ISA e linguaggio assembler

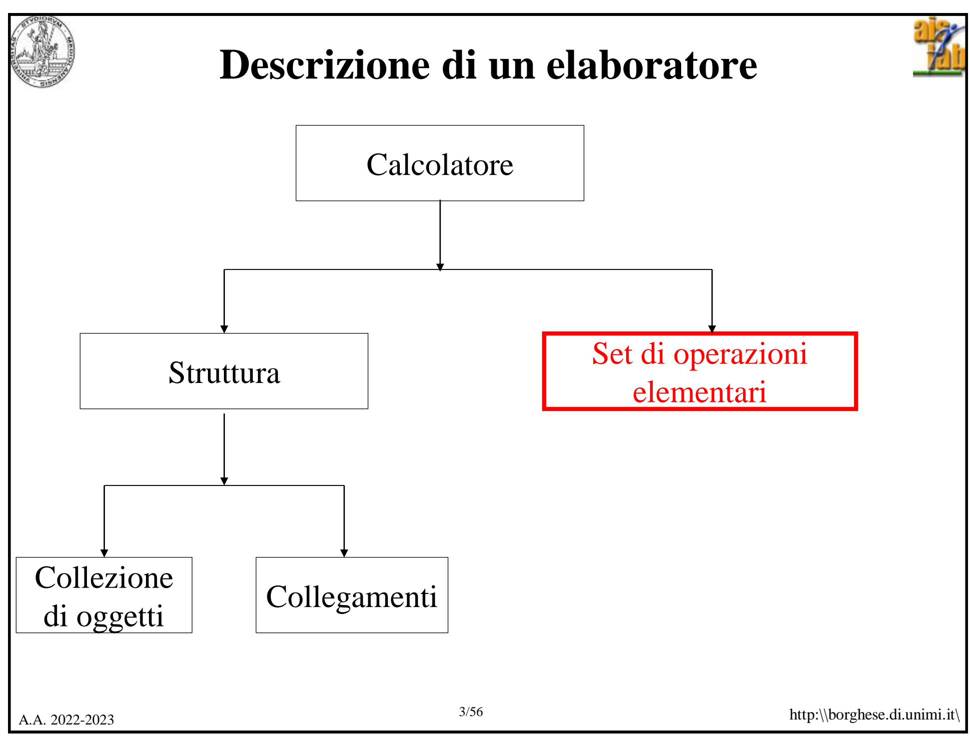
Prof. Alberto Borghese
Dipartimento di Informatica
borghese@di.unimi.it

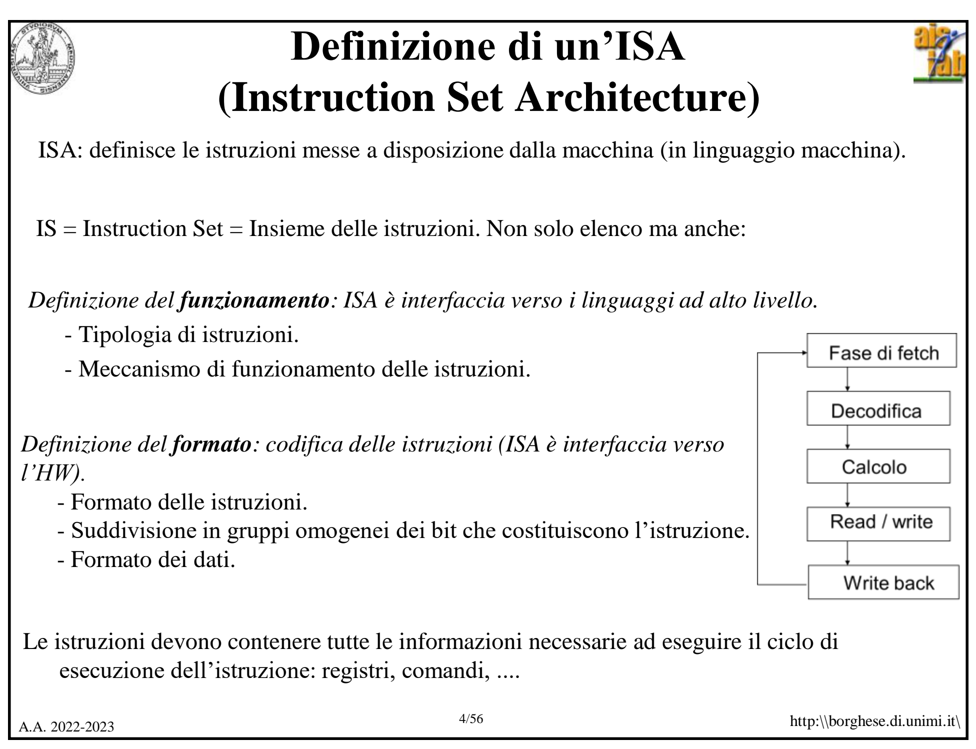
Università degli Studi di Milano

Riferimento sul Patterson: capitolo 4.2 , 4.4, D1, D2.

Sommario

- ISA
- Istruzioni aritmetico-logiche
- Istruzioni di accesso alla memoria
- Istruzioni di salto


Descrizione di un elaboratore
Calcolatore
Struttura
Set di operazioni elementari
Collezione di oggetti
Collegamenti
A.A. 2022-2023
3/56
http:\\borghese.di.unimi.it\


Definizione di un'ISA (Instruction Set Architecture)
ISA: definisce le istruzioni messe a disposizione dalla macchina (in linguaggio macchina).
IS = Instruction Set = Insieme delle istruzioni. Non solo elenco ma anche:
Definizione del funzionamento: ISA è interfaccia verso i linguaggi ad alto livello.
- Tipologia di istruzioni.
- Meccanismo di funzionamento delle istruzioni.
Definizione del formato: codifica delle istruzioni (ISA è interfaccia verso l'HW).
- Formato delle istruzioni.
- Suddivisione in gruppi omogenei dei bit che costituiscono l'istruzione.
- Formato dei dati.
Fase di fetch
Decodifica
Calcolo
Read / write
Write back
Le istruzioni devono contenere tutte le informazioni necessarie ad eseguire il ciclo di esecuzione dell'istruzione: registri, comandi, ...
A.A. 2022-2023
4/56
http:\\borghese.di.unimi.it\

ISA - IPR

ARM (**Advanced RISC Machine** and originally **Acorn RISC Machine**) è una famiglia di architetture di istruzioni.

Acorn, poi diventata RISC Limited, vende le licenze sull' ISA a società che poi realizzano i loro processori RISC. Tra i più diffusi i processori Cortex, alcuni realizzati come SoC – Systems on Chip), FPGA che comprendono memorie, interfacce, radio, ecc..

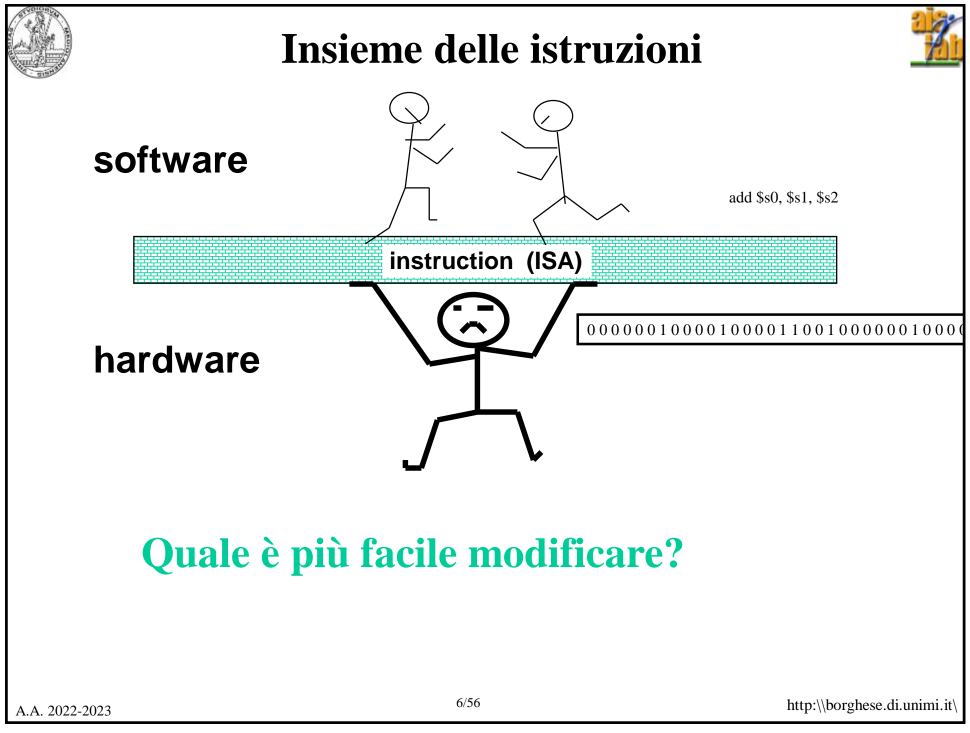
IPR – Intellectual Property Rights (proprietà intellettuale).

Nel 2019. RISC concede la licenza per lo sviluppo con pagamento delle royalities solo a partire dal primo prototipo delle CPU.

Non solo insieme di istruzioni elementari messe a disposizione dalla macchina (in linguaggio macchina).

L'architettura delle istruzioni, specifica come devono essere strutturate le istruzioni in modo tale che siano comprensibili alla macchina (in linguaggio macchina).

