POLITECNICO
MILANO 1863

# Architettura dei calcolatori e sistemi operativi

## Assemblatore e Collegatore (Linker)

## Capitolo 2 P&H
## Appendice 2 P&H

## Sommario

Il processo di assemblaggio

Il collegatore (linker)

# Assemblatore: traduzione in linguaggio macchina

L'***ASSEMBLATORE*** traduce in linguaggio macchina un programma scritto in linguaggio assemblatore simbolico, e genera la ***rappresentazione oggetto*** del programma (che non è ancora il formato eseguibile).

L'assemblatore traduce le pseudo-istruzioni (e anche le macro) nella corrispondente sequenza di istruzioni native.

Come si è visto, le pseudo-istruzioni forniscono un insieme di istruzioni macchina simboliche più ricco di quello realizzato nativamente in HW, con il solo "costo" aggiuntivo dato dal registro **$*at*** riservato all'uso da parte dell'assemblatore per espandere la pseudo-istruzione (o macro).

# Processo di assemblaggio

- È applicato modulo per modulo al programma, e per ogni ***modulo (file) sorgente*** costruisce il ***modulo (file) oggetto*** corrispondente.
- Esamina, riga per riga, il codice sorgente assemblatore di un modulo, e traduce le istruzioni simboliche nel corrispondente ***formato del linguaggio macchina.***
  - i codici mnemonici sono tradotti nei corrispondenti codici binari
  - i riferimenti ai registri nei corrispondenti "numeri" di registro
  - i ***riferimenti simbolici*** del modulo (identificatori di variabili, etichette di salto, nomi di funzioni) sono tradotti – se possibile – neegli indirizzi binari corrispondenti; per questa operazione l'assemblatore genera la ***tabella dei simboli*** del modulo
- Ogni segmento di ogni modulo è assemblato con indirizzi virtuali rilocabili di segmento (gli indirizzi di ogni segmento partono da 0).

# Processo di assemblaggio (continua)

Un programma eseguibile è generato a partire da più moduli (p. es. le librerie standard come la `stdio` del linguaggio C).

Non è sempre possibile tradurre tutti i riferimenti simbolici del modulo.

I riferimenti relativi a:

- identificatori definiti in altri moduli (***simboli esterni***)
- identificatori dipendenti dalla rilocazione del modulo (***simboli rilocabili***)

costituiscono i ***riferimenti non risolti*** del modulo e vengono gestiti dal ***COLLEGATORE (LINKER)***

Quindi il file oggetto prodotto dall'assemblatore deve contenere anche:

