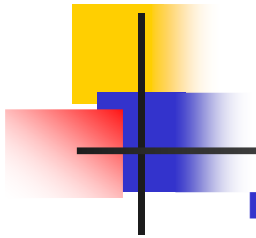
Università degli Studi di Roma "La Sapienza"

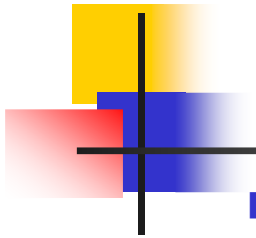
Architettura degli elaboratori II

Introduzione ai concetti ed
al simulatore SPIM



Indice degli argomenti

- **Introduzione**
  - Assembler, compilatore, linker, programma eseguibile
- **Elementi di un programma assembly**
  - Direttive all'assembler
  - Etichette e riferimenti
  - Registri generali e speciali
  - Istruzioni
- **Uso del tool SPIM**
  - Un semplice simulatore del processore MIPS R2000/R3000
  - Come utilizzare SPIM



Alcuni concetti fondamentali

- **Linguaggio macchina**
  - Basato su stringhe binarie
  - utilizzato dai computer per memorizzare ed eseguire programmi.
- **Linguaggio assembly**
  - Rappresentazione simbolica del linguaggio macchina (simboli per rappresentare istruzioni, registri e dati)
  - usato dai programmatori.
- **Linguaggi ad alto livello**
  - Costrutti più astratti che permettono al programmatore di non specificare vari dettagli implementativi.

Linguaggio basso/alto livello

Linguaggio C

```
/* esemp1.c */

void main()
{
    int x, y, z;
    x=4;
    y=6;
    z=x+y;
}
```

Linguaggio assembly

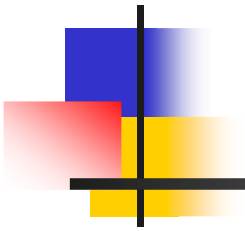
```
/* esemp1.s */

.text
lw $t1, x
lw $t2, y
add $t0,$t1,$t2
store $t0,z

.data
x:    .word 4
y:    .word 6
z:    .word 0
```

Linguaggio macchina

```
0000000100000100
0000001000000110
0000000000001010
```



Concetti di base

Assembler

Linker

Compilatore

Compilatore vs Assemblatore

- Compilatore:

  ling. alto livello $\rightarrow$ *ling. macchina*

- Assemblatore (+ linker):

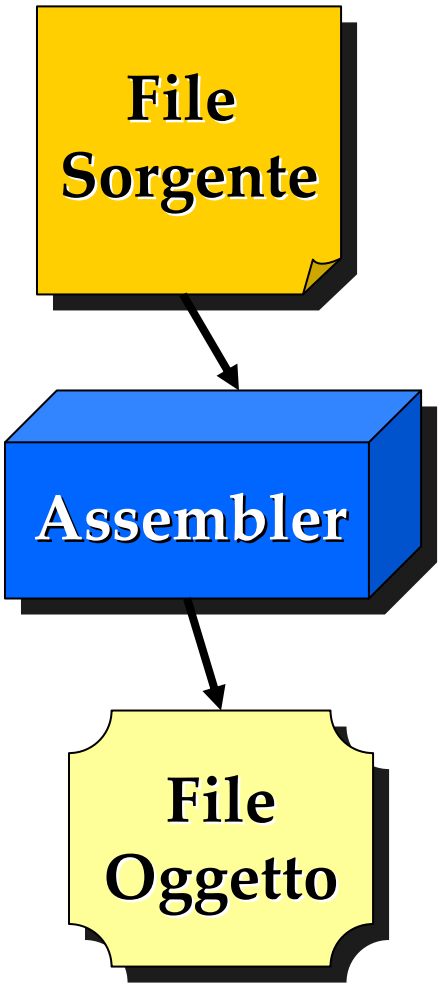
  ling. assembly $\rightarrow$ *ling. macchina*

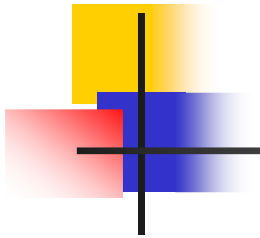
Di solito, per semplificare il compilatore, il passaggio a ling. macchina avviene in 2 passi:

L. alto liv. $\xrightarrow[\text{compilatore}]{}$ *L. assembly* $\xrightarrow[\text{assemblatore (+ linker)}]{}$ *L. macchina*

Assembler (assemblatore)

- L'**Assembler** traduce programmi scritti nel linguaggio assembly in linguaggio macchina.
- L'Assembler:
  - legge un ***file sorgente scritto in linguaggio assembly***
  - produce un ***file oggetto (o modulo)*** contenente linguaggio macchine ed altre informazioni necessarie per trasformare uno o più file oggetto in un programma eseguibile





File oggetto (o modulo)

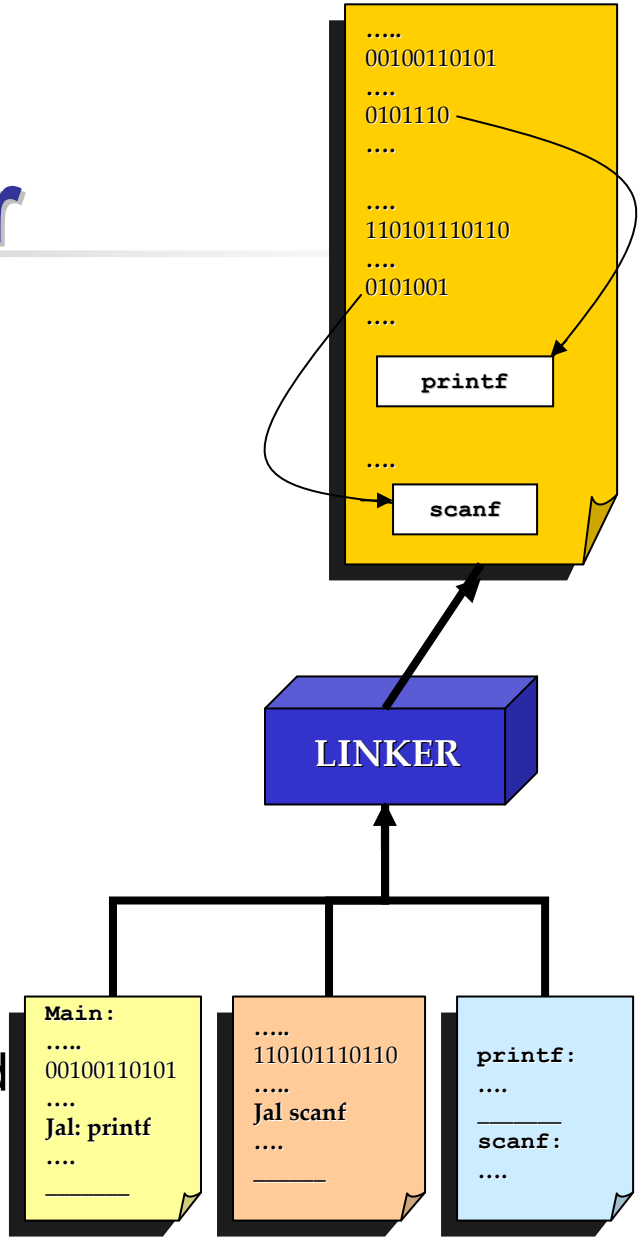
- **Un modulo può contenere**
  - Istruzioni (operazioni, routine, subroutine, ecc.)
  - Dati, variabili globali
  - Riferimenti a subroutines e dati di altri moduli o di librerie (*unresolved references*)

- **PROBL.:** il programma eseguibile lavora su indirizzi fisici
  - non deve contenere unresolved references
  - è necessario tradurre riferimenti simbolici in indirizzi fisici