Insieme delle istruzioni



Quale è più facile modificare?

Tipi di istruzioni

Il linguaggio macchina (di una calcolatore) è costituito da un insieme di istruzioni che possono essere eseguite dalla macchina.

Le istruzioni comprese nel linguaggio macchina di ogni calcolatore possono essere classificate nelle seguenti quattro categorie:

- Istruzioni aritmetico-logiche;
- Istruzioni di trasferimento da/verso la memoria *(load/store);*
- Istruzioni di salto condizionato e non condizionato per il controllo del flusso di programma;
- Istruzioni di trasferimento in ingresso/uscita (I/O).

- Le istruzioni e la loro codifica costituiscono l'ISA di un calcolatore.

Tipi di istruzioni

```
for (i=0; i<N; i++)          // Istruzioni di controllo
{   elem = i*N + j;           // Istruzioni aritmetico-logiche
    s = v[elem];              // Istruzioni di accesso a memoria
    z[elem] = s;              // Istruzioni di accesso a memoria
}
```

Le istruzioni in linguaggio macchina

- Linguaggio di programmazione direttamente comprensibile dalla macchina
  - Le parole di memoria sono interpretate come *istruzioni*
  - Vocabolario è *l'insieme delle istruzioni (instruction set)*

Codice in linguaggio ad alto livello (C)

```
a = a + c
b = b + a
var = m[a]
```

Codice in linguaggio macchina

```
00000001...0101010
00000010...1000111
10001000...00010000
00000010...0010000
```

Linguaggio Assembler

- Le istruzioni assembler sono una **rappresentazione simbolica** del **linguaggio macchina** comprensibile dall'HW.

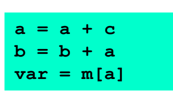
- Rappresentazione simbolica del linguaggio macchina
  - Più comprensibile del linguaggio macchina in quanto utilizza simboli invece che sequenze di bit

- Rispetto ai linguaggi ad alto livello, l'assembler fornisce limitate forme di controllo del flusso e non prevede articolate strutture dati

- Linguaggio usato come linguaggio target nella fase di compilazione di un programma scritto in un linguaggio ad alto livello (es: C, Pascal,ecc.)

- Vero e proprio linguaggio di programmazione che fornisce la visibilità diretta sull'hardware.

Codice in linguaggio ad alto livello (C)



Codice Assembler

```
add $t0,$s0,$s1
add $s2,$s2,$t0
muli $t1,$s4,4
add $t2,$s5,$t1
lw $s3,0($t2)
```

Codice in linguaggio macchina

```
011100010101010
000110101000111
000010000010000
001000100010000
```

Assembler come linguaggio di programmazione

- Principali *svantaggi* della programmazione in linguaggio assembler:
  - Mancanza di portabilità dei programmi su macchine diverse
  - Maggiore lunghezza, difficoltà di comprensione, facilità d'errore rispetto ai programmi scritti in un linguaggio ad alto livello

- Principali *vantaggi* della programmazione in linguaggio assembler:
  - Ottimizzazione delle prestazioni.
  - Massimo sfruttamento delle potenzialità dell'hardware sottostante.

- Le strutture di controllo hanno forme limitate
- Non esistono tipi di dati all'infuori di interi, virgola mobile e caratteri.
- La gestione delle strutture dati e delle chiamate a procedura deve essere fatta in modo esplicito dal programmatore.

Assembler come linguaggio di programmazione

- Alcune applicazioni richiedono un approccio *ibrido* nel quale le parti più critiche del programma sono scritte in assembly (per massimizzare le prestazioni) e le altre parti sono scritte in un linguaggio ad alto livello (le prestazioni dipendono dalle capacità di ottimizzazione del compilatore).

  Esempio: Sistemi embedded o dedicati

Sistemi "automatici" di traduzione da linguaggio ad alto livello (linguaggio C) ad Assembly e codice binario ed implementazione circuitale (e.g. sistemi di sviluppo per FPGA).

I registri

- Un registro è un insieme di celle di memora che vengono lette / scritte in parallelo.
- I registri sono associati alle variabili di un programma dal compilatore. Contengono i **dati**.
- Un processore possiede un numero limitato di registri: ad esempio il processore MIPS possiede **32 registri composti da 32-bit (word), register file.**
- I registri possono essere direttamente indirizzati mediante il loro numero progressivo (0, …, 31) preceduto da $: ad es.
  `$0, $1, …, $31`
- Per convenzione di utilizzo, sono stati introdotti nomi simbolici significativi. Sono preceduti da $, ad esempio:

  `$s0,$s1,...,$s7($s8)` Per indicare variabili in C

  `$t0, $t1, ... $t9` Per indicare variabili temporanee

I registri del register file

Nome	Numero	Utilizzo
→ $zero	0	costante zero
$at	1	riservato per l'assemblatore
$v0-$v1	2-3	valori di ritorno di una procedura
$a0-$a3	4-7	argomenti di una procedura
→ $t0-$t7	8-15	registri temporanei (non salvati)
→ $s0-$s7	16-23	registri salvati
→ $t8-$t9	24-25	registri temporanei (non salvati)
$k0-$k1	26-27	gestione delle eccezioni
$gp	28	puntatore alla global area (dati)
$sp	29	stack pointer
$s8	30	registro salvato (fp)
$ra	31	indirizzo di ritorno

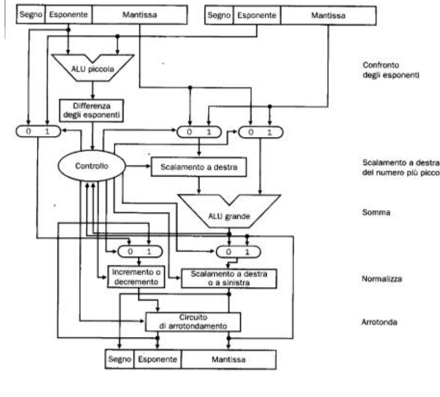
I registri per le operazioni floating point

- Esistono 32 registri utilizzati per l'esecuzione delle istruzioni.

- Esistono **32** registri per le operazioni floating point (virgola mobile) indicati come

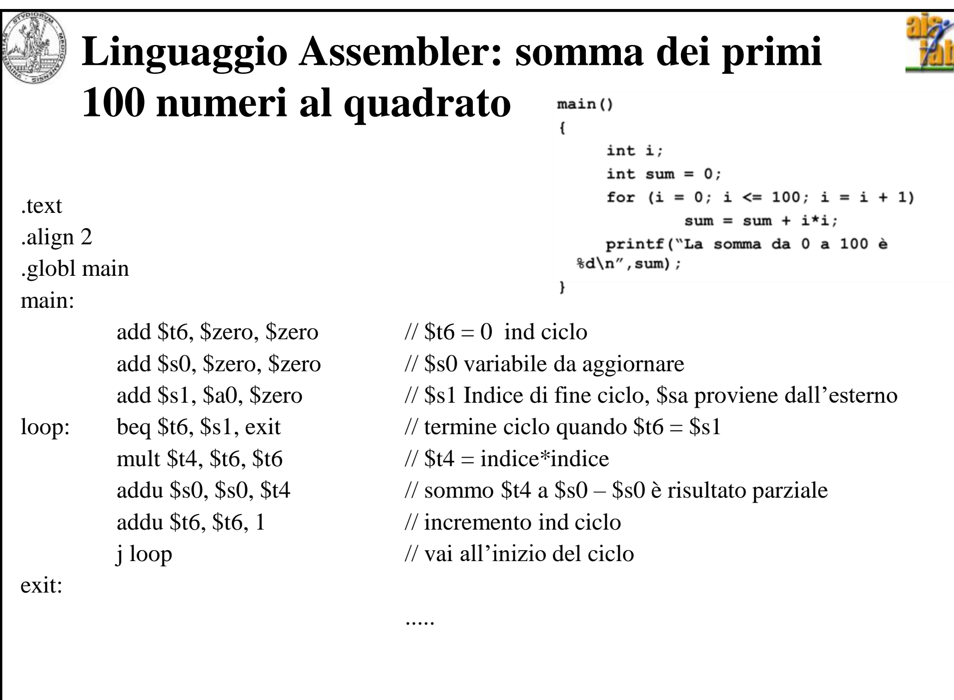
  `$f0, …, $f31`

  – Per le operazioni in doppia precisione si usano i registri contigui **`$f0, $f2, $f4, …`**



Linguaggio C: somma dei primi 100 numeri al quadrato

```
main()
{
      int i;
      int sum = 0;
      for (i = 0; i <= 100; i = i + 1)
              sum = sum + i*i;
      printf("La somma da 0 a 100 è
  %d\n",sum);
}
```



Sommario

- ISA
- Istruzioni aritmetico-logiche
- Istruzioni di accesso alla memoria
- Istruzioni di salto

Tipi di istruzioni

- Le istruzioni comprese nel linguaggio macchina di ogni calcolatore possono essere classificate nelle seguenti quattro categorie:
  - Istruzioni aritmetico-logiche;
  - Istruzioni di trasferimento da/verso la memoria *(load/store);*
  - Istruzioni di salto condizionato e non condizionato per il controllo del flusso di programma;
  - Istruzioni di trasferimento in ingresso/uscita (I/O).

