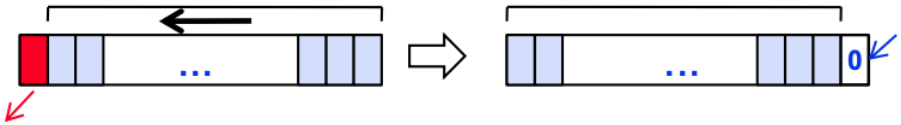
Architettura degli Elaboratori
Corso A
Lab 2

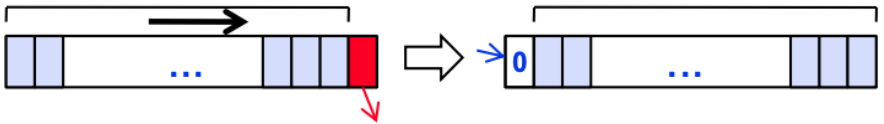
Operazioni logiche e di shift

Operazioni logiche

- Shift logico
  - A sinistra



  - A destra



Shift Left Logical

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

x9 = x22 << x19

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

x9 = x22 << 5

Shift Left Logical Immediate

Shift Right Logical

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

x9 = x22 >> x19

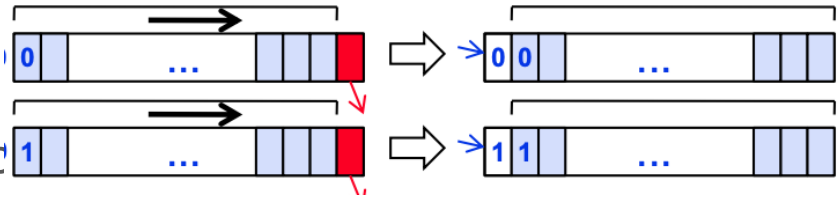
```
srli x9, x22, 5
```

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

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`

$\mathbf{x5 = \overline{x6}}$

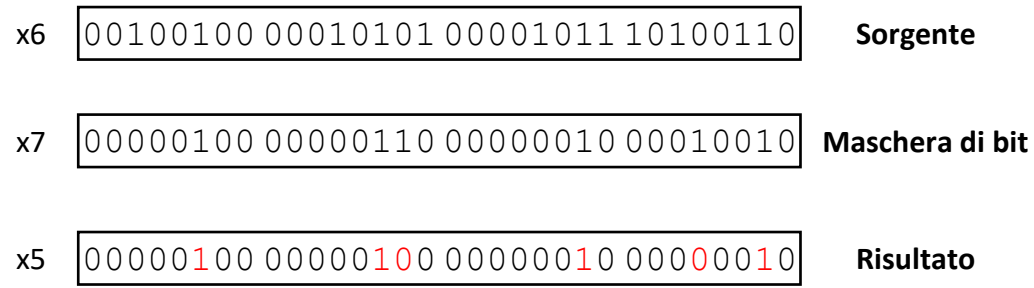
Operazioni logiche

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

Operazioni logiche

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

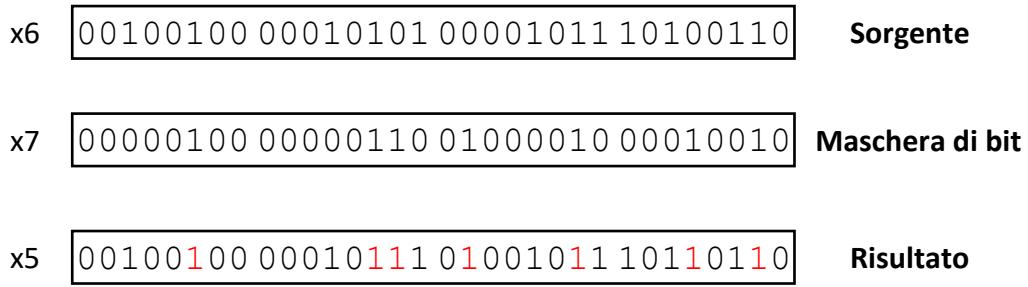
```
and x5,x6,x7
```



Operazioni logiche

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

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



Esercizio 1 - Operazioni Logiche

Si supponga che i registri seguenti contengano i valori:

`x5 = 0x00000000AAAAAAAA, x6 = 0x1234567812345678`

- Determinare il contenuto di **x7** dopo l'esecuzione delle seguenti istruzioni:

```
slli x7, x5, 4
or   x7, x7, x6
```

- Supponendo che i registri x5 e x6 contengano i valori riportati sopra, determinare il contenuto di x7 dopo l'esecuzione di:

```
slli x7, x6, 4
```

- Supponendo che i registri x5 e x6 contengano i valori riportati sopra, determinare il contenuto di x7 dopo l'esecuzione di:

```
srli x7, x5, 3
andi x7, x7, 0xFF
```

Estrarre Bit - Esempio

Determinare una sequenza di istruzioni RISC-V che consenta di estrarre i bit da b2 a b3 dal registro **x5** e li sostituisca ai bit da b6 a b7 del registro x6 senza modificare gli altri bit del registro **x5** e **x6**.