Linker

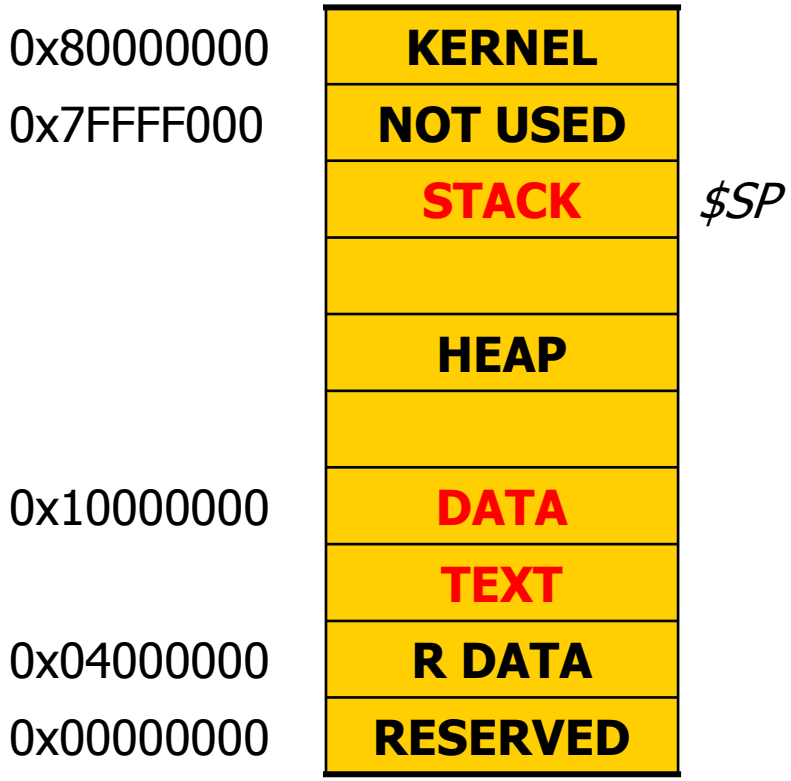
- Il **Linker** combina un *insieme di file oggetto* e *file libreria* in un *programma eseguibile*

- Il linker ha tre compiti:
  - Risolvere i riferimenti tra i diversi file
  - Ricercare nei file libreria le routine di libreria utilizzate dal programma (es. *printf*)
  - Determinare le locazioni di memoria che il codice di ogni modulo andrà ad utilizzare e inserire i corrispondenti indirizzi fisici



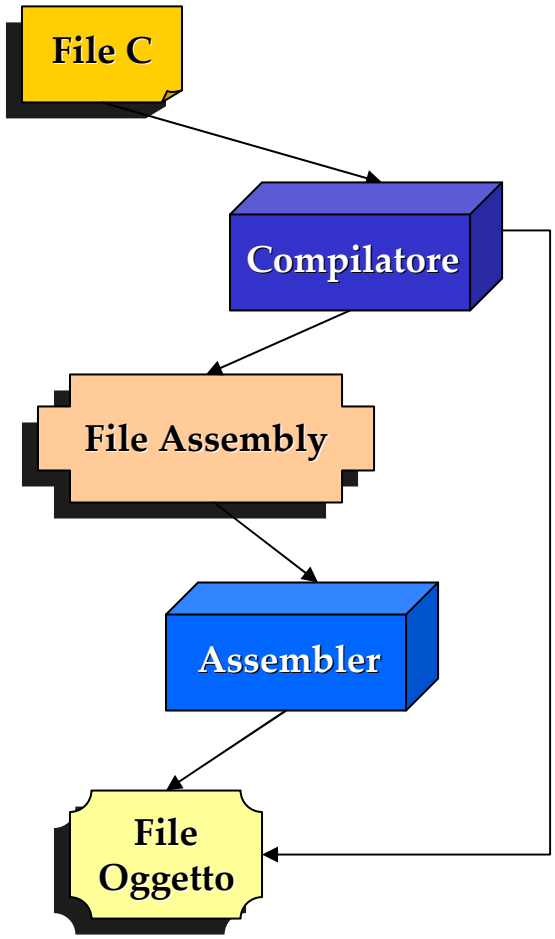
Organizzazione memoria

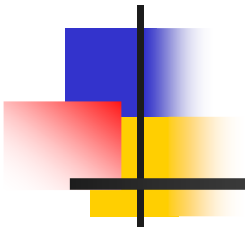
- La memoria viene suddivisa in segmenti
- Ogni segmento viene utilizzato per un particolare scopo
- **Segmenti principali:**
  - ***Text***: Contiene il codice dei programmi
  - ***Data***: Contiene i dati "globali" dei programmi
  - ***Stack*** Contiene i dati "locali" delle funzioni