Istruzioni aritmetico-logiche

- In MIPS, un'istruzione aritmetico-logica possiede in generale *tre* operandi: i due registri contenenti i valori da elaborare *(registri sorgente)* e il registro contenente il risultato *(registro destinazione).*
- L'ordine degli operandi è **fisso:** prima il registro contenente il r**isultato** dell'operazione e poi i due operandi.

```
OPCODE  DEST,  SORG1, SORG2

OPCODE rd, rs, rt
       rd = registro destinazione (DEST)
       rs = registro source (SORG1)
       rt = registro target (SORG2)
```

- La codifica prevede il codice operativo e tre campi relativi ai tre operandi:

Le operazioni vengono eseguite esclusivamente su dati presenti nella CPU, non su dati residenti nella memoria.

Esempi: istruzioni `add` e `sub`

Codice C: **R = A + B;**

Codice assembler MIPS:

```
add $s6,  $s7, $s8
add rd,   rs,  rt
```

mette la somma del contenuto di rs e rt in rd:

```
add rd, rs, rt         # rd  ← rs +  rt
add $s6, $s7, $s8      # $s6 ← $s7 + $s8
```

Nella traduzione da linguaggio ad alto livello a linguaggio assembler, le variabili sono associate ai registri dal compilatore

sub serve per sottrarre il contenuto di due registri sorgente rs e rt:

```
sub rd rs rt
```

e mettere la differenza del contenuto di rs e rt in rd

```
sub rd,   rs,  rt      # rd  ← rs  - rt
sub $s6, $s7, $s8      # $s6 ← $s7 - $s8
```

Istruzioni aritmetico-logiche in sequenza

Il fatto che ogni istruzione aritmetica ha tre operandi sempre nella stessa posizione consente di semplificare l'hw, ma complica alcune cose…

Codice C: **Z = A – (B + C + D); =>**
E = B + C + D; Z = A – E;

Suppongo che le variabili siano contenute nei seguenti registri:
A -> \$s0 B -> \$s1 C -> \$s2 D -> \$s3 Z -> \$s5

Occcorre spezzare la catena di operazioni in tante operazioni su 2 operandi. Codice MIPS:

```
add $t0, $s1, $s2
add $t1, $t0, $s3
sub $s5, $s0, $t1
```

Istruzioni aritmetico-logiche

- Operazioni con un numero di operandi maggiore di tre possono essere effettuate scomponendole in operazioni più semplici.
- Ad esempio, per eseguire la somma e sottrazione delle variabili `A.. D` nella variabile `Z` servono tre istruzioni che eseguono le operazioni in sequenza da sinistra a destra:

Codice C: **`Z = A + B - C + D;`**

Codice MIPS:

```
add $t0, $s0, $s1
sub $t1, $t0, $s2
add $s5, $t1, $s3
```

Suppongo che le variabili siano contenute nei seguenti registri:
A -> \$s0 B -> \$s1 C -> \$s2 D -> \$s3 Z -> \$s5

Implementazione alternativa

- Operazioni con un numero di operandi maggiore di tre possono essere effettuate scomponendole in operazioni più semplici.
- Ad esempio, per eseguire la somma e sottrazione delle variabili A.. D nella variabile Z servono tre istruzioni :

Codice C: **Z = A + B - C + D;**

Può essere riscritta con il seguente codice C: **Z = (A + B) – (C – D);**
Suppongo che le variabili siano contenute nei seguenti registri:
A -> \$s0 B -> \$s1 C -> \$s2 D -> \$s3 Z -> \$s5

Codice MIPS:

```
add $t0, $s0, $s1        add $t0, $s0, $s1
sub $t1, $s2, $s3        sub $t1, $t0, $s2
sub $s5, $t0, $t1        add $s5, $t1, $s3
```

Sono implementazioni equivalenti. Quale implementazione è la migliore? Sceglierà il compilatore il quale cerca di massimizzare la parallelizzazione del codice.

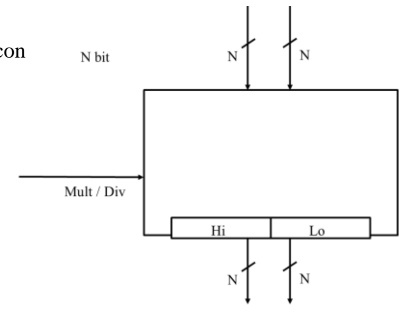
`add`: varianti

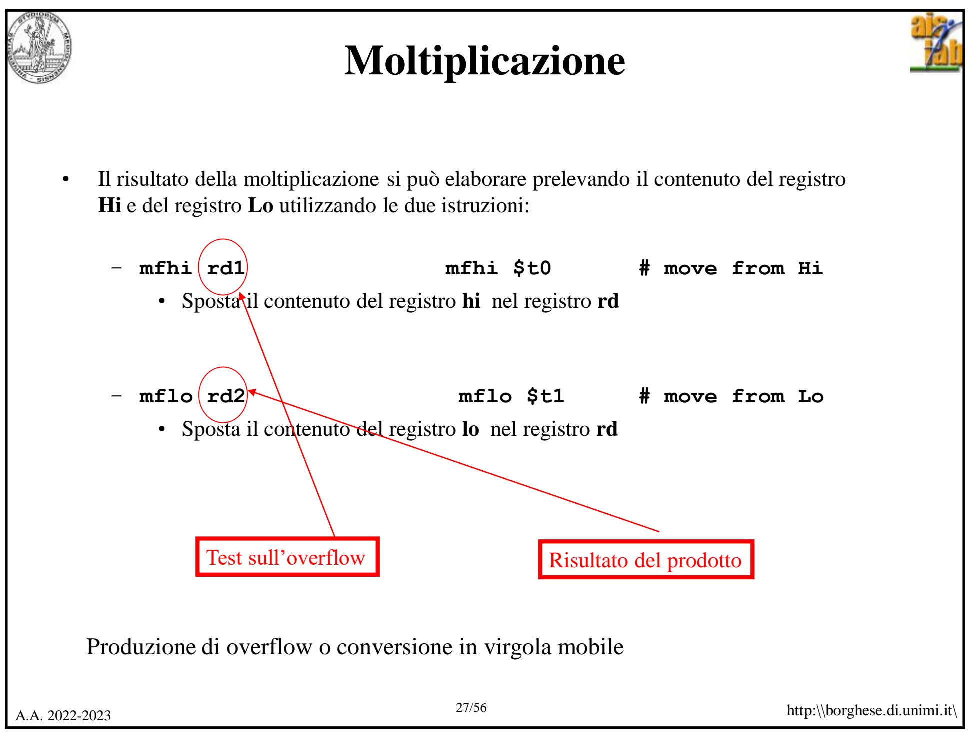
- `add $s0, $s1, $s2          #add: $s0 = $s1+$s2`
- `addi $s1, $s2, 100         #add immediate: $s1 = $s2+100`
  - Somma una costante: il valore del secondo operando è presente nell'istruzione come costante e sommata estesa in segno.
    `rt ← rs + costante`
- `addiu $s0, $s1, 100        #add immediate unsigned: $s0 = $s1+100`
  - Somma una costante ed evita overflow.
- `addu $s0, $s1, $s2         #add unsigned: $s0 = $s1+$s2`
  - Evita overflow: la somma viene eseguita considerando gli addendi sempre positivi. Il bit più significativo è parte del numero e non è bit di segno.

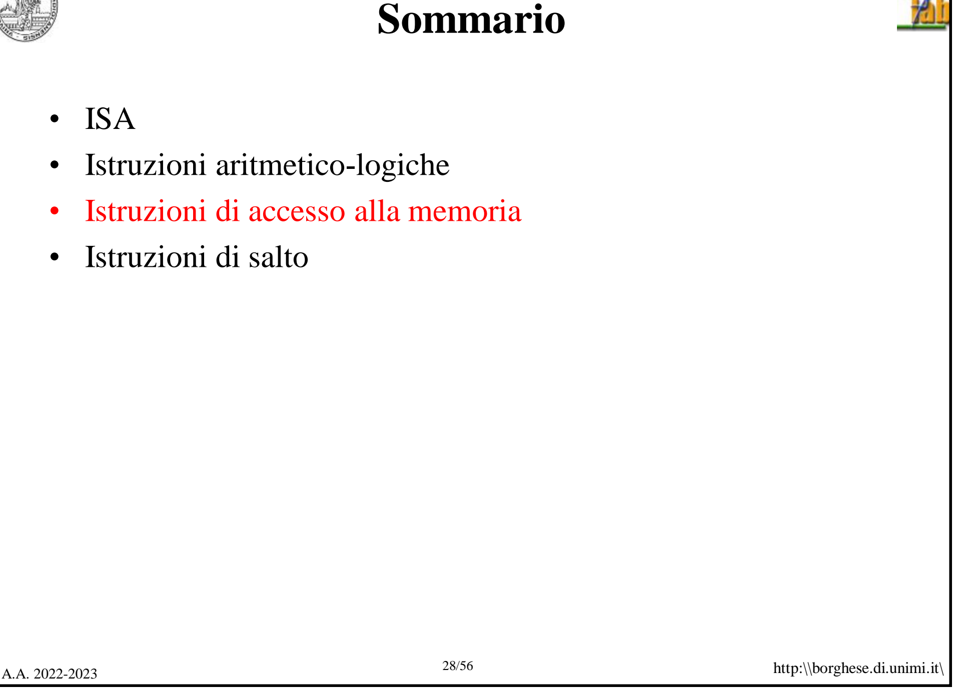
Non esiste un'istruzione di subi. Perchè?
- `addi $s1, $s2, -20             #add immediate: $s1 = $s2-20`

