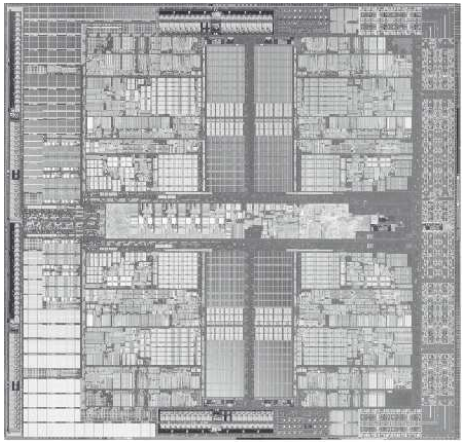
Università degli Studi di Cassino e del Lazio Meridionale



Corso di Calcolatori Elettronici

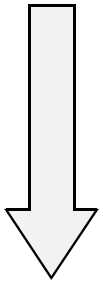
Assemblatori, Linker Loader

Anno Accademico 2011/2012

Francesco Tortorella

Dal produttore all'…esecutore

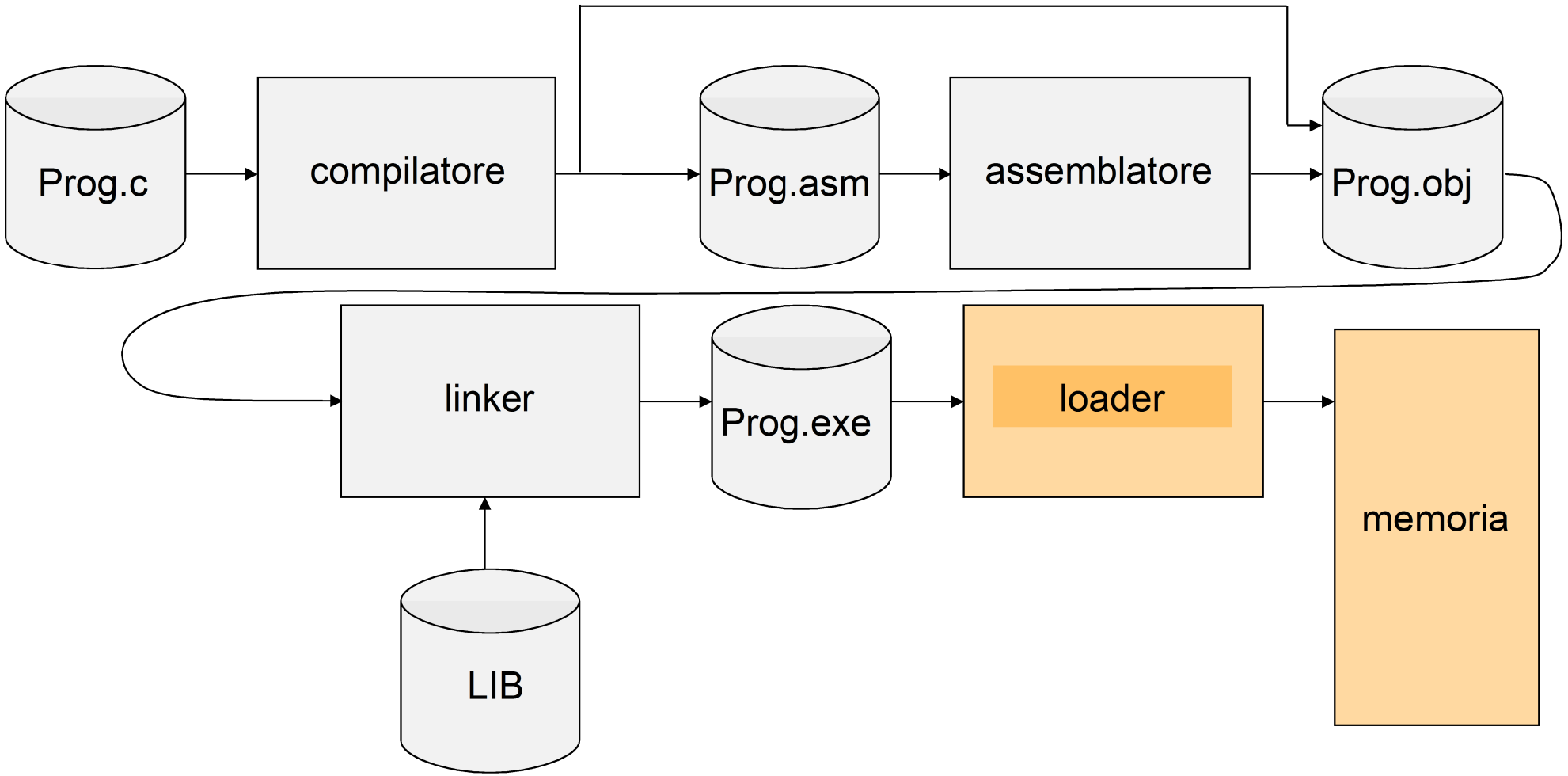
- Prima di essere eseguito, un programma attraversa le seguenti fasi:
  - Traduzione
    - Compilazione
    - Assemblaggio
  - Collegamento
  - Caricamento in memoria
- I prodotti delle varie fasi sono ospitati in files
  - .c
  - .asm
  - .obj
  - .exe



