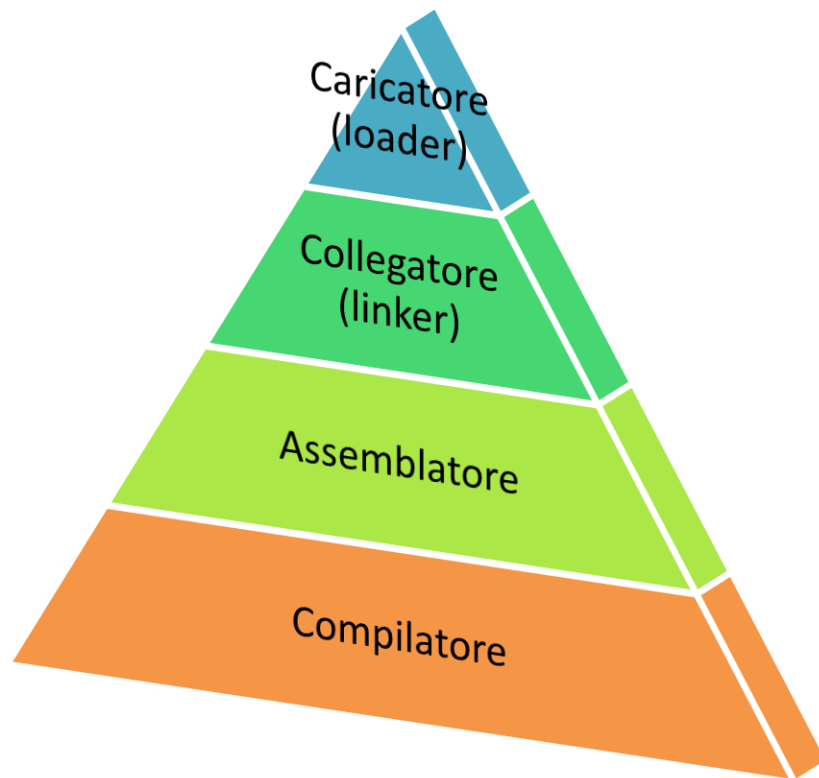


# Architettura degli Elaboratori Elettronici

*Dott. Franco Liberati*  
*liberati@di.uniroma1.it*

# ARGOMENTI DELLA LEZIONE

- ☐ Compilatore
- ☐ Assemblatore
- ☐ Collegatore (linker)
- ☐ Caricatore (loader)

