



**UNIVERSITÀ  
DEGLI STUDI  
DI TRIESTE**

# Catena programmatica e Assembly

**Prof.ssa Giulia Cisotto**

[giulia.cisotto@units.it](mailto:giulia.cisotto@units.it)

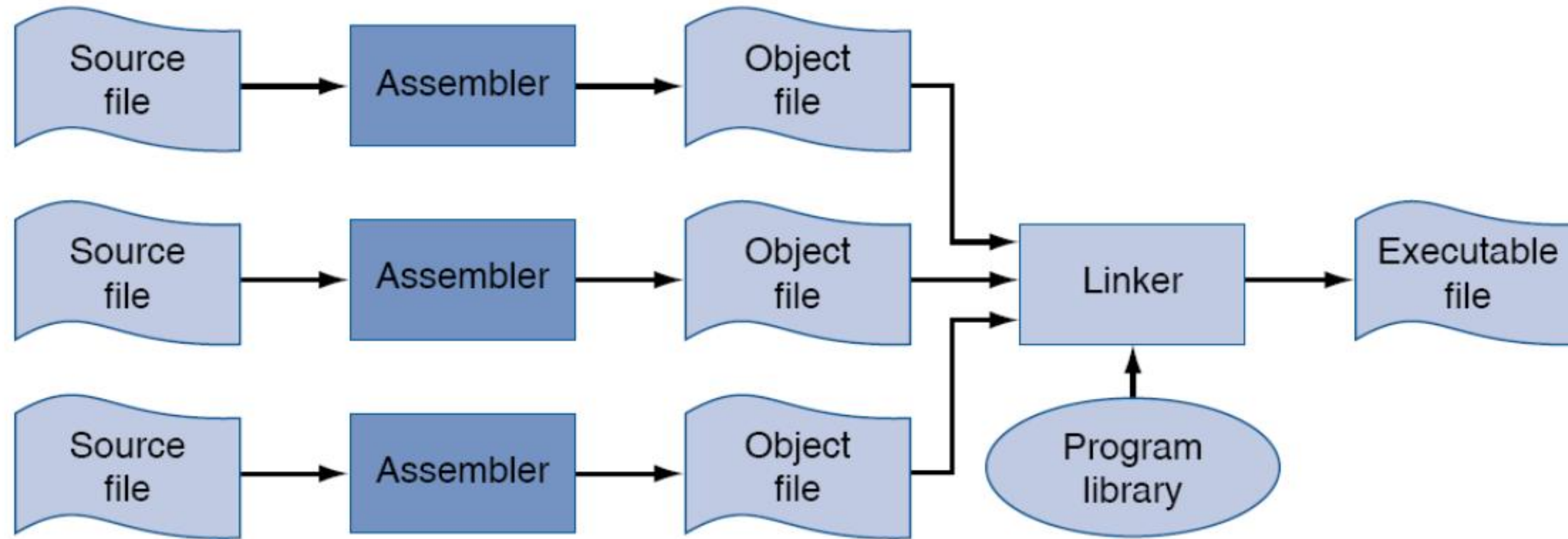
Trieste, 11 marzo 2026

# AGENDA DI OGGI

- *Quiz autovalutazione con peer-review*
- *Struttura di un programma in Assembly (parte 2)*
- **Catena programmatica (parte 2)**
- *Lab#2*

# CATENA PROGRAMMATIVA

## Assembler, Linker



**FIGURE A.1.1** The process that produces an executable file. An assembler translates a file of assembly language into an object file, which is linked with other files and libraries into an executable file.

# ASSEMBLER (1/3)

- Converte un programma assembler (*file sorgente*) in linguaggio macchina (*file oggetto*)
- Gestisce le etichette
- Gestisce pseudo-istruzioni
- Gestisce numeri in base diverse (e.g., binario, decimale, esadecimale)

**L'assemblaggio è un procedimento sequenziale** che esamina, riga per riga, il codice sorgente Assembly, traducendo ciascuna riga in un'istruzione del linguaggio macchina.

Applicato modulo per modulo al programma e costituisce per ogni modulo la **Tabella dei simboli** del modulo.

## ASSEMBLER (2/3)

### 2 operazioni importanti:

1. Traduce i codici simbolici delle istruzioni nei corrispondenti codici binari (sequenze di 32 bit)
2. Traduce i riferimenti simbolici (variabili, registri, etichette di salto, parametri) nei corrispondenti indirizzi numerici.

