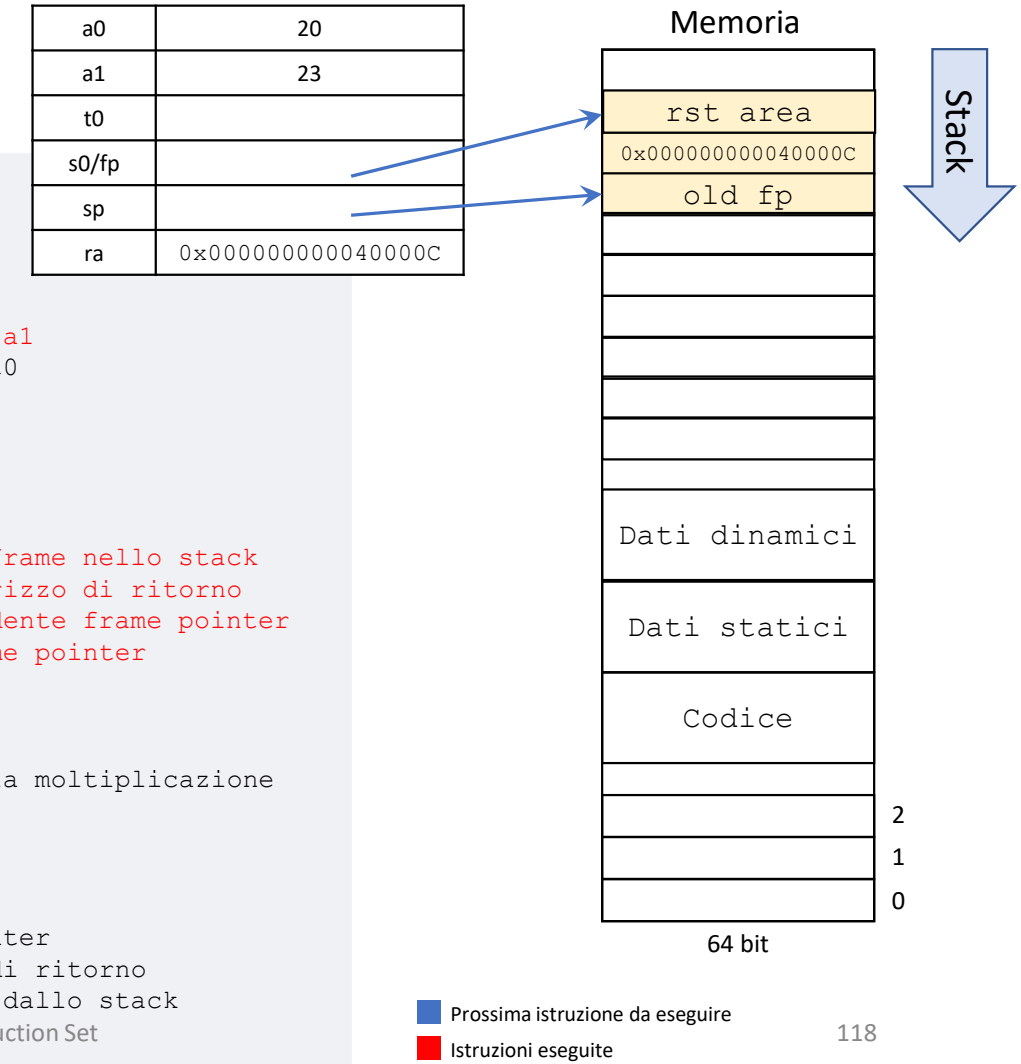
Un esempio

```
_start:
0x0000000000400000  li a0, 20 # salvo la base in a0
0x0000000000400004  li a1, 23 # salvo l'altezza in a1

# chiama triangolo(base,altezza)
0x0000000000400008  jal  ra, area       # altezza in a0, base in a1
0x000000000040000C  add  t0, a0, zero   # salva il risultato in t0
…
area:
# crea il call frame sullo stack (24 byte=ra+fp+rst)
# lo stack cresce verso il basso
0x0000000000400010  addi sp, sp, -24     # allocazione del call frame nello stack
0x0000000000400014  sd   ra, 8(sp)       # salvataggio dell'indirizzo di ritorno
0x0000000000400018  sd   fp, 0(sp)       # salvataggio del precedente frame pointer
0x000000000040001C  addi  fp, sp, 16     # aggiornamento del frame pointer

# calcolo dell'area
0x0000000000400020  jal ra, moltiplicazione
0x0000000000400024  sd a0, 0(fp)         #salva il risultato della moltiplicazione
                         #nella variabile locale rst
0x0000000000400028  srai a0,a0,1

# uscita dalla funzione
0x000000000040002C  ld   fp, 0(sp)       # recupera il frame pointer
0x0000000000400030  ld   ra, 8(sp)       # recupera l'indirizzo di ritorno
0x0000000000400034  addi sp, sp, 24      # elimina il call frame dallo stack
0x0000000000400038  jr   ra              # ritorna al chiamante
…
```



Un esempio

```
_start:
0x0000000000400000  li a0, 20 # salvo la base in a0
0x0000000000400004  li a1, 23 # salvo l'altezza in a1

# chiama triangolo(base,altezza)
0x0000000000400008  jal  ra, area       # altezza in a0, base in a1
0x000000000040000C  add  t0, a0, zero   # salva il risultato in t0
…

area:
# crea il call frame sullo stack (24 byte=ra+fp+rst)
# lo stack cresce verso il basso
0x0000000000400010  addi sp, sp, -24     # allocazione del call frame nello stack
0x0000000000400014  sd   ra, 8(sp)       # salvataggio dell'indirizzo di ritorno
0x0000000000400018  sd   fp, 0(sp)       # salvataggio del precedente frame pointer
0x000000000040001C  addi  fp, sp, 16     # aggiornamento del frame pointer

# calcolo dell'area
0x0000000000400020  jal ra, moltiplicazione
0x0000000000400024  sd a0, 0(fp)         #salva il risultato della moltiplicazione
                         #nella variabile locale rst
0x0000000000400028  srai a0,a0,1

# uscita dalla funzione
0x000000000040002C  ld   fp, 0(sp)       # recupera il frame pointer
0x0000000000400030  ld   ra, 8(sp)       # recupera l'indirizzo di ritorno
0x0000000000400034  addi sp, sp, 24      # elimina il call frame dallo stack
0x0000000000400038  jr   ra              # ritorna al chiamante
…
```



Un esempio

```
_start:
0x0000000000400000  li a0, 20 # salvo la base in a0
0x0000000000400004  li a1, 23 # salvo l'altezza in a1

# chiama triangolo(base,altezza)
0x0000000000400008  jal  ra, area       # altezza in a0, base in a1
0x000000000040000C  add  t0, a0, zero   # salva il risultato in t0
…

area:
# crea il call frame sullo stack (24 byte=ra+fp+rst)
# lo stack cresce verso il basso
0x0000000000400010  addi sp, sp, -24     # allocazione del call frame nello stack
0x0000000000400014  sd   ra, 8(sp)       # salvataggio dell'indirizzo di ritorno
0x0000000000400018  sd   fp, 0(sp)       # salvataggio del precedente frame pointer
0x000000000040001C  addi  fp, sp, 16     # aggiornamento del frame pointer

# calcolo dell'area
0x0000000000400020  jal ra, moltiplicazione
0x0000000000400024  sd a0, 0(fp)         #salva il risultato della moltiplicazione
                         #nella variabile locale rst
0x0000000000400028  srai a0,a0,1

# uscita dalla funzione
0x000000000040002C  ld   fp, 0(sp)       # recupera il frame pointer
0x0000000000400030  ld   ra, 8(sp)       # recupera l'indirizzo di ritorno
0x0000000000400034  addi sp, sp, 24      # elimina il call frame dallo stack
0x0000000000400038  jr   ra              # ritorna al chiamante
…
```