Assicuratevi di verificare il vostro codice utilizzando

`x5 = 0 e x6 = 0xffffffffffffffff`

Questo potrebbe rivelare una svista comune.

Estrarre Bit - Esempio

```
x5 = 0x …    0     0     0
x5 = 0b … 0000 0000 0000
```

bit 2-3

```
x6 = 0x …    f     f     f
x6 = 0b … 1111 1111 1111
```

bit 6-7

Estrarre Bit - Esempio

```
x5 = 0x …    0     0     0
x5 = 0b … 0000 0000 0000
```

bit 2-3

```
x6 = 0x …    f     f     f
x6 = 0b … 1111 1111 1111
```

bit 6-7

Vogliamo ottenere:

```
x6 = 0x …    f     3     f
x6 = 0b … 1111 0011 1111
```

Esercizio 2 - Estrarre Bit

Determinare una sequenza di istruzioni RISC-V che consenta di estrarre i bit da b11 a b16 dal registro **x5** e li sostituisca ai bit da b26 a b31 del registro x6 senza modificare gli altri bit del registro **x5** e **x6**.

Assicuratevi di verificare il vostro codice utilizzando

`x5 = 0 e x6 = 0xffffffffffffffff`

Questo potrebbe rivelare una svista comune.

Esercizio 3 - Media di pari

Si scriva un programma in linguaggio RISC-V che carichi due numeri interi pari su **x5** e **x6** e calcoli il valore della loro media aritmetica. Il valore calcolato va inserito nel registro **x7**.

In questo esercizio, utilizzare soltanto il set delle istruzioni "intere di base rv32i"
(usare un "shift" per fare la divisione per 2)

Operandi Allocati in Memoria (load/store)

Obiettivi

- Imparare a trasferire dati tra la memoria ed i registri in assembler
- Capire come funzionano le istruzioni "load" e "store"
- Capire il modello di memoria di un elaboratore RISC-V

Istruzione di trasferimento dati: un comando che sposta i dati tra la memoria e i registri.