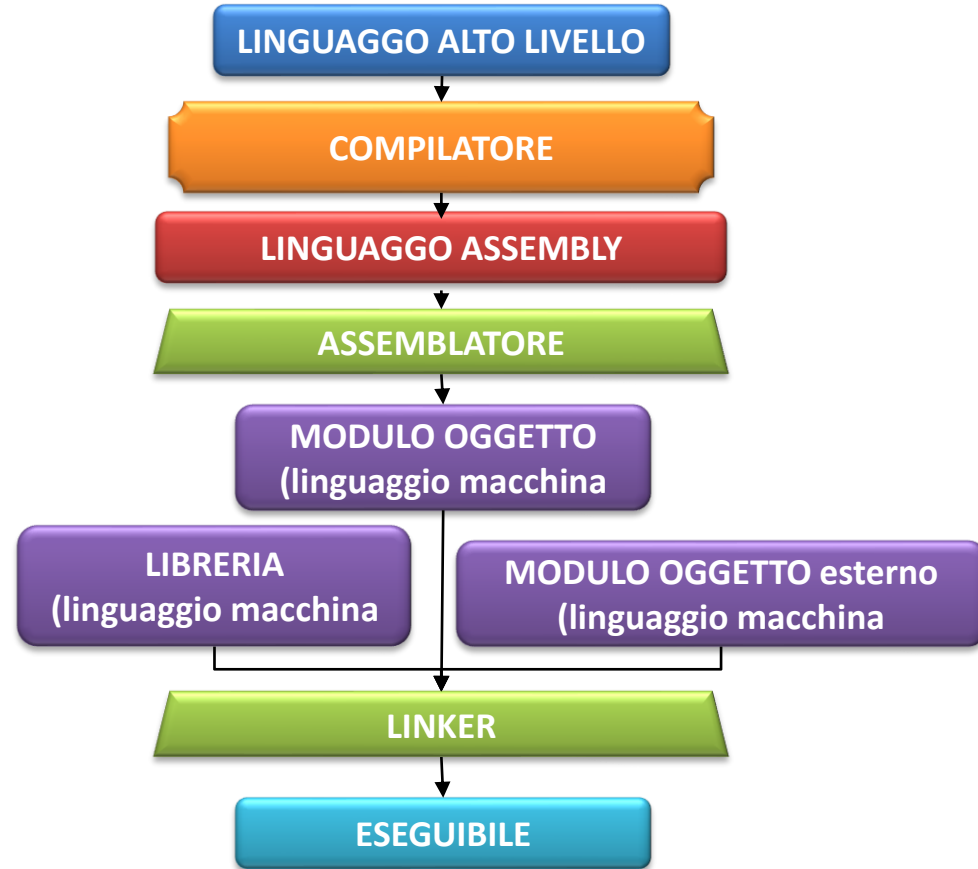


*Processo di produzione  
di un documento eseguibile*

# PROGRAMMA

## Generalità

- ❑ L'**esecuzione di un programma** è il punto di arrivo di una sequenza di azioni che nella maggior parte dei casi iniziano con la scrittura di un programma in un linguaggio simbolico di alto livello
- ❑ Le azioni principali che compongono tale sequenza nel caso si parta da un linguaggio ad alto livello sono quelle che vedono in gioco il **compilatore**, l'**assemblatore** e il **collegatore** (linker)
- ❑ Alcuni calcolatori raggruppano queste azioni per ridurre il tempo di traduzione, ma concettualmente tutti i **programmi compilati** passano sempre attraverso le fasi mostrate



# PROGRAMMA

## Compilatore

- ❑ Il **compilatore** trasforma, dopo un controllo sintattico, il programma scritto in un linguaggio ad alto livello in uno in linguaggio assembly, cioè in una forma simbolica che il calcolatore è in grado di capire ma, ancora, non eseguire
- ❑ Durante la generazione del codice, il compilatore effettua il **riordino delle istruzioni** cioè quali istruzioni sono trasmesse al processore e in quale ordine (utile nella canalizzazione o per il calcolo parallelo)
- ❑ Infine il compilatore **ottimizza il codice**: toglie istruzioni inutili o variabili non utilizzate



# PROGRAMMA

## Compilatore

Linguaggio  
alto livello  
(linguaggio C)

```
ad Main ()
{
    int ris=Pow (2,3);
}

int Pow(int b,int e)
{
    int t=1;
    for(i=0;i<e;i++)
    {
        t=t*b;
    }
    return(t);
}
```

Linguaggio  
assembly  
(SPIM)

```
.text
.globl main
main:
    lw $a0,base    #caricamento valore
    lw $a1,espo    #caricamento valore
    jal pow        #salto a funzione
    sw $a2,ris     #spostamento risultato in memoria
    li $v0,10
    syscall
pow:
    li $t0,0       #inizializzazione contatore
    li $t1,1       #inizializzazione risultato temporaneo
    move $t3,$a0
ciclo:
    bge $t0,$a1,fine #confronto contatore-esponente
    mul $t1,$t1,$t3  #moltiplicazione per la base
    addi $t0,1      #incremento contatore
    j ciclo        #salto

fine:
    move $a2,$t1
    jr $ra         #ritorno a funzione
.data             #dichiarazione variabili
base: .word 2
espo: .word 3
ris: .word 0
```