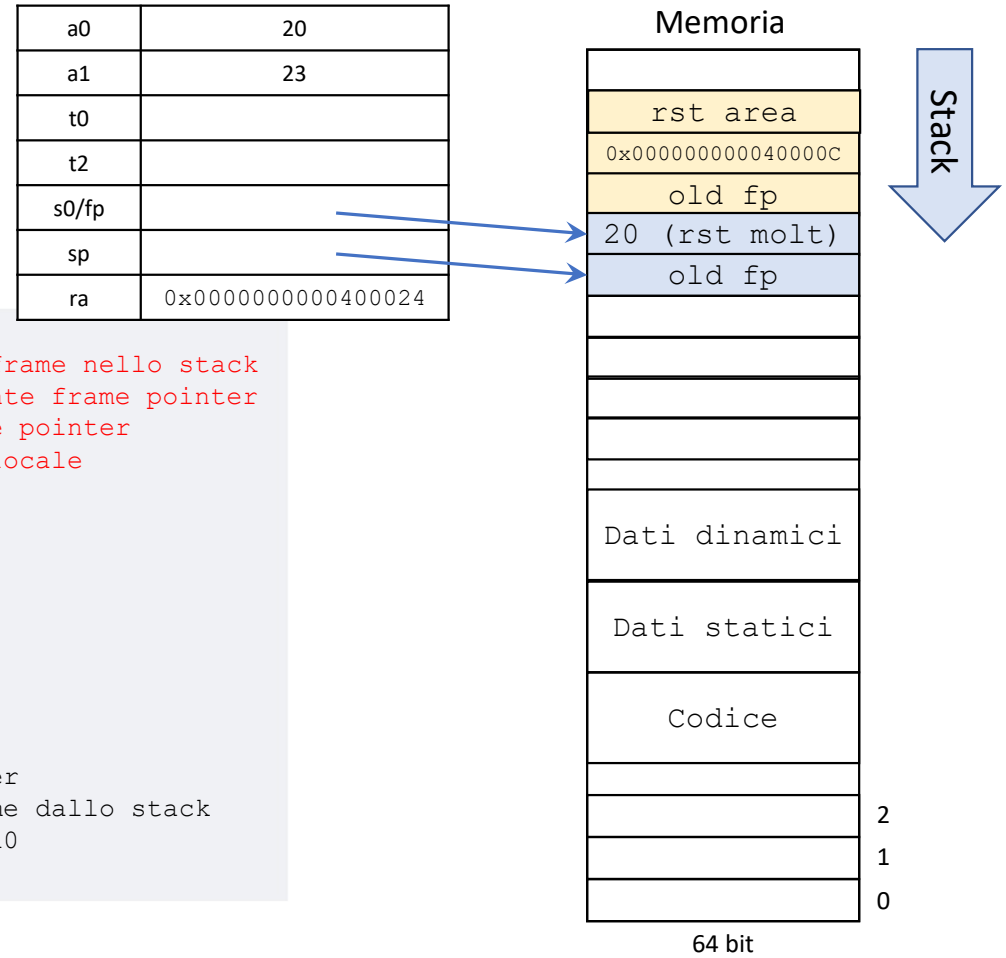
a0	20
a1	23
t0	
s0/fp	
sp	
ra	0x0000000000400024

Prossima istruzione da eseguire

Istruzioni eseguite

Un esempio

```
moltiplicazione:
0x0000000000400038  addi sp, sp, -16     # allocazione del call frame nello stack
0x000000000040003C  sd   fp, 0(sp)      # salvataggio del precedente frame pointer
0x0000000000400040  addi  fp, sp, 8      # aggiornamento del frame pointer
0x0000000000400044  sd a0, 0(fp)         # salvataggio variabile locale

0x0000000000400048  ld t2, 0(fp)
0x000000000040004C  li t0, 1
for:
0x0000000000400050  bge t0,a1,endfor
0x0000000000400054  add t2,a0,t2
0x0000000000400058  addi t0,t0,1
0x000000000040005C  j for
endfor:
0x0000000000400060  ld fp, 0(sp)        # recupera il frame pointer
0x0000000000400064  addi sp, sp, 16        # elimina il call frame dallo stack
0x0000000000400068  add a0,t2,zero        #valore di ritorno in a0
0x000000000040006C  jr   ra               # ritorna al chiamante
```

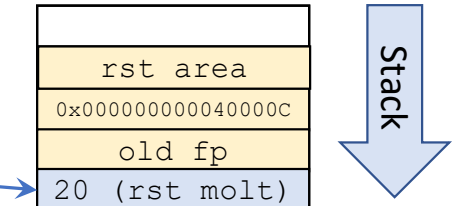


Prossima istruzione da eseguire

Istruzioni eseguite

Un esempio

a0	20
a1	23
t0	23
t2	460
s0/fp	
sp	
ra	0x0000000000400024



Memoria

rst area
0x000000000040000C
old fp
20 (rst molt)
old fp
Dati dinamici
Dati statici
Codice

2
1
0

64 bit

```
moltiplicazione:
0x0000000000400038  addi sp, sp, -16     # allocazione del call frame nello stack
0x000000000040003C  sd   fp, 0(sp)      # salvataggio del precedente frame pointer
0x0000000000400040  addi  fp, sp, 8      # aggiornamento del frame pointer
0x0000000000400044  sd a0, 0(fp)         # salvataggio variabile locale

0x0000000000400048  ld t2, 0(fp)
0x000000000040004C  li t0, 1
for:
0x0000000000400050  bge t0,a1,endfor
0x0000000000400054  add t2,a0,t2
0x0000000000400058  addi t0,t0,1
0x000000000040005C  j for
endfor:
0x0000000000400060  ld fp, 0(sp)        # recupera il frame pointer
0x0000000000400064  addi sp, sp, 16        # elimina il call frame dallo stack
0x0000000000400068  add a0,t2,zero        #valore di ritorno in a0
0x000000000040006C  jr   ra               # ritorna al chiamante
```

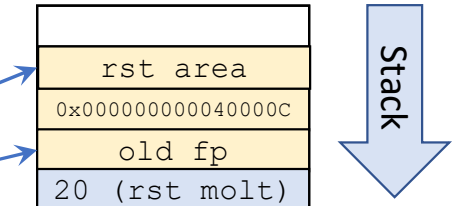
Al termine del ciclo for…

Prossima istruzione da eseguire

Istruzioni eseguite

Un esempio

a0	460
a1	23
t0	23
t2	460
s0/fp	
sp	
ra	0x0000000000400024



```
moltiplicazione:
0x0000000000400038  addi sp, sp, -16     # allocazione del call frame nello stack
0x000000000040003C  sd   fp, 0(sp)      # salvataggio del precedente frame pointer
0x0000000000400040  addi  fp, sp, 8      # aggiornamento del frame pointer
0x0000000000400044  sd a0, 0(fp)         # salvataggio variabile locale

0x0000000000400048  ld t2, 0(fp)
0x000000000040004C  li t0, 1
for:
0x0000000000400050  bge t0,a1,endfor
0x0000000000400054  add t2,a0,t2
0x0000000000400058  addi t0,t0,1
0x000000000040005C  j for
endfor:
0x0000000000400060  ld fp, 0(sp)        # recupera il frame pointer
0x0000000000400064  addi sp, sp, 16        # elimina il call frame dallo stack
0x0000000000400068  add a0,t2,zero        #valore di ritorno in a0
0x000000000040006C  jr   ra               # ritorna al chiamante
```

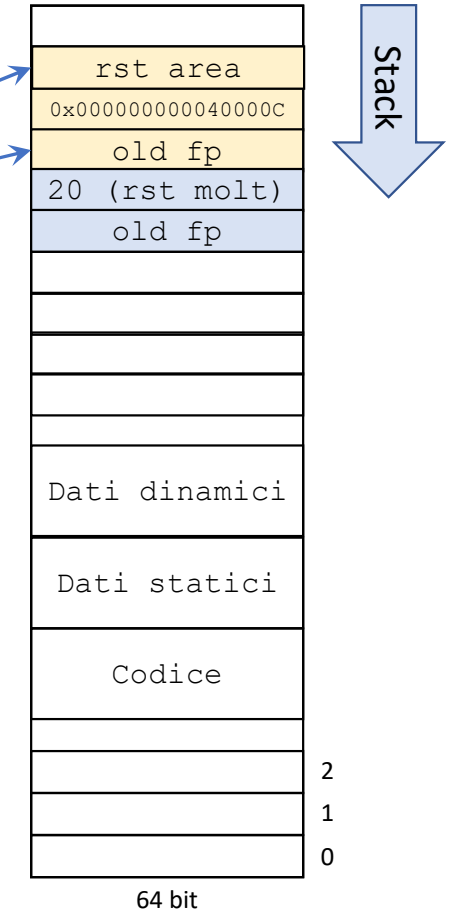
Prossima istruzione da eseguire

Istruzioni eseguite

Un esempio

```
moltiplicazione:
0x0000000000400038  addi sp, sp, -16     # allocazione del call frame nello stack
0x000000000040003C  sd   fp, 0(sp)      # salvataggio del precedente frame pointer
0x0000000000400040  addi  fp, sp, 8      # aggiornamento del frame pointer
0x0000000000400044  sd a0, 0(fp)         # salvataggio variabile locale

0x0000000000400048  ld t2, 0(fp)
0x000000000040004C  li t0, 1
for:
0x0000000000400050  bge t0,a1,endfor
0x0000000000400054  add t2,a0,t2
0x0000000000400058  addi t0,t0,1
0x000000000040005C  j for
endfor:
0x0000000000400060  ld fp, 0(sp)        # recupera il frame pointer
0x0000000000400064  addi sp, sp, 16        # elimina il call frame dallo stack
0x0000000000400068  add a0,t2,zero        #valore di ritorno in a0
0x000000000040006C  jr   ra               # ritorna al chiamante
```

