Costruzione di un file eseguibile

Argomenti:

- Creazione di un programma eseguibile
- Assembler
- Linker
- Loader

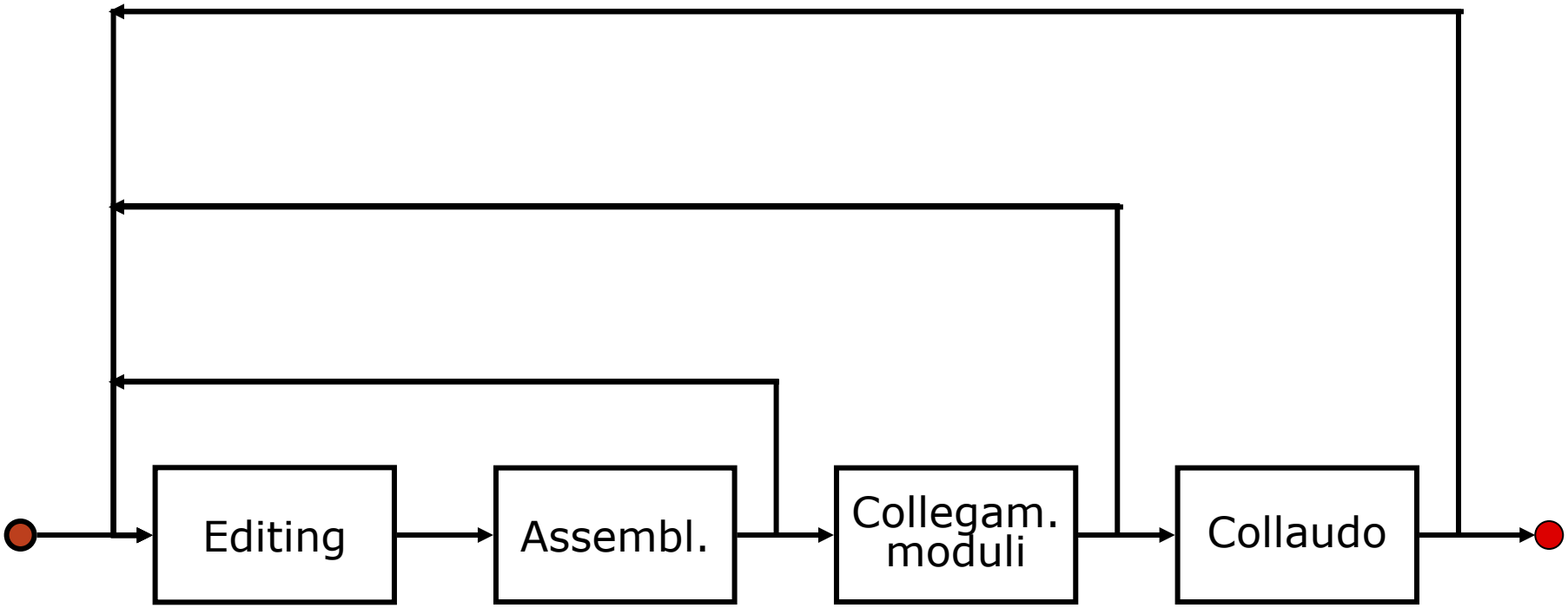
Materiale didattico:

- Appendice B

Processo di produzione del software

- **Analisi** del problema
- **Progettazione** del programma
  - Definizione dei moduli software
  - Progettazione singoli moduli
  - Documentazione
- **Scrittura** del programma
  - Scrittura dei moduli
  - Traduzione e collaudo di ciascun modulo
  - Costruzione dell'intero programma
  - Collaudo del programma