- la tabella dei simboli
- informazioni sulla rilocazione

# Processo di assemblaggio (continua)

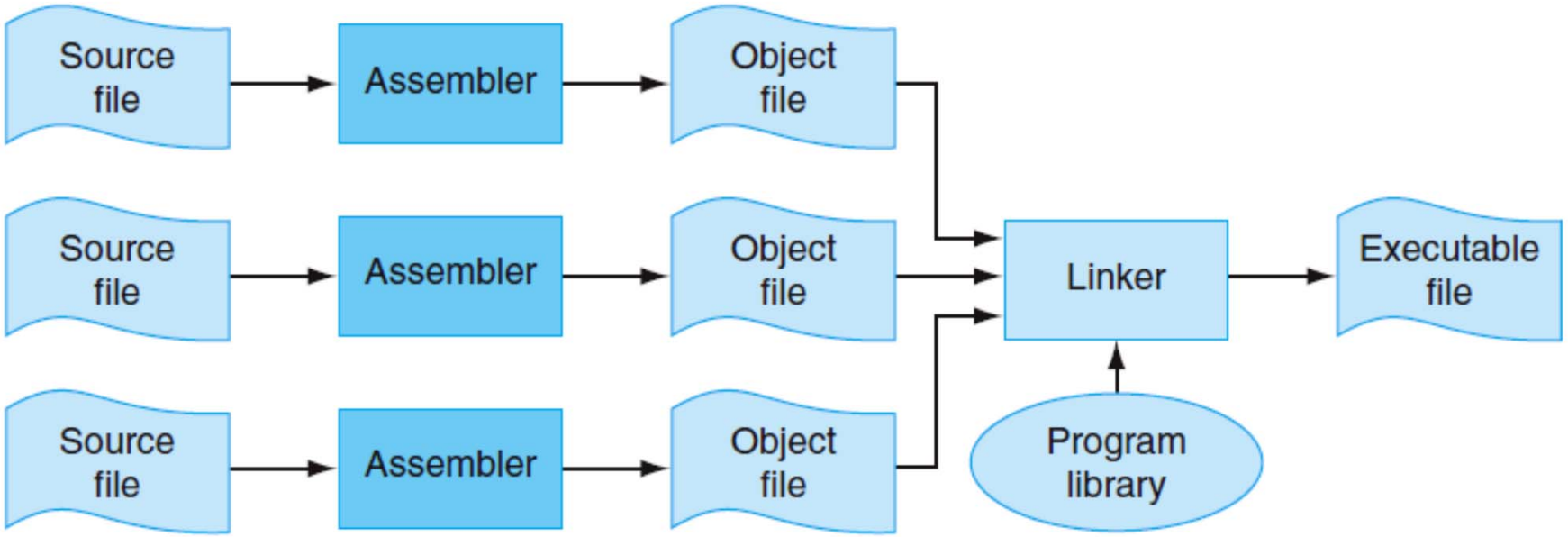
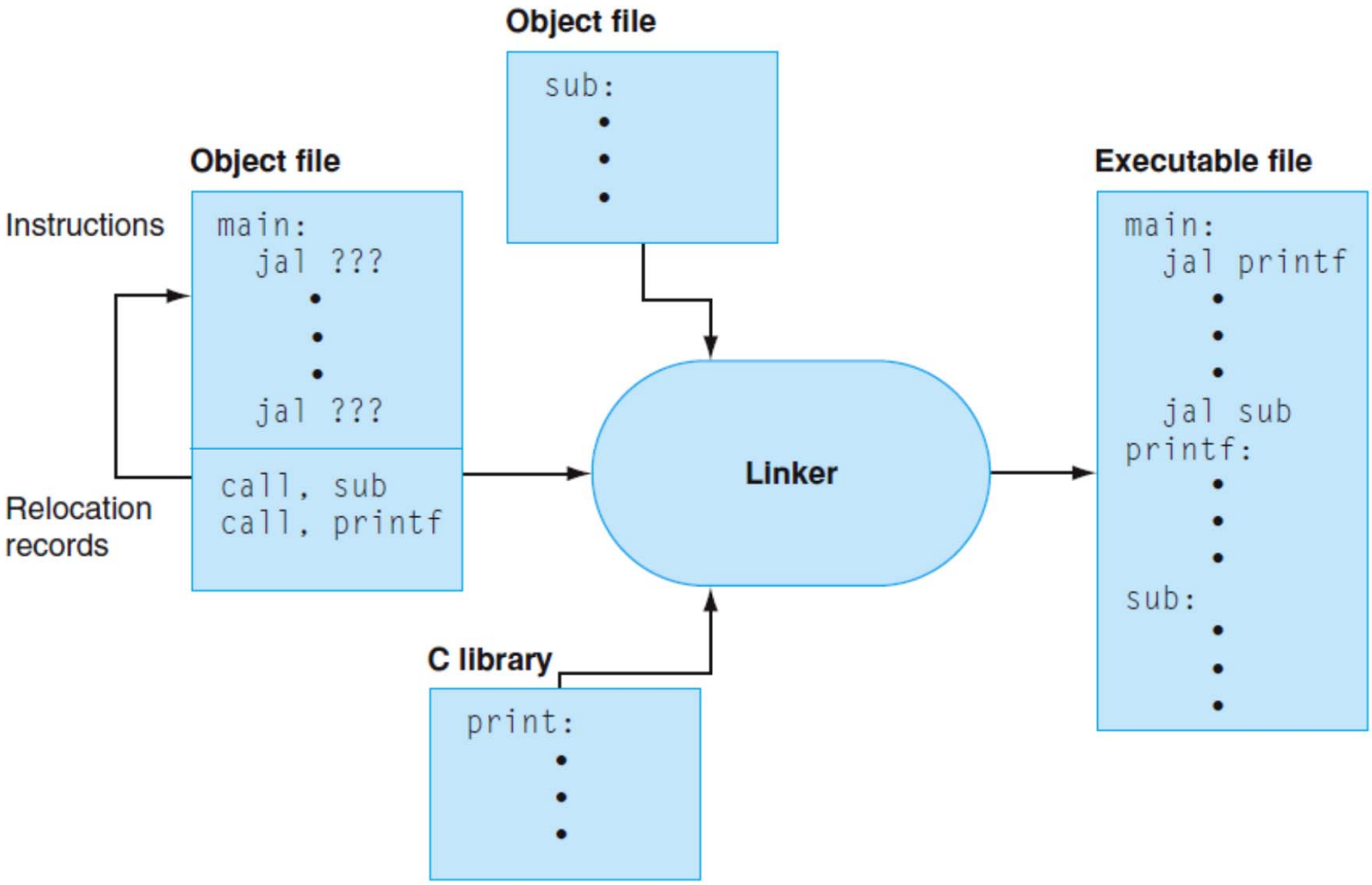



