

# Architettura degli Elaboratori

## Corso A

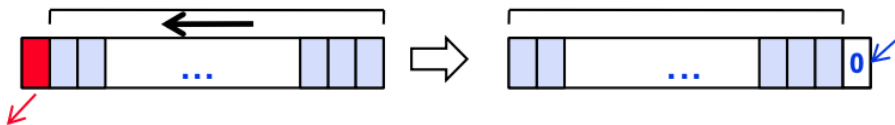
### Lab 2

# Operazioni logiche e di shift

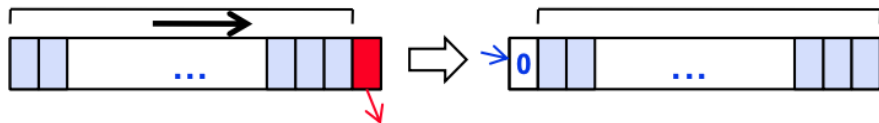
# Operazioni logiche

- Shift logico

- A sinistra



- A destra



## Shift Left Logical

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

$x9 = x22 \ll x19$

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

$x9 = x22 \ll 5$

## Shift Left Logical Immediate

## Shift Right Logical

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

$x9 = x22 \gg x19$

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

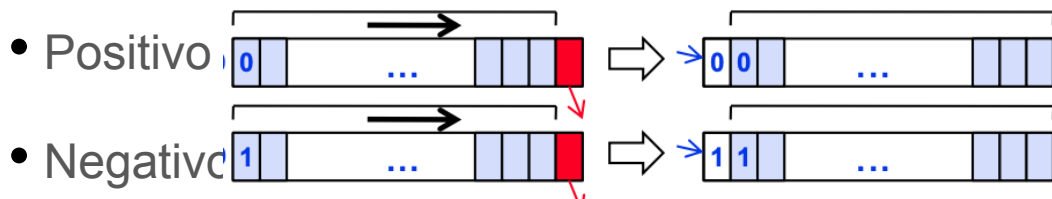
$x9 = x22 \gg 5$

## Shift Right Logical Immediate<sub>3</sub>

# Operazioni logiche

- Shift aritmetico

- A destra



- 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 `srl` 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  $\oplus$  x19
```

```
xori x9,x22,5  
x9 = x22  $\oplus$  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 di bit
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 di bit
x5	00100100 00010111 01001011 10110110	Risultato

# 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 **00**00

**bit 2-3**

x6 = 0x ... f f f  
x6 = 0b ... 1111 **11**11 1111

**bit 6-7**

# Estrarre Bit - Esempio

x5 = 0x ... 0 0 0  
x5 = 0b ... 0000 0000 **00**00

**bit 2-3**

x6 = 0x ... f f f  
x6 = 0b ... 1111 **11**11 1111

**bit 6-7**

Vogliamo ottenere:

x6 = 0x ... f 3 f  
x6 = 0b ... 1111 **00**11 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

File Edit Run Settings Tools Help

- ☒ Show Labels Window (symbol table)
- ☒ Program arguments provided to program
- ☒ Popup dialog for input syscalls (5, 6, 7, 8, 12)
- ☒ Addresses displayed in hexadecimal
- ☒ Values displayed in hexadecimal
- ☐ Assemble file upon opening
- ☐ Assemble all files in directory
- ☐ Assemble all files currently open
- ☐ Assembler warnings are considered errors
- ☐ Initialize Program Counter to global 'main' if defined
- ☒ Permit extended (pseudo) instructions and formats
- ☐ Self-modifying code
- ☒ 64 bit
- Editor...
- Highlighting...
- Exception Handler...
- Memory Configuration...

View and modify memory segment base addresses for the simulated processor

Run speed at max (no interaction)

Control and Status

Floating Point

Registers

Name	Num...	Value
ustatus	0	0x00000...
fflags	1	0x00000...
frm	2	0x00000...
fcsr	3	0x00000...
uie	4	0x00000...
utvec	5	0x00000...
uscratch	64	0x00000...
uepc	65	0x00000...
ucause	66	0x00000...
utval	67	0x00000...
uip	68	0x00000...