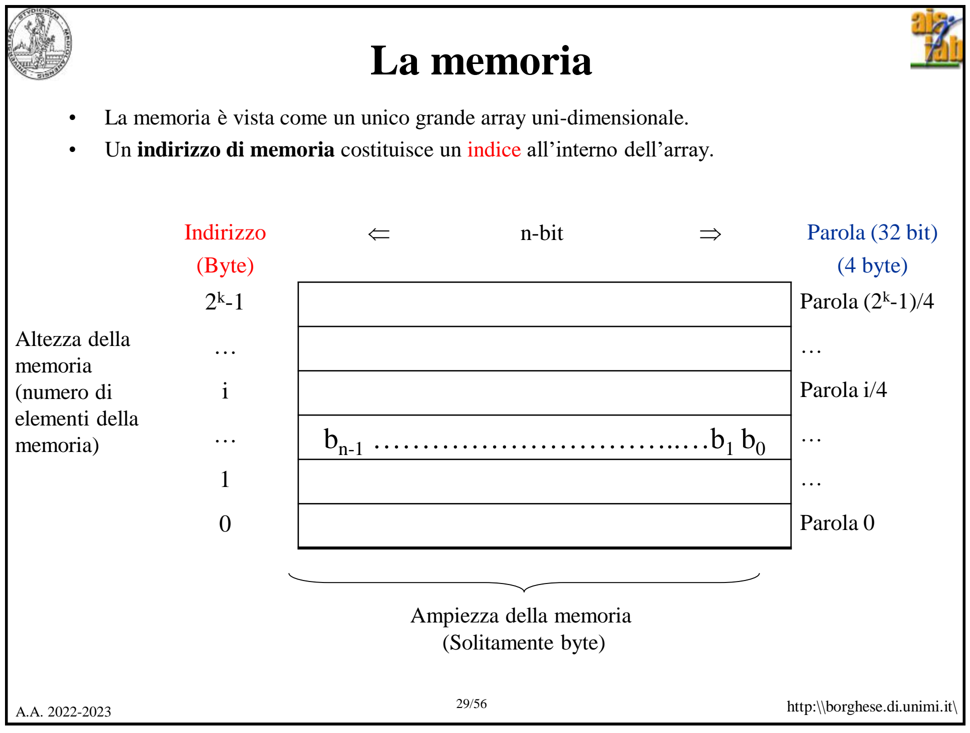
Moltiplicazione

- Due istruzioni:
  - `mult rs rt                mult $s0, $s1`
  - `multu rs rt               multu $s0, $s1       # unsigned`
- Il registro destinazione è ***implicito.*** Il risultato della moltiplicazione viene posto sempre in due registri dedicati di una parola (special purpose) denominati ***Hi (High order word)*** e ***Lo (Low order word)***
- La moltiplicazione di due numeri rappresentabili con 32 bit può dare come risultato un numero non rappresentabile in 32 bit.




Moltiplicazione
• Il risultato della moltiplicazione si può elaborare prelevando il contenuto del registro Hi e del registro Lo utilizzando le due istruzioni:
– mfhi rd1 mfhi $t0 # move from Hi
• Sposta il contenuto del registro hi nel registro rd
– mflo rd2 mflo $t1 # move from Lo
• Sposta il contenuto del registro lo nel registro rd
Test sull'overflow
Risultato del prodotto
Produzione di overflow o conversione in virgola mobile
A.A. 2022-2023
27/56
http:\\borghese.di.unimi.it\


Sommario
• ISA
• Istruzioni aritmetico-logiche
• Istruzioni di accesso alla memoria
• Istruzioni di salto
A.A. 2022-2023
28/56
http:\\borghese.di.unimi.it\

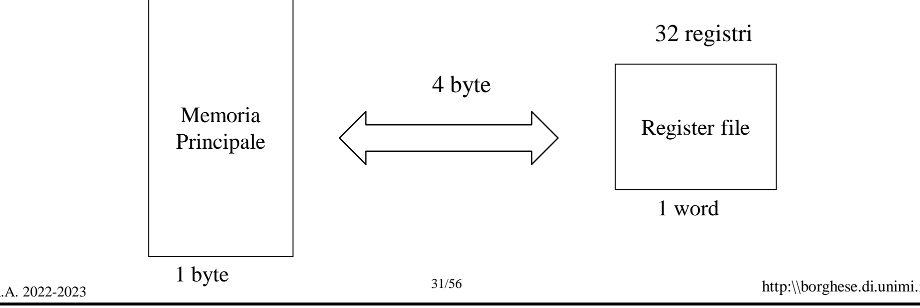


Indirizzi nella memoria principale

- La memoria è organizzata in ***parole di memoria*** composte da *n-bit* che possono essere indirizzate come un unicum (tipicamente 1 Byte)
- Ogni parola di memoria è associata ad un indirizzo composto da *k-bit.*
- I 2^k indirizzi costituiscono lo *spazio di indirizzamento* del calcolatore. Ad esempio un indirizzo di memoria composto da *32-bit* genera uno spazio di indirizzamento di 2^{32} Byte o *4Gbyte*.

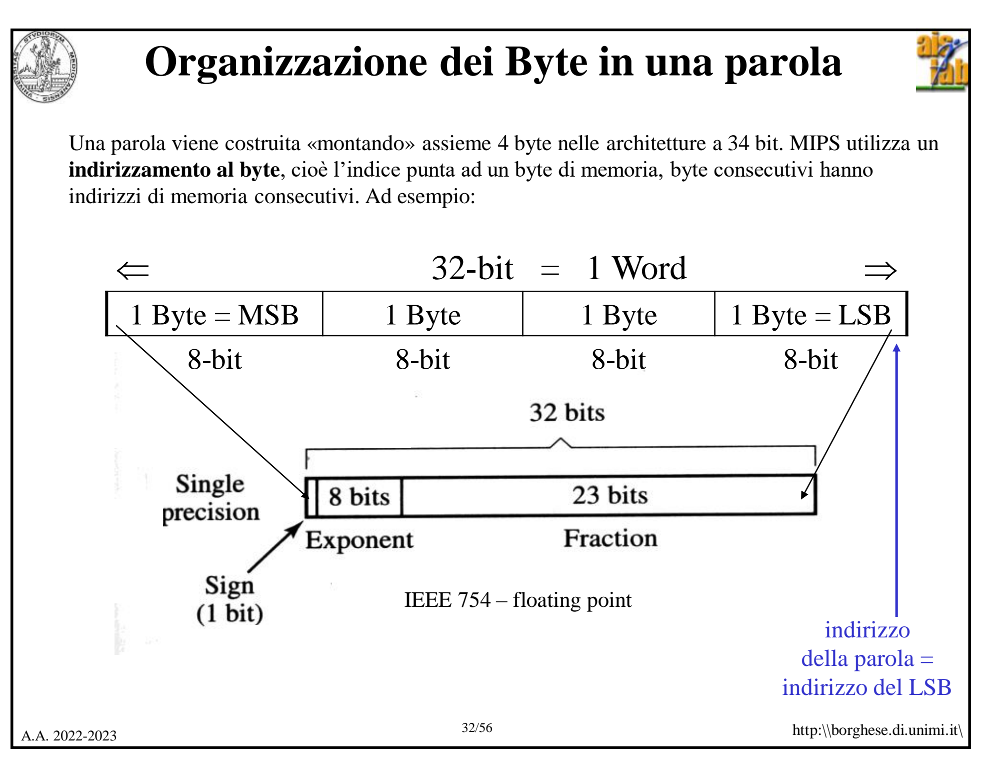
Memoria Principale e parole

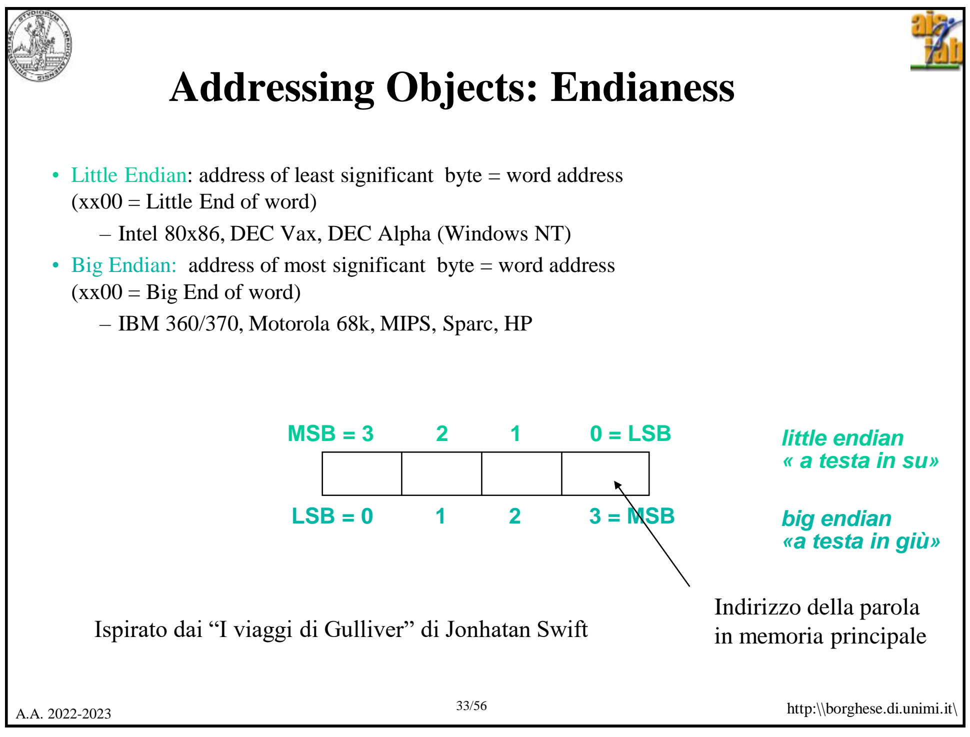
- In genere, la dimensione della parola di memoria (1 Byte) non coincide con la dimensione della parola della CPU e dei registri contenuti all'interno della *CPU* (1 word)
- Supponiamo che a ogni trasferimento tra Memoria Principale e Registri, venga trasferito contemporaneamente in parallelo un numero di Byte pari alla dimensione dei registri dell'architettura (1 word).
    - ⇒ l'operazione di *load/store* di una parola avviene in un singolo ciclo di clock del bus.
    - => Vengono trasferiti in parallelo 4 Byte
- Le parole hanno quindi generalmente indirizzo in memoria che è multiplo del numero di byte di una parola (32 bit => indirizzi spaziati di 4, 64 bit => indirizzi spaziati di 8).
- Alcuni dati possono essere rappresentati su singolo Byte (e.g. caratteri) o su coppie di Byte (e.g. audio). Può nascere un problema di allineamento dei dati.