# PROGRAMMA

## Assemblatore

- ❑ L'**assemblatore** converte un programma assembly in un **file oggetto**, che è una combinazione di istruzioni in linguaggio macchina, di dati e di informazioni necessarie a collocare le istruzioni in memoria nella posizione opportuna
- ❑ Un programma assembly è tradotto in una sequenza di istruzioni (opcode, indirizzi, costanti, ecc.) attraverso il **processo di assemblaggio** (assembler) costituito da due passi logici successivi ed in parte indipendenti :
  1. il programma assembly è letto sequenzialmente, si identificano le istruzioni e i loro operandi, si calcola la lunghezza e si assegna un indirizzo (relativo) a ciascuna istruzione; inoltre, quando è letto un simbolo (un indirizzo simbolico, cioè una etichetta), nome e indirizzo sono inseriti in una **tabella dei simboli** (*symbol table*): nome e indirizzo di un simbolo possono essere inseriti nella symbol table in momenti diversi se un simbolo è usato prima di essere definito
  2. il programma assembly è letto sequenzialmente, a tutti i simboli è sostituito il valore numerico corrispondente presente nella symbol table, a tutte le istruzioni e ai relativi operandi ancora in forma simbolica è sostituito il valore numerico corrispondente (opcode, ecc.).

**Il processo di assemblaggio prende il nome di assegnazione interna delle locazioni (*internal relocate symbol o internal reference*)**

# PROGRAMMA

## Assemblatore doppio passaggio (esempio)

| Programma       |                 |
|-----------------|-----------------|
| lw \$t0,pippo   |                 |
| ciclo:          | beqz \$t0,salto |
| add \$t0,\$t0,1 |                 |
| j ciclo         |                 |
| salto:          | sw \$t0,pluto   |
|                 |                 |
| pippo:.word 4   |                 |
| pluto:.word 0   |                 |

### PASSO 1

| Indirizzo | Istruzione      |
|-----------|-----------------|
| 100       | lw \$t0,pippo   |
| 101       | ciclo:          |
| 102       | beqz \$t0,salto |
| 103       | add \$t0,\$t0,1 |
| 104       | j 101           |
| 105       | salto:          |
| 106       | sw \$t0,pluto   |
|           |                 |
| 300       | pippo:.word 4   |
| 301       | pluto:.word 0   |

### TABELLA SIMBOLI

|       |     |
|-------|-----|
| pippo | 300 |
| ciclo | 101 |
| salto | 105 |
| pluto | 301 |



# PROGRAMMA

## Assemblatore doppio passaggio (esempio)

| Programma       |                 |
|-----------------|-----------------|
| lw \$t0,pippo   |                 |
| ciclo:          | beqz \$t0,salto |
| add \$t0,\$t0,1 |                 |
| j ciclo         |                 |
| salto:          | sw \$t0,pluto   |
|                 |                 |
| pippo:.word 4   |                 |
| pluto:.word 0   |                 |

### PASSO 2

| Indirizzo | Istruzione      |
|-----------|-----------------|
| 100       | lw \$t0, 300    |
| 101       |                 |
| 102       | beqz \$t0, 105  |
| 103       | add \$t0,\$t0,1 |
| 104       | j 101           |
| 105       |                 |
| 106       | sw \$t0, 301    |

### TABELLA SIMBOLI

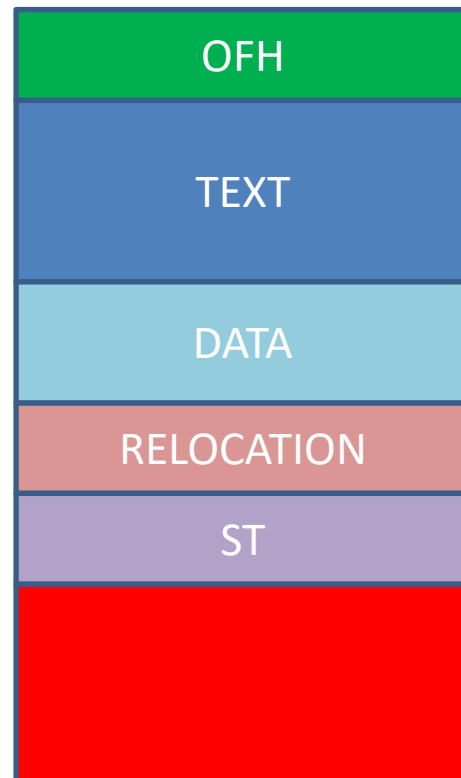
|       |     |
|-------|-----|
| pippo | 300 |
| ciclo | 101 |
| salto | 105 |
| pluto | 301 |

|     |               |
|-----|---------------|
| 300 | pippo:.word 4 |
| 301 | pluto:.word 0 |

