

# *CALCOLATORI*

## *Toolchain: Come generare applicazioni in linguaggio macchina*

Marco Roveri  
[marco.roveri@unitn.it](mailto:marco.roveri@unitn.it)

*Lezione basata su materiale preparato  
dai Prof. Luca Abeni, Luigi Palopoli e Giovanni Iacca*



UNIVERSITÀ DEGLI STUDI DI TRENTO  
Dipartimento di Ingegneria  
e Scienza dell'Informazione

# La lingua della CPU

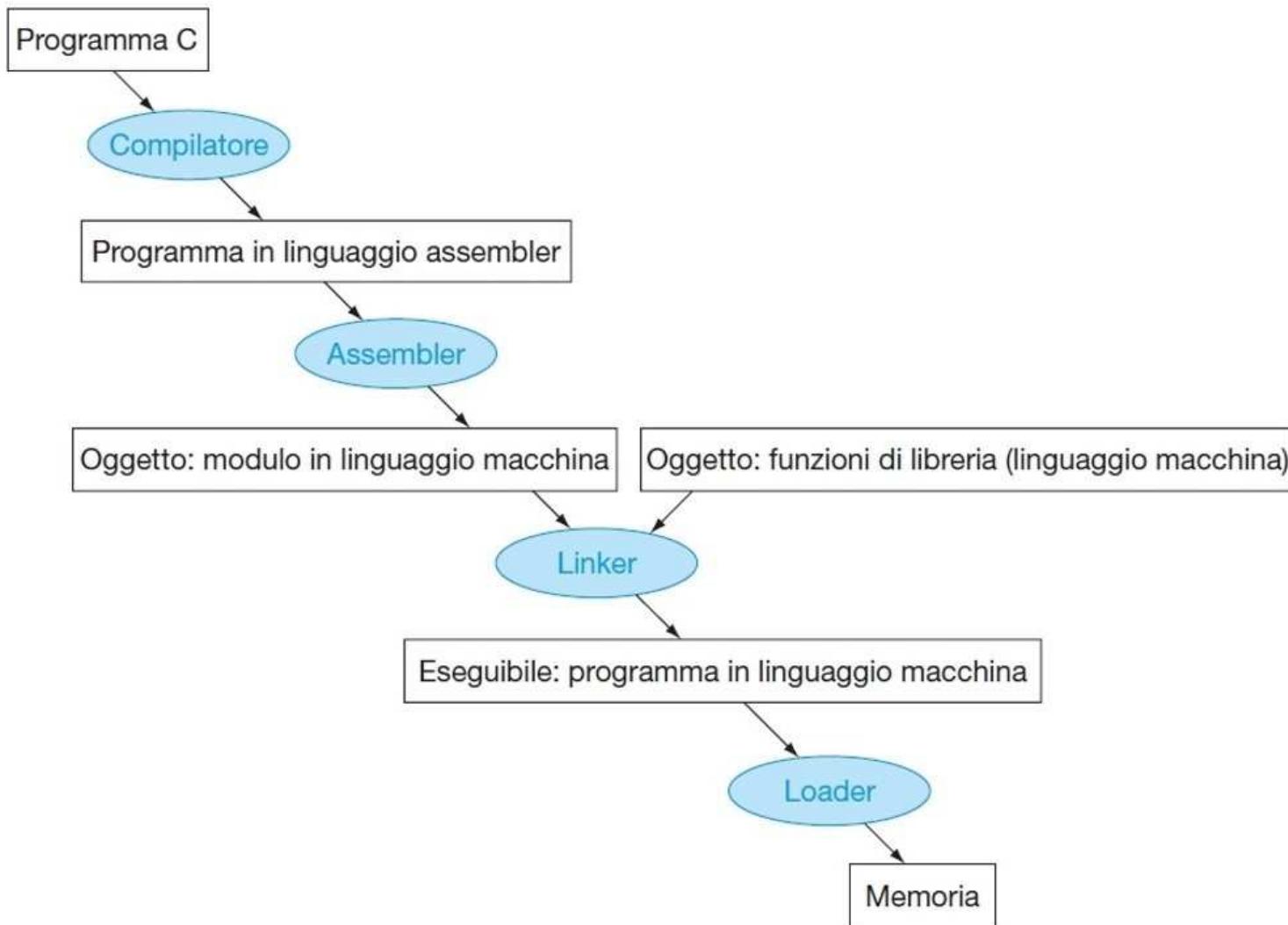
- Una CPU “capisce” e riesce ad eseguire solo il **linguaggio macchina**
  - Linguaggio di (estremamente!) basso livello
  - Sequenza di 0 e 1
- Assembly: codici **mnemonici** (add, addi, etc.) invece di cifre binarie
  - Più “gestibile” del linguaggio macchina...
  - ... ma sempre troppo complesso per noi!
  - In realtà, anche più complesso di quanto visto finora
- In genere, non si programma direttamente in assembly!
  - Assembly generato a partire da linguaggio di alto livello...
  - Chi fa la conversione? **Compilatore!**