High Level Language

Machine Language

Dal produttore all'…esecutore



Compilatore (Compiler)

- **Input**: codice in HLL
- **Output**:
  - Codice assembly
  - Codice oggetto
- Fasi di analisi
  - Analisi lessicale
  - Analisi sintattica
  - Analisi semantica
- Criteri di Ottimizzazione
  - spaziale
  - temporale

Assemblatore (Assembler)

- **Input**: codice in linguaggio assembly

- **Output**: codice oggetto

- Interpreta ed esegue le direttive
- Sostituisce le pseudoistruzioni
- Produce le istruzioni in linguaggio macchina
- Crea un file oggetto

Tabelle dell'assemblatore

- Per realizzare la traduzione in LM, l'assemblatore fa uso di tabelle.
- **Tabelle statiche**: sono integrate nel codice dell'assemblatore
  - Tabella dei codici operativi: riporta la corrispondenza tra codici operativi simbolici e codici in LM
  - Tabella delle direttive: riporta la corrispondenza tra direttive ed azioni da eseguire
- **Tabelle dinamiche**: sono impiegate per gestire le etichette, vengono costruite dinamicamente durante la traduzione e riportano, per ogni etichetta, l'indirizzo corrispondente e gli indirizzi ed i codici operativi delle istruzioni che usano l'etichetta **(tabella dei simboli)**. I simboli si riferiscono a:
  - Istruzioni → label da utilizzare nelle istruzioni di salto
  - Dati → etichette per accedere ai dati presenti nella sezione `.data`

```
            .data
eol:        .word     10     # cod. ASCII del LINE FEED
string:     .ascii    “paperino\n”

            .text
            la $a0,string    # prepara i parametri
            jal strlen       # salta alla subroutine
            move $a0,$v0     # preleva il valore restituito
            li $v0,1
            syscall          # stampa
            li $v0,10
            syscall          # termina

strlen:     li $v0,0         # Inizializza la lunghezza in $v0
            lw $t0,eol       # Pone in $t0 il terminatore di linea
loop:       lb $t1,($a0)     # Carica un carattere in $t1
            beq $t1,$t0,brk  # Esce se è quello di fine linea
            beqz $t1,brk     # oppure se il byte è zero
            addi $v0,$v0,1   # Conteggia un carattere valido
            addiu $a0,$a0,1  # Incrementa il puntatore alla stringa
            b loop           # Ripete il ciclo
brk:        jr $ra           # Ritorna al chiamante
```

```
[0x00400064] 0x34020000  ori $2, $0, 0
[0x00400068] 0x3c011001  lui $1, 4097
[0x0040006c] 0x8c280038  lw $8, 56($1)
[0x00400070] 0x80890000  lb $9, 0($4)
[0x00400074] 0x11280000  beq $9, $8, 0
[0x00400078] 0x11200000  beq $9, $0, 0
[0x0040007c] 0x20420001  addi $2, $2, 1
[0x00400080] 0x20840001  addi $4, $4, 1
[0x00400084] 0x0401fffb  bgez $0 -20 [loop-0x00400084]
             brk:        jr $31
```

Istruzioni non completate

Istruzione non ancora tradotta

Tabella dei simboli

Simbolo	Valore	Istruzione	
		indirizzo	tipo
strlen	0x400064		
loop	0x400070	0x400084	bgez
brk		0x400074	beq
brk		0x400078	beq

Problema dei riferimenti futuri

- A volte nel codice un'etichetta viene usata prima che sia definita (riferimento futuro).

```
        b label
        ⋮
label lw ...
```

- Soluzione: assemblatore a due passi
  - 1° passo: compilazione della tabella dei simboli e traduzione delle istruzioni definite
  - 2° passo: completamento della traduzione e produzione del LM