a0	460
a1	23
t0	23
t2	460
s0/fp	
sp	
ra	0x0000000000400024



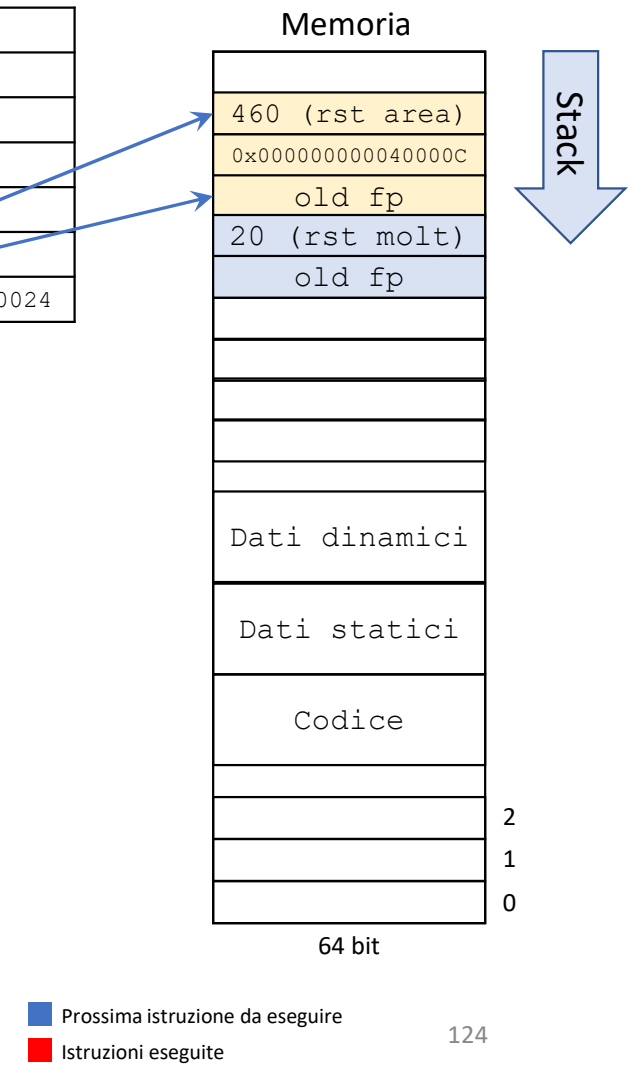
Prossima istruzione da eseguire

Istruzioni eseguite

Un esempio

```
_start:
0x0000000000400000  li a0, 20 # salvo la base in a0
0x0000000000400004  li a1, 23 # salvo l'altezza in a1

# chiama triangolo(base,altezza)
0x0000000000400008  jal  ra, area       # altezza in a0, base in a1
0x000000000040000C  add  t0, a0, zero   # salva il risultato in t0
…

area:
# crea il call frame sullo stack (24 byte=ra+fp+rst)
# lo stack cresce verso il basso
0x0000000000400010  addi sp, sp, -24     # allocazione del call frame nello stack
0x0000000000400014  sd   ra, 8(sp)       # salvataggio dell'indirizzo di ritorno
0x0000000000400018  sd   fp, 0(sp)       # salvataggio del precedente frame pointer
0x000000000040001C  addi  fp, sp, 16     # aggiornamento del frame pointer

# calcolo dell'area
0x0000000000400020  jal ra, moltiplicazione
0x0000000000400024  sd a0, 0(fp)         #salva il risultato della moltiplicazione
                         #nella variabile locale rst
0x0000000000400028  srai a0,a0,1         #divide per due

# uscita dalla funzione
0x000000000040002C  ld   fp, 0(sp)       # recupera il frame pointer
0x0000000000400030  ld   ra, 8(sp)       # recupera l'indirizzo di ritorno
0x0000000000400034  addi sp, sp, 24      # elimina il call frame dallo stack
0x0000000000400038  jr   ra              # ritorna al chiamante
…
```

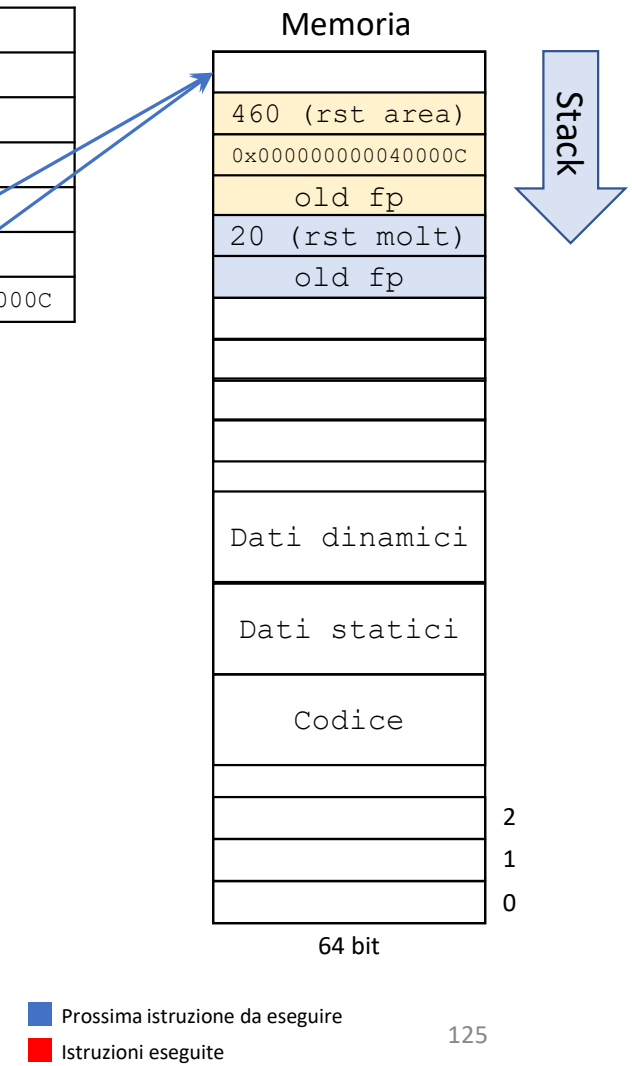
a0	230
a1	23
t0	23
t2	460
s0/fp	
sp	
ra	0x0000000000400024



Un esempio

```
_start:
0x0000000000400000  li a0, 20 # salvo la base in a0
0x0000000000400004  li a1, 23 # salvo l'altezza in a1

# chiama triangolo(base,altezza)
0x0000000000400008  jal  ra, area       # altezza in a0, base in a1
0x000000000040000C  add  t0, a0, zero   # salva il risultato in t0
…

area:
# crea il call frame sullo stack (24 byte=ra+fp+rst)
# lo stack cresce verso il basso
0x0000000000400010  addi sp, sp, -24     # allocazione del call frame nello stack
0x0000000000400014  sd   ra, 8(sp)       # salvataggio dell'indirizzo di ritorno
0x0000000000400018  sd   fp, 0(sp)       # salvataggio del precedente frame pointer
0x000000000040001C  addi  fp, sp, 16     # aggiornamento del frame pointer

# calcolo dell'area
0x0000000000400020  jal ra, moltiplicazione
0x0000000000400024  sd a0, 0(fp)         #salva il risultato della moltiplicazione
                         #nella variabile locale rst
0x0000000000400028  srai a0,a0,1         #divide per due

# uscita dalla funzione
0x000000000040002C  ld   fp, 0(sp)       # recupera il frame pointer
0x0000000000400030  ld   ra, 8(sp)       # recupera l'indirizzo di ritorno
0x0000000000400034  addi sp, sp, 24      # elimina il call frame dallo stack
0x0000000000400038  jr   ra              # ritorna al chiamante
…
```