# PROGRAMMA

## Disposizione dei file oggetto in memoria

- ❑ I file oggetto sono suddivisi e disposti in memoria di solito in sei sezioni distinte:
1. *object file header*: descrive la dimensione e la posizione delle altre sezioni del file oggetto;
  2. *text segment*: contiene le istruzioni in linguaggio macchina;
  3. *data segment*: contiene tutti i dati che fanno parte del programma;
  4. *relocation information*: identifica le istruzioni e i dati che dipendono da indirizzi assoluti e che dovranno essere rilocati dal linker
  5. *symbol table*: contiene i simboli che non sono ancora definiti, ad esempio le etichette che fanno riferimento a moduli esterni;
  6. *debugging information*: contiene informazioni per il debugger.



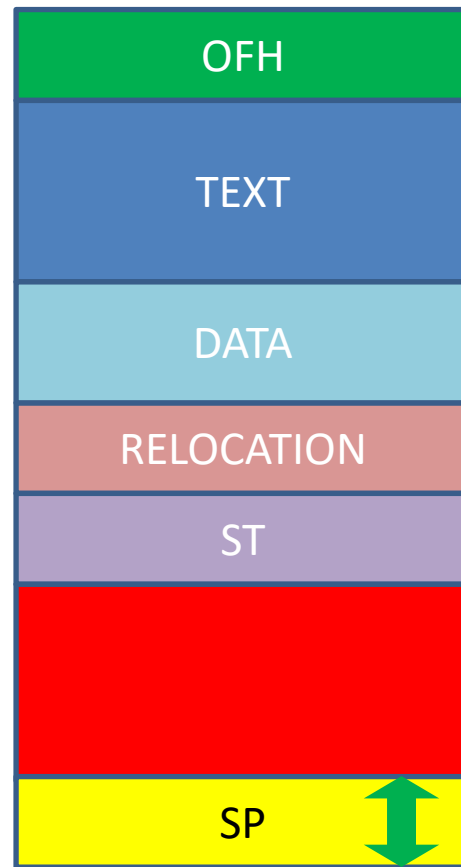
# PROGRAMMA

## Disposizione dei file oggetto in memoria

❑ I file oggetto sono suddivisi e disposti in memoria di solito in sei sezioni distinte:

1. *object file header*: descrive la dimensione e la posizione delle altre sezioni del file oggetto;
2. *text segment*: contiene le istruzioni in linguaggio macchina;
3. *data segment*: contiene tutti i dati che fanno parte del programma;
4. *relocation information*: identifica le istruzioni e i dati che dipendono da indirizzi assoluti e che dovranno essere rilocati dal linker
5. *symbol table*: contiene i simboli che non sono ancora definiti, ad esempio le etichette che fanno riferimento a moduli esterni;
6. *debugging information*: contiene informazioni per il debugger.