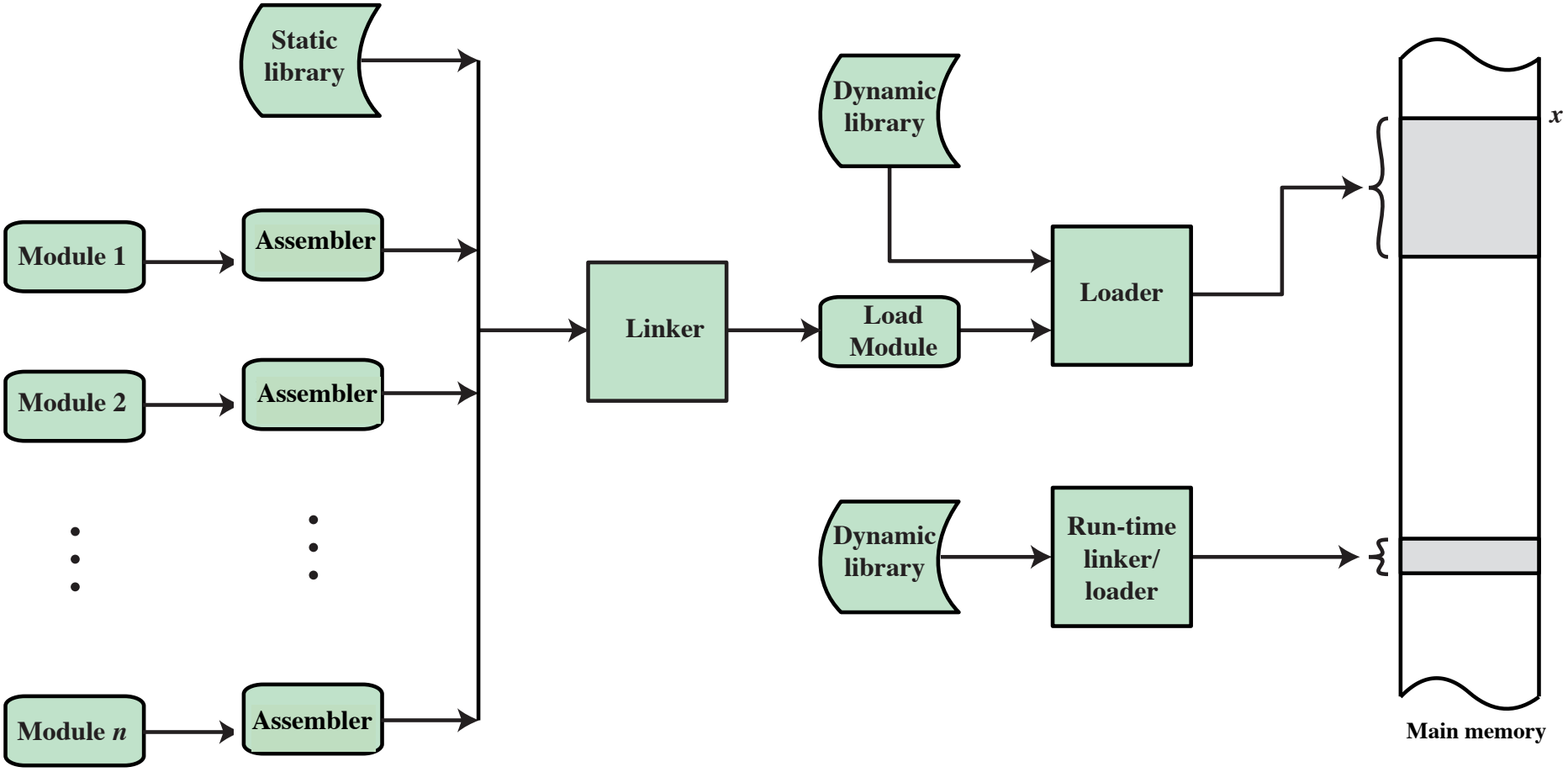
Scrittura di un programma



Sistema di sviluppo in linguaggio assembly

- Text Editor
  - Programma per la scrittura e la modifica del testo sorgente
- Assemblatore
  - Programma per traduce il testo sorgente in modulo oggetto
- Linker
  - Programma per costruire il programma eseguibile
- Loader
  - Programma per caricare in memoria il programma eseguibile
- Debugger
  - Esegue il programma sotto controllo del programmatore

Sistema di sviluppo in linguaggio assembly



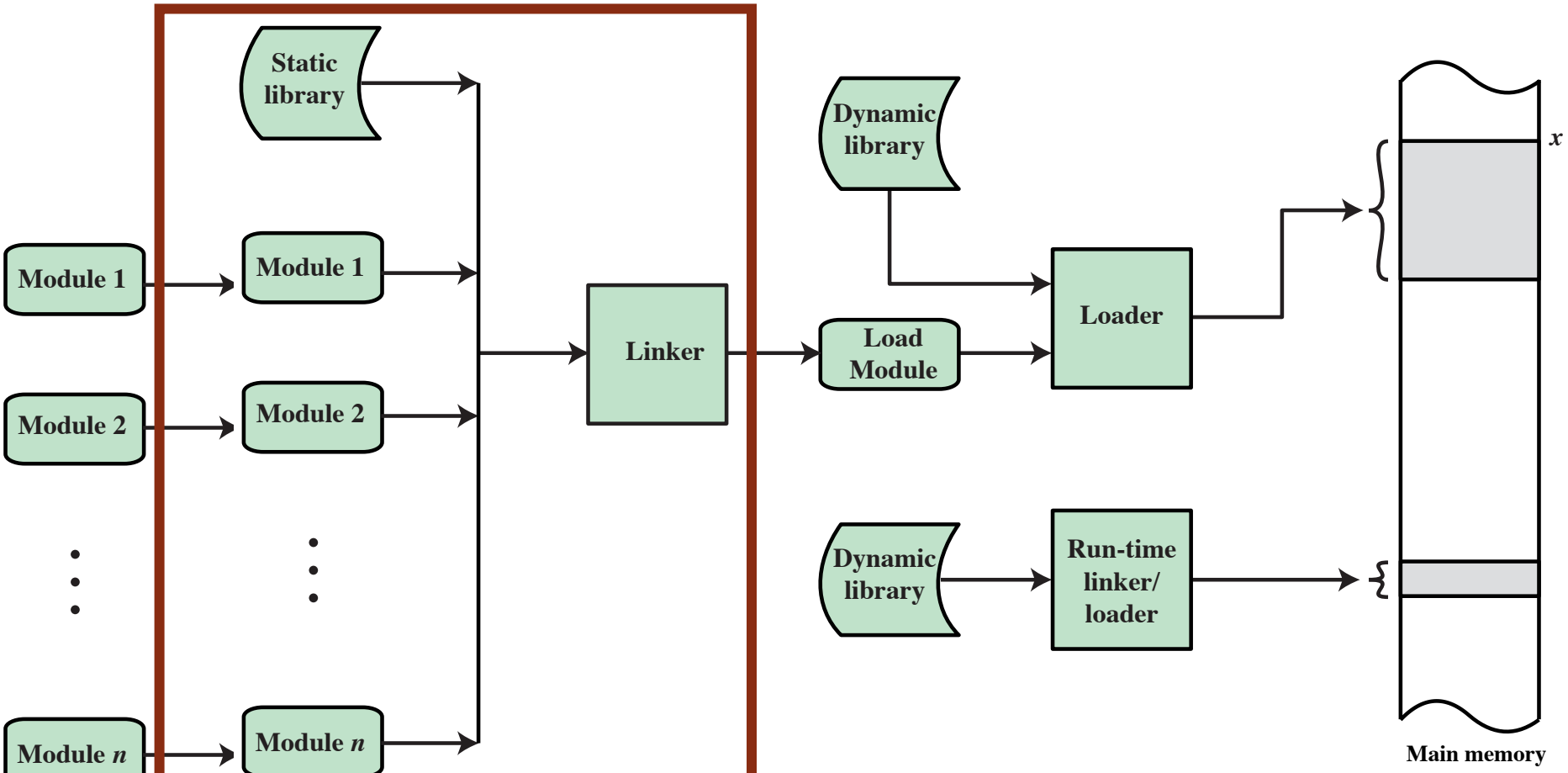
Sistema di sviluppo in ARM

- Text Editor
  - vi, textmate, notepad++, …
- Assemblatore
  - as
- Linker
  - ld

integrati nel commando gcc

- Loader
  - Avviato dal sistema operativo quando si esegue un programma (ad esempio con .nome_programma o con qemu-arm)
- Debugger
  - gdb (o interfacce grafiche tipo gdbgui)

Sistema di sviluppo in ARM (arm-gcc)



Due passi:
1 – Assembling dei file sorgente
2 – Linking dei moduli oggetto

Sistema di sviluppo in ARM

- Text Editor
  - vi, textmate, notepad++, …
- Assemblatore
  - as
- Linker
  - ld
- Loader
  - Avviato dal sistema operativo quando si esegue un programma (ad esempio con .nome_programma o con qemu-arm)
- Debugger
  - gdb

gdbgui:
- Interfaccia grafica per gdb

Moduli sorgenti

- Il codice di un programma consiste di uno o più moduli sorgente.

- Un **modulo sorgente** è costituito da istruzioni di due tipi:
  - Istruzioni macchina,
  - Istruzioni per l'assemblatore (direttive, pseudo-istruzioni).