Generazione degli indirizzi

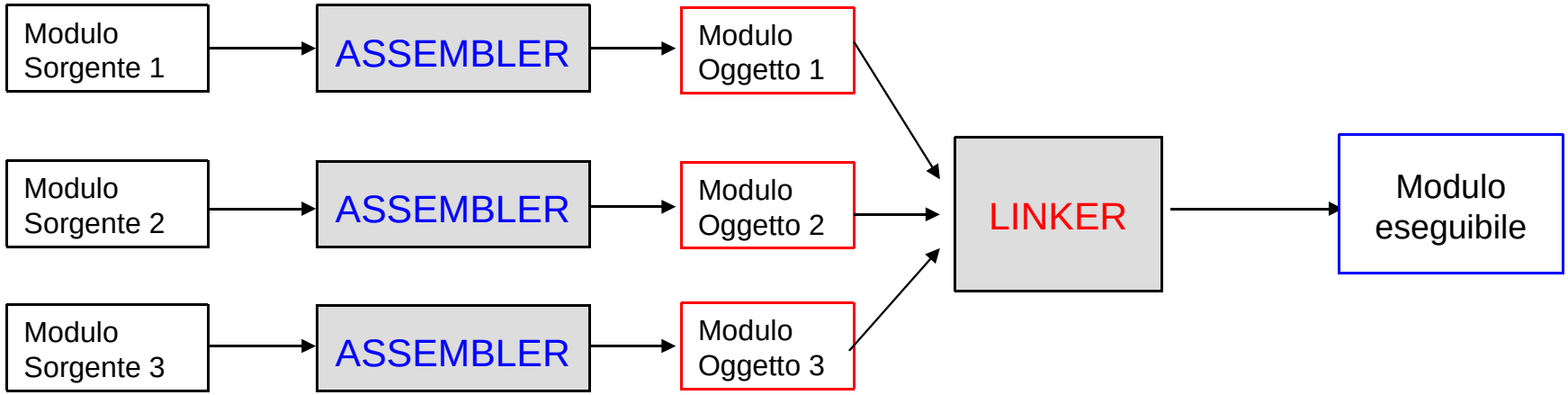
- Per generare gli indirizzi di istruzioni e dati, l'assemblatore utilizza delle variabili interne (location counter) che tengono traccia degli indirizzi utilizzati.
- Ogni segmento ha il proprio location counter.
- Nel segmento dati il LC tiene conto dello spazio allocato dalle varie direttive
- Nel segmento testo, il LC si incrementa ad ogni istruzione prodotta; viene inoltre utilizzato per calcolare gli offset nelle istruzioni di branch e di jump.

Traduzione separata e linking

- Per rendere più agevole la realizzazione di programmi di grandi dimensioni è opportuno dividerli in moduli da tradurre separatamente.
- In questo modo, modifiche locali ad un modulo non costringono ad una traduzione dell'intero programma.
- Conseguenza: all'interno di un modulo ci possono essere riferimenti ad oggetti definiti in altri moduli
  - Chiamate a funzioni
  - Riferimenti a dati
- La traduzione del singolo modulo non può quindi generare direttamente un programma eseguibile. Genera invece un file in formato intermedio (file oggetto) che deve attraversare una successiva fase di collegamento (linking).

Traduzione separata e linking

- Ogni modulo tradotto dall'assemblatore genera un file oggetto.
- I file oggetto sono gestiti da un collegatore (linker) che genera il programma eseguibile



Formato del file oggetto

- Intestazione (header): dimensione e posizione degli altri componenti presenti all'interno del file oggetto.
- Segmento codice: istruzioni in linguaggio macchina
- Segmento dati: rappresentazioni in binario dei dati presenti nel file sorgente
- Tabella dei simboli: contiene i simboli pubblici (es. indirizzi di funzioni definite nel modulo e invocabili da altri moduli) ed i riferimenti esterni (es. etichette di funzioni invocate nel modulo ma definite in altri moduli)
- Informazioni per la rilocazione: identifica istruzioni che devono essere modificate a seguito del collegamento (es. contengono riferimenti esterni)
- Informazioni per il debugging