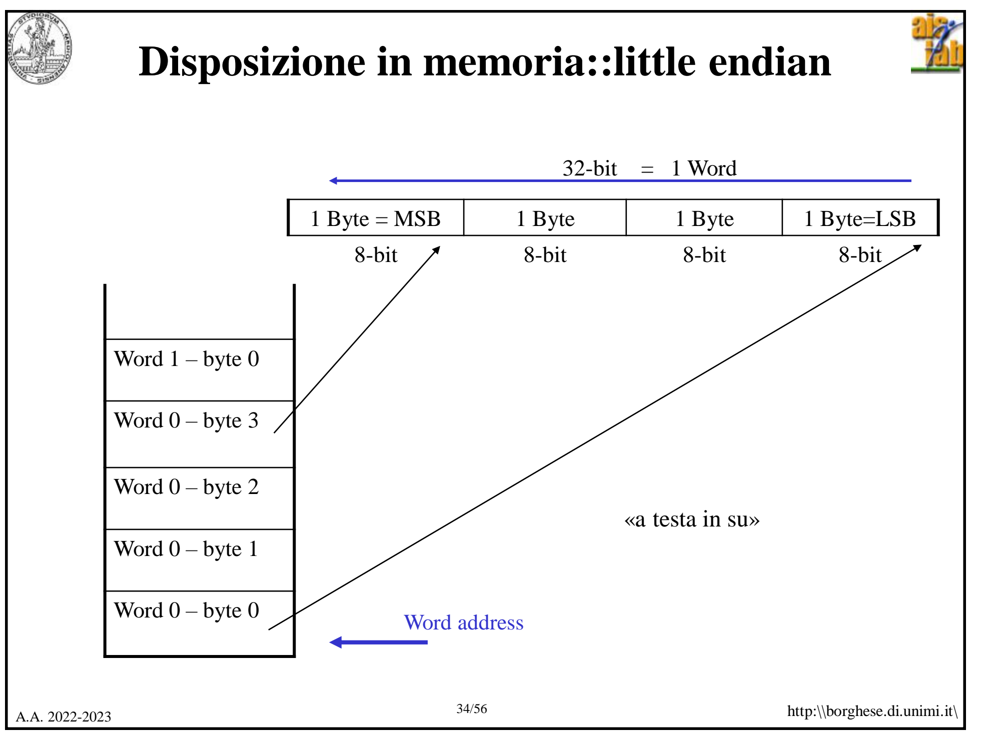


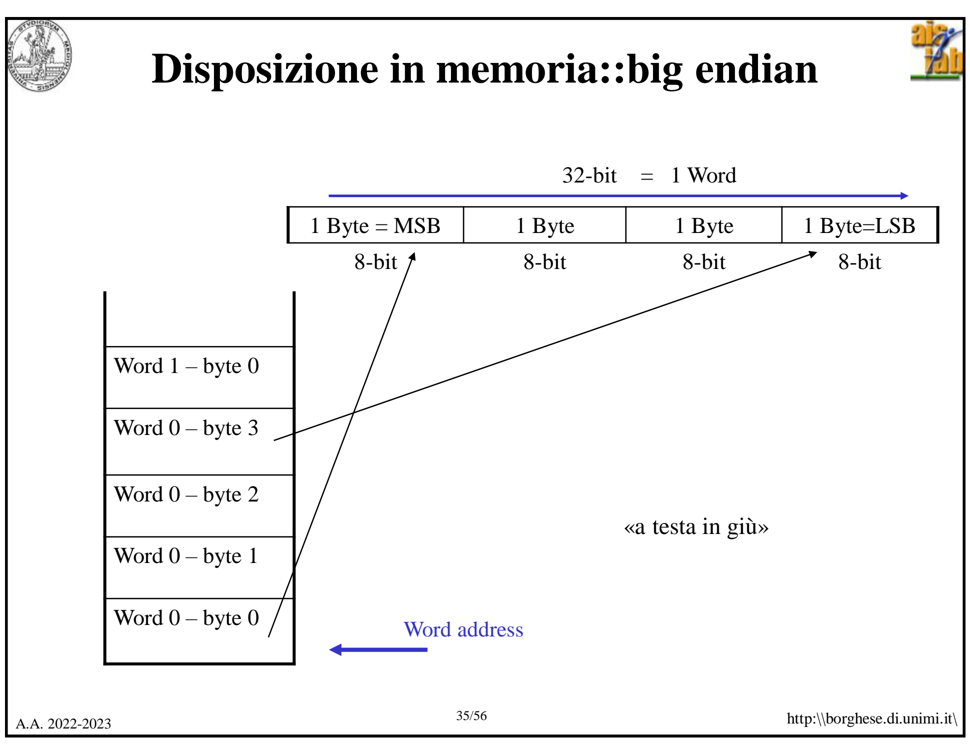
Organizzazione dei Byte in una parola

Una parola viene costruita «montando» assieme 4 byte nelle architetture a 34 bit. MIPS utilizza un **indirizzamento al byte**, cioè l'indice punta ad un byte di memoria, byte consecutivi hanno indirizzi di memoria consecutivi. Ad esempio:




Addressing Objects: Endianess
• Little Endian: address of least significant byte = word address
(xx00 = Little End of word)
– Intel 80x86, DEC Vax, DEC Alpha (Windows NT)
• Big Endian: address of most significant byte = word address
(xx00 = Big End of word)
– IBM 360/370, Motorola 68k, MIPS, Sparc, HP
MSB = 3
2
1
0 = LSB
LSB = 0
1
2
3 = MSB
little endian
« a testa in su»
big endian
«a testa in giù»
Indirizzo della parola
in memoria principale
Ispirato dai "I viaggi di Gulliver" di Jonhatan Swift
A.A. 2022-2023
33/56
http:\\borghese.di.unimi.it\

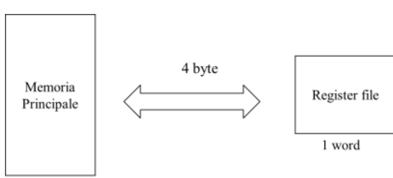

Disposizione in memoria::little endian
32-bit = 1 Word
1 Byte = MSB
1 Byte
1 Byte
1 Byte=LSB
8-bit
8-bit
8-bit
8-bit
Word 1 – byte 0
Word 0 – byte 3
Word 0 – byte 2
Word 0 – byte 1
Word 0 – byte 0
«a testa in su»
Word address
A.A. 2022-2023
34/56
http:\\borghese.di.unimi.it\



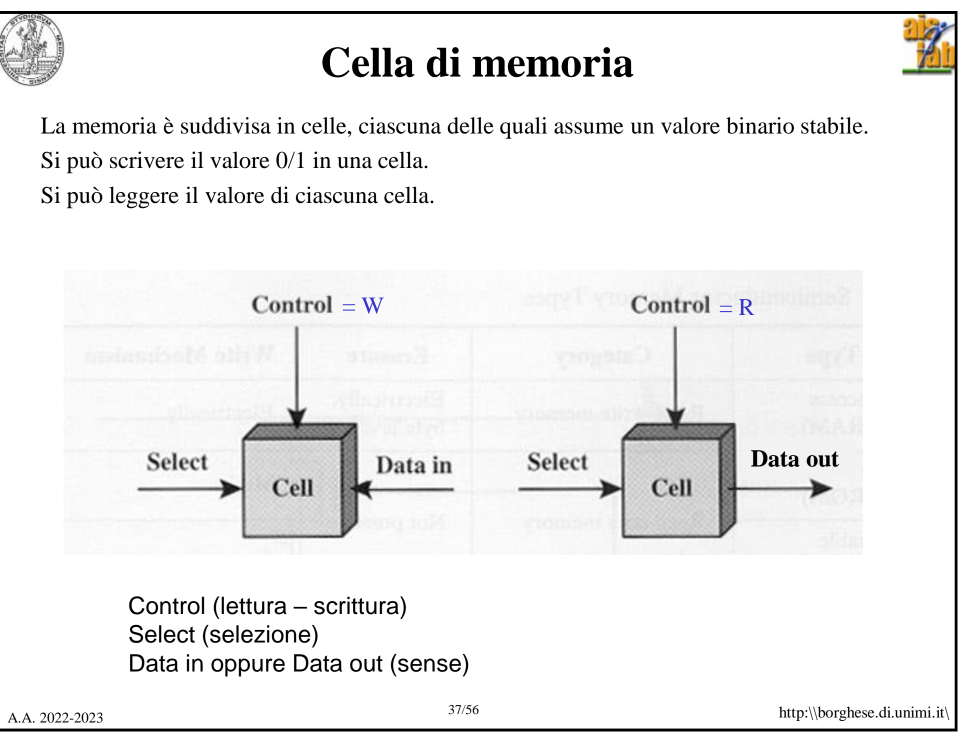
Istruzioni di trasferimento dati

- Gli operandi di una istruzione aritmetica **devono risiedere nei registri (architettura load/store)** che sono in numero limitato (32 nel MIPS). I programmi in genere richiedono un numero maggiore di variabili.
- Cosa succede ai programmi i cui dati richiedono più di 32 registri (32 variabili)?
  Alcuni dati risiederanno in memoria.
- La tecnica di trasferire le variabili meno usate (o usate successivamente) in memoria viene chiamata ***Register Spilling***.