# Compilazione: Esempio

- Esempio di generazione di codice in linguaggio macchina da linguaggio di alto livello: linguaggio C
  1. Preprocessore: gestisce direttive # . . . . Generalmente, sostituzione di codice
  2. Compilatore: da C ad assembly (file . s)
  3. Assembler: da assembly a linguaggio macchina (file . o)
  4. Linker: mette assieme codice in linguaggio macchina e librerie per generare un eseguibile
- Normalmente, un *driver* gestisce tutto questo in automatico
- Il file eseguibile può ora essere caricato in memoria con un'apposita system call (in unix, la famiglia exec () )

# Un compilatore C

- Preprocessore: poco interessante per noi, ignoriamolo

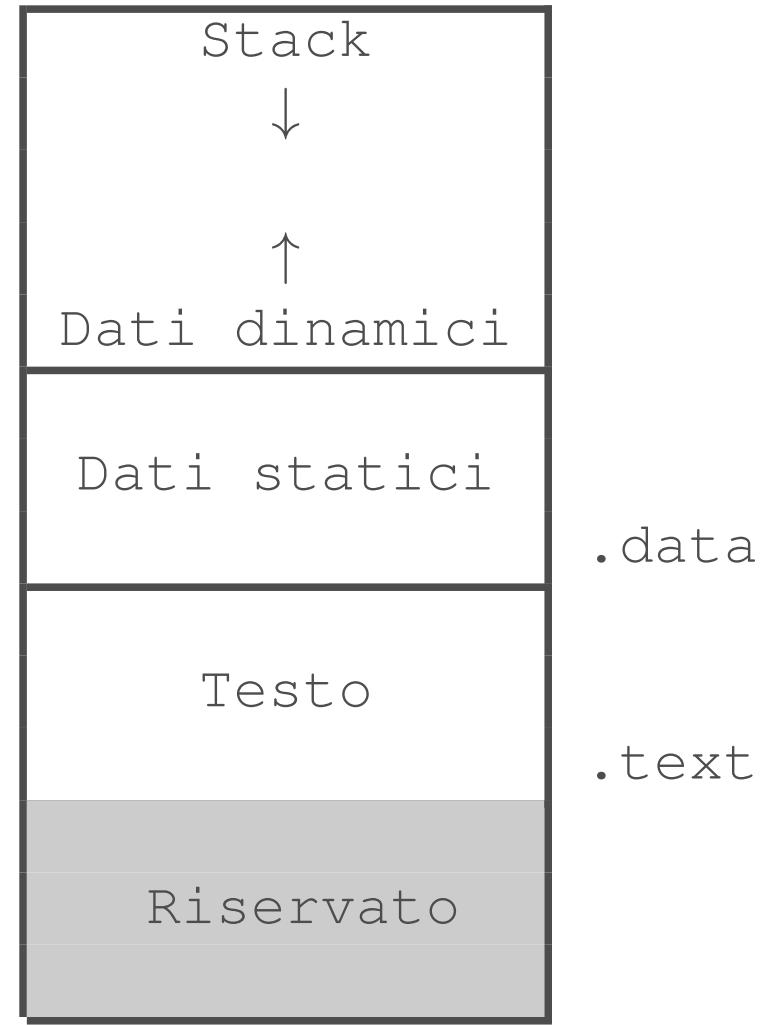


# Allocazione della memoria per programmi e dati nel RISC-V

SP → 0000 003f ffff ffff0<sub>16</sub>

0000 0000 1000 0000<sub>16</sub>

PC → 0000 0000 0040 0000<sub>16</sub>



## Usando gcc...

- gcc: gnu compiler collection
  - Può compilare vari linguaggi di alto livello...
  - ... generando linguaggio macchina per varie CPU (RISC-V compreso!)
- Vari passaggi ad opera di diversi programmi: `cpp`, `cc`, `as`, `ld`
- gcc invoca i vari comandi usando i giusti parametri
  - Default: invoca tutti i programmi necessari
  - `gcc -S` si ferma dopo aver invocato `cc` (genera file assembly `.s`)
  - `gcc -c` si ferma dopo aver invocato `as` (genera file oggetto `.o`)

# Da C ad assembly

- Dato un file `<file>.c`, comando `gcc -S` invoca `cc` per generare un file assembly `<file>.s`
  - Sintassi: `gcc -S <file>.c [-o <nomefile>]`
    - Senza opzione `-o` genera `<file>.s`
    - Con `-o` salva il risultato della compilazione in `<nomefile>`
  - `cc` è il **Compilatore** propriamente detto
- `cc` conosce l'architettura target (RISC-V, ARM, Intel X86, nel nostro caso) meglio di un programmatore umano