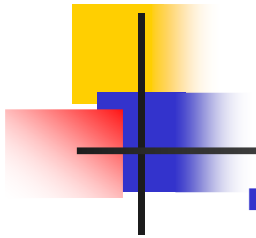
Compilatore

- Il **Compilatore** traduce un programma scritto in un linguaggio ad alto livello in un:
  - programma equivalente scritto in linguaggio assembly, che può essere trasformato in un file oggetto da un assembler
  - oppure, direttamente in un file oggetto





Assembler



Assembler (Assemblatore)

- **Compito principale di un assembler è quello di consentire una programmazione immediata ad un livello prossimo alla macchina**
- Utilizzo di parole mnemoniche per identificare le istruzioni del linguaggio macchina

  addiu $29, $29, -32

  piuttosto che

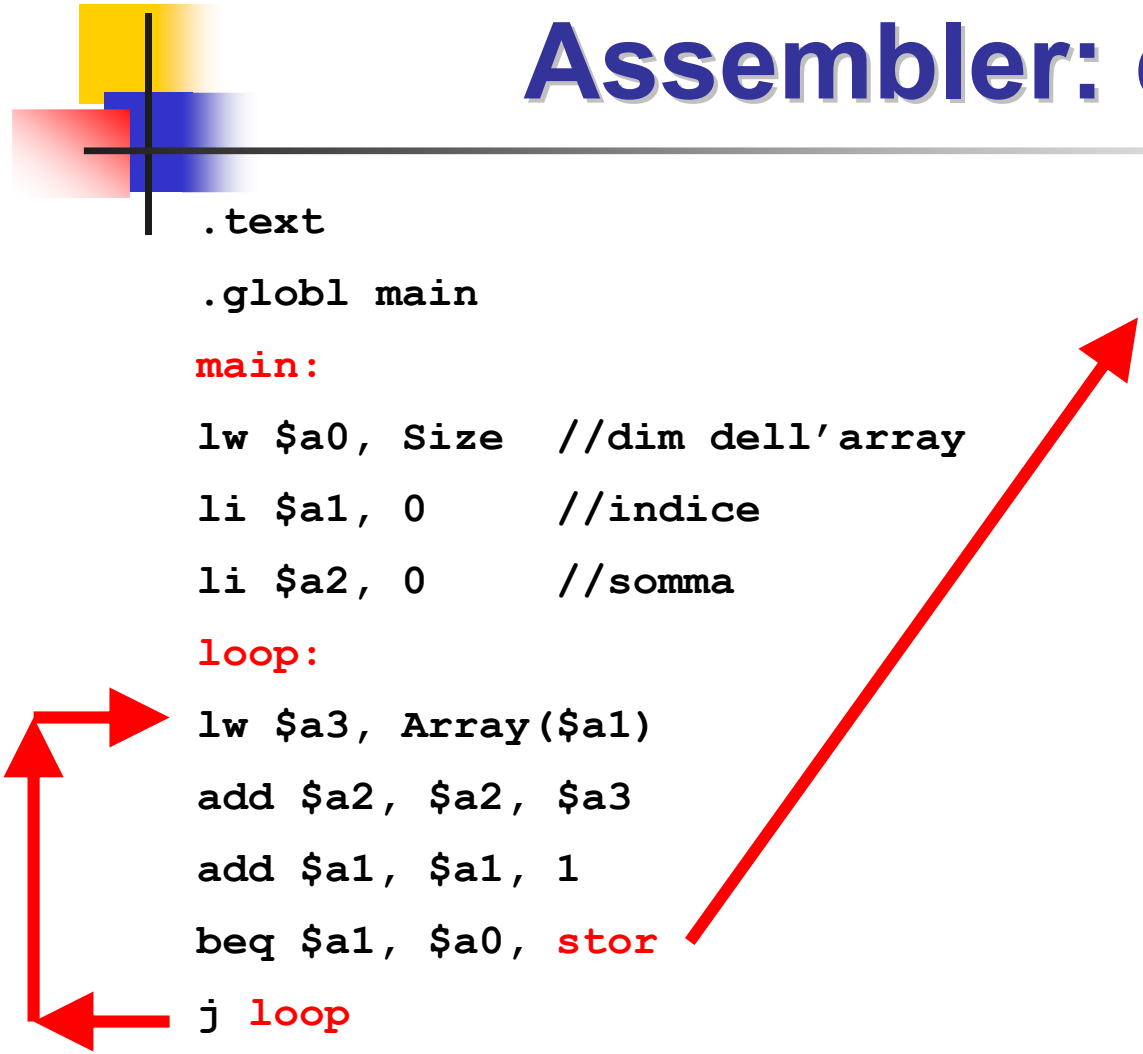
  00100111101111011111111111100000
- Possibilità di definire **macro** **(non in SPIM)**
- Possibilità di definire **etichette** **(sì)**
- Possibilità di aggiungere **commenti** **(sì)**