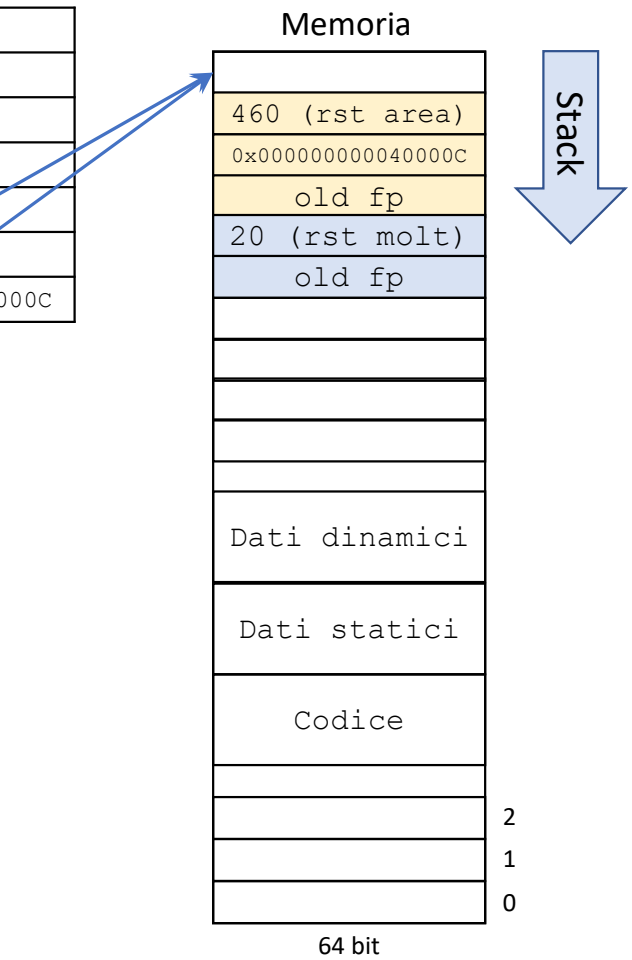
a0	230
a1	23
t0	23
t2	460
s0/fp	
sp	
ra	0x000000000040000C



Un esempio

```
_start:
0x0000000000400000  li a0, 20 # salvo la base in a0
0x0000000000400004  li a1, 23 # salvo l'altezza in a1

# chiama triangolo(base,altezza)
0x0000000000400008  jal  ra, area       # altezza in a0, base in a1
0x000000000040000C  add  t0, a0, zero   # salva il risultato in t0
…

area:
# crea il call frame sullo stack (24 byte=ra+fp+rst)
# lo stack cresce verso il basso
0x0000000000400010  addi sp, sp, -24     # allocazione del call frame nello stack
0x0000000000400014  sd   ra, 8(sp)       # salvataggio dell'indirizzo di ritorno
0x0000000000400018  sd   fp, 0(sp)       # salvataggio del precedente frame pointer
0x000000000040001C  addi  fp, sp, 16     # aggiornamento del frame pointer

# calcolo dell'area
0x0000000000400020  jal ra, moltiplicazione
0x0000000000400024  sd a0, 0(fp)         #salva il risultato della moltiplicazione
                         #nella variabile locale rst
0x0000000000400028  srai a0,a0,1         #divide per due

# uscita dalla funzione
0x000000000040002C  ld   fp, 0(sp)       # recupera il frame pointer
0x0000000000400030  ld   ra, 8(sp)       # recupera l'indirizzo di ritorno
0x0000000000400034  addi sp, sp, 24      # elimina il call frame dallo stack
0x0000000000400038  jr   ra              # ritorna al chiamante
…
```

a0	230
a1	23
t0	23
t2	460
s0/fp	
sp	
ra	0x000000000040000C



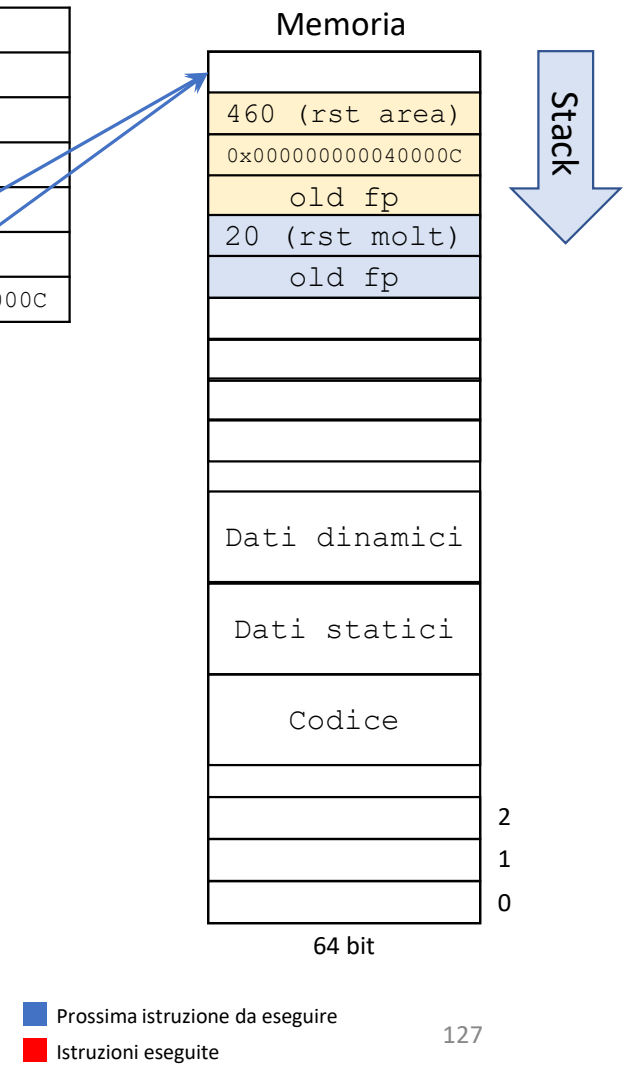
Prossima istruzione da eseguire

Istruzioni eseguite

Un esempio

```
_start:
0x0000000000400000  li a0, 20 # salvo la base in a0
0x0000000000400004  li a1, 23 # salvo l'altezza in a1

# chiama triangolo(base,altezza)
0x0000000000400008  jal  ra, area       # altezza in a0, base in a1
0x000000000040000C  add  t0, a0, zero   # salva il risultato in t0
…

area:
# crea il call frame sullo stack (24 byte=ra+fp+rst)
# lo stack cresce verso il basso
0x0000000000400010  addi sp, sp, -24     # allocazione del call frame nello stack
0x0000000000400014  sd   ra, 8(sp)       # salvataggio dell'indirizzo di ritorno
0x0000000000400018  sd   fp, 0(sp)       # salvataggio del precedente frame pointer
0x000000000040001C  addi  fp, sp, 16     # aggiornamento del frame pointer

# calcolo dell'area
0x0000000000400020  jal ra, moltiplicazione
0x0000000000400024  sd a0, 0(fp)         #salva il risultato della moltiplicazione
                         #nella variabile locale rst
0x0000000000400028  srai a0,a0,1         #divide per due

# uscita dalla funzione
0x000000000040002C  ld   fp, 0(sp)       # recupera il frame pointer
0x0000000000400030  ld   ra, 8(sp)       # recupera l'indirizzo di ritorno
0x0000000000400034  addi sp, sp, 24      # elimina il call frame dallo stack
0x0000000000400038  jr   ra              # ritorna al chiamante
…
```

a0	230
a1	23
t0	230
t2	460
s0/fp	
sp	
ra	0x000000000040000C



Esempio: calcolo del fattoriale

```
int fact(int n) {
  if (n==0)
    return 1;
  else return n*fact(n-1)

}
```

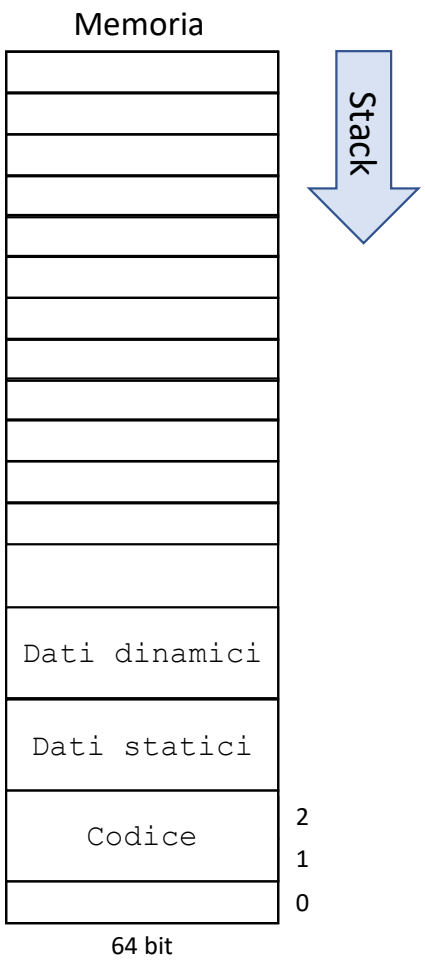
```
fact:
# crea il call frame sullo stack (24 byte)
# lo stack cresce verso il basso
    addi sp, sp, -24     # allocazione del call frame nello stack
    sd   a0, 16(sp)       # salvataggio di n nel call frame
    sd   ra, 8(sp)       # salvataggio dell'indirizzo di ritorno
    sd   fp, 0(sp)       # salvataggio del precedente frame pointer
    addi  fp, sp, 16    # aggiornamento del frame pointer

# calcolo del fattoriale
    bne  a0, zero,  Ric  # test fine ricorsione n!=0
    addi a0, zero, 1     # 0! = 1
    j    Fine

Ric:                     # chiamata ricorsiva per il calcolo di (n-1)!
    addi a0, a0, -1      # a0 <- (n - 1)  passaggio del parametro in a0 per fact(n-1)
    jal  fact            # chiama fact(n-1) -> risultato in a0
    ld   t0, 0(fp)       # t0 <- n
    mul  a0, a0, t0      # n! = (n-1)! x n

# uscita dalla funzione
Fine:
    ld   fp, 0(sp)       # recupera il frame pointer
    ld   ra, 8(sp)       # recupera l'indirizzo di ritorno
    addi sp, sp, 24      # elimina il call frame dallo stack
    jr   ra              # ritorna al chiamante
```