In più si riserva uno spazio nel quale può avvenire uno scambio di dati (STACK)



# PROGRAMMA

## Collegatore (Linker)

- ❑ Quando un programma simbolico è costituito da più moduli contenuti in diversi file sorgenti, il processo di traduzione (compilazione e assemblaggio) è ripetuto per ciascun modulo
- ❑ I **file oggetto** (object) risultanti devono essere **collegati** (linked) opportunamente tra di loro all'interno di unico file eseguibile, che solo allora può essere caricato in memoria
- ❑ Per ogni modulo tradotto separatamente l'indirizzo iniziale è lo stesso: è compito del linker modificare gli indirizzi di ciascun modulo in modo che non ci siano sovrapposizioni

| Modulo A |       |
|----------|-------|
| 000      | ...   |
| 001      | x     |
| ...      | ...   |
| 150      | Jsr B |
| ...      | ...   |
| 200      | ...   |

| Modulo C |       |
|----------|-------|
| 000      | x     |
| 001      | y     |
| ...      | ...   |
| 250      | Ret B |

| Modulo B |       |
|----------|-------|
| 000      | ...   |
| 001      | y     |
| ...      | ...   |
| 120      | x     |
| ...      | ...   |
| 175      | Jsr C |
| ...      | ...   |
| 300      | Ret A |

| Eseguibile |           |
|------------|-----------|
| 000        | ...       |
| 001        | x         |
| ...        | ...       |
| 150        | Jsr 201   |
| ...        | ...       |
| 200        | ...       |
| 201        | ...       |
| 202        | y         |
| ...        | ...       |
| 321        | Loc (001) |
| ...        | ...       |
| 376        | Jsr 502   |
| ...        | ...       |
| 501        | Ret 151   |
| 502        | Loc (001) |
| 503        | Loc (202) |
| ...        | ...       |
| 752        | Ret 377   |

# PROGRAMMA

## Collegatore (Linker)

- ❑ La traslazione dell'indirizzo di ogni istruzione in ciascun modulo permette di unire tutti i moduli ma non è sufficiente, infatti: è necessario traslare in maniera consistente anche tutti gli indirizzi (assoluti) che compaiono come operandi
- ❑ Per ogni riferimento da parte di un modulo a un indirizzo di un altro modulo è necessario calcolare coerentemente l'**indirizzo esterno** (riferimento esterno o *external reference*). Esempi in questo senso sono le **variabili globali** (che devono essere viste da tutti i moduli) e le chiamate tra procedura appartenenti a moduli diversi
- ❑ Per questo, il linker costruisce una **tabella dei moduli** grazie alla quale è possibile procedere alla rilocazione e al calcolo dei riferimenti esterni a ciascun modulo
- ❑ Infine il linker produce un file eseguibile che di norma ha la stessa struttura di un file oggetto, ma non contiene riferimenti non risolti

| Modulo A |       |
|----------|-------|
| 000      | ...   |
| 001      | x     |
| ...      | ...   |
| 150      | Jsr B |
| ...      | ...   |
| 200      | ...   |

| Modulo C |       |
|----------|-------|
| 000      | x     |
| 001      | y     |
| ...      | ...   |
| 250      | Ret B |

| Modulo B |       |
|----------|-------|
| 000      | ...   |
| 001      | y     |
| ...      | ...   |
| 120      | x     |
| ...      | ...   |
| 175      | Jsr C |
| ...      | ...   |
| 300      | Ret A |

| Eseguibile |           |
|------------|-----------|
| 000        | ...       |
| 001        | x         |
| ...        | ...       |
| 150        | Jsr 201   |
| ...        | ...       |
| 200        | ...       |
| 201        | ...       |
| 202        | y         |
| ...        | ...       |
| 321        | Loc (001) |
| ...        | ...       |
| 376        | Jsr 502   |
| ...        | ...       |
| 501        | Ret 151   |
| 502        | Loc (001) |
| 503        | Loc (202) |
| ...        | ...       |
| 752        | Ret 377   |

# PROGRAMMA

## Caricatore (loader)

- ❑ Una volta che il file eseguibile è memorizzato sul supporto di massa (generalmente il disco magnetico), il **caricatore**, o *loader*, (un programma afferente al sistema operativo) può caricarlo in memoria per l'esecuzione e quindi effettuare:
  - ❑ la lettura dell'intestazione del file eseguibile per determinare la dimensione dei segmenti testo (istruzioni) e dati
  - ❑ la creazione un nuovo spazio di indirizzamento, grande a sufficienza per contenere istruzioni, dati e stack
  - ❑ la copia delle istruzioni e dei dati dal file oggetto al nuovo spazio di indirizzamento
  - ❑ la copia sullo stack degli eventuali argomenti del programma
  - ❑ l'inizializzazione dei registri della CPU
  - ❑ l'inizio dell'esecuzione a partire da una direttiva di inizio che copia gli argomenti del programma dallo stack agli opportuni registri e che chiama la funzione `main()` o la **direttiva di inizio** (BEGIN); fino ad una **direttiva di terminazione** (END)