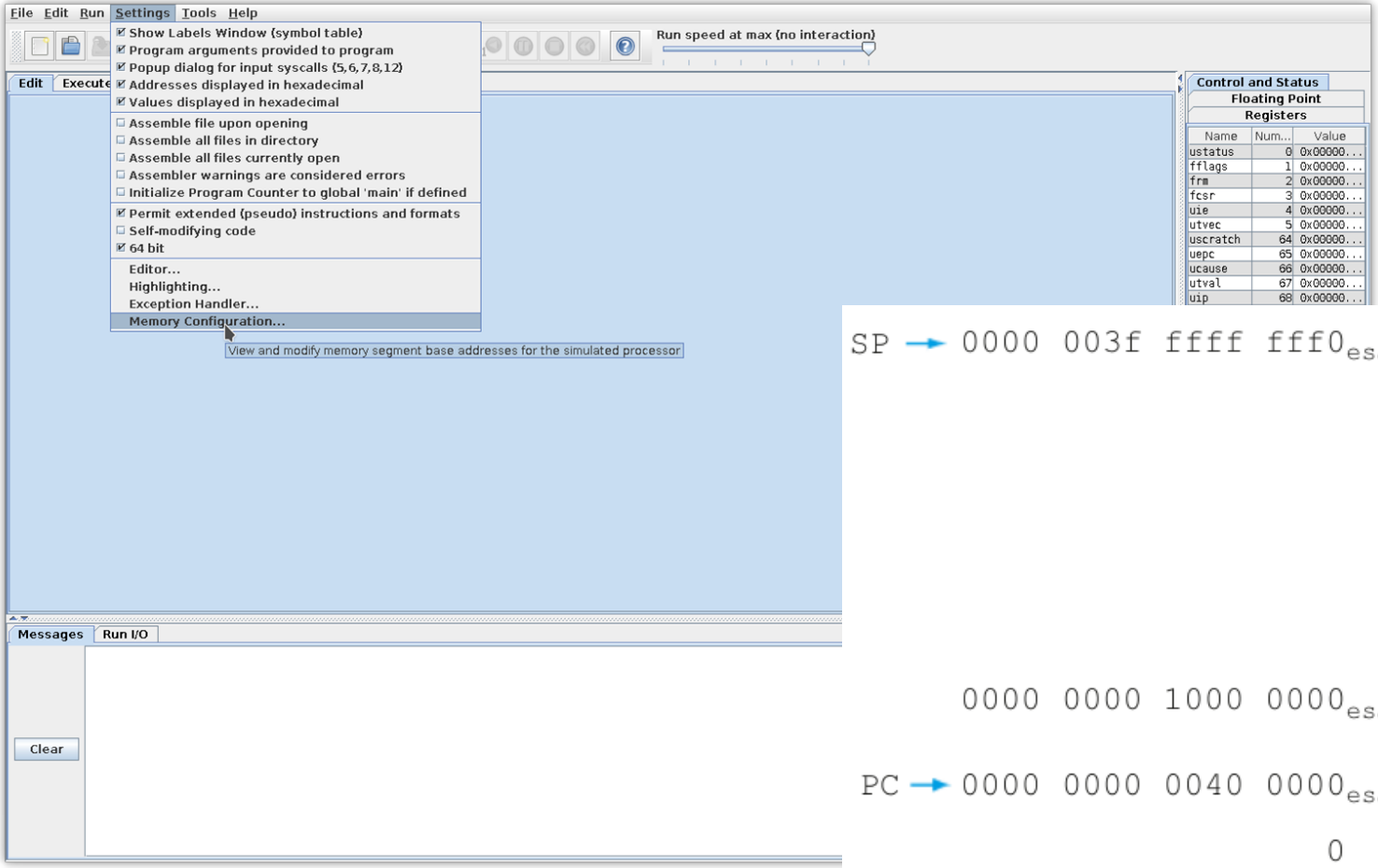
Indirizzo: un numero utilizzato per identificare la posizione di uno specifico elemento all'interno di una memoria, vista come un vettore unidimensionale.

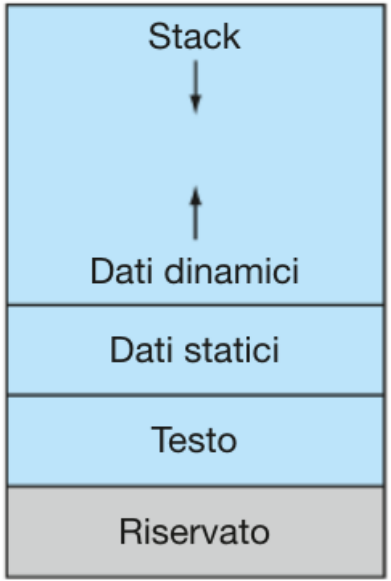
Esempio: Load & Store words

Si scriva un programma in linguaggio RISC-V che carichi due numeri interi presenti nella memoria in word contigue e calcoli la loro somma. Il valore calcolato va salvato in una terza posizione della memoria contigua alle due usate per la somma.

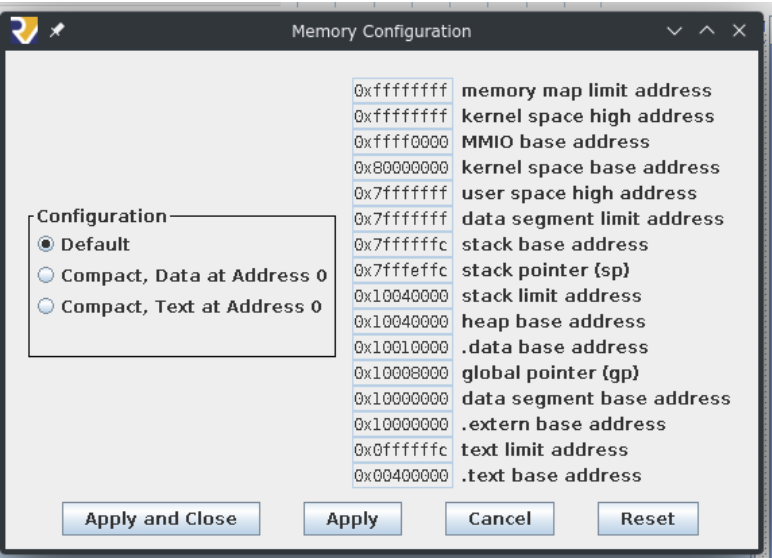
- Per risolvere questo esercizio ci serve capire **dove in memoria** sono i nostri dati **(l'indirizzo)**