Esempio: calcolo del fattoriale

```
_start:
0x400000  li a0, 3 # salvo n in a0

0x400004  jal  ra, fact          # n in a0
0x400008  add  t0, a0, zero      # salva il risultato in t0

fact:
0x40000c  addi sp, sp, -24       # allocazione del call frame nello stack
0x400010  sd   a0, 16(sp)        # salvataggio di n nel call frame
0x400014  sd   ra, 8(sp)         # salvataggio dell'indirizzo di ritorno
0x400018  sd   fp, 0(sp)         # salvataggio del precedente frame pointer
0x40001c  addi  fp, sp, 16       # aggiornamento del frame pointer

0x400020  bne  a0, zero,  Ric # test fine ricorsione n!=0
0x400024  addi a0, zero, 1       # 0! = 1
0x400028  j    Fine

ric:                             # chiamata ricorsiva per il calcolo di (n-1)!
0x40002c  addi a0, a0, -1        # a0 <- (n - 1) parametro per fact(n-1)
0x400030  jal  fact              # chiama fact(n-1) -> risultato in a0
0x400034  ld   t0, 0(fp)         # t0 <- n
0x400038  mul  a0, a0, t0        # n! = (n-1)! x n

fine:
0x40003c  ld   fp, 0(sp)         # recupera il frame pointer
0x400040  ld   ra, 8(sp)         # recupera l'indirizzo di ritorno
0x400044  addi sp, sp, 24        # elimina il call frame dallo stack
0x400048  jr   ra                # ritorna al chiamante
```

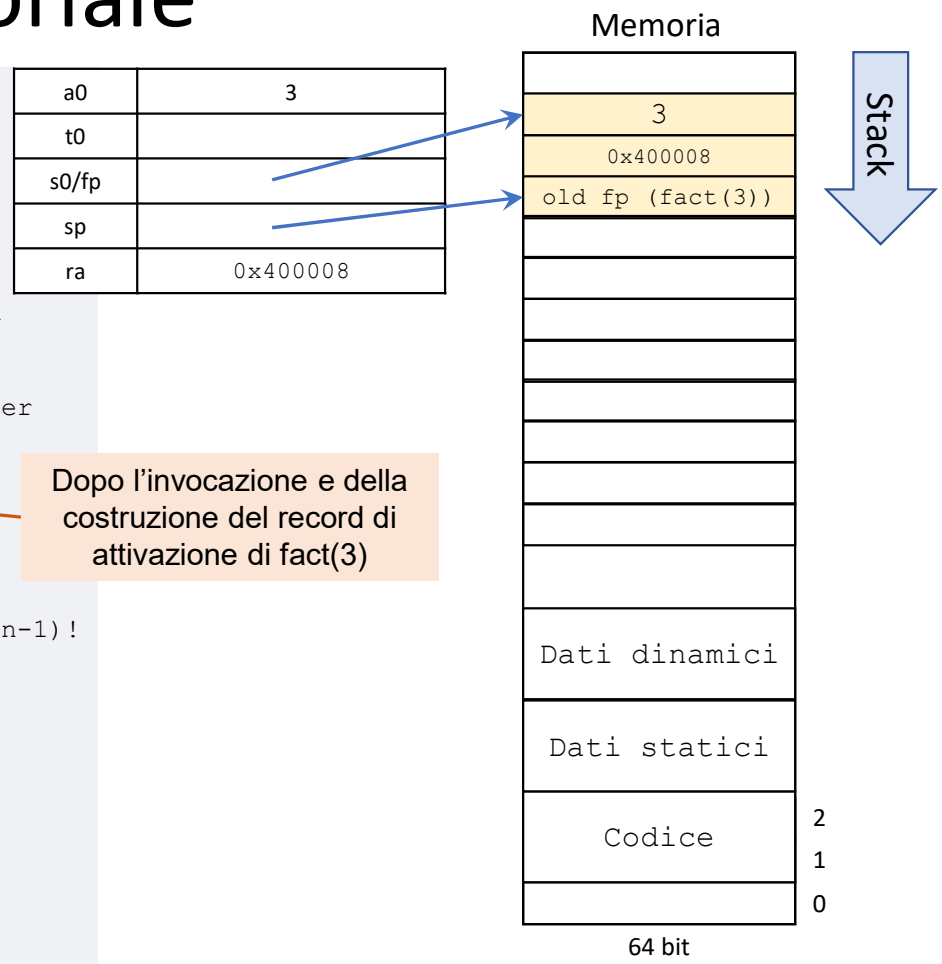
Prima dell'invocazione di fact(3)

a0	3
t0	
s0/fp	
sp	
ra	



Esempio: calcolo del fattoriale

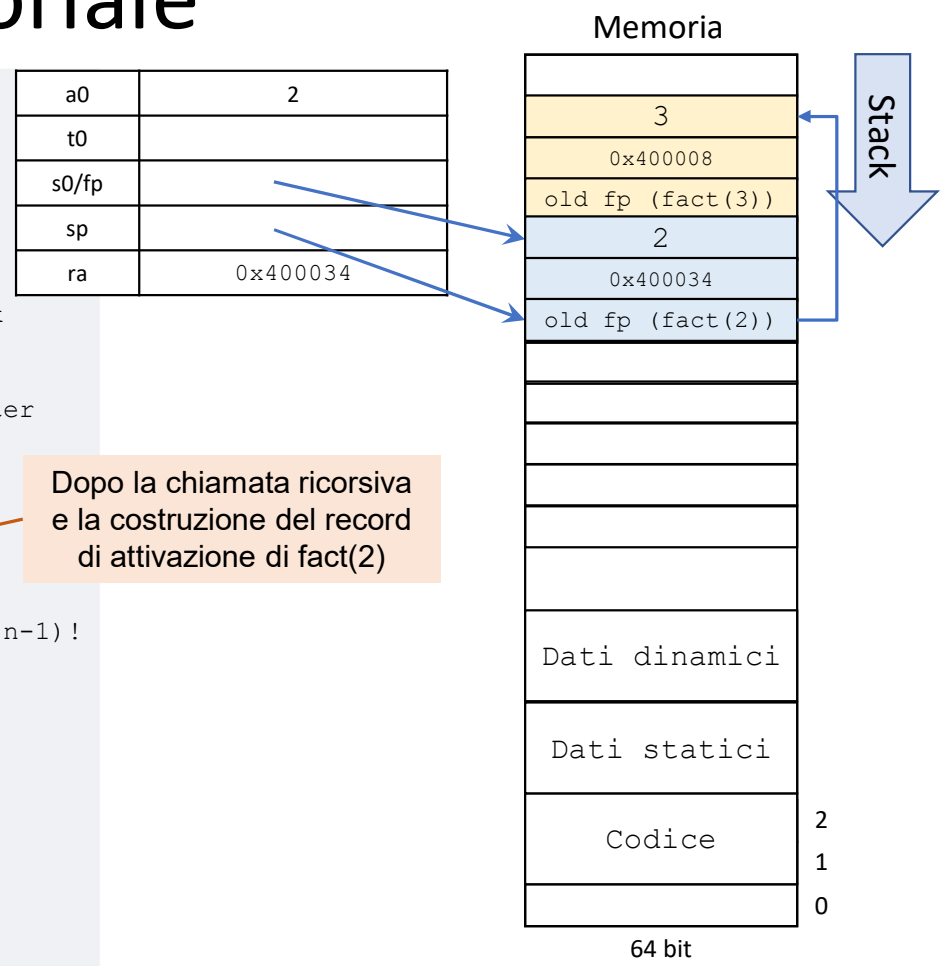
```
_start:
0x400000  li a0, 3 # salvo n in a0

0x400004  jal  ra, fact       # n in a0
0x400008  add  t0, a0, zero   # salva il risultato in t0

fact:
0x40000c  addi sp, sp, -24    # allocazione del call frame nello stack
0x400010  sd   a0, 16(sp)     # salvataggio di n nel call frame
0x400014  sd   ra, 8(sp)      # salvataggio dell'indirizzo di ritorno
0x400018  sd   fp, 0(sp)      # salvataggio del precedente frame pointer
0x40001c  addi  fp, sp, 16    # aggiornamento del frame pointer

0x400020  bne  a0, zero,  Ric # test fine ricorsione n!=0
0x400024  addi a0, zero, 1    # 0! = 1
0x400028  j    Fine

ric:                          # chiamata ricorsiva per il calcolo di (n-1)!
0x40002c  addi a0, a0, -1     # a0 <- (n - 1) parametro per fact(n-1)
0x400030  jal  fact           # chiama fact(n-1) -> risultato in a0
0x400034  ld   t0, 0(fp)      # t0 <- n
0x400038  mul  a0, a0, t0     # n! = (n-1)! x n

fine:
0x40003c  ld   fp, 0(sp)      # recupera il frame pointer
0x400040  ld   ra, 8(sp)      # recupera l'indirizzo di ritorno
0x400044  addi sp, sp, 24     # elimina il call frame dallo stack
0x400048  jr   ra             # ritorna al chiamante
```