- Spesso il codice assembly generato da `cc` è migliore di quello generato “a mano”
  - Possibili diversi livelli di ottimizzazione (`-O...`)
    - `O0` No ottimizzazioni, `-ON` con  $N \geq 1$  diversi e sofisticati livelli di ottimizzazione
  - `gcc -c -Q -ON --help=optimizers` mostra le ottimizzazioni abilitate con livello  $N$ 
    - Provare con  $N=0, 1, \dots$  e vedere le differenze
  - Assembly generato potrebbe essere di non facile lettura
    - Ottimizzazioni potrebbero riordinare istruzioni per sfruttare al meglio il processore

# Da assembly a linguaggio macchina

- Dato un file `<file>.c` o `<file>.s`, `gcc -c` invoca `cc` e `as` o solo `as` (risp.) per generare un file oggetto `<file>.o`
  - Sintassi: `gcc -c <file.{s|c}> [-o <nomefile>]`
    - Senza opzione `-o` genera `<file>.o`
    - Con `-o` salva il risultato della compilazione in `<nomefile>`
  - **Assembler**
- `as` fa spesso qualcosa in più rispetto alla semplice sostituzione di codici mnemonici con sequenze di bit
  - Pseudo-istruzioni
    - Convertite in istruzioni riconosciute dalla CPU
  - Converte numeri da decimale / esadecimale a binario
  - Gestisce label
  - Gestisce salti: se destinazione troppo lontana, `j DEST` va convertita in caricamento di registro + `jr`
  - Genera *metadati*

# Pseudo-istruzioni

- Non corrispondono a vere e proprie istruzioni in Linguaggio Macchina
  - Esempio: RISC-V non ha istruzioni native tipo `mv` fra registri.
- Ma sono utili per il programmatore (o il compilatore)
- L'assembler sa come convertirle in una o più istruzioni macchina esistenti
- Esempi:
  - `mv x10, x11` // x10 assume valore di x11  
  ↓  
  `addi x10, x11, 0` // x10 riceve il contenuto di x11 + 0
  - `li x9, 123` // carica il valore 123 in x9  
  ↓  
  `addi x9, x0, 123` // x9 assume il valore x0 + 123
  - `j LABEL` // Jump non condizionato a LABEL  
  ↓  
  `jal x0, LABEL`
  - etc.