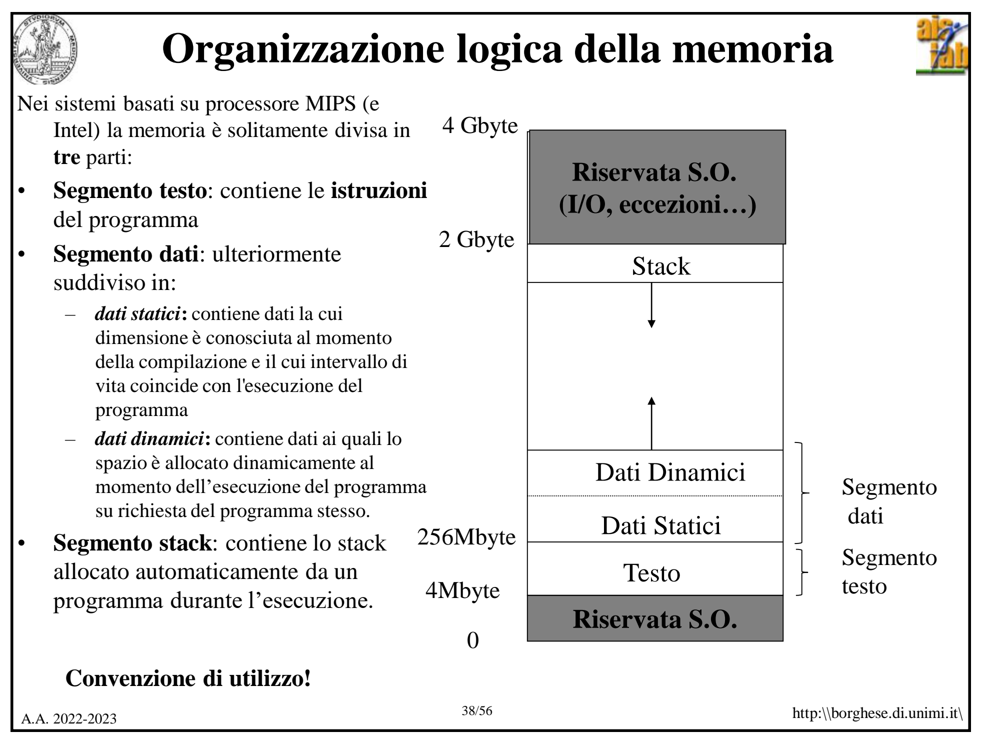
Servono istruzioni apposite per trasferire dati da memoria a registri e viceversa



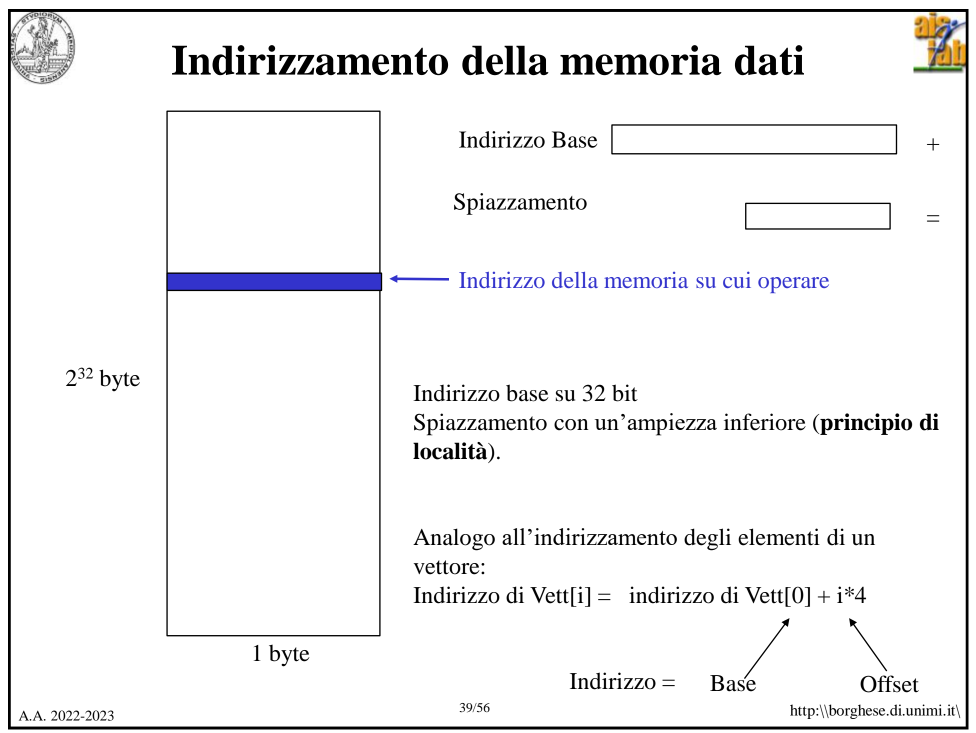
A.A. 2022-2023 http:\\borghese.di.unimi.it\

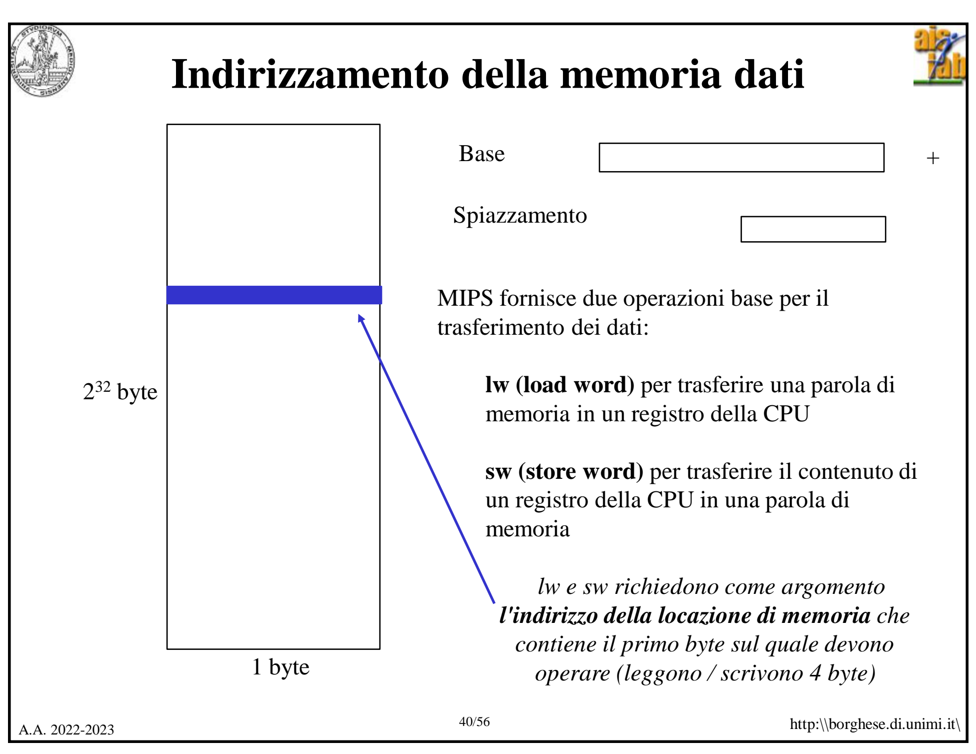


Cella di memoria
La memoria è suddivisa in celle, ciascuna delle quali assume un valore binario stabile.
Si può scrivere il valore 0/1 in una cella.
Si può leggere il valore di ciascuna cella.
Control = W
Select
Cell
Data in
Control = R
Select
Cell
Data out
Control (lettura – scrittura)
Select (selezione)
Data in oppure Data out (sense)
A.A. 2022-2023
37/56
http:\\borghese.di.unimi.it\



Organizzazione logica della memoria
Nei sistemi basati su processore MIPS (e Intel) la memoria è solitamente divisa in tre parti:
• Segmento testo: contiene le istruzioni del programma
• Segmento dati: ulteriormente suddiviso in:
– dati statici: contiene dati la cui dimensione è conosciuta al momento della compilazione e il cui intervallo di vita coincide con l'esecuzione del programma
– dati dinamici: contiene dati ai quali lo spazio è allocato dinamicamente al momento dell'esecuzione del programma su richiesta del programma stesso.
• Segmento stack: contiene lo stack allocato automaticamente da un programma durante l'esecuzione.
Convenzione di utilizzo!
4 Gbyte
Riservata S.O. (I/O, eccezioni…)
2 Gbyte
Stack
Dati Dinamici
Dati Statici
256Mbyte
Testo
4Mbyte
Riservata S.O.
0
Segmento dati
Segmento testo
A.A. 2022-2023
38/56
http:\\borghese.di.unimi.it\