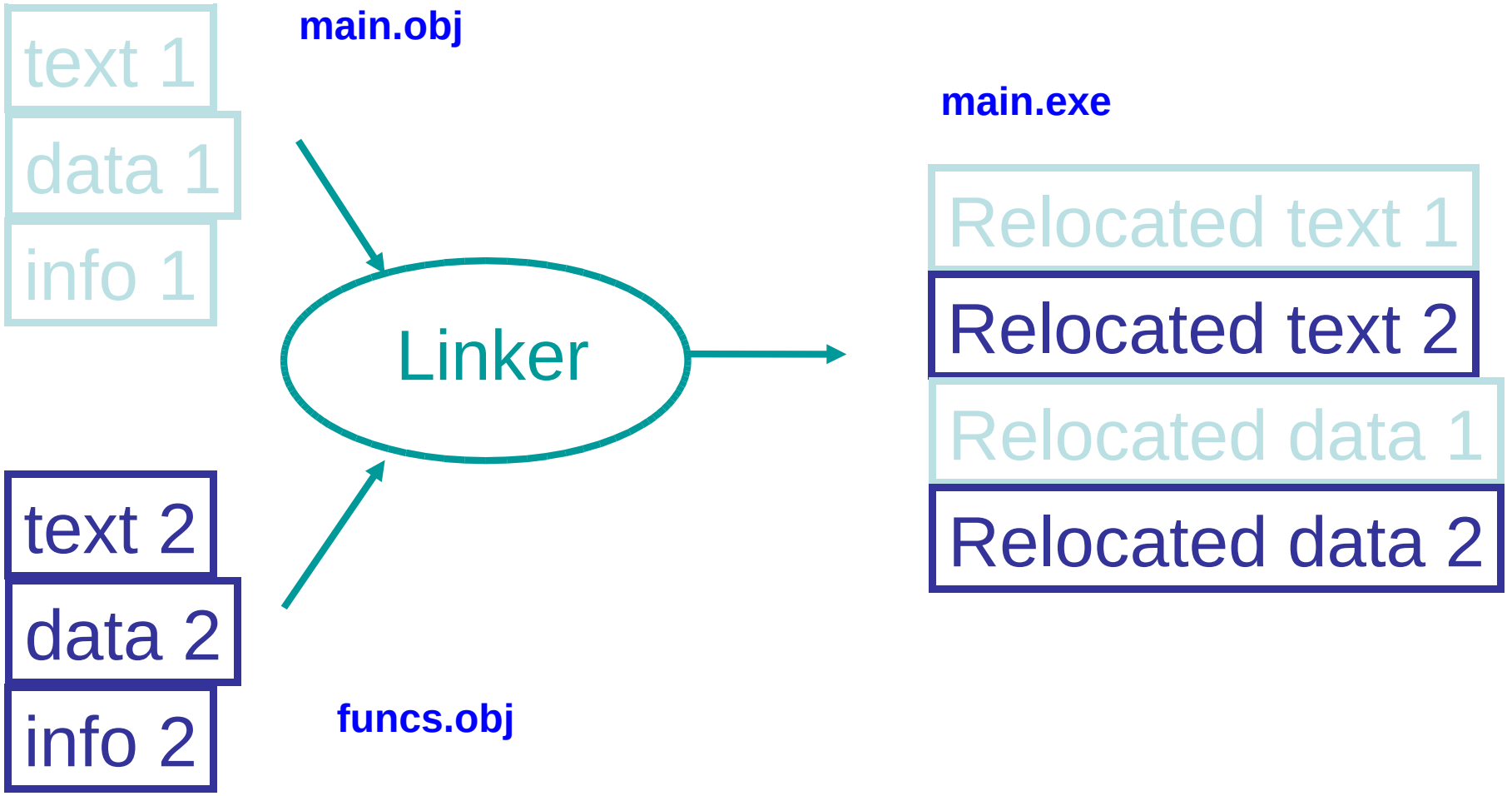
Linker

- **Input:** codice in formato oggetto, tabelle
- **Output:** programma eseguibile

- Combina i file oggetto in un unico programma eseguibile.

- Problemi affrontati:
  - Rilocazione dei singoli moduli
  - Risoluzione dei riferimenti esterni

Linker


main.obj
text 1
data 1
info 1
Linker
text 2
data 2
info 2
funcs.obj
main.exe
Relocated text 1
Relocated text 2
Relocated data 1
Relocated data 2

Operazioni eseguite dal linker

1. Costruisce una tabella contenente la lunghezza dei moduli oggetto (codice e dati)
2. In base alla tabella, assegna un indirizzo di caricamento ad ogni modulo oggetto (codice e dati)
3. Preleva i segmenti codice dai file oggetto e li fonde insieme in accordo agli indirizzi di caricamento calcolati nel passo 2; fa lo stesso con i segmenti dati
4. Aggiorna tutti gli indirizzi rilocabili
5. Aggiorna i riferimenti a procedure esterne