Dopo l'invocazione e della costruzione del record di attivazione di fact(3)

Esempio: calcolo del fattoriale

```
_start:
0x400000  li a0, 3 # salvo n in a0

0x400004  jal  ra, fact       # n in a0
0x400008  add  t0, a0, zero   # salva il risultato in t0

fact:
0x40000c  addi sp, sp, -24    # allocazione del call frame nello stack
0x400010  sd   a0, 16(sp)     # salvataggio di n nel call frame
0x400014  sd   ra, 8(sp)      # salvataggio dell'indirizzo di ritorno
0x400018  sd   fp, 0(sp)      # salvataggio del precedente frame pointer
0x40001c  addi  fp, sp, 16    # aggiornamento del frame pointer

0x400020  bne  a0, zero,  Ric # test fine ricorsione n!=0
0x400024  addi a0, zero, 1    # 0! = 1
0x400028  j    Fine

ric:                          # chiamata ricorsiva per il calcolo di (n-1)!
0x40002c  addi a0, a0, -1     # a0 <- (n - 1) parametro per fact(n-1)
0x400030  jal  fact           # chiama fact(n-1) -> risultato in a0
0x400034  ld   t0, 0(fp)      # t0 <- n
0x400038  mul  a0, a0, t0     # n! = (n-1)! x n

fine:
0x40003c  ld   fp, 0(sp)      # recupera il frame pointer
0x400040  ld   ra, 8(sp)      # recupera l'indirizzo di ritorno
0x400044  addi sp, sp, 24     # elimina il call frame dallo stack
0x400048  jr   ra             # ritorna al chiamante
```



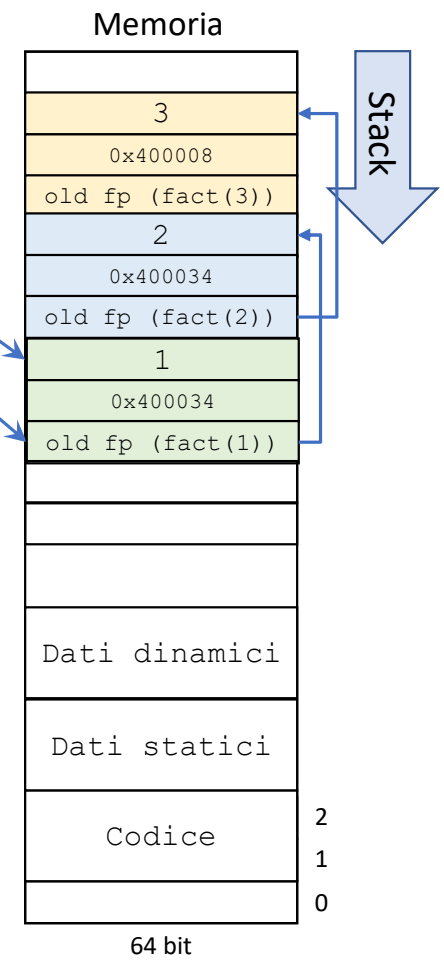
Dopo la chiamata ricorsiva e la costruzione del record di attivazione di fact(2)

Esempio: calcolo del fattoriale

```
_start:
0x400000  li a0, 3 # salvo n in a0

0x400004  jal  ra, fact       # n in a0
0x400008  add  t0, a0, zero   # salva il risultato in t0

fact:
0x40000c  addi sp, sp, -24    # allocazione del call frame nello stack
0x400010  sd   a0, 16(sp)     # salvataggio di n nel call frame
0x400014  sd   ra, 8(sp)      # salvataggio dell'indirizzo di ritorno
0x400018  sd   fp, 0(sp)      # salvataggio del precedente frame pointer
0x40001c  addi  fp, sp, 16    # aggiornamento del frame pointer

0x400020  bne  a0, zero,  Ric # test fine ricorsione n!=0
0x400024  addi a0, zero, 1    # 0! = 1
0x400028  j    Fine

ric:                          # chiamata ricorsiva per il calcolo di (n-1)!
0x40002c  addi a0, a0, -1     # a0 <- (n - 1) parametro per fact(n-1)
0x400030  jal  fact           # chiama fact(n-1) -> risultato in a0
0x400034  ld   t0, 0(fp)      # t0 <- n
0x400038  mul  a0, a0, t0     # n! = (n-1)! x n

fine:
0x40003c  ld   fp, 0(sp)      # recupera il frame pointer
0x400040  ld   ra, 8(sp)      # recupera l'indirizzo di ritorno
0x400044  addi sp, sp, 24     # elimina il call frame dallo stack
0x400048  jr   ra             # ritorna al chiamante
```

a0	1
t0	
s0/fp	
sp	
ra	0x400034

Dopo la chiamata ricorsiva e la costruzione del record di attivazione di fact(1)

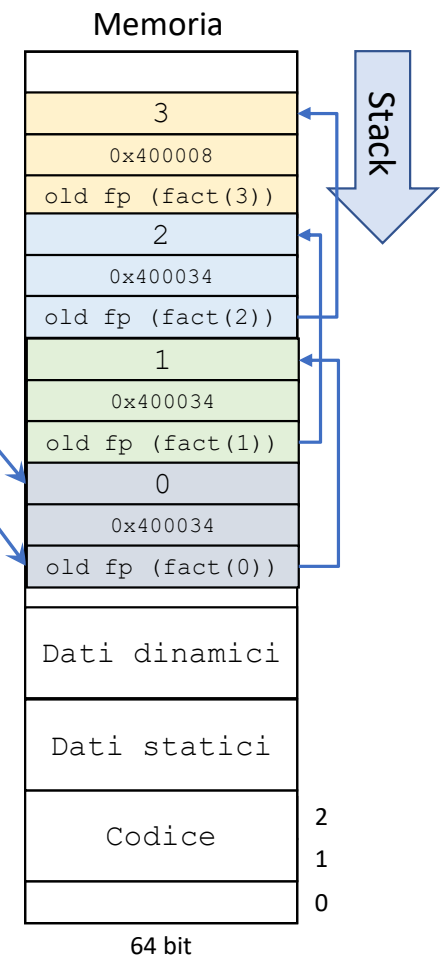


Esempio: calcolo del fattoriale

```
_start:
0x400000  li a0, 3 # salvo n in a0

0x400004  jal  ra, fact       # n in a0
0x400008  add  t0, a0, zero   # salva il risultato in t0

fact:
0x40000c  addi sp, sp, -24    # allocazione del call frame nello stack
0x400010  sd   a0, 16(sp)     # salvataggio di n nel call frame
0x400014  sd   ra, 8(sp)      # salvataggio dell'indirizzo di ritorno
0x400018  sd   fp, 0(sp)      # salvataggio del precedente frame pointer
0x40001c  addi  fp, sp, 16    # aggiornamento del frame pointer

0x400020  bne  a0, zero,  Ric # test fine ricorsione n!=0
0x400024  addi a0, zero, 1    # 0! = 1
0x400028  j    Fine

ric:                          # chiamata ricorsiva per il calcolo di (n-1)!
0x40002c  addi a0, a0, -1     # a0 <- (n - 1) parametro per fact(n-1)
0x400030  jal  fact           # chiama fact(n-1) -> risultato in a0
0x400034  ld   t0, 0(fp)      # t0 <- n
0x400038  mul  a0, a0, t0     # n! = (n-1)! x n

fine:
0x40003c  ld   fp, 0(sp)      # recupera il frame pointer
0x400040  ld   ra, 8(sp)      # recupera l'indirizzo di ritorno
0x400044  addi sp, sp, 24     # elimina il call frame dallo stack
0x400048  jr   ra             # ritorna al chiamante
```

a0	0
t0	
s0/fp	
sp	
ra	0x400034

Dopo la chiamata ricorsiva e la costruzione del record di attivazione di fact(0)