Indirizzamento della memoria dati
Indirizzo Base
+
Spiazzamento
=
Indirizzo della memoria su cui operare
2^32 byte
Indirizzo base su 32 bit
Spiazzamento con un'ampiezza inferiore (principio di località).
Analogo all'indirizzamento degli elementi di un vettore:
Indirizzo di Vett[i] = indirizzo di Vett[0] + i*4
1 byte
Indirizzo = Base Offset
A.A. 2022-2023
39/56
http:\\borghese.di.unimi.it\

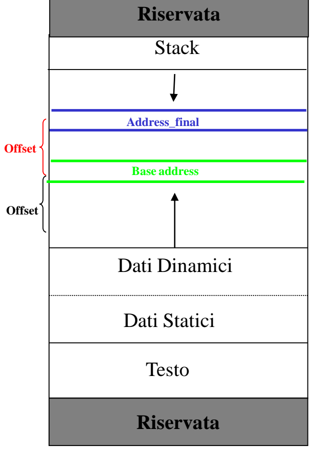

Indirizzamento della memoria dati
Base
+
Spiazzamento
MIPS fornisce due operazioni base per il trasferimento dei dati:
2^32 byte
lw (load word) per trasferire una parola di memoria in un registro della CPU
sw (store word) per trasferire il contenuto di un registro della CPU in una parola di memoria
lw e sw richiedono come argomento l'indirizzo della locazione di memoria che contiene il primo byte sul quale devono operare (leggono / scrivono 4 byte)
1 byte
A.A. 2022-2023
40/56
http:\\borghese.di.unimi.it\

Indirizzamento della memoria dati

Base + spiazzamento
- Come base può essere utilizzato il registro $gp
- Offeset positivo / negativo

Address_final = Base_address + Offset

Nome	Numero	Utilizzo
$zero	0	costante zero
$at	1	riservato per l'assemblatore
$v0-$v1	2-3	valori di ritorno di una procedura
$a0-$a3	4-7	argomenti di una procedura
$t0-$t7	8-15	registri temporanei (non salvati)
$s0-$s7	16-23	registri salvati
$t8-$t9	24-25	registri temporanei (non salvati)
$k0-$k1	26-27	gestione delle eccezioni
$gp	28	puntatore alla global area (dati)
$sp	29	stack pointer
$s8	30	registro salvato (fp)
$ra	31	indirizzo di ritorno



A volte come base address si considera il registro $gp (Global Pointer)

Istruzione *load*

- L'istruzione di *load* trasferisce una copia di un dato/istruzione, contenuto in una specifica locazione di memoria, a un registro della *CPU,* lasciando inalterata la parola di memoria:

```
load LOC, reg                # reg ← [LOC]
```

- La *CPU* invia l'indirizzo della locazione desiderata alla memoria e richiede un'operazione di lettura del suo contenuto.
- La memoria effettua la lettura del dato memorizzato all'indirizzo specificato e lo invia alla *CPU*.

Implementazione MIPS

Nel MIPS l'istruzione di caricamento di un dato dalla memoria è: "load word" (lw):

- Nel MIPS, l'istruzione **lw** ha tre argomenti:
  - un registro base (*base register*) che contiene il valore dell'indirizzo base *(base address)* da sommare all'offset.
  - una costante o *spiazzamento* (*offset)*
  - il *registro destinazione* in cui caricare la parola letta dalla memoria

```
lw rt, costante(rs)    # rt ← M[ [rs] + costante ]
lw $s1, 100($s2)       # $s1 ← M[ [$s2] + 100 ]
```

Al registro destinazione $s1 è assegnato il valore contenuto all'indirizzo della memoria principale: ($s2 + 100). L'indirizzo è espresso ***in byte****.*

Questo spiega la semantica di «registro target» per il registro SORG2

Istruzione di *sw*

- L'istruzione di *store* trasferisce una parola di dato/istruzione da un registro della *CPU* in una specifica locazione di memoria, sovrascrivendo il contenuto precedente di quella locazione:

```
store reg, LOC         # [LOC] ← reg
```

- La *CPU* invia l'indirizzo della locazione di memoria, assieme con i dati che vi devono essere scritti e richiede un'operazione di scrittura.

- La memoria effettua la scrittura dei dati all'indirizzo specificato.

L'istruzione MIPS per la scrittura di un registro in memoria è la sw (store word). Essa possiede argomenti analoghi alla lw

Esempio:

```
sw rt, costante(rs)    # M[ [rs] + costante ] ← rt
sw $s1, 100($s2)       # M[ [$s2] + 100 ] ← $s1
```

 Alla locazione di memoria di indirizzo ($s2 + 100) è caricato il contenuto del registro $s1

lw & sw: esempio

Elaborazione di dati di un vettore A.

Codice C: `A[12] = h + A[8];`

- Si suppone che:
  - la variabile **h** sia associata al registro **$s2**
  - l'indirizzo del primo elemento dell'array *(base address)* sia contenuto nel registro **$s3 (A[0])**

Codice MIPS:

```
lw $t0, 32($s3)          # $t0 ← M[ [$s3] + 32]
add $t0, $s2, $t0        # $t0 ← $s2 + $t0
sw $t0, 48($s3)          # M[ [$s3] + 48] ← $t0
```

Memorizzazione di un vettore

- L'elemento **i-esimo** di un array di N elementi, si troverà nella locazione **`br + 4 * i`** dove:
  - **`br`** è il registro base;
  - **`i`** è l'indice del vettore (e.g. codice C);
  - il fattore **`4`** dipende dall'indirizzamento al byte della memoria nel MIPS e si riferisce ad architetture a 32 bit

Assembler (puntatori)	C
s3	A[0]
s3 + 4	A[1]
s3 + 8	A[2]
	…..

A[0]	3	2	1	0	0x40000
A[1]	7	6	5	4	0x40004
A[2]	11	10	9	8	0x40008
					……
A[N-1]	$2^{N*4}-1$	$2^{N*4}-2$	$2^{N*4}-3$	$2^{(N-1)*4}$	

Frammento di gestione di un vettore

- Sia A un array di N word. **Realizziamo l'istruzione C:** g = h + A[i]
- Si suppone che:
  - le variabili **g, h, i** siano associate rispettivamente ai registri **$s1, $s2,** ed **$s4**
  - l'indirizzo del primo elemento dell'array *(base address)* sia contenuto nel registro **$s3**
- L'elemento **i-esimo** dell'array si trova nella locazione di memoria di indirizzo **($s3+ 4*i)**
- Caricamento dell'indirizzo di A[i] nel registro temporaneo **$t1:**

```
muli $t1, $s4, 4        # $t1 ← 4 * i
add $t1, $t1, $s3       # $t1 ← address of A[i]
                        # that is ($s3 + 4 * i)
```

- Per trasferire A[i] nel registro temporaneo **$t0:**

```
lw $t0, 0($t1)          # $t0 ← A[i]
```

- Per sommare h e A[i] e mettere il risultato in g:

```
add $s1, $s2, $t0               # g = h + A[i]
```

Vettori: aritmetica dei puntatori

Codice C:

```
for (i=0; i<N; i+=2)
    g = h + A[i];
```

Supponiamo che l'indirizzo del primo elemento dell'array A *(base address)* sia contenuto nel registro **$s3**

Codice Assembler:

First iteration:

```
lw $t0, 0($s3)          # Carico l'indirizzo dell'elemento 0
                        # di A (base address)
```

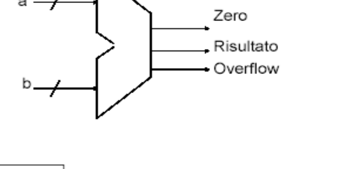
All the other iterations:

```
addi $s3, $s3, 8        # Carico l'elemento successivo (+=2)
lw $t0, 0($s3)
...
```

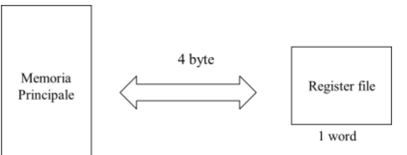
- Increment of the address of the location of A[i], inside $s3, by adding the proper offset (here 4 Byte * 2 elements = 8 Byte, as we supposed a 32 bit architecture)

Istruzioni aritmetiche vs. load/store

- Le istruzioni aritmetiche leggono il contenuto di due registri (operandi) , eseguono una computazione e scrivono il risultato in un terzo registro (destinazione o risultato)



- Le operazioni di trasferimento dati leggono e scrivono un solo operando senza effettuare nessuna computazione. Tuttavia utilizzano 2 registri, di cui uno viene utilizzato per costruire l'indirizzo.