Simboli

I **simboli** sono stringhe alfanumeriche con un significato particolare definito dal linguaggio assembly o dal programmatore

- Simboli che rappresentano opcode (ADD, OR,…), registri (R0, R1, …), tecniche di indirizzamento (#, []), …
- Simboli che rappresentano indirizzi di memoria (tramite label)
- Simboli che rappresentano valori numerici generici (tramite direttiva .equ)

Simboli: classificazione

- Simboli predefiniti: codici operativi, identificatori di registri di CPU, …
- Simboli definiti dal programmatore/compilatore (indirizzi, costanti numeriche):
  - **locali** (visibili solo nel modulo corrente):
  - **globali** (visibili anche in altri moduli):
  - **esterni** (definiti in altri moduli sorgente).
- Un simbolo può avere valore
  - **assoluto**: il suo valore non cambia (e.g., direttiva .EQU)
  - **da rilocare**: il loro valore dipende dalla posizione del codice in memoria (e.g., tutte le label)

Forward reference

Forward reference: quando un simbolo viene utilizzato prima della sua definizione

- In molti casi, la forward reference è necessaria

```
            . . .
            B   AVANTI
            . . .
            . . .
AVANTI:     . . .
            . . .
```

Simboli esterni

In molti casi è necessario usare simboli definiti in altri moduli sorgenti

- Ad esempio: chiamate a subroutine contenute in librerie
- Permette la scrittura di codice modulare

- **Simbolo esterno**: un simbolo usato in un file, ma definito in un altro file
- **Simbolo globale**: un simbolo definito in un file che può essere usato in altri file
- **Simbolo locale**: un simbolo definito in un file che non può essere usato in altri file

Esempio: somma (file main.s)

```
/*main.s*************************************************/
/* somma di due numeri                                   */
/* addendi in memoria, risultato in memoria              */
/*************************************************/
.text
.global main
main: push {r0-r2,lr}
      ldr   r2, =in1
      ldr   r0, [r8]
      ldr   r2, =in2
      ldr   r1, [r8]
      ldr   r2, =out
      bl addf
      pop {r0-r2, lr}
```

Main viene definito come simbolo globale

Definizione di main

Simbolo esterno: addf non è definito in main.s

Esempio: somma (file main.s) (2)

```
end_main: mov pc, lr
```

Definizione di un simbolo locale, non globale

```
.data
in1:  .word 0x00000012
in2:  .word 0x00000034

.bss
out: .space 4
     .space 256
```

Esempio: somma (file addf.s)

```
/*addf.s*************************************************/
/* somma di due numeri                                   */
/* subroutine                                            */
/*************************************************/
        .text
        .global  addf
addf:
        push {r0}
        add   r0,  r0, r1      @ esegue la somma
        str   r0,  [r2]        @ memorizza il risultato
        pop {r0}
        mov pc, lr
```

Label addf viene definita come globale

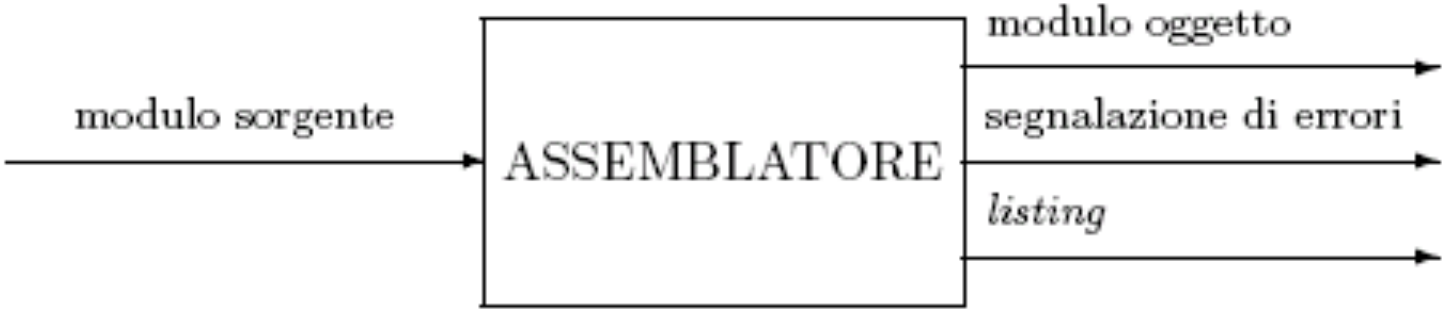
Definizione della label addf

Il programma assemblatore

- L'assemblatore:
  - Traduce istruzioni assembly in istruzioni macchina
  - Esegue le istruzioni per l'assemblatore (ad esempio .word, .skip)
  - Sostituisce simboli (label, costanti) con il loro valore se definite nel file, o restano pendenti se simboli esterni.

- Viene generato un **modulo oggetto** per ogni modulo sorgente
  - file1.s → file1.o
  - file2.s → file2.o
  - …

Il programma assemblatore



- Oltre ai moduli oggetto, l'assemblatore segnala eventuali errori e file di listing
- Il file di listing mostra come le istruzioni assembly sono state tradotte in istruzioni macchina

Funzionamento di un assemblatore

- L'assemblatore effettua **due scansioni** del file di input:
  1. La prima scansione cerca tutte le definizione di simboli
  2. La seconda scansione converte le istruzioni e risolve tutti i simboli
- Due scansioni sono necessarie per gestire le forward references
- L'assemblatore mantiene il **contatore LC** (location counter) che indica a che indirizzo verrà salvata la prossima istruzione
  - Il valore iniziale di LC è 0
  - LC viene opportunamente incrementato ad ogni istruzione letta

Prima scansione

Nella prima scansione viene costruita la **tabella dei simboli** che contiene per ogni simbolo definito nel sorgente le seguenti informazioni:

- Il nome del simbolo
- Il suo valore
- Locale/globale

Operazioni della prima scansione:

1. Leggi la prossima riga del sorgente
2. Se contiene la definizione di un simbolo X: inserisci il nome del simbolo, il suo valore, e se è globale/locale
3. Incrementa LC del numero di byte necessari per codificare la riga letta
4. Ritorna al passo 1