Poichè le *etichette di salto* generano il problema dei riferimenti in avanti (ossia, riferimenti ad etichette successive o contenute in altri file), **l'assemblatore deve leggere il programma sorgente due volte.**

Ogni lettura del programma sorgente è chiamata **passo** e **l'assemblatore è chiamato *traduttore a due passi*.**

# ASSEMBLER (3/3)

## Tabella dei simboli

Contiene i **referimenti simbolici delle istruzioni** presenti nel modulo da tradurre e, al termine del primo passo, conterrà gli indirizzi numerici di tutti i simboli, tranne quelli esterni al modulo in esame.

Per le **etichette associate a direttive dell'assemblatore** che definiscono costanti simboliche, nella tabella dei simboli viene creata la coppia **< etichetta, valore >** e in ogni istruzione che fa riferimento al simbolo viene sostituito il valore.

Per **etichette che definiscono variabili** (spazio di memoria + eventuale inizializzazione), l'assemblatore riserva spazio, eventualmente inizializza la zona di memoria e crea nella tabella la coppia **<etichetta, indirizzo>**. In ogni istruzione che fa riferimento al simbolo (all'etichetta), il simbolo viene sostituito con l'indirizzo.

Nelle **etichette presenti nelle istruzioni di salto**, l'assemblatore deve generare un riferimento all'indirizzo dell'istruzione destinazione di salto.

**Etichette interne (local)** sono visibili solo all'interno di un modulo.

**Etichette esterne (global)** al modulo possono essere usate da moduli esterni; *rimangono non risolte.*

L'assembler non è disturbato da questo aspetto. **Le etichette esterne saranno risolte dal linker.**

# ASSEMBLAGGIO: ESEMPIO (1/3)

```
.data
array: .space 40      # Alloca spazio per 10 word (40 byte)
size:  .word 10     # Definisce una costante di valore 10

.text
main:
    li $t0, 0          # Inizializza $t0 a 0
loop:
    # Corpo del loop
    beq $t0, $t1, end  # Se $t0 == $t1, salta a 'end'
    j loop             # Salta all'inizio del loop
end:
    # Fine del programma
```

- **array** è un'etichetta che rappresenta l'inizio di un'area di memoria allocata per un array.
- **size** è un'etichetta associata a una parola che contiene il valore costante 10.
- **main**, **loop** ed **end** sono etichette che identificano indirizzi specifici nel segmento di codice.

## ASSEMBLAGGIO: ESEMPIO (2/3)

### Tabella dei Simboli (va nel file oggetto)

Nome	Tipo	Sezione	Valore (indirizzo virtuale)
array	OBJECT	.data	0x10010000 (inizio area dati)
size	OBJECT	.data	0x10010028 (dopo 40 byte)
main	FUNCTION	.text	0x00400000
loop	LABEL	.text	0x00400004
end	LABEL	.text	0x0040000C

- **Nome:** Il nome simbolico definito nel codice assembly (es. etichette, variabili, costanti).
- **Tipo:** La categoria del simbolo, come etichetta (label), variabile o costante.
- **Sezione:** La sezione del programma in cui il simbolo è definito, tipicamente .text per il codice e .data per i dati.
- **Indirizzo/Valore:** L'indirizzo di memoria assegnato al simbolo o il suo valore, nel caso di una costante

# ASSEMBLAGGIO: ESEMPIO (3/3)

Associazione istruzioni → codice macchina (.Text)

INDIRIZZO	ISTRUZIONE ASSEMBLY	ISTRUZIONE BINARIA (SEMPLIFICATA)
0x00400000	li \$t0, 0	→ addiu \$t0, \$zero, 0 (pseudoistruz.)
0x00400004	beq \$t0, \$t1, end	000100 01000 01001 0000 0000 0000 0001
0x00400008	j loop	000010 0000 0000 0000 0000 0000 0001
0x0040000C	end:	(nessuna istruzione)

AL TERMINE DELL'ASSEMBLAGGIO VIENE PRODOTTO IL **FILE OGGETTO** che contiene:

- la *tabella dei simboli* con indirizzi virtuali per etichette e variabili
- *codice macchina* nella sezione .text
- *dati* riservati nella sezione .data

**Nota.** L'assembler espande *prima* di assegnare gli indirizzi, in modo da considerare 8 byte per una pseudoistruzione in .Text

# FILE OGGETTO

Un **file oggetto** (.o oppure .obj) è il risultato della **compilazione o assemblaggio** di un file sorgente (.c, .asm, ecc.) ed è quindi creato dall'assemblatore.