Assembler: vantaggi/svantaggi

- Realizzare sistemi real time
- Ottimizzare sezioni critiche dal punto di vista della performance di un programma
- Utilizzare *istruzioni particolari* del processore altrimenti non utilizzate dai compilatori
- Sviluppare parte del *kernel* di un sistema operativo, che necessita di istruzioni particolari per gestire la protezione della memoria

- Complesso
  - Specificare tutti i dettagli implementativi
  - Illeggibilità del codice
  - Scarsa gestibilità
- Non portabilità del codice
- Programmatore è Supervisore e supergestore
- Scarsa produttività
- Massima efficienza dei compilatori moderni

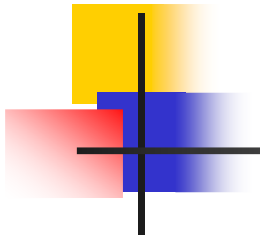
Assembler: esempio



```
.text
.globl main
main:
lw $a0, Size   //dim dell'array
li $a1, 0      //indice
li $a2, 0      //somma
loop:
lw $a3, Array($a1)
add $a2, $a2, $a3
add $a1, $a1, 1
beq $a1, $a0, stor
j loop
```

```
stor:
sw $a2, Result
…


.data
Array:  .word 1, 2, 3, 4, 5
Size:   .word 5
Result: .word 0
```



Direttive

Direttive

- Forniscono informazioni addizionali utili all'assembler per gestire l'organizzazione del codice
- Iniziano con un *punto*

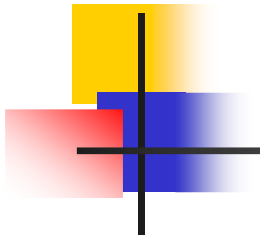
Esempi:

.text

indica che le linee successive contengono **istruzioni**

.data

indica che le linee successive contengono **dati**

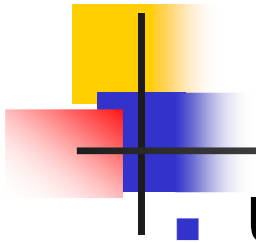


Identificatori

Identificatori

- Un identificatore è un nome associato ad una particolare posizione del programma assembly come l'indirizzo di una istruzione o di un dato.
- Un identificatore consiste in una sequenza case sensitive di caratteri alfanumerici.

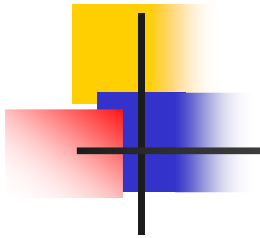
  Esempio: main, loop, stor2, Size, size, Array, Result
- Ogni istruzione o dato si trova in un particolare indirizzo di memoria. Un identificatore ci permette di fare riferimento ad una particolare posizione senza sapere il suo indirizzo in memoria (che non ci interessa)



Etichette

- Una etichetta *introduce* un identificatore e lo associa al punto del programma in cui si trova.
- Un'etichetta consiste in un identificatore seguito dal simbolo *due punti*.

  Esempio: ***main:***, ***loop:***, ***stor:***, ***Size:***, ***Array:***, ***Result:***
- L'identificatore introdotto può avere una visibilità locale o globale. Le etichette sono locali; l'uso della direttiva *.globl* rende un'etichetta globale
- Una etichetta **locale** può essere referenziata solo dall'interno del file in cui è definita; una etichetta **globale** può essere referenziata anche da file diversi da quello in cui è definita.