# File oggetto

Composti da *segmenti* distinti:

- **Header**
  - Specifica dimensione e posizione degli altri segmenti del file oggetto
- **Segmenti**
  - **Segmento di Testo/Text segment**: contiene il codice in linguaggio macchina
  - **Segmento dati statici/Static data segment**: contiene tutti i dati (sia statici che dinamici) allocati per la durata del programma (codice)
- **Tabella dei simboli/Symbol table**
  - Associa simboli ad indirizzi (relativi)
  - Enumera simboli non definiti (sono in altri moduli)
- **Tabella di rilocazione/Relocation table**
  - Enumera istruzioni che fanno riferimento a istruzioni e dati che quando programma messo in memoria dipendono da indirizzi assoluti (da “patchare”)
- Altre informazioni (debugging, etc.)

# Da file oggetto ad eseguibili

- Dato un file `<file>.c` o `<file>.s` o `<file>.o`, `gcc` senza opzioni `-S` e `-c` invoca anche il linker (`ld`) per generare un eseguibile
  - Sintassi: `gcc -c <file.{s|c|o}> [-o <nomefile>]`
    - Senza opzione `-o` genera il file `a.out` (su Windows `a.exe`)
    - Con `-o` salva l'eseguibile in `<nomefile>`
- **Linker** `ld`: mette assieme uno o più file oggetto, eseguendo le necessarie rilocazioni
  - Decide come codice e dati sono disposti in memoria
  - Associa indirizzi assoluti a tutti i simboli
  - Risolve simboli che erano lasciati indefiniti in alcuni file `.o`
  - “Patcha” le istruzioni macchina citate nella tabella di rilocazione (in base agli indirizzi assegnati)
- Scopo di `ld` è quindi eliminare tavole dei simboli e tavole di rilocazione, generando codice macchina con i giusti riferimenti
  - Poiché un simbolo usato in un file può essere definito in un file diverso, `ld` mette quindi assieme più file `.o`

## Linker e simboli

- Un linker gestisce vari tipi di simboli:
  - **Simboli definiti** (defined): associati ad un indirizzo (relativo) nella tabella dei simboli
  - **Simboli non definiti** (undefined): usati in un file (e quindi presenti nella tabella dei simboli) ma definiti in un file diverso
  - **Simboli locali** (o non esportati): definiti ed usati in un file (quindi simili a simboli definiti), ma non usabili in altri file
- In tutti i casi, associa un indirizzo **assoluto** ad ogni simbolo
- Per simboli non definiti, cerca in altri file
  - Se non trovato, errore di linking!

## Linking in tre passi

1. Disporre in memoria i vari segmenti (.text, .data, etc.) dei file .o
    - Segmenti testo uno dopo l'altro, idem per i segmenti dati, etc.
  2. In base al passo precedente, assegnare un indirizzo assoluto ad ogni simbolo contenuto nelle varie tavole dei simboli
  3. In base alle tavole di rilocazione, correggere le varie istruzioni con gli indirizzi calcolati
- Il risultato viene poi “incapsulato” in un file eseguibile
    - Segmenti (testo, dati, etc.)
    - Informazioni per il caricamento in memoria (indirizzo di caricamento dei segmenti, indirizzo entry point, etc.)
    - Altre informazioni (es. per debugging)

# Librerie

- Esistono funzioni “predefinite” fornite dal compilatore / sistema
- Definite in file .o inclusi in ogni eseguibile che viene prodotto
  - Buon numero di file oggetto linkati “di default” in ogni eseguibile
  - Poco pratico!
- **Libreria:** collezione di file .o
  - Invece di linkare un'enormità di file oggetto, si linka un'unica libreria!
- Librerie: statiche o dinamiche
  - **Librerie statiche** (.a):
    - semplici collezioni di file oggetto; ld fa tutto il lavoro!
  - **Librerie dinamiche** (.so):
    - ld non fa molto... il vero linking avviene a tempo di esecuzione (caricamento dinamico)!