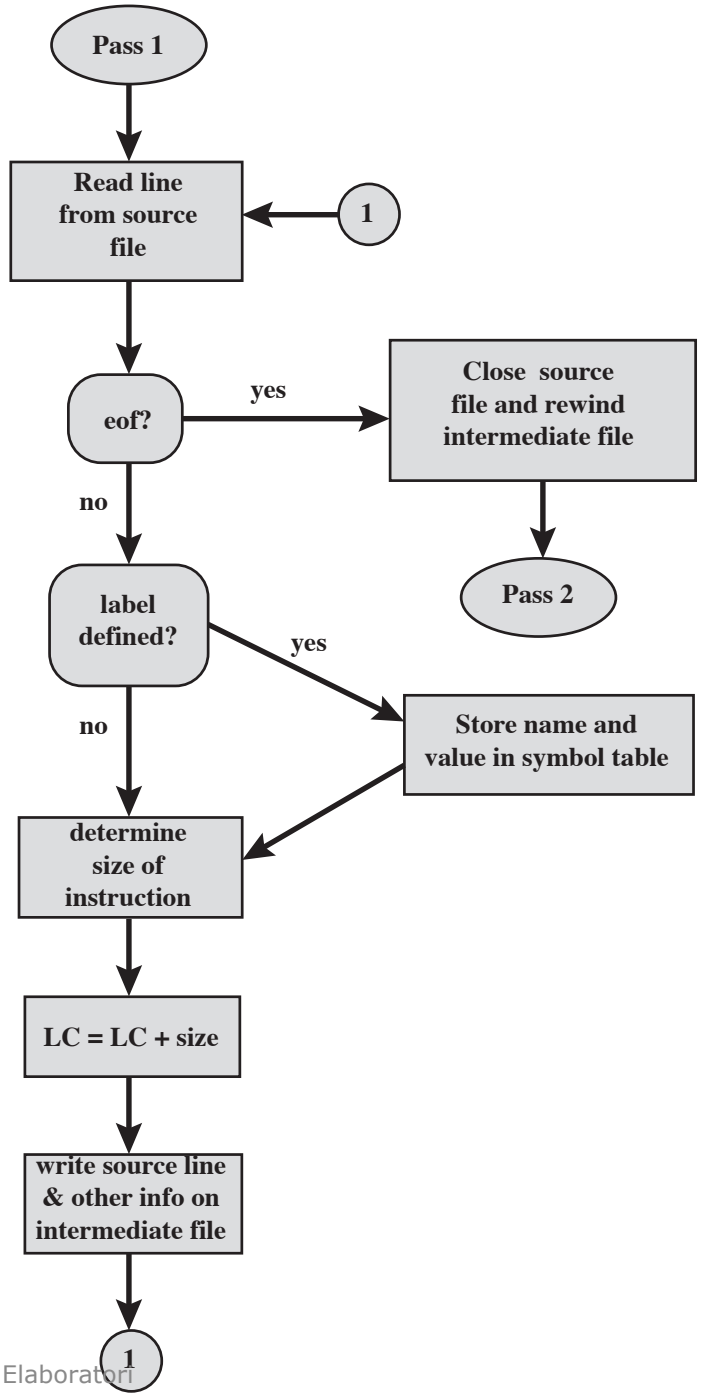
Seconda scansione

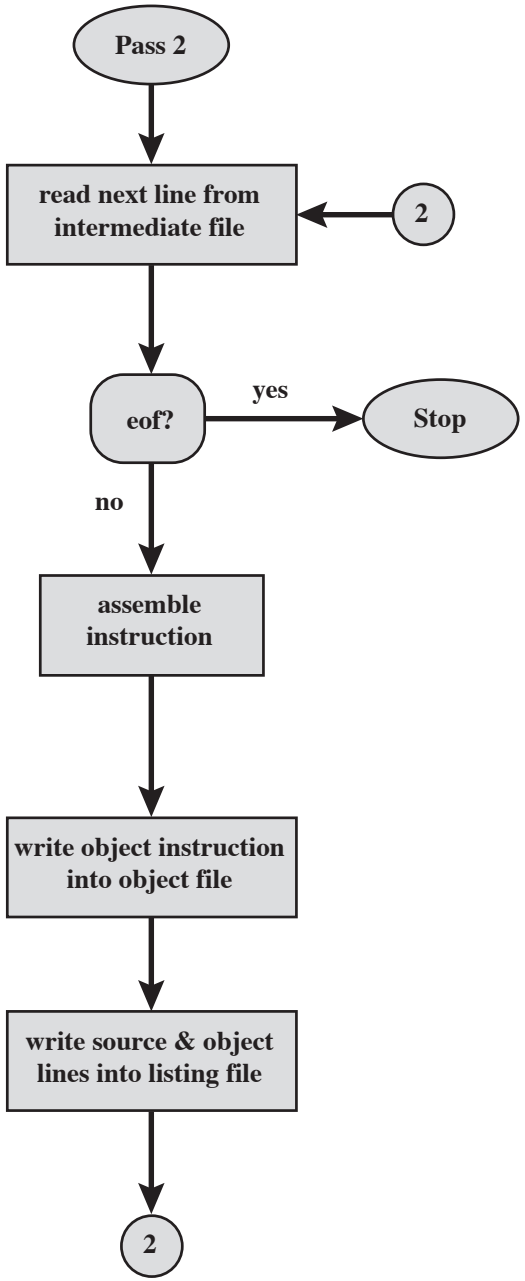
- Ogni simbolo presente nella tabella dei simboli viene sostituito con il suo valore:
  - Per ogni simbolo, si registra l'indirizzo dove è stato usato (ad esempio, nella tabella dei simboli)
- Ogni istruzione assembly viene codificata in linguaggio macchina
- Viene allocato lo spazio per i dati
- Viene mantenuto il contatore LC

Seconda scansione (2)

Operazioni della seconda scansione:

1. Leggi la prossima riga del sorgente
2. Sostituisci ogni simbolo presente nella tabella dei simboli con il suo valore; registra l'utilizzo dell'indirizzo
3. Codifica l'istruzione
4. Incrementa LC del numero di byte usati per codificare l'istruzione
5. Ritorna al passo 1


Pass 1
Read line from source file
1
eof?
yes
Close source file and rewind intermediate file
Pass 2
no
label defined?
yes
Store name and value in symbol table
no
determine size of instruction
LC = LC + size
write source line & other info on intermediate file
1


Pass 2
read next line from intermediate file
2
eof?
yes
Stop
no
assemble instruction
write object instruction into object file
write source & object lines into listing file
2

Funzionamento di un assemblatore

Codice di esempio con un unico segmento

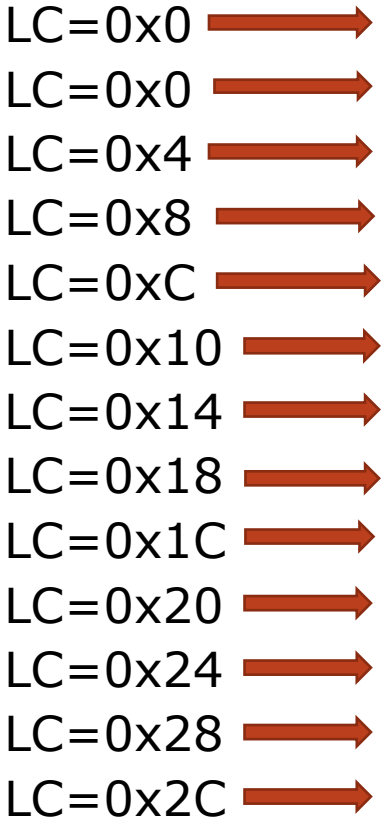
```
.global  main
main: push {r0-r2,lr}
ldr   r2, =in1
ldr   r0, [r8]
ldr   r2, =in2
ldr   r1, [r8]
ldr   r2, =out
bl addf
pop {r0-r2, lr}
end_main: mov pc, lr
in1: .word    0x00000012
in2: .word    0x00000034
out:  .space 4
```