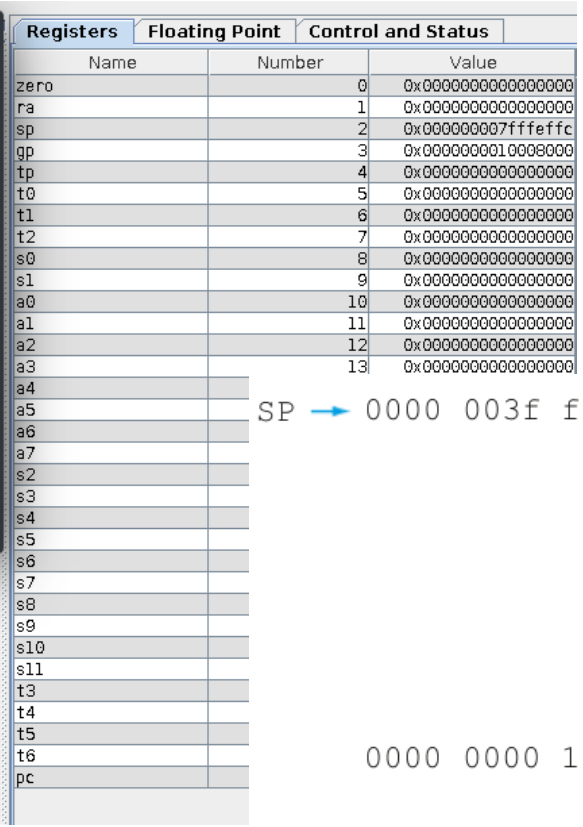
Esempio: Load & Store words



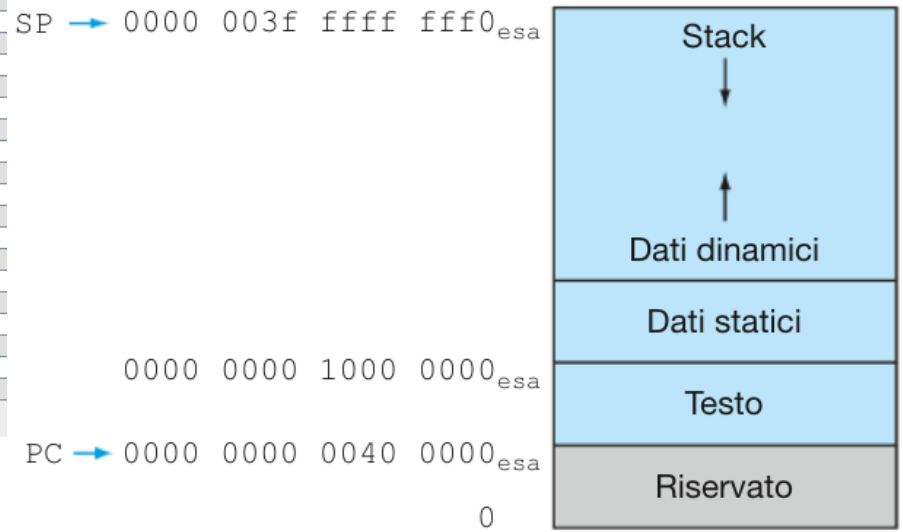


Esempio: Load & Store words



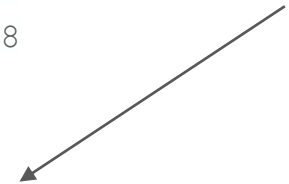


Name	Number	Value
zero	0	0x0000000000000000
ra	1	0x0000000000000000
sp	2	0x000000007fffeffc
gp	3	0x0000000010008000
tp	4	0x0000000000000000
t0	5	0x0000000000000000
t1	6	0x0000000000000000
t2	7	0x0000000000000000
s0	8	0x0000000000000000
s1	9	0x0000000000000000
a0	10	0x0000000000000000
a1	11	0x0000000000000000
a2	12	0x0000000000000000
a3	13	0x0000000000000000
a4		
a5		
a6		
a7		
s2		
s3		
s4		
s5		
s6		
s7		
s8		
s9		
s10		
s11		
t3		
t4		
t5		
t6		
pc		