## Librerie statiche

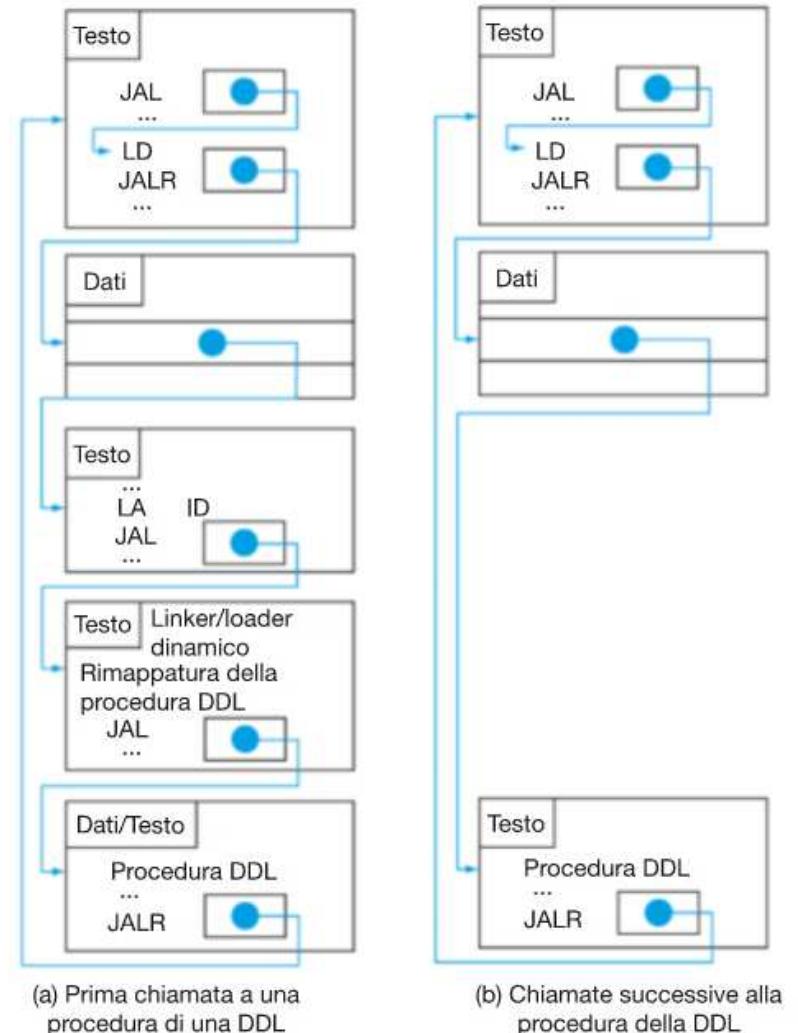
- ld inserisce nell'eseguibile tutto il codice della libreria utilizzato dal programma
  - La libreria serve solo durante il linking (codice autocontenuto)
  - Le dimensioni dell'eseguibile aumentano...
  - Esempio: ogni eseguibile contiene una copia del codice di printf...
- Caricamento del programma da parte del SO: semplice!

# Librerie dinamiche

- `ld` inserisce nell'eseguibile riferimenti alle librerie usate ed alle funzioni invocate...
  - ... ma non le include nell'eseguibile!
- Ogni eseguibile contiene un riferimento ad un *linker dinamico* (`/lib/ld-linux.so`)
  - All'esecuzione del programma, viene caricato ed eseguito `/lib/ld-linux.so` passandogli il programma stesso come argomento!
  - `ld-linux.so` caricherà quindi l'eseguibile e le librerie (`.so`) da cui dipende, e si occuperà di fare il linking
    - La libreria serve anche per eseguire il programma (codice non autocontenuto)
- Caricamento del programma da parte del SO: complesso!
- Vantaggi/Svantaggi
  - + Le dimensioni dell'eseguibile sono piccole
  - + Possibile aggiornare librerie senza ricompilare
  - - Il programma non è autocontenuto

# Possibile complicazione: “lazy linking”

- A noi informatici piace complicare le cose...
  - ... e siamo pigri!
- Invece di fare le operazioni di linking a tempo di caricamento, posporle il più possibile
  - Se un eseguibile è linkato ad una libreria, ma non ne invoca mai i servizi a runtime, forse si può evitare di linkarla...
- Invece di chiamare la vera funzione, si chiama uno *stub* che esegue caricamento, rilocazione e linking quando serve
  - La seconda volta che si chiama la procedura, il processo sarà più semplice perchè la procedura ora è già stata caricata