Prima scansione: esempio

```
LC=0x0  ⟶  .global  main
LC=0x0  ⟶  main: push {r0-r2,lr}
LC=0x4  ⟶  ldr   r2, =in1
LC=0x8  ⟶  ldr   r0, [r8]
LC=0xC  ⟶  ldr   r2, =in2
LC=0x10 ⟶  ldr   r1, [r8]
LC=0x14 ⟶  ldr   r2, =out
LC=0x18 ⟶  bl addf
LC=0x1C ⟶  pop {r0-r2, lr}
LC=0x20 ⟶  end_main: mov pc, lr
LC=0x24 ⟶  in1: .word 0x00000012
LC=0x28 ⟶  in2: .word  0x00000034
LC=0x2C ⟶  out: .space 4
```

Simbolo	Valore	Locale/globale
main		G
end_main	0x20	L
in1	0x24	L
in2	0x28	L
out	0x2C	L

Prima scansione: esempio



```
LC=0x0  .global  main
LC=0x0  main: push {r0-r2,lr}
LC=0x4  ldr   r2, =in1
LC=0x8  ldr   r0, [r8]
LC=0xC  ldr   r2, =in2
LC=0x10 ldr   r1, [r8]
LC=0x14 ldr   r2, =out
LC=0x18 bl addf
LC=0x1C pop {r0-r2, lr}
LC=0x20 end_main: mov pc, lr
LC=0x24 in1: .word 0x00000012
LC=0x28 in2: .word 0x00000034
LC=0x2C out:  .space 4
```

INDIRIZZO	CODIFICA
0x00	e92d4007
0x04	e59f2024
0x08	e5980000
0x0C	e59f2020
0x10	e5981000
0x14	e59f201c
0x18	ebfffffe
0x1C	e8bd4007
0x20	e1a0f00e
0x24	00000012
0x28	00000034
0x2C	00000000
0x30	00000024
0x34	00000028
0x38	0000002c

Label non definite

- Alcuni simboli possono essere definiti in altri sorgenti
- Nella seconda scansione, l'assemblatore crea una **tabella dei simboli esterni** in cui elenca i simboli mancanti e gli indirizzi nel modulo oggetto in cui il valore mancante deve essere inserito.
- Il linker provvederà nella fase successiva ad inserire i simboli mancanti.

Seconda scansione: aggiornamento tabella dei simboli

Simboli definiti

Simbolo	Valore	Locale/globale	Indirizzi uso
main	0x0	G	-
end_main	0x20	L	-
in1	0x24	L	0x30
in2	0x28	L	0x34
out	0x2C	L	0x38

Simboli non definiti

Simbolo esterno	Lista di indirizzi in cui il simbolo è usato
addf	0x28

Assemblaggio di segmenti e file multipli

- L'assemblatore lavora **indipendentemente** su ogni file

- Per ogni file:
  - Si effettua la prima scansione di ogni segmento: ogni segmento è costruito in un proprio spazio di indirizzamento (LC parte da 0)
  - Si costruisce una tabella unica per tutti i simboli nei vari segmenti: simboli dichiarati in segmenti diversi potrebbero avere lo stesso valore (vedi esempio seguente di main.s)
  - Si effettua la seconda scansione di ogni simbolo
  - Viene generato un solo file oggetto.

Cosa contiene il modulo oggetto?

Il modulo oggetto contiene:

- La traduzione di ogni segmento
- La tabella dei simboli locali e globali
- La tabella dei simboli esterni (non definiti) con la lista degli indirizzi nel modulo oggetto da aggiornare

Assemblaggio di segmenti e file multipli

FILE main.s

```
.data
in1:  .word 0x00000012
in2:  .word 0x00000034

.bss
out: .space 4
     .space 256

.text
.global  main
main:    push {r0-r2, lr}
         ldr r2, =in1
         ldr r0, [r8]
         ldr r2, =in2
         ldr r1, [r8]
         ldr r2, =out
         bl addf
         pop {r0-r2, lr}
end_main:mov pc, lr
```

FILE addf.s

```
.text
.global  addf
addf:    push {r0}
         add r0, r0, r1
         str r0, [r2]
         pop {r0}
         mov pc, lr
```

Esempio

Vediamo un esempio di assemblaggio di 2 file: main.s e addf.s

- `as –o main.o main.s`
- `as –o addf.o addf.s`
- Aggiungendo i parametri '–gstabs –al', l'assemblatore produce anche il listato

Per analizzare i simboli:

- `nm main.o`
- `nm addf.o`

Assemblaggio .text in main.s

```
.text
.global  main
main: push {r0-r2, lr}
      ldr r2, =in1
      ldr r0, [r8]
      ldr r2, =in2
      ldr r1, [r8]
      ldr r2, =out
      bl addf
      pop {r0-r2, lr}
end_main:mov pc, lr
```

```
Disassembly of section .text:

00000000 <main>:
 0x0: e92d4007 push {r0, r1, r2, lr}
 0x4: e59f2018 ldr r2, [pc, #0x1C] ; 24 <main_end+0x4>
 0x8: e5980000 ldr r0, [r8]
 0xc: e59f2014 ldr r2, [pc, #0x18] ; 28 <main_end+0x8>
0x10: e5981000 ldr r1, [r8]
0x14: e59f2010 ldr r2, [pc, #0x14] ; 2c <main_end+0xc>
0x18: ebfffffe bl 0 <addf>
0x1c: e8bd4007 pop {r0, r1, r2, lr}

00000020 <end_main>:
0x20: e1a0f00e mov pc, lr
0x24: 00000000
0x28: 00000004
0x2c: 00000000
```

Assemblaggio .data in main.s

```
.data
in1:   .word     0x00000012
in2:   .word     0x00000034
```

```
00000000 <in1>:
   0: 00000012 …


00000004 <in2>:
   4: 00000034 …
```

Esempio main.o, segmento .bss

```
.bss
out:    .space   4
```

```
00000000 <out>: ...
```

Tabelle dei simboli (main.o/text)

Simbolo	Valore	Locale/globale	Indirizzi uso
main	0	G/text	-
end_main	0x20	l/text	-
in1	0x0	l/data	0x20/text
in2	0x4	l/data	0x34/text
out	0x0	l/bss	0x38/text