**Figura A.1.1. Il processo che produce un file eseguibile.** Un assemblatore traduce il file contenente codice assembler in un file oggetto, il quale viene collegato ad altri file e a funzioni di libreria per produrre un file eseguibile.

# Processo di assemblaggio (continua)



# Processo di assemblaggio – primo passo

Nel primo passo l'assemblatore non traduce nessuna istruzione, ma costruisce la ***tabella dei simboli del modulo.***

I riferimenti simbolici inseriti nella tabella possono essere:

- simboli associati a direttive dell'assemblatore che definiscono costanti simboliche (.***eqv***) – nella tabella si crea la coppia ⟨ *simbolo*, *valore* ⟩
- etichette che definiscono variabili del segmento dati – nella tabella si crea la coppia ⟨ *simbolo*, *indirizzo* ⟩
- etichette che contrassegnano istruzioni destinazioni di salto – nella tabella si crea la coppia ⟨ *simbolo*, *indirizzo* ⟩

I ***valori degli indirizzi*** inseriti sono quelli ***rilocabili*** rispetto al segmento considerato (T o D).

# Processo di assemblaggio – secondo passo

È la fase di traduzione vera e propria: usa la tabella dei simboli del modulo e genera – oltre alla traduzione – la ***tabella di rilocazione*** del modulo.

Un'istruzione ***è tradotta in modo «incompleto»*** (e quindi deve essere processata anche dal collegatore) se:

- il riferimento simbolico presente in essa è relativo a variabili del segmento ***.data***
  - per variabili scalari la traduzione convenzionale è **`0($gp)`**
  - per vettori (e altre) la traduzione è tramite la pseudoistruzione **`la`**, ossia tramite **`lui`** e **`ori`** (**`addiu`**) con immediati convenzionali a **`0`**
- il riferimento è relativo a simboli non presenti nella tabella dei simboli del modulo: simbolo posto a **`0`** per convenzione
- è un'istruzione di salto di tipo J (non autorilocante): simbolo posto a **`0`** per convenzione

In corrispondenza di ogni traduzione "incompleta", nella tabella di rilocazione viene inserito un elemento (una tripletta) nella forma seguente:

⟨ ***indirizzo rilocabile istruzione, codice op. istruzione, simbolo da risolvere*** ⟩

# ESEMPIO

**Sorgente procedura B:**

```
.text
B:   bne $a0, $0, E
     sw  $a0, Y
E:   lui $t0, W
     ori $t0, $t0, W
.data
Y:   .word 0
```

**Oggetto procedura B:**

(passo 1 assemblatore)

dimensione testo: ----

dimensione dati: ----

segmento di testo:

```
0       bne $a0, $0, E
4       sw  $a0, Y
8       lui $t0, W
C       ori $t0, $t0, W
```

segmento dei dati:

| | | |
|---|---|---|
| 0 | 0 | |

tabella dei simboli:

| | | |
|---|---|---|
| B | 0 | T |
| E | 8 | T |
| Y | 0 | D |

**Oggetto procedura B:**

(passo 2 assemblatore)

dimensione testo: 0x10

dimensione dati: 0x04

segmento di testo:

```
0       bne $a0, $0, 1
4       sw  $a0, 0($gp)
8       lui $t0, 0
C       ori $t0,$t0, 0
```

segmento dei dati:

| | | |
|---|---|---|
| 0 | 0 | |

tabella dei simboli:

| | | |
|---|---|---|
| B | 0 | T |
| E | 8 | T |
| Y | 0 | D |

tabella di rilocazione:

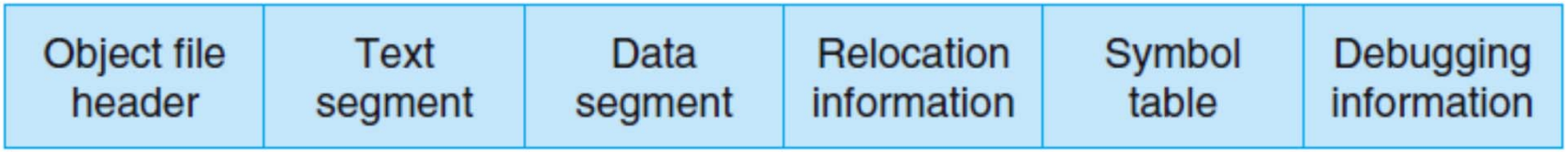
| | | |
|---|---|---|
| 4 | **sw** | Y |
| 8 | **lui** | W |
| C | **ori** | W |

Se l'istruzione è di salto in formato I,
il simbolo viene risolto come

**(VS_REL – (IADDR_REL + 4)) / 4**

Per esempio in «**bne** `$a0, $0, E`» il simbolo `E` viene tradotto come (8 – (0 + 4)) / 4 = 1

# Il file oggetto di un modulo




L'**intestazione** descrive le *dimensioni* del segmento testo e del segmento dati del modulo.

Il **segmento testo** contiene il codice in linguaggio macchina delle procedure (o funzioni) del file sorgente. Queste procedure potrebbero essere non eseguibili a causa di riferimenti non risolti.

Il **segmento dati statici** contiene una rappresentazione binaria dei dati definiti nel file sorgente. Anche i dati potrebbero essere incompleti a causa di riferimenti non risolti a etichette definite in altri file.

Le **informazioni di rilocazione** identificano le istruzioni e le parole di dati che dipendono da **indirizzi assoluti all'interno del file eseguibile** del programma completo. La posizione di queste istruzioni e dati deve essere modificata se parti del programma vengono spostate in memoria.

La **tabella dei simboli** associa un indirizzo alle etichette «esterne» contenute nel modulo sorgente e contiene l'elenco dei riferimenti non risolti del modulo.

Le **informazioni di debug**, che non consideriamo ulteriormente.

# Esempio di riferimento

**sorgente MAIN:**

```
        .text
MAIN:   lw  $a0, X
        beq $a0, $0, E
        jal B
        .globl MAIN
        .data
X:      .word 128
W:      .word 0x12345678
```

**sorgente procedura B:**

```
        .text
B:      bne $a0, $0, E
        sw  $a0, Y
E:      lui $t0, W
        ori $t0, $t0, W
        .data
Y:      .word 0
```

**oggetto MAIN:**

dimensione testo: 0x0C

dimensione dati: 0x08

segmento testo:

```
0       lw  $a0, 0($gp)
4       beq $a0, $0, 0
8       jal 0
```

segmento dati:

| | |
|---|---|
| 0 | 0x80 |
| 4 | 0x12345678 |

tabella dei simboli:

| | | |
|---|---|---|
| MAIN | 0 | T |
| X | 0 | D |
| W | 4 | D |

tabella di rilocazione:

| | | |
|---|---|---|
| 0 | lw | X |
| 4 | beq | E |
| 8 | jal | B |

**oggetto procedura B:**

dimensione testo: 0x10

dimensione dati: 0x04

segmento esto:

```
0       bne $a0, $0, 1
4       sw  $a0, 0($gp)
8       lui $t0, 0
C       ori $t0, $t0, 0
```

segmento dati:

| | |
|---|---|
| 0 | 0 |

tabella dei simboli:

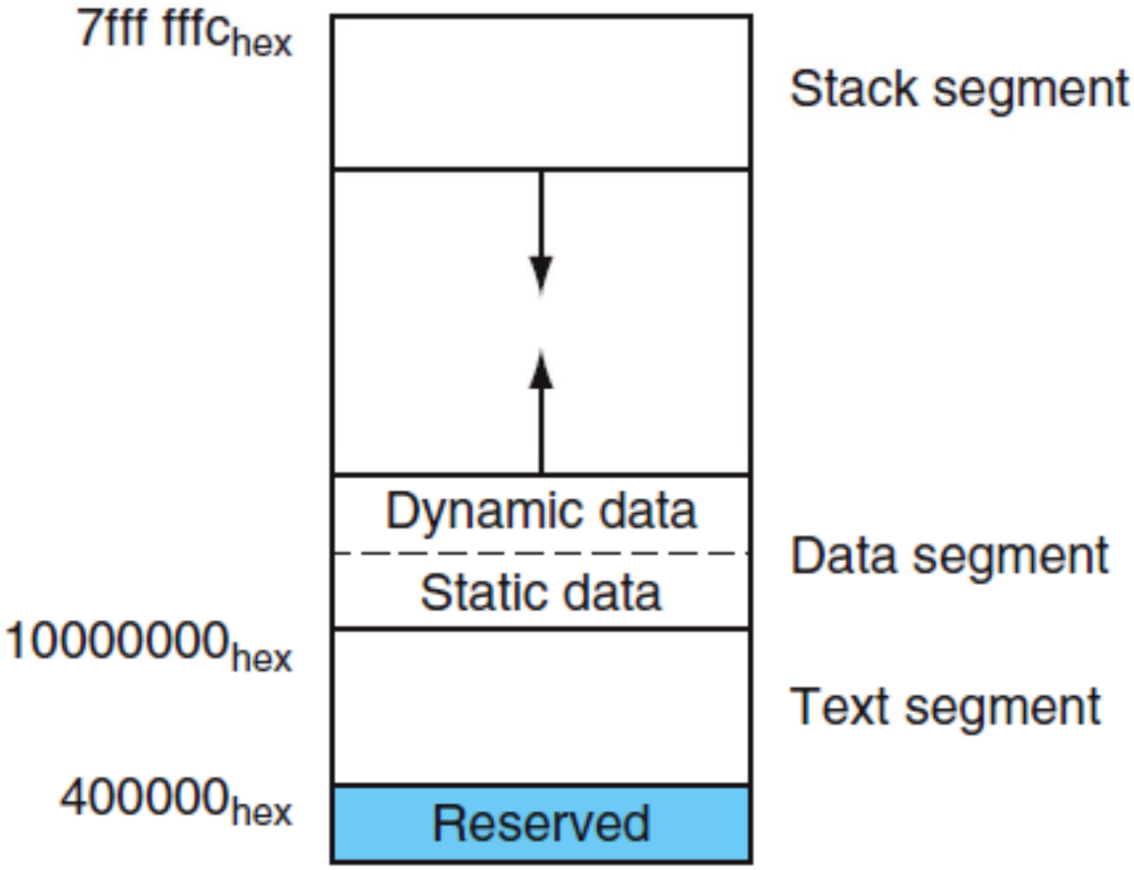
| | | |
|---|---|---|
| B | 0 | T |
| E | 8 | T |
| Y | 0 | D |

tabella di rilocazione:

| | | |
|---|---|---|
| 4 | sw | Y |
| 8 | lui | W |
| C | ori | W |

# Organizzazione della memoria

spazio virtuale assoluto del programma



# Accesso all'area dati statici

- Il segmento dati inizia all'indirizzo `0x1000 0000`:
  - dunque le istruzioni di accesso alla memoria non possono fare riferimento direttamente agli oggetti in esso contenuti, avendo campi di 16 bit
  - esempio: per caricare la parola all'indirizzo `0x1001 0020` nel registro `$v0` sono necessarie due istruzioni:

```
lui $s0, 0x1001          # 0x1001 significa 1001 in base 16
lw  $v0, 0x0020($s0)     # 0x1001 0000 + 0x0020 = 0x1001 0020
```

- Ottimizzazione: uso di un registro ($gp) come **puntatore globale** (*global pointer*) al segmento dei dati statici. Ecco come si usa:
  - questo registro viene inizializzato con l'indirizzo `0x1000 8000`
  - le istruzioni di *load* e *store* possono utilizzare il loro campo spiazzamento di 16 bit dotato di segno per accedere ai primi 64 kB del segmento dei dati statici
  - ora l'esempio precedente richiede una sola istruzione:

```
lw  $v0, 0x8020($gp)
```

- Il registro *global pointer* accelera l'indirizzamento delle locazioni di memoria comprese tra `0x1000 0000` e `0x1001 0000` rispetto a quello delle altre locazioni dell'area dati.
- Di solito il compilatore memorizza in questa area le **variabili globali di tipo scalare**, poiché esse sono memorizzate in locazioni fisse e sono più adatte a questa modalità di indirizzamento rispetto ad altri dati globali (come i vettori).

# Processo di collegamento

Il collegatore (linker) ha il compito di generare un ***solo programma binario eseguibile*** (in formato rilocabile) partendo dai diversi moduli.