## Esempio: funzioni da compilare / linkare

- Programma composto da due file assembly (.s)

file1.o

```
.comm x,4,4
...
.text
.globl func_1
func_1:
    ld x10, 0(x3)
    jal x1, 0
    ...
    ...
```

file2.o

```
.comm y,4,4
...
.text
.globl func_2
func_2:
    sd x11, 0(x3)
    jal x1, 0
    ...
    ...
```

# Esempio: file oggetto 1

header	campo	valore	
	nome	file1	
	text size	$100_{16}$	
	data size	$20_{16}$	
text	indirizzo (rel.)	istruzione	
	0	ld x10, 0(x3)	
	4	jal x1, 0	
	8	...	
data	indirizzo (rel.)	simbolo	
	0	x	
	...	...	
tabella simboli	simbolo	indirizzo	
	x	*UND*	
	func_2	*UND*	
	...	...	
tabella rilocazione	indirizzo	tipo istruzione	simbolo
	0	ld	x
	4	jal	func_2

Procedura func\_1 necessita indirizzo di x da mettere nella ld e indirizzo func\_2 da mettere nella jal.

## Esempio: file oggetto 2

header	campo	valore	
	nome	file2	
	text size	$200_{16}$	
	data size	$30_{16}$	
text	indirizzo (rel.)	istruzione	
	0	sd x11, 0(x3)	
	4	jal x1, 0	
	8	...	
data	indirizzo (rel.)	simbolo	
	0	y	
	...	...	
tabella simboli	simbolo	indirizzo	
	y	*UND*	
	func_1	*UND*	
	...	...	
tabella rilocazione	indirizzo	tipo istruzione	simbolo
	0	sd	y
	4	jal	func_1

Procedura func\_2 necessita indirizzo di y per la sd e indirizzo di func\_1 per la sua jal.

# Linker: mettendo tutto assieme...

Prima file1 poi file2

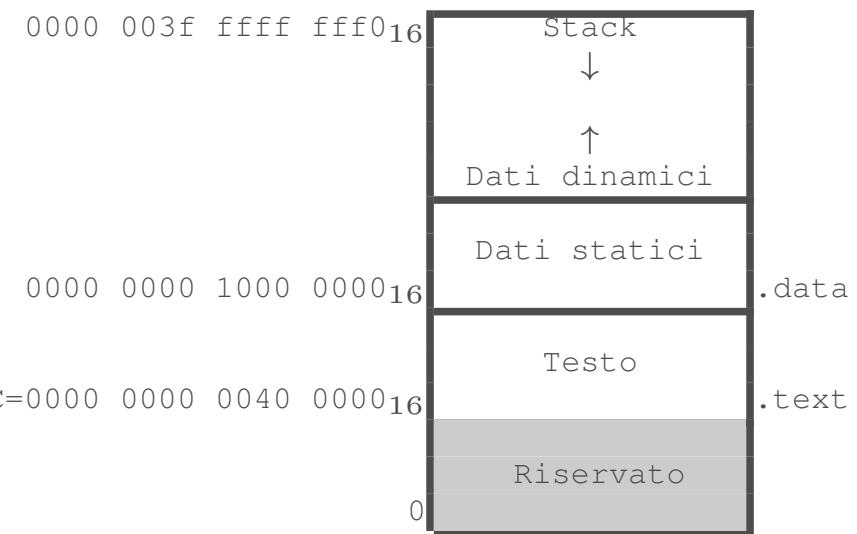
header	campo	valore
	text size	AAA
	data size	BBB
	...	...
text	indirizzo	istruzione
	KKKKKKKKKKKKKKKKKK <sub>16</sub>	ld x10, UUU(x3)
	LLLLLLLLLLLLLLL <sub>16</sub>	jal x1, YYY
	...	...
	MMMMMMMMMMMMMM <sub>16</sub>	sd x11, VVV(x3)
	NNNNNNNNNNNNNNNN <sub>16</sub>	jal x1, TTT
	...	...
data	indirizzo	simbolo
	PPPPPPPPPPPPPPPP <sub>16</sub>	x
	...	...
	JJJJJJJJJJJJJJJ <sub>16</sub>	y
	...	...