Simbolo esterno	Lista di indirizzi in cui il simbolo è usato
addf	0x18/text

Esempio addf.o, segmento .text

```
.text
.global  addf
addf:
   push {r0}
   add   r0,  r0, r1
   str   r0,  [r2]
   pop {r0}
   mov pc, lr
```

```
00000000 <addf>:
   0: e52d0004 push {r0}
   4: e0800001 add r0, r0, r1
   8: e5820000 str r0, [r2]
   c: e49d0004 pop {r0}
  10: e1a0f00e mov pc, lr
```

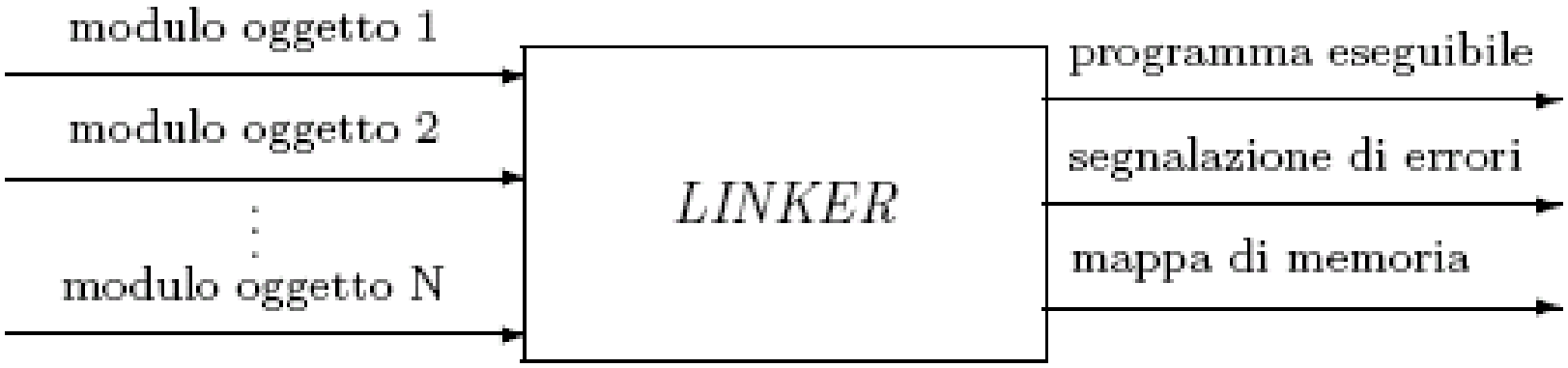
Tabelle dei simboli (addf.o/text)

Simbolo	Valore	Locale/globale	Indirizzi uso
addf	0x0	G/text	-

Il programma linker (1 di 2)

Il **linker** unisce gli N moduli oggetto in un unico file eseguibile contenente:

- Il programma in linguaggio macchina
- Informazioni di supporto (e.g., indirizzo di partenza per eseguire il file)



Costruzione del programma eseguibile

Genera i segmenti TEXT, DATA e BSS del modulo eseguibile, dove vengono collocate le porzioni di quel segmento provenienti da ciascun modulo.

Operazioni eseguite dal linker:

1. Calcola l'estensione di memoria occupata da ciascun segmento di ciascun modulo
2. Posiziona i segmenti in memoria e calcola il nuovo indirizzo iniziale
3. Ogni indirizzo di memoria definito da una label viene riallocato.
4. Tutti i i riferimenti a simboli esterni vengono risolti.

Esempio

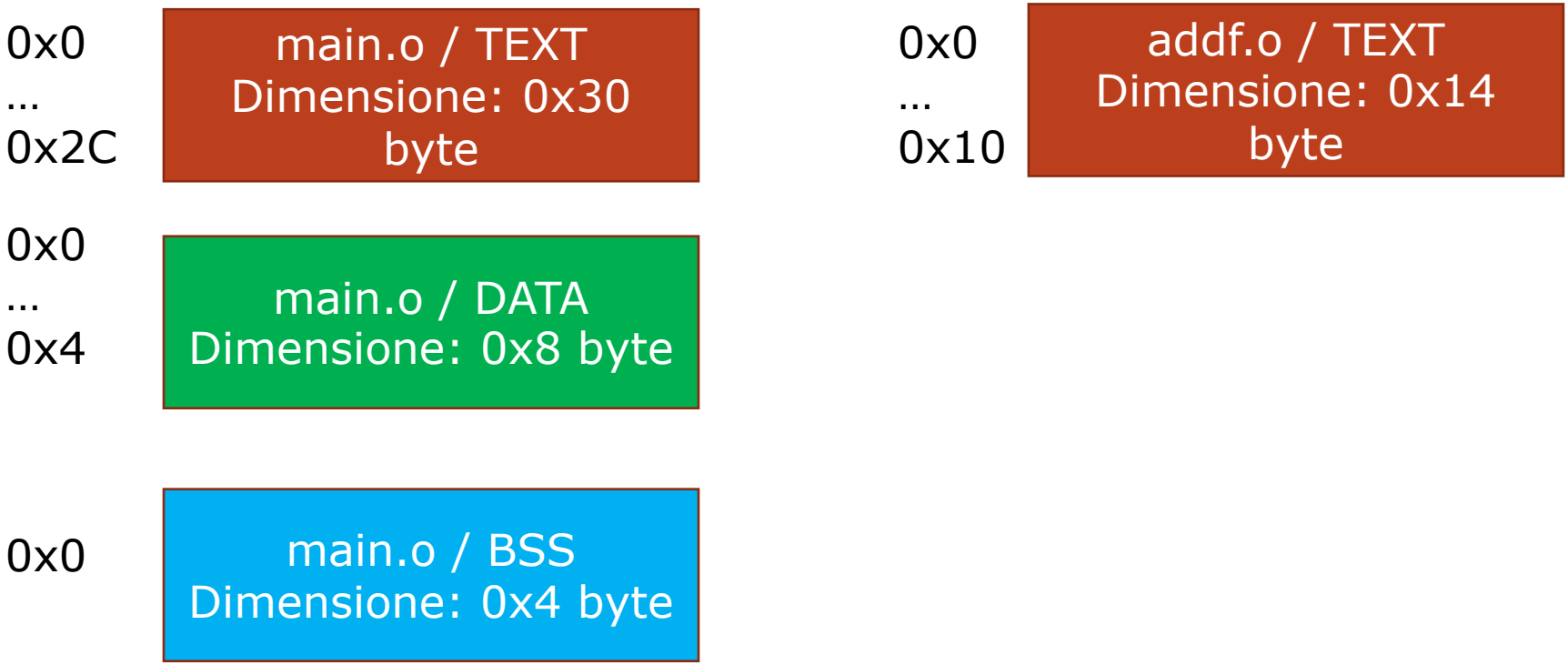
Vediamo un esempio di linkaggio di 2 file oggetto (main.o e addf.o)