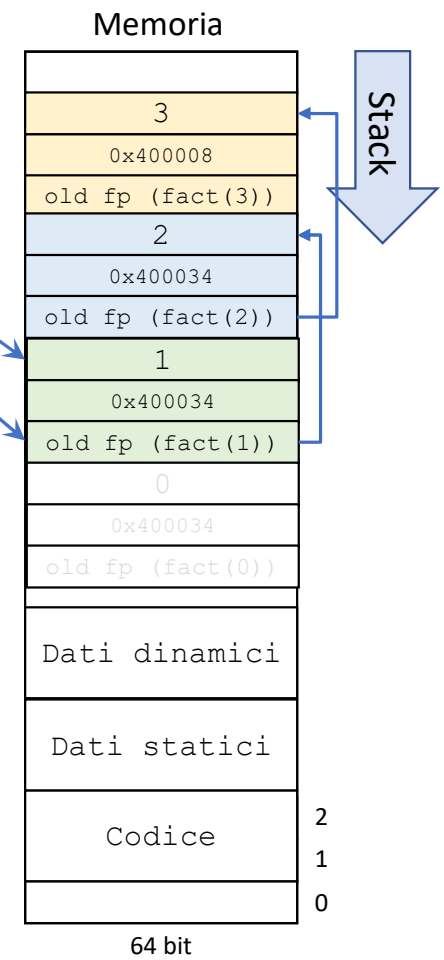


Esempio: calcolo del fattoriale

```
_start:
0x400000  li a0, 3 # salvo n in a0

0x400004  jal  ra, fact        # n in a0
0x400008  add  t0, a0, zero    # salva il risultato in t0

fact:
0x40000c  addi sp, sp, -24     # allocazione del call frame nello stack
0x400010  sd   a0, 16(sp)      # salvataggio di n nel call frame
0x400014  sd   ra, 8(sp)       # salvataggio dell'indirizzo di ritorno
0x400018  sd   fp, 0(sp)       # salvataggio del precedente frame pointer
0x40001c  addi  fp, sp, 16     # aggiornamento del frame pointer

0x400020  bne  a0, zero,  Ric # test fine ricorsione n!=0
0x400024  addi a0, zero, 1     # 0! = 1
0x400028  j    Fine

ric:                           # chiamata ricorsiva per il calcolo di (n-1)!
0x40002c  addi a0, a0, -1      # a0 <- (n - 1) parametro per fact(n-1)
0x400030  jal  fact            # chiama fact(n-1) -> risultato in a0
0x400034  ld   t0, 0(fp)       # t0 <- n
0x400038  mul  a0, a0, t0      # n! = (n-1)! x n

fine:
0x40003c  ld   fp, 0(sp)       # recupera il frame pointer
0x400040  ld   ra, 8(sp)       # recupera l'indirizzo di ritorno
0x400044  addi sp, sp, 24      # elimina il call frame dallo stack
0x400048  jr   ra              # ritorna al chiamante
```

a0	1
t0	
s0/fp	
sp	
ra	0x400034

n=0 e quindi la funzione ritorna 1, deallocando il record di attivazione

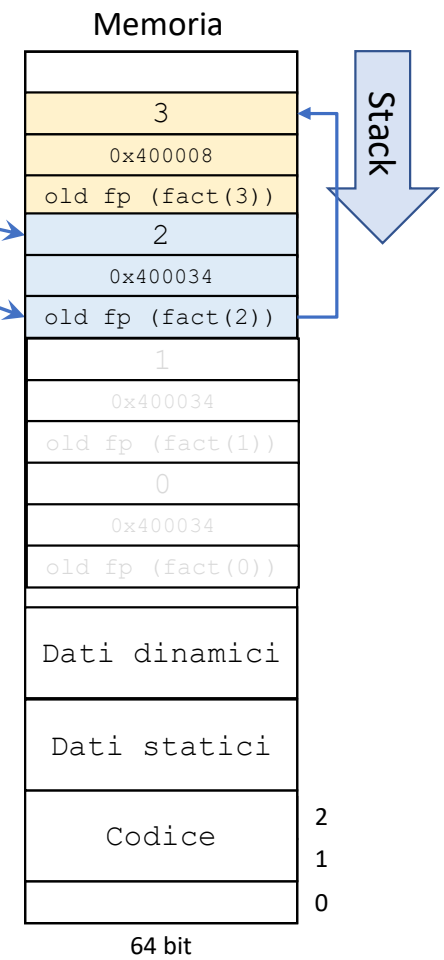


Esempio: calcolo del fattoriale

```
_start:
0x400000  li a0, 3 # salvo n in a0

0x400004  jal  ra, fact        # n in a0
0x400008  add  t0, a0, zero    # salva il risultato in t0

fact:
0x40000c  addi sp, sp, -24     # allocazione del call frame nello stack
0x400010  sd   a0, 16(sp)      # salvataggio di n nel call frame
0x400014  sd   ra, 8(sp)       # salvataggio dell'indirizzo di ritorno
0x400018  sd   fp, 0(sp)       # salvataggio del precedente frame pointer
0x40001c  addi  fp, sp, 16     # aggiornamento del frame pointer

0x400020  bne  a0, zero,  Ric # test fine ricorsione n!=0
0x400024  addi a0, zero, 1     # 0! = 1
0x400028  j    Fine

ric:                           # chiamata ricorsiva per il calcolo di (n-
0x40002c  addi a0, a0, -1      # a0 <- (n - 1) parametro per fact(n-1)
0x400030  jal  fact            # chiama fact(n-1) -> risultato in a0
0x400034  ld   t0, 0(fp)       # t0 <- n
0x400038  mul  a0, a0, t0      # n! = (n-1)! x n

fine:
0x40003c  ld   fp, 0(sp)       # recupera il frame pointer
0x400040  ld   ra, 8(sp)       # recupera l'indirizzo di ritorno
0x400044  addi sp, sp, 24      # elimina il call frame dallo stack
0x400048  jr   ra              # ritorna al chiamante
```

a0	1
t0	
s0/fp	
sp	
ra	0x400034

Viene calcolata la moltiplicazione 1x1 e viene ritornato il risultato, deallocando il record di attivazione



Esempio: calcolo del fattoriale

```
_start:
0x400000  li a0, 3 # salvo n in a0

0x400004  jal  ra, fact        # n in a0
0x400008  add  t0, a0, zero    # salva il risultato in t0

fact:
0x40000c  addi sp, sp, -24     # allocazione del call frame nello stack
0x400010  sd   a0, 16(sp)      # salvataggio di n nel call frame
0x400014  sd   ra, 8(sp)       # salvataggio dell'indirizzo di ritorno
0x400018  sd   fp, 0(sp)       # salvataggio del precedente frame pointer
0x40001c  addi  fp, sp, 16     # aggiornamento del frame pointer

0x400020  bne  a0, zero,  Ric # test fine ricorsione n!=0
0x400024  addi a0, zero, 1     # 0! = 1
0x400028  j    Fine

ric:                           # chiamata ricorsiva per il calcolo di (n-
0x40002c  addi a0, a0, -1      # a0 <- (n - 1) parametro per fact(n-1)
0x400030  jal  fact            # chiama fact(n-1) -> risultato in a0
0x400034  ld   t0, 0(fp)       # t0 <- n
0x400038  mul  a0, a0, t0      # n! = (n-1)! x n

fine:
0x40003c  ld   fp, 0(sp)       # recupera il frame pointer
0x400040  ld   ra, 8(sp)       # recupera l'indirizzo di ritorno
0x400044  addi sp, sp, 24      # elimina il call frame dallo stack
0x400048  jr   ra              # ritorna al chiamante
```

a0	2
t0	
s0/fp	
sp	
ra	0x400008

Viene calcolata la moltiplicazione 1x2 e viene ritornato il risultato, deallocando il record di attivazione