Linker

- Per realizzare i punti 3 e 4, il linker utilizza le informazioni prodotte dall'assemblatore e contenute nei moduli oggetto.
- In particolare, vengono costruite due tabelle generali, che raccolgono i simboli pubblici ed i riferimenti esterni di tutti i moduli.

tabella dei simboli pubblici

Simbolo pubblico	Modulo	Indirizzo

tabella dei riferimenti esterni

Modulo	Indirizzo	Rif. esterno

Linker

- Per risolvere i riferimenti esterni (etichette o dati):
  - Cerca nella tabella dei simboli pubblici
  - Se la ricerca fallisce, cerca nei file di libreria (es. `printf`)
  - Una volta determinato l'indirizzo assoluto, completa l'istruzione in linguaggio macchina
- E' in questa fase che vengono scoperti eventuali errori di collegamento:
  - Unresolved external
  - Already defined symbol

Che cosa rilocare ?

- Indirizzamento PC-Relative (`beq`, `bne`): da non rilocare
- Indirizzamento assoluto (`j`, `jal`): da rilocare
- Riferimento ai dati (lw, sw): da rilocare

Codice eseguibile rilocabile

- In alcuni casi (linking separato dal loading) la rilocazione effettuata non è quella definitiva ed il modulo eseguibile è ancora rilocabile.
- In questo caso, il linker produce una tabella dei riferimenti rilocabili, memorizzata nel modulo eseguibile e utilizzata dal loader.

Header del file oggetto A

	Nome	Procedura A	
	Dim. del testo	100_{16}	
	Dim. dei dati	20_{16}	
Segmento di testo (codice)	**Indirizzo**	**Istruzione**	
	0	lw $a0, **0**($gp)	
	4	jal **0**	
	...	...	
Segmento dati	**0**	**(X)**	
	...	...	
Informazioni di rilocazione	**Indirizzo**	**Tipo istruzione**	**Dipendenza**
	0	lw	X
	4	jal	B
Tabella dei simboli	**Etichetta**	**Indirizzo**	
	X	0 (segm. dati)	
	A	0 (segm. text)	
	B	- (esterno)	

Header del file oggetto B

	Nome	Procedura B	
	Dim. del testo	200_{16}	
	Dim. dei dati	30_{16}	
Segmento di testo (codice)	**Indirizzo**	**Istruzione**	
	0	sw $a1, **0**($gp)	
	4	jal **0**	
	...	...	
Segmento dati	**0**	**(Y)**	
	...	...	
Informazioni di rilocazione	**Indirizzo**	**Tipo istruzione**	**Dipendenza**
	0	lw	Y
	4	jal	A
Tabella dei simboli	**Etichetta**	**Indirizzo**	
	Y	0 (segm. dati)	
	B	0 (segm. text)	
	A	- (esterno)	

File eseguibile

	Dim. del testo	300_{16}
	Dim. dei dati	50_{16}
Segmento di testo (codice)	**Indirizzo$_{16}$**	**Istruzione**
	0040 0000	lw $a0, **8000**($gp)
	0040 0004	jal **40 0100**
	...	...
	0040 0100	sw $a1, **8020**($gp)
	0040 0104	jal **40 0000**
Segmento dati	1000 0000	**(X)**
	...	...
	1000 0020	**(Y)**
	...	...

Le librerie

- Nella fase di collegamento è particolarmente importante la gestione delle “librerie”.
- La libreria (maldestra traduzione dell’inglese *library*) è un insieme di funzioni già implementate che possono essere immediatamente collegate al resto del programma.
- Alcune librerie sono tipicamente disponibili con i compilatori (es. libreria matematica).

Librerie statiche e dinamiche

- Sono possibili due tecniche diverse per il collegamento della libreria
- Libreria caricata staticamente (“statically-linked” o libreria statica): la libreria diventa parte dell’eseguibile.
  - In caso di aggiornamenti della libreria, è necessario rifare il linking per ottenere l’eseguibile aggiornato
  - Viene inclusa l’intera libreria, anche se non tutte le funzioni saranno utilizzate
  - Il file eseguibile è autoconsistente (non ha bisogno di altri file per essere eseguito)
- L’alternativa è costituita dalle **librerie caricate dinamicamente (Dynamically Linked Libraries, DLL)**, diffuse sui sistemi Windows e UNIX.