# PROGRAMMA

Eseguibile

Linguaggio  
assembly  
(MIPS)

```
.text
.globl main
main:
lw $a0,base
lw $a1,espo
jal pow
sw $a2,ris
li $v0,10
syscall
pow:
li $t0,0
li $t1,1
move $t3,$a0
ciclo:
    bge $t0,$a1,fine
    mul $t1,$t1,$t3
    j ciclo

fine:
move $a2,$t1
jr $ra
.data
base: .word 2
espo: .word 3
ris: .word 0
```

**Linguaggio  
macchina  
(MIPS)**  
*Area istruzioni*

```
0x3c011001 00000000000001001001100000101100
0x8c240000 10001100001001000000000000000000
0x3c011001 00111100000000010001000000000001
0x8c250004 10001100001001010000000000000100
0x0c100009 00001100000100000000000000001001
0x3c011001 00111100000000010001000000000001
0xac260008 101011000010011000000000000001000
0x3402000a 00110100000000100000000000001010
0x0000000c 00000000000000000000000000001100
0x34080000 000000100000100000000010100000000
0x34090001 00110100000010010000000000000001
0x00045821 000000000000001000101100000100001
0x0105082a 00000001000001010000100000101010
0x10200004 00010000001000000000000000000100
0x712b4802 01110001001010110100100000000010
0x21080001 00100001000010000000000000000001
0x0810000c 00001000000100000000000000001100
0x00093021 00000000000010010011000000100001
0x03e00008 00000011111000000000000000001000
```

Interprete



# PROGRAMMA

## Interprete

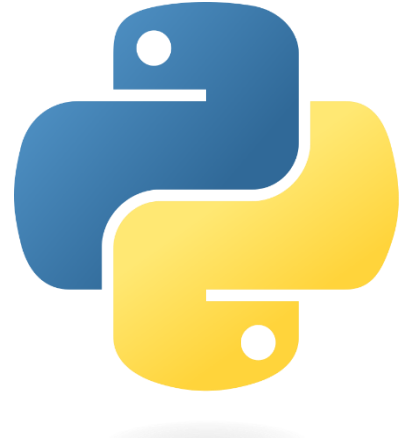
- ❑ Un **interprete** non svolge le operazioni di compilazione e di assemblaggio, ma traduce le istruzioni in linguaggio macchina (memorizzando su file il codice oggetto per essere eseguito dal processore) effettuando solamente una analisi sintattica prima della traslitterazione
- ❑ L'uso di un interprete comporta una minore efficienza durante l'esecuzione del programma (*run-time*); un programma interpretato, in esecuzione, richiede più memoria ed è meno veloce, a causa dell'overhead (maggior numero di operazioni da compiere) introdotto dall'interprete stesso
- ❑ Durante l'esecuzione, l'interprete deve infatti analizzare le istruzioni a partire dal livello sintattico, identificare le azioni da eseguire (eventualmente trasformando i nomi simbolici delle variabili coinvolte nei corrispondenti indirizzi di memoria), ed eseguirle; mentre le istruzioni del codice compilato, già in linguaggio macchina, sono caricate e istantaneamente eseguite dal processore
- ❑ L'uso di un interprete consente all'utente di agire sul programma in esecuzione sospendendolo, ispezionando o modificando i contenuti delle sue variabili, e così via, in modo spesso più flessibile e potente di quanto si possa ottenere, per il codice compilato

# PROGRAMMA

Linguaggi compilati e interpretati



**Pascal**



Fine