Contiene:

- *Codice macchina* parzialmente compilato
- *Tabelle dei simboli* (etichette, variabili)
- *Dati* riservati nella sezione .data
- *Riferimenti a simboli esterni non ancora risolti* (che verranno risolti dal **linker**)

# FILE OGGETTO: ESEMPIO

## FILE OGGETTO: loop.o

### [Symbol Table]

```
-----  
array    @ 0x10010000 (.data)  
size     @ 0x10010028 (.data)  
main   @ 0x00400000 (.text)  
loop     @ 0x00400004 (.text)  
end      @ 0x0040000C (.text)
```

### [.text Section]

```
-----  
0x00400000: addi $t0, $zero, 0  
0x00400004: beq $t0, $t1, 0x0040000C  
0x00400008: j 0x00400004
```

### [.data Section]

```
-----  
0x10010000–0x10010027: spazio vuoto (40 byte per 'array')  
0x10010028: .word 10 (valore di 'size')
```

$$0x27_{16} = 39_{10}$$

indirizzi inclusivi (cioè dall'indirizzo iniziale fino a quello finale), i byte sono:  $39 + 1 = 40$

# ESEMPIO

```
main:
    li $t0, 0
loop:
    addi $t0, $t0, 1
    beq $t0, $t1, end
    j loop
end:
    li $v0, 10
    syscall
```

Durante la **prima passata**, l'assemblatore:

- Nota che `main` si trova a `0x00400000`
- `loop` a `0x00400004`
- `end` a `0x00400010`

Nella **seconda passata**, quando trova `beq $t0, $t1, end`

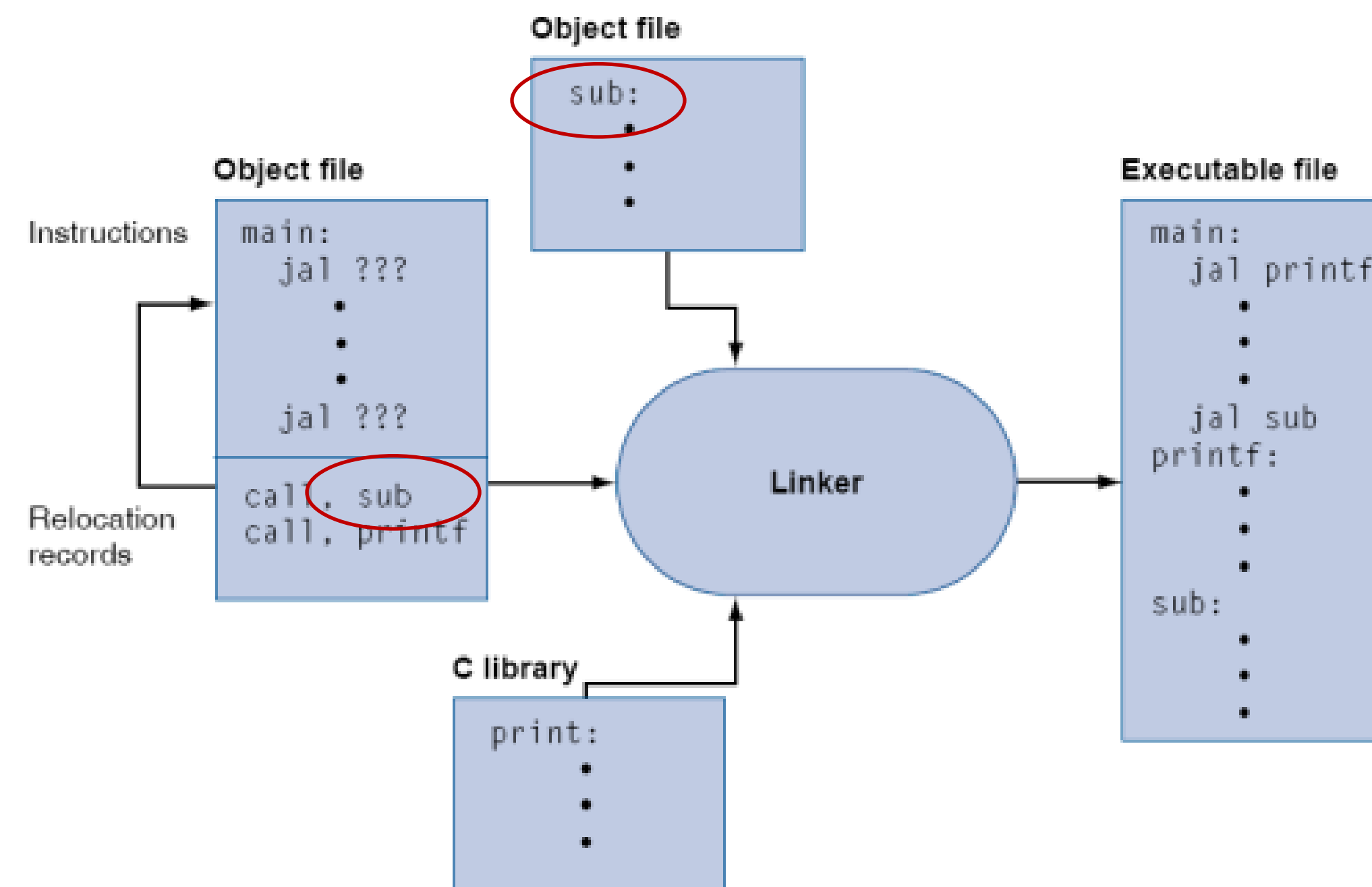
...può finalmente calcolare l'**offset** corretto da inserire nell'istruzione `beq`, perché **ora conosce**