➡ un ***solo spazio di indirizzamento*** per tutto il programma (spazio di indirizzamento virtuale del programma)

Il collegatore (linker) opera in base a queste informazioni:

- la lunghezza del segmento testo e del segmento dati di ciascun modulo tradotto
- e gli indirizzi di impianto

Il collegatore (linker) calcola gli indirizzi dei riferimenti non risolti e completa la traduzione delle istruzioni presenti nelle tabelle di rilocazione.

# Operazioni svolte dal collegatore (linker)

- Determinare lo posizione in memoria, cioè l'indirizzo iniziale o di base, delle sezioni codice e dati dei diversi moduli (vedi l'esempio precedente).

- Determinare il nuovo valore di tutti gli indirizzi simbolici che risultano modificati dallo spostamento della base, ossia creare una ***tabella globale dei simboli***.

- Correggere in tutti i moduli i riferimenti agli indirizzi simbolici che sono stati modificati, in base alle tabelle di rilocazione.

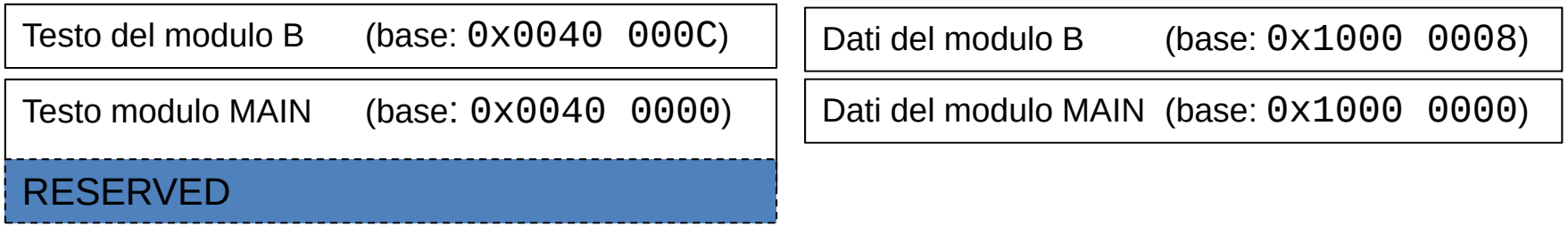
# Determinazione della posizione in memoria dei moduli

L'assemblatore alloca la sezione testo e la sezione dati a partire dall'indirizzo base 0.

Ma i moduli non possono essere tutti caricati nella stessa zona di memoria; essi vanno invece caricati sequenzialmente.

Inoltre essi devono rispettare la struttura generale della memoria.

Esempio di riferimento:



# Creazione della tabella globale dei simboli

È costituita dall'unione delle tabelle dei simboli di tutti i moduli da collegare, modificati (rilocati) in base all'indirizzo di base del modulo cui appartengono.

Esempio di riferimento:

| Simbolo | Valore iniziale | Base di rilocazione | Valore finale |
|---|---|---|---|
| MAIN | 0 | 0040 0000 | 0040 0000 |
| X | 0 | 1000 0000 | 1000 0000 |
| W | 4 | 1000 0000 | 1000 0004 |
| B | 0 | 0040 000C | 0040 000C |
| E | 8 | 0040 000C | 0040 0014 |
| Y | 0 | 1000 0008 | 1000 0008 |

# Correzione dei riferimenti nei moduli

Siano:

- ISTR un'istruzione riferita dalla tabella di rilocazione di un modulo M, con simbolo S e indirizzo IND
- IADDR l'indirizzo di tale istruzione nell'eseguibile finale:

  IADDR = IND + BASE_M

  dove BASE_M è l'indirizzo di base del modulo M
- VS il valore di S nella tabella globale dei simboli
- GP il valore del registro *global pointer*

Regole da applicare in base al tipo di istruzione:

- ISTR è in formato J: inserire VS / 4 nell'istruzione
- ISTR è di salto in formato I: inserire (VS – (IADDR + 4)) / 4
- ISTR è aritmetico-logica in formato I:

  inserire i 16 bit meno significativi di VS (VS_low)