Librerie caricate dinamicamente

- Una libreria DLL non è collegata staticamente, ma viene caricata dinamicamente durante l'esecuzione del programma.
- Questo aggiunge un po' di complessità al compilatore, al linker e al sistema operativo, ma permette di ottenere notevoli vantaggi

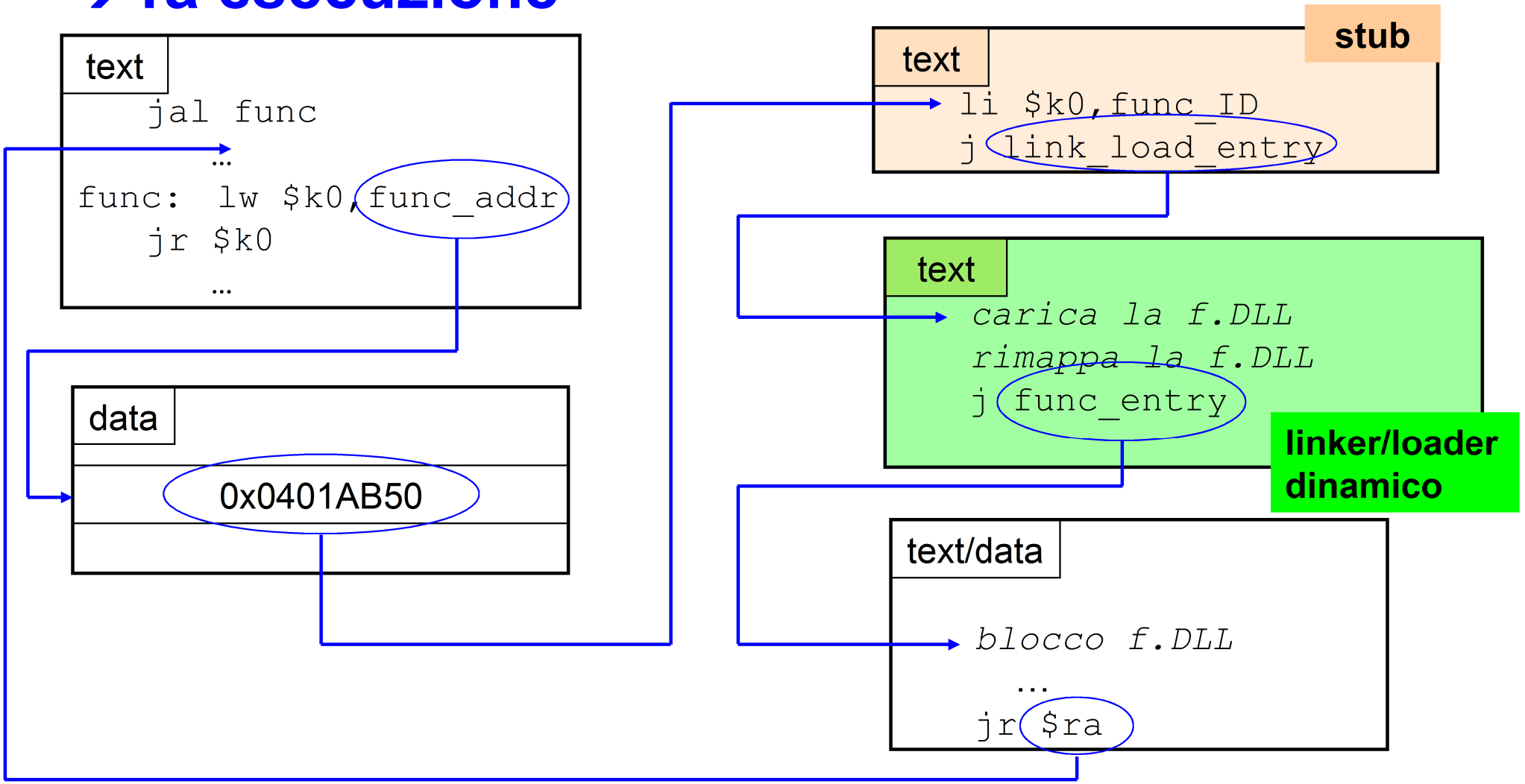
Librerie caricate dinamicamente: vantaggi e svantaggi

- Efficienza
  - ☺Il programma eseguibile occupa meno spazio
  - ☺ L'esecuzione di più programmi che usano la stessa DLL richiede meno memoria
  - ☹ A tempo di esecuzione inevitabile un overhead per eseguire il collegamento
- Aggiornamento
  - ☺ L'aggiornamento di una DLL aggiorna automaticamente tutti i programmi che usano quella DLL
  - ☹ Il file "eseguibile" del programma non è autonomo