**l'indirizzo di end!**

# LINKER

Il **linker** è uno strumento che prende **più file oggetto** (generati da più file .asm) e:

1. **Unisce** il codice e i dati in un *unico programma eseguibile*.
2. **Risolve i riferimenti tra simboli** (es. se un file usa un'etichetta *func* definita in un altro).
3. **Assegna indirizzi finali** a tutte le etichette, dati e istruzioni, aggiornando la memoria.



**FIGURE A.3.1** The linker searches a collection of object files and program libraries to find nonlocal routines used in a program, combines them into a single executable file, and resolves references between routines in different files.

# LINKER vs ASSEMBLER

<b>Assemblatore (.asm → .o)</b>	<b>Linker (o. → .exe)</b>
Lavora su <b>un file alla volta</b>	Lavora su <b>più file oggetto</b> insieme
Risolve <b>etichette locali</b>	Risolve <b>simboli globali ed esterni</b>
Crea codice <b>parzialmente assemblato</b>	Genera <b>codice finale eseguibile</b>
Crea <b>referimenti simbolici non risolti</b>	Risolve quegli stessi riferimenti

# LOADER

- Lettura dell'intestazione del file eseguibile per **determinare la lunghezza del segmento di testo** (cioè delle istruzioni) e **del segmento dati** (cioè le variabili)
- **Creazione di uno spazio di indirizzamento** sufficiente a contenere testo e dati
- Copia delle istruzioni e dati dal file eseguibile in memoria
- Copia nello stack degli eventuali parametri passati al programma principale
- **Inizializzazione dei registri e impostazione dello stack pointer** affinché punti alla prima locazione libera
- **Impostare il Program Counter (PC)** al punto di inizio (es. etichetta main)
- Salto a una **procedura di startup** la quale copia i parametri nei registri argomento e chiama la procedura principale del programma
- Quando la procedura principale restituisce il controllo, la procedura di startup **termina il programma con una chiamata alla funzione di sistema exit**

# LOADER: ESEMPIO (1/2)

Supponiamo che il loader segua la convenzione MIPS32:

- .text inizia da **0x00400000**
- .data inizia da **0x10010000**
- Stack da **0x7FFFFFFC** verso il basso

```
.data
array: .space 40      # Alloca spazio per 10 word (40 byte)
size:  .word 10      # Definisce una costante di valore 10

.text
main:
    li $t0, 0        # Inizializza $t0 a 0
loop:
    # Corpo del loop
    beq $t0, $t1, end # Se $t0 == $t1, salta a 'end'
    j loop           # Salta all'inizio del loop
end:
    # Fine del programma
```

*Codice in slide #7*

## 1. Caricamento del segmento .data

ETICHETTA	INDIRIZZO	CONTENUTO
array	0x10010000	40 byte (vuoti, per 10 word)
size	0x10010028	0x0000000A (valore 10)

# LOADER: ESEMPIO (2/2)

```
.data
array: .space 40      # Alloca spazio per 10 word (40 byte)
size:  .word 10      # Definisce una costante di valore 10

.text
main:
    li $t0, 0          # Inizializza $t0 a 0
loop:
    # Corpo del loop
    beq $t0, $t1, end # Se $t0 == $t1, salta a 'end'
    j loop             # Salta all'inizio del loop
end:
    # Fine del programma
```

## 2. Caricamento del segmento .text

INDIRIZZO	ISTRUZIONE ASSEMBLY	NOTE
0X00400000	li \$t0, 0	pseudoistruz. → addiu \$t0, \$zero, 0
0X00400004	beq \$t0, \$t1, end	salto condizionato
0X00400008	j loop	salto assoluto
0X0040000C	# end: (nessuna istruz.)	etichetta di arrivo

L'etichetta loop è a 0x00400004, end è a 0x0040000C.

## 3. Loader imposta i registri

REGISTRO	VALORE	MOTIVO
\$pc	0x00400000	Inizio del programma (main)
\$sp	0x7FFFFFFC	Stack pointer iniziale

# Materiale per la lezione

- *Hennessy-Patterson, cap. 1 pp.14-16*
- *Appendix A.2, A.3, A.4, A.5*

*Prossima lezione: 12 marzo, h.14:00, aula 4C*