Esempio: calcolo del fattoriale

```
_start:
0x400000  li a0, 3 # salvo n in a0

0x400004  jal  ra, fact       # n in a0
0x400008  add  t0, a0, zero   # salva il risultato in t0

fact:
0x40000c  addi sp, sp, -24    # allocazione del call frame nello stack
0x400010  sd   a0, 16(sp)     # salvataggio di n nel call frame
0x400014  sd   ra, 8(sp)      # salvataggio dell'indirizzo di ritorno
0x400018  sd   fp, 0(sp)      # salvataggio del precedente frame pointer
0x40001c  addi  fp, sp, 16    # aggiornamento del frame pointer

0x400020  bne  a0, zero,  Ric # test fine ricorsione n!=0
0x400024  addi a0, zero, 1    # 0! = 1
0x400028  j    Fine

ric:                          # chiamata ricorsiva per il calcolo di (n-
0x40002c  addi a0, a0, -1     # a0 <- (n - 1) parametro per fact(n-1)
0x400030  jal  fact           # chiama fact(n-1) -> risultato in a0
0x400034  ld   t0, 0(fp)      # t0 <- n
0x400038  mul  a0, a0, t0     # n! = (n-1)! x n

fine:
0x40003c  ld   fp, 0(sp)      # recupera il frame pointer
0x400040  ld   ra, 8(sp)      # recupera l'indirizzo di ritorno
0x400044  addi sp, sp, 24     # elimina il call frame dallo stack
0x400048  jr   ra             # ritorna al chiamante
```

a0	6
t0	
s0/fp	
sp	
ra	

Viene calcolata la moltiplicazione 2x3 e viene ritornato il risultato al main, deallocando il record di attivazione

Operandi immediati ampi

- Problema: è possibile caricare in un registro una costante a 32 bit? Supponiamo di voler caricare nel registro x5 il valore 0x12345678
- Soluzione
  - si introduce una nuova istruzione `lui` (load upper immediate, tipo U) che carica i 20 bit più significativi della costante nei bit da 12 a 31 di un registro e pone quelli a sinistra a zero (i 32 bit più significativi hanno lo stesso valore del bit 31)

```
lui x5,0x12345
```

x5 ..... 00010010 00110100 01010000 00000000

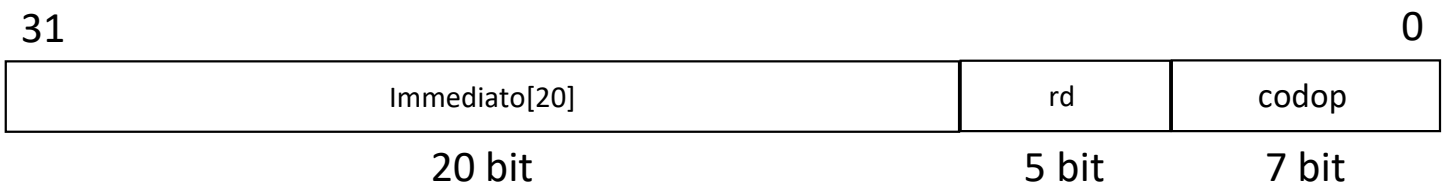
  - Con una operazione di or immediato si impostano i 12 bit meno significativi rimasti

```
ori x5,x5,0x678
```

x5 ..... 00010010 00110100 01010110 01111000

LUI e linguaggio macchina

- Viene introdotto un nuovo tipo: U

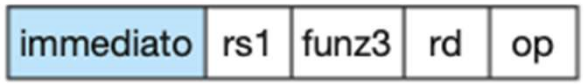


Operandi immediati ampi

- Potreste realizzare il caricamento con le istruzioni `lui` e `addi` (al posto di `ori`)?
  - Supponete di voler caricare nel registro x5 il valore 0x82345678: succede qualcosa?
  - Supponete di voler caricare nel registro x5 il valore 0x12345878: succede qualcosa?
  - Supponete di voler caricare nel registro x5 il valore 0x82345878: succede qualcosa?

Modalità di indirizzamento

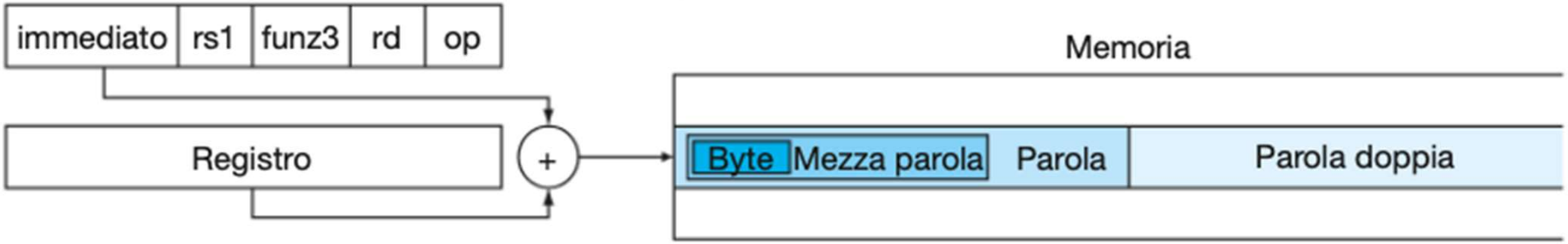
- Indirizzamento immediato



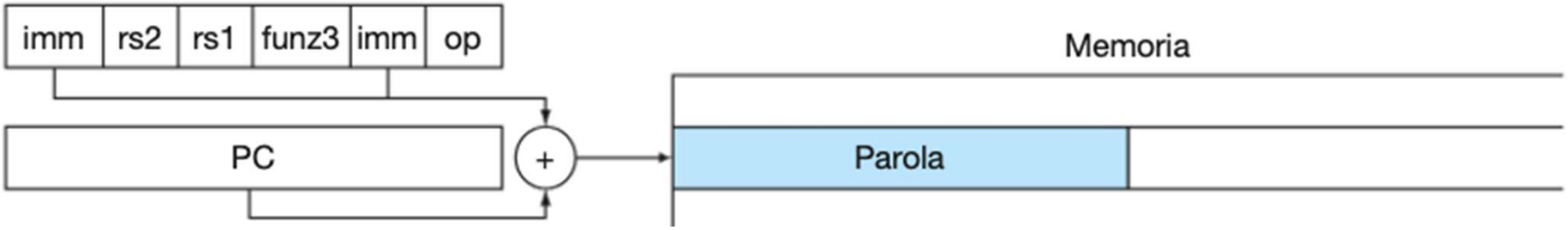
- Indirizzamento tramite registro

Modalità di indirizzamento

- Indirizzamento tramite base e spiazzamento



- Indirizzamento relativo al program counter



Salti con offset più grandi

- Come per le costanti, anche i salti possono essere eseguiti anche ad istruzioni più lontane
- RISC-V introduce la possibilità di salto in un intervallo pari a 2^{32}
- Nuova istruzione

```
auipc rd, offset
```

Add Upper Immediate PC

Tipo U in linguaggio macchina

  - Inserisce nel registro `rd` l'indirizzo di PC + (offset << 12)

Esempio: `auipc x5, 0x12345` → `x5 = PC + 0x12345000`

Salti con offset più grandi

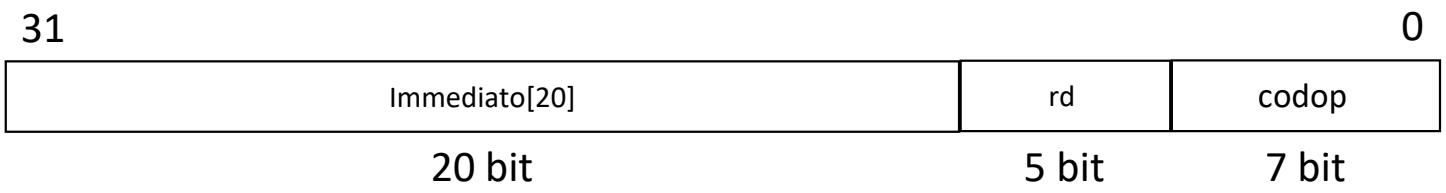
- Se usiamo `auipc` con i 20 bit più significativi dell'offset, allora possiamo aggiungere questa istruzione una istruzione che calcola
  - PC = rd + offset[31..0]
- Otteniamo come risultato un salto incondizionato con offset più esteso
- L'istruzione da considerare è `jalr`! Ricapitolando:

```
auipc rd, offset[31..12]
jalr x0, offset[0..11](rd)
```

- Realizza un salto incondizionato a `PC + offset[31..0]`

AUIPC e linguaggio macchina

- Viene usato il tipo U



Riassunto dei formati delle istruzioni RISC-V

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.