- `ld –o main addf.o main.o`

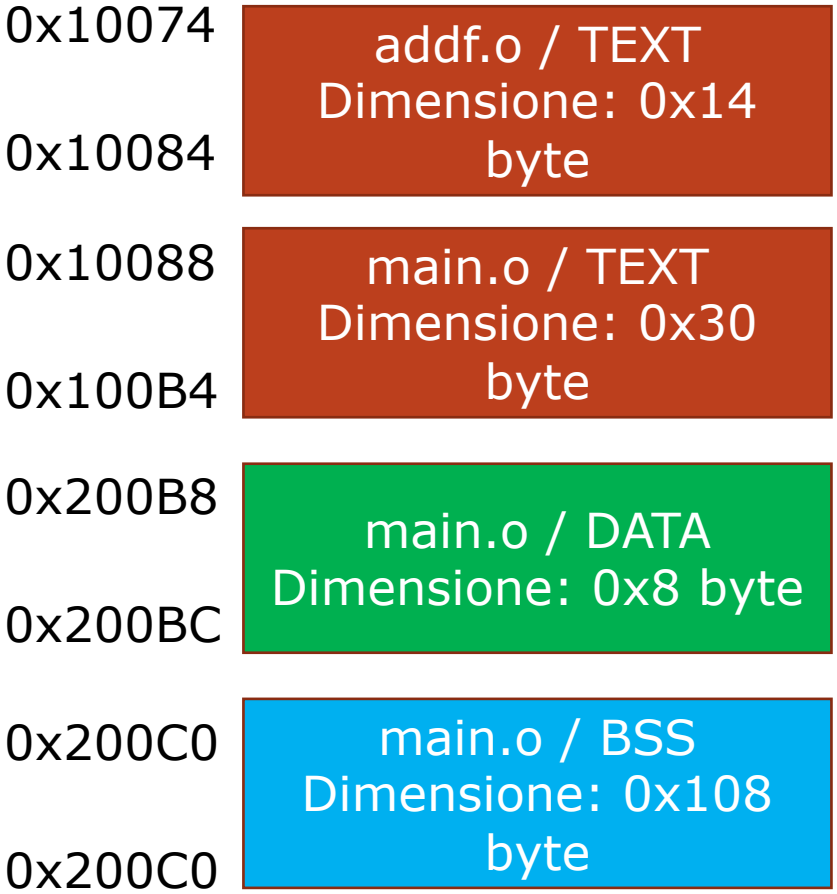
E’ necessario specificare anche l’entry point con i parametri

- ‘-e main’

Segmenti e dimensione



Linker: unione dei segmenti



Indirizzo iniziale di ogni segmento:

- addf.o/TEXT → +0x10074 byte
- main.o/TEXT → +0x10088 byte
- main.o/DATA → +0x200B8 byte
- main.o/BSS → +0x200C0 byte

Riallocazione

- Ogni riferimento a label (i.e. indirizzi in memoria) deve essere aggiornato in seguito all'unione dei segmenti → **Riallocazione**

Non vengono riallocati:

- Indirizzi/valori assoluti, ovvero definiti da costati (.EQU)
- Indirizzi definiti da offset (autorelativo, frame pointer)

Riallocazione (2)

Per ogni label X con valore V all'interno di un segmento con indirizzo iniziale ADR, il linker sostituisce tutte le occorrenze di X con il valore V+ADR

Esempio: si consideri una label X con valore 0x105 all'interno di un segmento che avrà indirizzo iniziale 0xAA000. Il linker sostituirà tutte le occorrenze di 0x105 con il valore 0xAA000+0x105 = 0xAA105.

Riallocazione degli indirizzi

Simbolo	Valore	Locale/globale
main	0X0	globale
end_main	0x20	locale

main.o/TEXT
offset: 0x10088

Simbolo	Valore	Locale/globale
main	0X10088	globale
end_main	0x100A8	locale

Simbolo	Valore	Locale/globale
in1	0x0	locale
in2	0x4	locale

main.o/DATA
offset: 0x200B8

Simbolo	Valore	Locale/globale
in1	0x200B8	locale
in2	0x200BC	locale

Simbolo	Valore	Locale/globale
out	0x0	locale

main.o/BSS
offset: 0x200C0

Simbolo	Valore	Locale/globale
out	0x200C0	locale

Simbolo	Valore	Locale/globale
addf	0x0	globale

addf.o/TEXT
offset: 0x10074

Simbolo	Valore	Locale/globale
addf	0x10074	globale

Sostituzione simboli esterni

Ogni riferimento a simbolo esterno viene risolto con l'utilizzo della tabella dei simboli esterni e di tutte le tabelle dei simboli (tabelle aggiornate con la riallocazione)

Simbolo esterno	Lista di indirizzi in cui il simbolo è usato
addf	0x28

Simbolo esterno	Lista di indirizzi in cui il simbolo è usato (DOPO RIALLOCAZIONE)
addf	0x100A0

Riallocazione in addf/text

```
00000000 <addf>:
   0: e52d0004 push {r0}
   4: e0800001 add r0, r0, r1
   8: e5820000 str r0, [r2]
   c: e49d0004 pop {r0}
  10: e1a0f00e mov pc, lr
```

```
00000000 <addf>:
   0: e52d0004 push {r0}
   4: e0800001 add r0, r0, r1
   8: e5820000 str r0, [r2]
   c: e49d0004 pop {r0}
  10: e1a0f00e mov pc, lr
```

Riallocazione in main/text

```
00000000 <main>:
   0: e92d4007 push {r0, r1, r2, lr}
   4: e59f2018 ldr r2, [pc, #0x1C] ; 24
<end_main+0x4>
   8: e5980000 ldr r0, [r8]
   c: e59f2014 ldr r2, [pc, #0x18] ; 28
<end_main+0x8>
  10: e5981000 ldr r1, [r8]
  14: e59f2010 ldr
<end_main+0xc>
  18: ebfffffe bl 0 <addf>
  1c: e8bd4007 pop {r0, r1, r2, lr}


00000020 <end_main>:
  20: e1a0f00e mov pc, lr
  24: 00000000
  28: 00000004
  2c: 00000000
```

Assegnazione di addf

Riallocazione di end_main

Riallocazione di in1, in2, out

```
00010088 <main>:
   10088: e92d4007 push {r0, r1, r2, lr}
   1008c: e59f2018 ldr r2, [pc, #0x1C] ;
100ac <end_main+0x4>
   10090: e5980000 ldr r0, [r8]
   10094: e59f2014 ldr r
100b0 <end_main+0x8>
   10098: e5981000 ldr r
   1009c: e59f2010 ldr r2, [pc, #0x14] ;
100b4 <end_main+0xc>
   100a0: ebfffff3 bl 10074 <addf>
   100a4: e8bd4007 pop {r0, r1, r2, lr}


000100a8 <end_main>:
   100a8: e1a0f00e mov pc, lr
   100ac: 000200b8
   100b0: 000200bc
   100b4: 000200c0
```