Esempio: Load & Store words (prima idea)

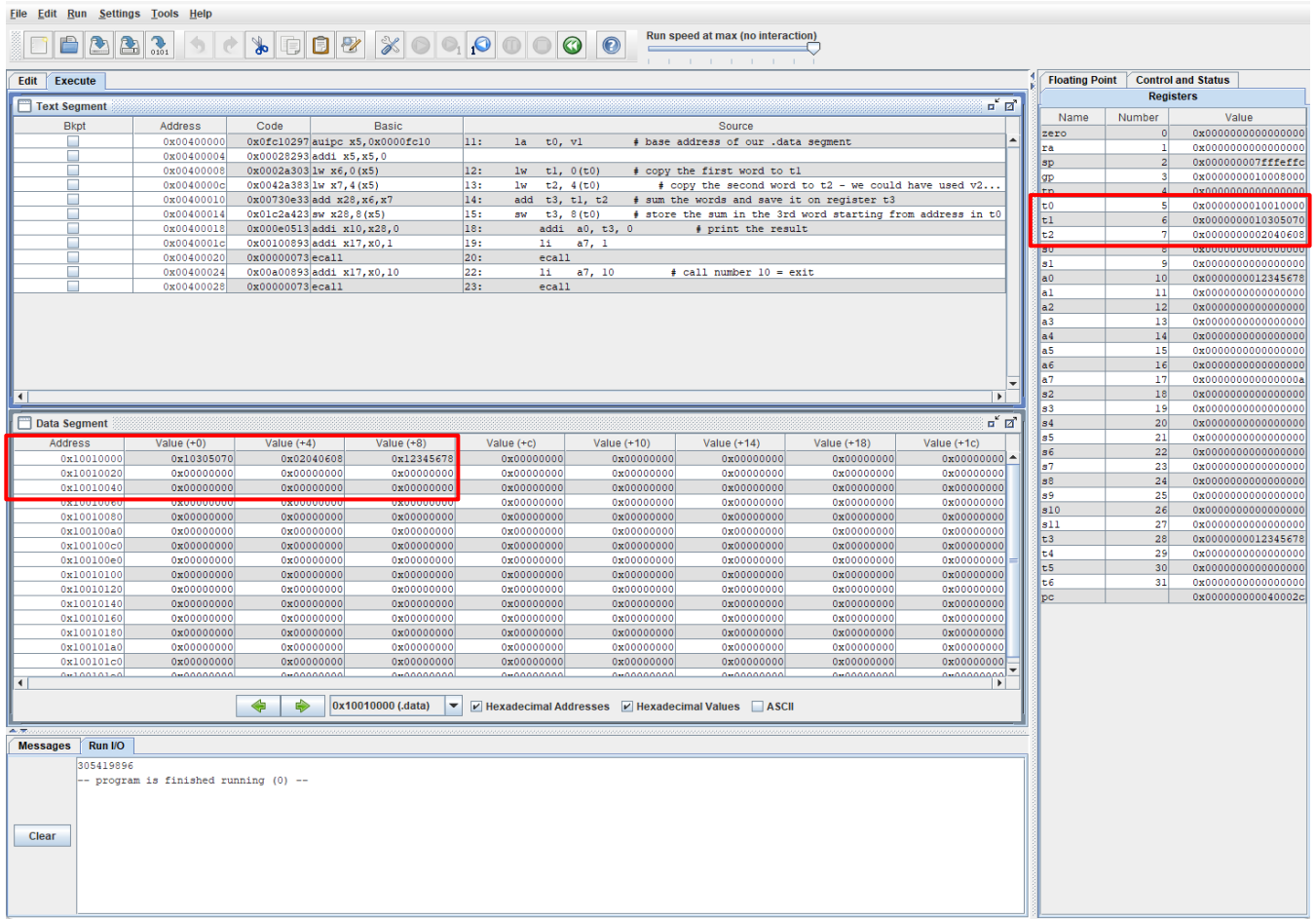
```
.globl _start
.data
    v1: .word 0x10305070
    v2: .word 0x02040608
    v3: .word 0
.text
_start:
    la   t0, 0x10010000     # base address of our .data segment
    lw   t1, 0(t0)          # copy the first word to t1
    lw   t2, 4(t0)          # copy the second word to t2
    add  t3, t1, t2         # sum the words and save it on register t3
    sw   t3, 8(t0)          # store the sum in the 3rd word starting from address in t0
…
```



Indirizzo "hard-coded"?
Perché funziona?
Come evitarlo?