- Le operazioni di trasferimento dati sono necessarie per eseguire le istruzioni aritmetiche!! (cf. Roof model)

Sommario

- ISA
- Istruzioni artmetico-logiche
- Istruzioni di accesso alla memoria
- Istruzioni di salto

Istruzioni di salto in ciclo for

Ciclo a condizione iniziale di uscita (può essere eseguito 0 volte)

```
for (i=0; i<N; i++)          // Istruzioni di controllo (i != N → (i – N) !=0)
{   elem = i*N + j;          // Istruzioni aritmetico-logiche
    s = v[elem];             // Istruzioni di accesso a memoria
    z[elem] = s;             // Istruzioni di accesso a memoria
}

inizia:   0x40000  beq $t0, $s0, esci       // $s0 conteggio fine ciclo
          0x40004  ..
                   ..
                   ..
          0x40068  j inizia                 ; torna in ciclo
esci:     0x4006C  …
```

Istruzioni di salto condizionato

- Salti condizionati relativi:
  - **beq rs, rt, Etichetta** (*branch on equal*)
  - **bne rs, rt, Etichetta** (*branch on not equal*)

- Salti condizionati relativi:
  - Il flusso sequenziale di controllo cambia solo se la condizione è vera. (beq)

```
beq $s1, $s0, esci   # if (s1 == s0) esci

bne $s1, $s0, esci   # if (s0 ≠ s0) esci
```

Condizioni di minoranza

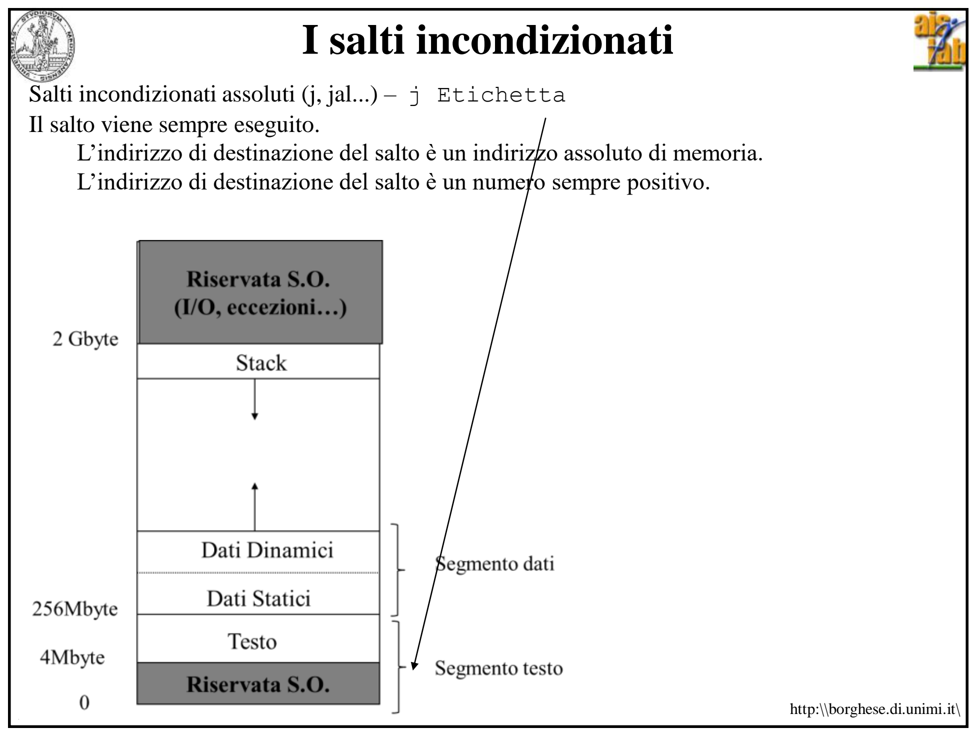
```
blt $s1, $s0, esci  # if (s1 < s0) esci
```

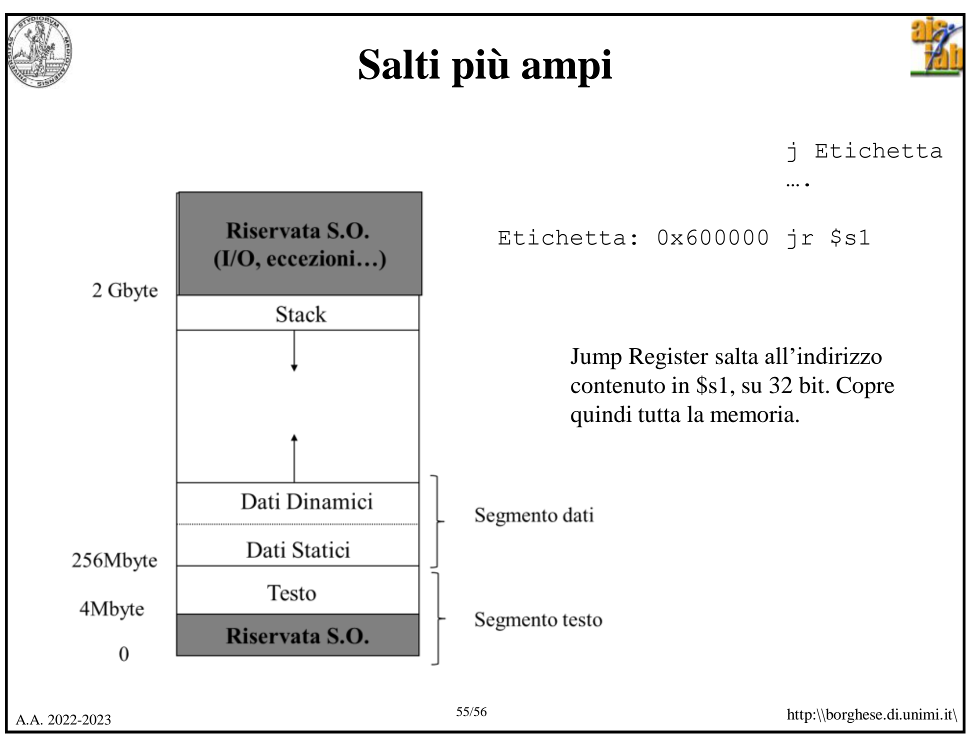
blt è una **pseudo-istruzione:**
- Non fa parte dell'ISA
- E' un'istruzione molto utilizzata
- Equivale a due o più istruzioni dell'ISA

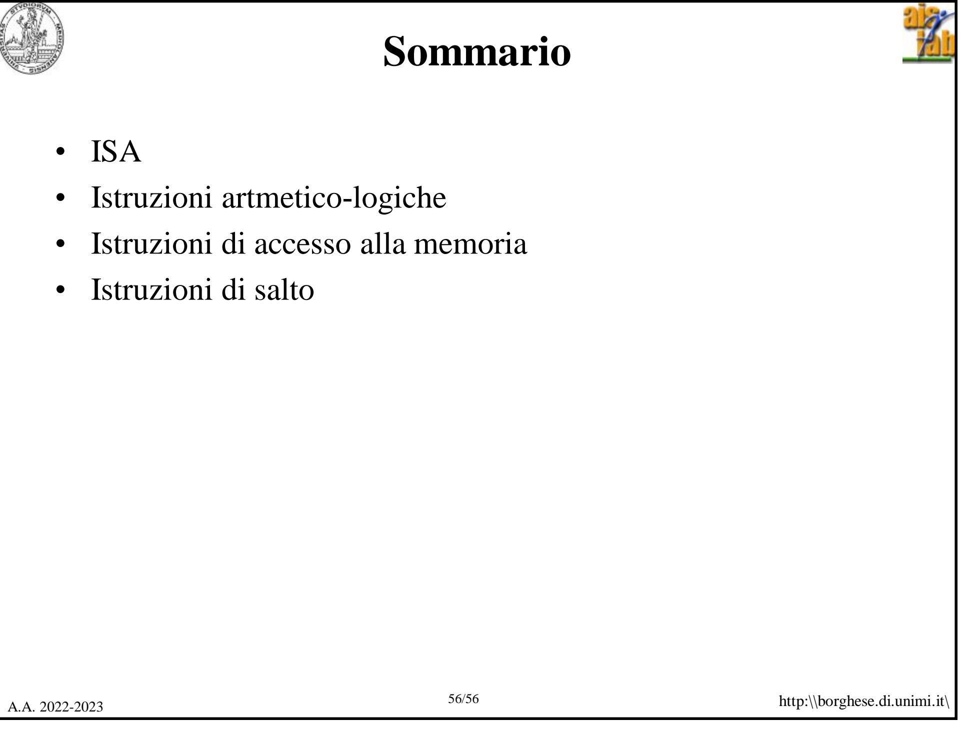
```
slt $t0, $s1, $s0          # if (s1 < s0) t0 = 1
bne $t0, $zero, esci       # if (t0 ≠ 0) esci
```


I salti incondizionati

Salti incondizionati assoluti (j, jal...) – `j Etichetta`
Il salto viene sempre eseguito.
L'indirizzo di destinazione del salto è un indirizzo assoluto di memoria.
L'indirizzo di destinazione del salto è un numero sempre positivo.




Salti più ampi
j Etichetta
….
Etichetta: 0x600000 jr $s1
Riservata S.O.
(I/O, eccezioni…)
2 Gbyte
Stack
Jump Register salta all'indirizzo contenuto in $s1, su 32 bit. Copre quindi tutta la memoria.
Dati Dinamici
Segmento dati
Dati Statici
256Mbyte
Testo
4Mbyte
Segmento testo
Riservata S.O.
0
A.A. 2022-2023
55/56
http:\\borghese.di.unimi.it\


Sommario
• ISA
• Istruzioni artmetico-logiche
• Istruzioni di accesso alla memoria
• Istruzioni di salto
A.A. 2022-2023
56/56
http:\\borghese.di.unimi.it\