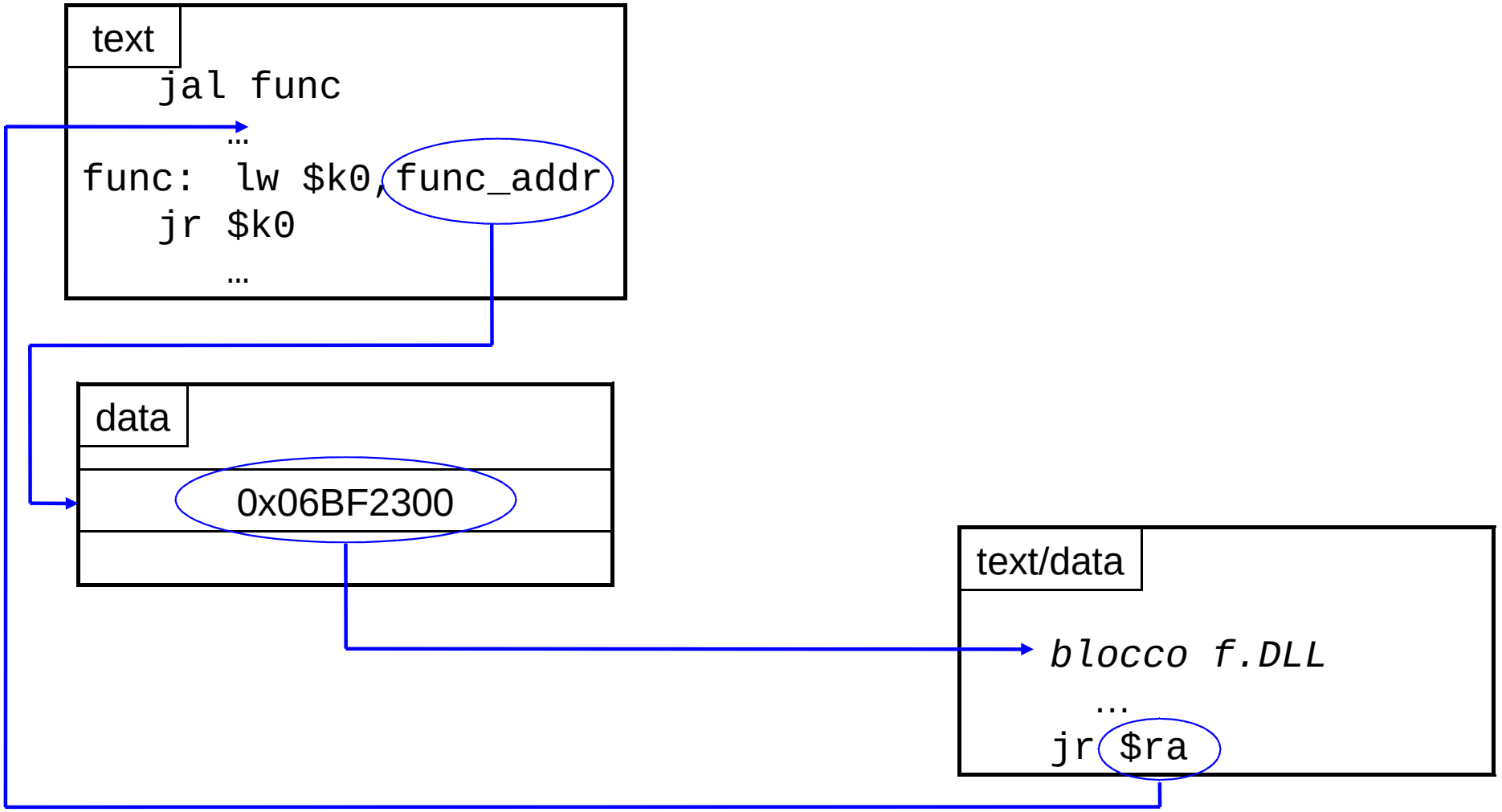
DLL: meccanismo di caricamento

- Nelle versioni iniziali delle DLL, il loader (programma caricatore) lanciava un linker dinamico per cercare le librerie adeguate e per aggiornare i riferimenti esterni.
- Questo meccanismo aveva però l'inconveniente di caricare tutta le funzioni della libreria, anche quelle che non sarebbero state usate.
- Alternativa: tecnica di caricamento **lazy** ("pigra"). Ogni procedura è caricata solo dopo la sua chiamata.