No riallocazione perché autorelativo

Riallocazione in main/data e bss

```
00000000 <in1>:
   0: 00000012 …
00000004 <in2>:
   4: 00000034 …


00000000 <out>:
...
```

```
Disassembly of section .data:

000200b8 <in1>:
   200b8: 00000012

000200bc <in2>:
   200bc: 00000034


Disassembly of section .bss:

000200c0 <__bss_start>:
   200c0: 00000000
```

Programma eseguibile

- Il programma eseguibile contiene tutte le informazioni necessarie per l'esecuzione
  - Istruzioni macchina in text
  - Dati in data/bss
  - Punto di inizio (main)
  - Dimensioni segmenti text, data, bss
  - Tabelle dei simboli
  - Informazioni di debug
  - …

Loader

Per eseguire un programma, viene prima invocato il loader

- Il **loader** carica in memoria le istruzioni macchina e dati del programma agli indirizzi indicati nel file eseguibile e carica nel registro PC l'indirizzo del punto di inizio (main)

- In certi casi, il loader può riallocare il programma in nuovi indirizzi
  - La riallocazione viene eseguita come nel linker

Librerie

- Le **librerie software** sono collezioni di moduli oggetto che contengono subroutine che possono essere invocate da altri programmi

- Il codice presente nelle librerie può essere inserito nel proprio programma in modi diversi:
  - Librerie statiche
  - Librerie dinamiche

Librerie statiche

Librarie statiche: il codice delle librarie viene incluso al momento dell'assemblaggio e linking

- Il file addf.s è un esempio di libraria statica

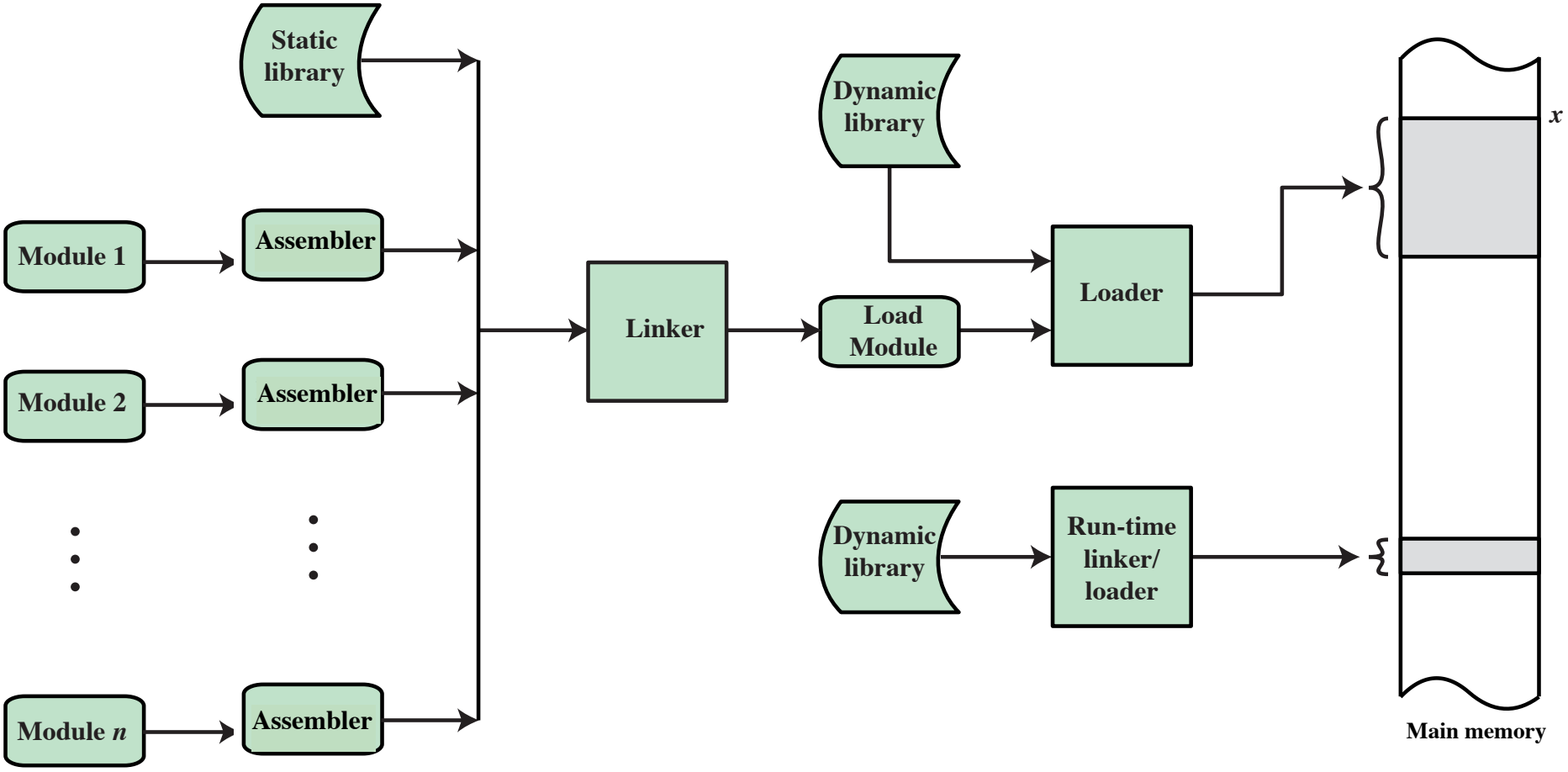
- Permette di creare programmi autonomi
- Ad ogni modifica di una libreria statica, è necessario ricompilare il programma che la utilizza.

Librerie dinamiche

Libreria dinamica: il codice delle librerie viene incluso in un momento successivo al linking

- Se un programma utilizza una libreria dinamica, il linker non risolve i simboli della libreria dinamica, che restano non definiti.
- I simboli della libraria dinamica vengono risolti:
  - Quando il programma viene caricato in memoria, il loader inserisce i simboli mancanti (load-time dynamic library)
  - Oppure quando il programma esegue l'istruzione con il simbolo mancante (run-time dynamic library) → gestito dal sistema operativo

Librerie statiche e dinamiche



Debugger

- Il debugger è un programma che permette l'analisi dell'esecuzione di un programma

- Utilizzato per:
  - Risolvere bug
  - Analizzare prestazioni e punti critici

- Permettono l'esecuzione passo-passo e la visione del contenuto dei registri e della memoria