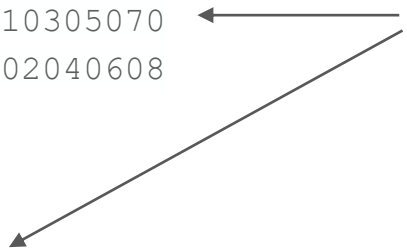
Esempio: Load & Store words (prima idea)

Usare il simulatore per individuare i dati in memoria e nei registri dopo ogni istruzione



Esempio: Load & Store words (usando etichette)

```
.globl _start
.data
    v1: .word 0x10305070
    v2: .word 0x02040608
    v3: .word 0
.text
_start:
    la   t0, v1
    lw   t1, 0(t0)
    lw   t2, 4(t0)
    add  t3, t1, t2
    sw   t3, 8(t0)
...
```



Le etichette sono gli offset per arrivare agli indirizzi!

- **la → load address è una pseudo-istruzione**
- **Verificare nel simulatore RARS come viene tradotta l'istruzione**

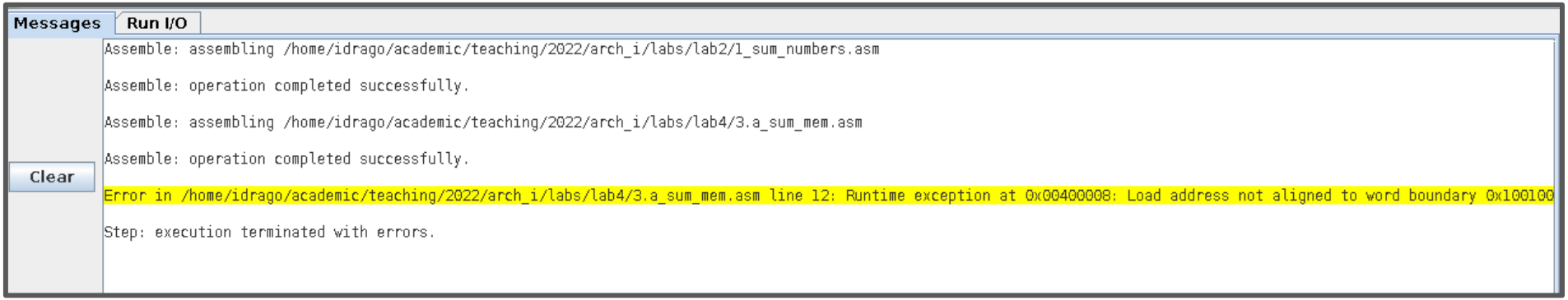
P.S: Parleremo di "auipc" anche in altro lab

Esempio: Load & Store words (usando etichette)

```
.globl _start
.data
    v1: .word 0x10305070
    v2: .word 0x02040608
    v3: .word 0
.text
_start:
    la   t0, v1
    lw   t1, 1(t0)
    lw   t2, 4(t0)
    add  t3, t1, t2
    sw   t3, 8(t0)
…
```

Address + offset → lw?
Provare su RARS: `lw t1, 1(t0)`

Esempio: Load & Store words (usando etichette)




```
.text
_start:
    la   t0, v1
    lw   t1, 1(t0)
    lw   t2, 4(t0)
    add  t3, t1, t2
    sw   t3, 8(t0)
```

Address + offset → lw → word aligned

Vincolo di allineamento: requisito per cui, in molte architetture, le parole devono iniziare sempre a indirizzi multipli di 4 e le parole doppie a indirizzi multipli di 8.

Approfondimento. In molte architetture, le parole devono iniziare sempre a indirizzi multipli di 4 e le parole doppie a indirizzi multipli di 8. Questo requisito si chiama **vincolo di allineamento** ed è comune a molte altre architetture (nel Capitolo 4 viene suggerito come l'allineamento produca una velocità maggiore nel trasferimento dei dati). I RISC-V e gli Intel x86 non hanno il vincolo di allineamento, mentre i MIPS sì.

Esercizio 5 - Media interi

Si scriva un programma in linguaggio RISC-V che carichi 4 numeri interi presente nella memoria in word contigue e calcoli il valore intero della loro media aritmetica (arrotondamento per difetto). Il valore calcolato va salvato in un’ulteriore posizione della memoria contigua a quelle usate per il calcolo.

- **In questo esercizio, utilizzare soltanto il set delle istruzioni "intere di base rv64i".**