Meccanismo di caricamento delle DLL
→1a esecuzione



Meccanismo di caricamento delle DLL
→ esecuzioni successive



Loader

- **Input:** programma eseguibile (`pippo.exe`)
- **Output:** (esecuzione del programma)

- I programmi eseguibili sono memorizzati come file sui dischi.
- Quando si richiede l'esecuzione di un programma, il loader preleva il file dal disco, ne carica il contenuto in memoria e ne inizia l'esecuzione.
- **ACHTUNG**: il loader fa parte del Sistema Operativo.

Loader: fase di caricamento

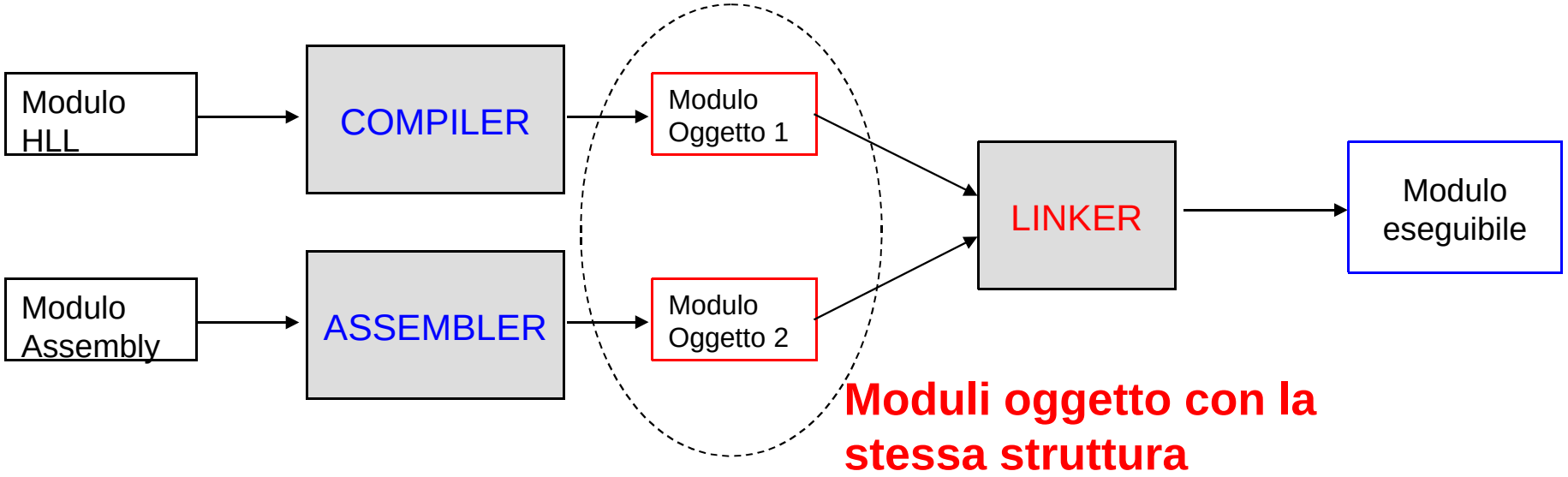
- All'atto del caricamento, il loader legge l'header del file eseguibile per determinare la dimensione dei segmenti codice e dati.
- Alloca un nuovo spazio in memoria sufficientemente ampio per contenere i segmenti codice, dati e stack.
- Copia istruzioni (se necessario operando una rilocazione) e dati dal file nei registri appartenenti allo spazio di memoria allocato

Loader: fase di attivazione

- Terminata il trasferimento in memoria, il loader copia eventuali argomenti passati al programma sullo stack (di supervisore).
- Inizializza i registri del processore
  - Azzera i registri generali
  - Assegna allo stack pointer l'indirizzo iniziale del segmento stack
- Salta alla routine di start-up che copia gli argomenti passati al programma nei registri e inizializza il Program Counter

Aggancio moduli HLL-moduli assembly

- Con lo schema visto, è possibile collegare oggetti provenienti sia da sorgenti assembly che da sorgenti HLL.



Aggancio moduli HLL-moduli assembly: condizioni necessarie

- Moduli oggetto con la stessa struttura
- Stesse convenzioni per la rappresentazione dei dati
- Stesse convenzioni per il subroutine linkage

Aggancio moduli HLL-moduli assembly: cose da sapere

- **Rappresentazione dei dati**
  - Come sono rappresentati i vari tipi in HLL ?
- **Subroutine linkage**
  - Relazione nome procedura → simbolo esterno
  - Come sono salvati i parametri ?
  - In che ordine sono salvati i parametri ?
  - Come viene gestito il valore restituito dalle funzioni ?