  ISTR è di tipo *load* o *store*: inserire VS – GP
- ISTR è l'istruzione **lui**:

  inserire i 16 bit più significativi di VS (VS_high)

| Indirizzo | Istruzione completa | Note |
|---|---|---|
| **Segmento codice (testo)** | | |
| 0040 0000 | `lw $a0, 0x8000($gp)` | X – GP; 0x1000 0000 – 0x1000 8000 |
| 0040 0004 | `beq $a0, $0, 0x0003` | (E – (IADDR+4)) / 4;<br>(0x0040 0014 – 0x0040 0008) / 4 |
| 0040 0008 | `jal 0x0010 0003` | B / 4; 0x0040 000C / 4 |
| 0040 000C | `bne $a0, $0, 1` | autorilocante; inizio modulo B |
| 0040 0010 | `sw $a0, 0x8008($gp)` | Y – GP; 0x1000 0008 – 0x1000 8000 |
| 0040 0014 | `lui $t0, 0x1000` | W_high |
| 0040 0018 | `ori $t0,$t0, 0x0004` | W_low |
| **Segmento dati** | | |
| 1000 0000 | 128 | valore iniziale di X |
| 1000 0004 | 0x1234 5678 | valore iniziale di W |
| 1000 0008 | 0 | valore iniziale di Y |

# Il file eseguibile del programma completo

| intestazione | codice eseguibile | valori iniziali dei dati statici (variabili globali) | tabella globale dei simboli (facoltativa) |
|---|---|---|---|

Somiglia al file oggetto di un singolo modulo, ma ormai contiene un *programma completo* (unione di tutti i moduli) *eseguibile*, e informazioni su come caricarlo in memoria e gestirlo.

L'**intestazione** ha dimensioni fisse e contiene alcuni campi necessari per guidare il lancio del programma. Specifica le *dimensioni* del *codice eseguibile* e dei *valori iniziali dei dati*, e *l'indirizzo dell'istruzione iniziale* del programma, cioè l'istruzione con etichetta MAIN.

Il **codice eseguibile** è la *lista delle istruzioni* del programma. Tutte le istruzioni sono ormai in forma numerica completa, ossia tutti i loro campi sono numericamente risolti. Al lancio del programma, esse verranno caricate in memoria *in sequenza* nel segmento di testo.

I **valori iniziali dei dati statici (variabili globali)** sono la *lista dei valori iniziali da dare ai dati* (a quelli inizializzati – gli altri vengono azzerati per default). Al lancio del programma, i valori vengono caricati in memoria nel segmento dati, *agli indirizzi dei dati corrispondenti*.

La **tabella globale dei simboli** è includibile facoltativamente a solo *scopo di debug*. Se è presente, al lancio del programma viene caricata con il codice. Eseguendo il programma passo-passo per farne il debug, quando un'istruzione genera un *indirizzo numerico*, consultando la tabella il sistema di debug trova il *simbolo corrispondente* e lo visualizza. Serve per aiutare il programmatore a comprendere meglio il tipo di errore e a correggerlo.

# Caricamento ed esecuzione

Nei Sistemi Operativi di famiglia UNIX (Linux), il *nucleo* (*kernel*) del SO carica il programma nella memoria principale e ne lancia l'esecuzione. Operazioni:

1. Legge l'intestazione del file eseguibile per determinare le dimensioni dei segmenti *testo* e *dati*, e per conoscere *l'indirizzo dell'istruzione iniziale* del programma.
2. Crea un nuovo spazio di indirizzamento per il programma. Questo spazio è abbastanza grande per contenere i segmenti di *testo* (con eventuale tabella globale dei simboli) e dei *dati*, nonché un segmento per la *pila* (*stack*).
3. Carica le istruzioni e i valori iniziali dei dati dal file eseguibile alla memoria, mettendo tutto quanto all'interno del nuovo spazio di indirizzamento.
4. Carica in pila gli eventuali argomenti passati al programma (`argc` e `argv` in ling. C).
5. Inizializza i registri dell'architettura. In generale, la maggior parte dei registri viene azzerata, tranne il registro *stack pointer*, cui viene assegnato l'indirizzo della prima cella libera della pila, e (per MIPS) il registro *global pointer*, come già visto.
6. Esegue un'apposita procedura di avvio, che fa parte del nucleo del SO. Tale procedura copia gli argomenti del programma dalla pila ai registri e poi chiama la procedura *main* del programma, saltando all'indirizzo dell'istruzione iniziale specificato nel file eseguibile (cioè salta all'istruzione con etichetta MAIN). Quando *main* termina, la procedura di avvio conclude il programma tramite la chiamata di sistema *exit*.