Procedura func\_1 necessita indirizzo di x da mettere nella ld e indirizzo func\_2 da mettere nella jal.

Procedura func\_2 necessita indirizzo di y per la sd e indirizzo di func\_1 per la sua jal.

# Linker: mettendo tutto assieme... (cont.)

- Header
  - Text size:  
 $100_{16}$  (file1) +  $200_{16}$  (file2) =  $300_{16}$
  - Data size:  
 $20_{16}$  (file1) +  $30_{16}$  (file2) =  $50_{16}$
- Disposizione segmenti in memoria
  - text: inizia a  $0000000000400000_{16}$   
prima file1 (dimensione  $100_{16}$ ),  
poi file2 (indirizzo  $0000000000400100_{16}$ )
  - data: inizia a  $00000001000000_{16}$ ;  
prima file1 (dimensione  $20_{16}$ ),  
poi file2 (indirizzo  $000000010000020_{16}$ )
- Assegnamento indirizzi a simboli:
  - func\_1:  $0000000000400000_{16}$   
func\_2:  $0000000000400100_{16}$
  - x:  $000000010000000_{16}$   
y:  $000000010000020_{16}$



## Linker: mettendo tutto assieme... (cont.)

header	campo	valore
	text size	$300_{16}$
	data size	$50_{16}$
	...	...
text	indirizzo	istruzione
	$0000000000400000_{16}$	ld x10, <b>UUU</b> (x3)
	$0000000000400004_{16}$	jal x1, <b>YYY</b>
	...	...
	$0000000000400100_{16}$	sd x11, <b>VVV</b> (x3)
	$00000000000000104_{16}$	jal x1, <b>TTT</b>
	...	...
data	indirizzo	simbolo
	$0000000010000000_{16}$	x
	...	...
	$0000000010000020_{16}$	y
	...	...

## Linker: mettendo tutto assieme... (cont.)

- Calcolo valore per `jal`:
  - le istruzioni utilizzano indirizzo relativo al PC, basta fare differenza tra indirizzo della `jal` e indirizzo della procedura;
    - il campo indirizzo di `jal` a  $400004_{16}$  che salta a  $400100_{16}$  (indirizzo procedura `func_2`), conterrà  $400100_{16} - 400004_{16} = 252_{10}$
    - il campo indirizzo di `jal` a  $400104_{16}$  che salta a  $400000_{16}$  (indirizzo procedura `func_1`), conterrà  $400000_{16} - 400104_{16} = -260_{10}$
  - Calcolo offset per `ld/sd`
    - Sono più complessi da calcolare perchè dipendono da indirizzo base ( $x3$  nel caso che assumiamo  $x3 = 0000000010000000_{16}$ ):
      - per ottenere indirizzo di `x` inseriamo  $0_{16}$  nel campo indirizzo di `ld` a  $400000_{16}$
      - per ottenere indirizzo di `y` inseriamo  $20_{16}$  nel campo indirizzo di `sd` a  $400100_{16}$
    - Gli indirizzi associati alle operazioni di store vengono gestiti come le load, ad eccezione del fatto che formato istruzioni tipo `S` rappresenta le costanti diversamente dal formato `I` delle load.
      - Inseriamo  $32_{10}$  (i.e.  $20_{16}$ ) nel campo indirizzo della `sd` all'indirizzo  $400100_{16}$  per ottenere indirizzo  $0000000010000020_{16}$  di `y`.

## Quindi...

<b>header</b>	campo	valore
	text size	$300_{16}$
	data size	$50_{16}$
	...	...
<b>text</b>	indirizzo	istruzione
	$0000000000400000_{16}$	ld x10, 0(x3)
	$0000000000400004_{16}$	jal x1, 252 <sub>10</sub>
	...	...
	$0000000000400100_{16}$	sd x11, 32(x3)
	$0000000000400104_{16}$	jal x1, -260 <sub>10</sub>
	...	...
<b>data</b>	indirizzo	simbolo
	$0000000010000000_{16}$	x
	...	...
	$0000000010000020_{16}$	y
	...	...