SP → 0000 003f ffff fff0<sub>esa</sub>

0000 0000 1000 0000<sub>esa</sub>

PC → 0000 0000 0040 0000<sub>esa</sub>

0

Messages Run I/O

Clear



# Esempio: Load & Store words

Memory Configuration

Configuration

- ☒ Default
- ☐ Compact, Data at Address 0
- ☐ Compact, Text at Address 0

0xffffffff memory map limit address

0xffffffff kernel space high address

0xffff0000 MMIO base address

0x80000000 kernel space base address

0x7fffffff user space high address

0x7fffffff data segment limit address

0x7fffffcc stack base address

0x7fffffcc stack pointer (sp)

0x10040000 stack limit address

0x10040000 heap base address

0x10010000 .data base address

0x10008000 global pointer (gp)

0x10000000 data segment base address

0x10000000 .extern base address

0x0ffffffc text limit address

0x00400000 .text base address

Apply and Close Apply Cancel Reset

Registers			Floating Point	Control and Status
Name	Number	Value		
zero	0	0x0000000000000000		
ra	1	0x0000000000000000		
sp	2	0x000000007fffffcc		
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				

SP → 0000 003f ffff fff0<sub>esa</sub>

0000 0000 1000 0000<sub>esa</sub>

PC → 0000 0000 0040 0000<sub>esa</sub>

0



# 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

The screenshot displays a MIPS simulator interface with three main panels:

- Text Segment (Assembly Code):** Shows instructions 11 through 23. Instruction 11 is `la t0, v1` (base address of .data segment). Instructions 12-15 perform word loads and store them into registers t1, t2, and t3. Instruction 16 stores the sum of t1 and t2 into t3. Instruction 17 prints the result. Instruction 18 calls a function to exit.
- Data Segment (Memory Dump):** A table showing memory addresses and their values. The first four rows are highlighted with a red box, showing the initial state of memory.
- Registers (Control and Status):** A table showing the current values of registers. Registers t0, t1, and t2 are highlighted with a red box, showing their values after the first three instructions.

**Registers Table:**

Name	Number	Value
zero	0	0x00000000
ra	1	0x00000000
sp	2	0x00000000
gp	3	0x00000000
tp	4	0x00000000
t0	5	0x00000001
t1	6	0x00000001
t2	7	0x00000002
t3	8	0x00000000
t4	9	0x00000000
t5	10	0x00000000
t6	11	0x00000000
t7	12	0x00000000
t8	13	0x00000000
t9	14	0x00000000
t10	15	0x00000000
t11	16	0x00000000
t12	17	0x00000000
t13	18	0x00000000
t14	19	0x00000000
t15	20	0x00000000
t16	21	0x00000000
t17	22	0x00000000
t18	23	0x00000000
t19	24	0x00000000
t20	25	0x00000000
t21	26	0x00000000
t22	27	0x00000000
t23	28	0x00000000
t24	29	0x00000000
t25	30	0x00000000
t26	31	0x00000000
pc		0x00000000

**Data Segment Table:**

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x10305070	0x02040608	0x12345678	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010004	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010008	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x1001000c	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010010	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010014	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010018	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x1001001c	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

**Messages:** 305419996  
-- program is finished running (0) --

# 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)

```
Messages Run I/O
Assemble: assembling /home/idrago/academic/teaching/2022/arch_i/labs/lab2/1_sum_numbers.asm
Assemble: operation completed successfully.
Assemble: assembling /home/idrago/academic/teaching/2022/arch_i/labs/lab4/3.a_sum_mem.asm
Assemble: operation completed successfully.
Error in /home/idrago/academic/teaching/2022/arch_i/labs/lab4/3.a_sum_mem.asm line 12: Runtime exception at 0x00400008: Load address not aligned to word boundary 0x100100
Step: execution terminated with errors.
```

```
.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".**