Riferimenti

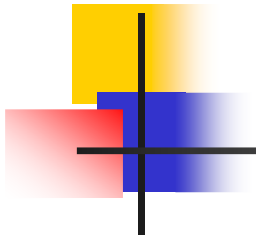
- Gli identificatori possono essere *usati* nelle istruzioni assembly e nei dati per fare riferimento alla posizione del programma associata all'identificatore.
- È sufficiente una etichetta anche per dati che occupano più byte; basta aggiungere uno *scostamento* o *offset* (calcolato in byte) al riferimento base.

  Esempio

  Array: .word 1, 2, 3, 4, 5

  # *Array* si può usare come riferimento al dato 1

  # *Array* + *4* è un riferimento al dato 2, ecc.



Allocazione di memoria per dati

Allocazione di memoria nel segmento *data*:

- È possibile allocare memoria nel segmento *data* utilizzando alcune direttive che permettono di specificare come la memoria sarà utilizzata e il valore iniziale della memoria stessa.
- Il valore iniziale può essere specificato tramite espressioni, costanti o stringhe.

  Esempi

  - “Hello World”
  - 0xAA+12
  - 10*10

Direttive per allocazione memoria

.byte b1 , …, bn	*Alloca n quantità a 8 bit in byte successivi in memoria*
.half h1, …, hn	*Alloca n quantità a 16 bit in halfword successive in memoria*
.word w1, …, wn	*Alloca n quantità a 32 bit in word successive in memoria*
.float f1, …, fn	*Alloca n valori floating point a singola precisione in locazioni successive in memoria*
.double d1, …, dn	*Alloca n valori floating point a doppia precisione in locazioni successive in memoria*
.asciiz str	*Alloca la stringa str in memoria, terminata con il valore 0*
.space n	*Alloca n byte, senza inizializzazione*

Registri generali

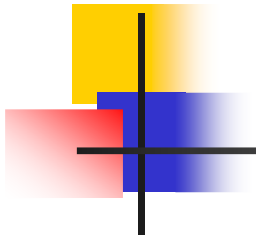
- 32 registri della CPU
- alcuni riservati (non utilizzabili dal programmatore)
- altri:

Numero	Nome Simbolico	Funzione
$0	**$zero**	*Valore fisso a 0*
$2-$3	**$v0-$v1**	*Risultati di una funzione*
$4-$7	**$a0-$a3**	*Argomenti di una funzione*
$8-$15, $24, $25	**$t0-$t9**	*Temporanei (non preservati fra chiamate)*
$16-$23	**$s0-$s7**	*Temporanei (preservati fra le chiamate)*
$28	**$gp**	*Pointer to global area*
$29	**$sp**	*Stack pointer*
$30	**$fp**	*Frame pointer*
$31	**$ra**	*Return address*

Registri speciali

PC	*Program counter*
HI	*Risultato di una moltiplicazione, parte più significativa*
LO	*Risultato di una moltiplicazione, parte meno significativa*

- **Nota**
- I registri generali possono essere utilizzati in qualunque istruzione, a scelta del programmatore (sebbene esistano delle convenzioni)
- I registri speciali *non* sono registri generali, quindi **non** possono essere utilizzati dalle istruzioni normali
- Esistono istruzioni speciali per gestire i registri speciali:
  - Istruzioni "Branch" e "Jump" per il PC
  - Istruzioni **mthi**, **mtlo**, **mfhi**, **mflo** per LO ed HI



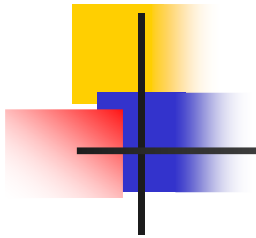
Istruzioni

Un'istruzione inizia con una parola riservata (keyword) e continua a seconda della sua sintassi

N.B.: Ad ogni istruzione del linguaggio macchina MIPS corrisponde un'istruzione del linguaggio assembly

Esempio:

bne reg1, reg2, address (branch if not equal)



Pseudo-Istruzioni

Il viceversa non vale: esistono istruzioni fornite dall'assembler ma non implementata in hardware

Esempio:

blt reg1, reg2, address (branch if less than)

diventa:

slt $at, reg1, reg2 (set less than)

bne $at, $zero, address (branch if not equal)