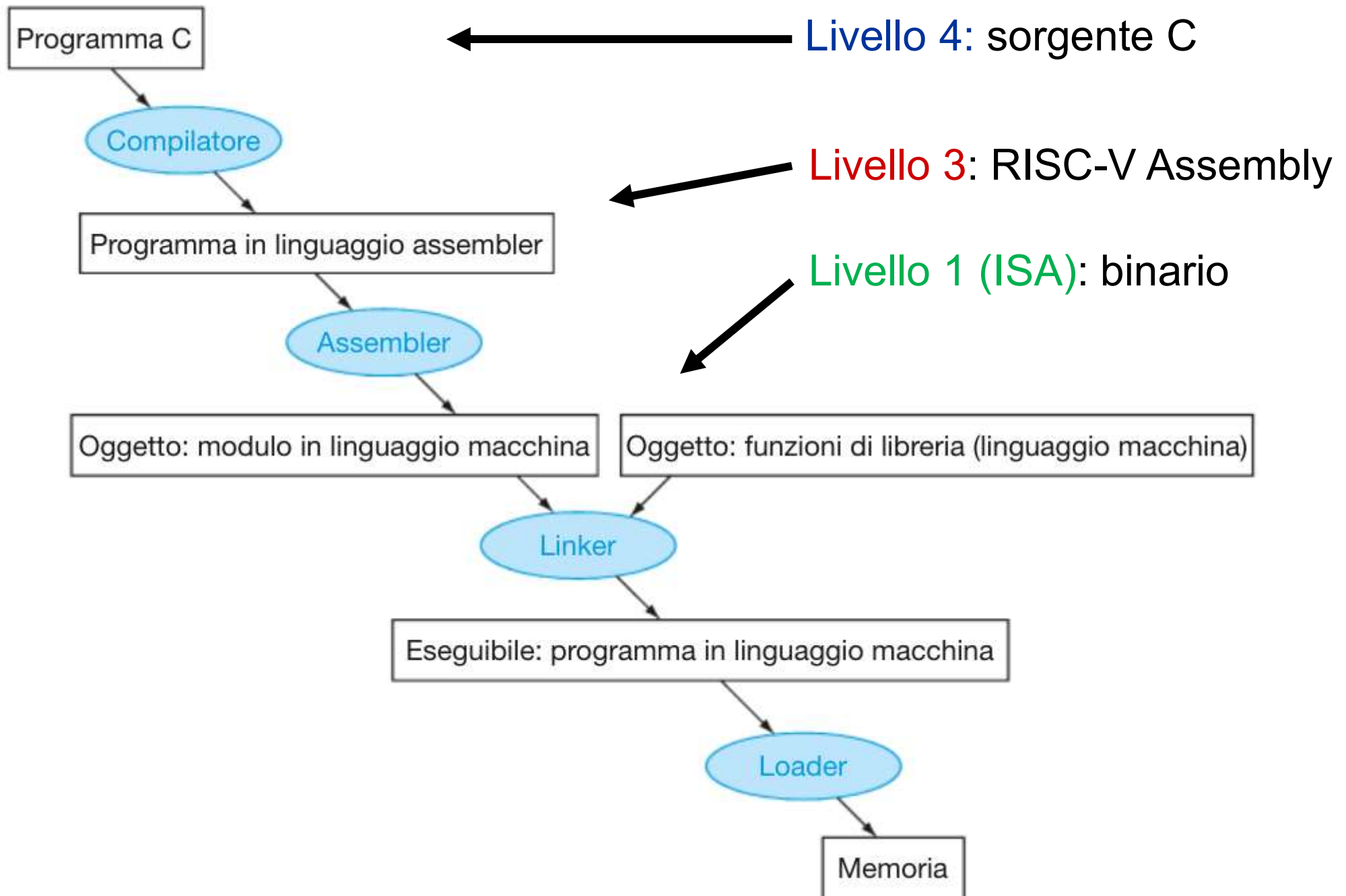


Corso di
Architettura degli Elaboratori
a.a. 2023/2024

Il livello del Linguaggio Assemblativo

Sequenza di passi di traduzione per il C



Compilazione da C

```
void scambia(long long int v[], int k) {  
    long long int temp;  
    temp = v[k];  
    v[k] = v[k + 1];  
    v[k+1] = temp;  
}
```

↑
Livello 4:
Sorgente C

Livello 3:
RISC-V Assembly

↓

```
scambia:  
    slli    x6, x11, 3    # reg x6 = k * 8  
    add     x6, x10, x6   # reg x6 = v + (k * 8)  
    ld      x5, 0(x6)     # reg x5 (temp) = v[k]  
    ld      x7, 8(x6)     # reg x7 = v[k + 1]  
    sd      x7, 0(x6)     # v[k] = reg x7  
    sd      x5, 8(x6)     # v[k+1] = reg x5 (temp)  
    jalr    x0, 0(x1)     # ritorno alla procedura chiamante
```

riscv64-linux-gnu-gcc -O1 -o- -S INPUT.c

Traduzione

scambia:

```
slli  x6, x11, 3  # reg x6 = k * 8
add   x6, x10, x6 # reg x6 = v + (k * 8)
ld    x5, 0(x6)   # reg x5 (temp) = v[k]
ld    x7, 8(x6)   # reg x7 = v[k + 1]
sd    x7, 0(x6)   # v[k] = reg x7
sd    x5, 8(x6)   # v[k+1] = reg x5 (temp)
jalr  x0, 0(x1)   # ritorno alla procedura chiamante
```

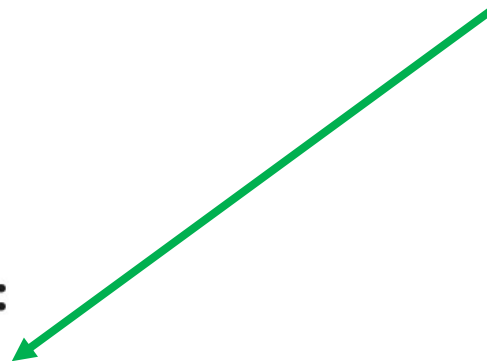


Livello 3:
RISC-V Assembly

Livello 1 (ISA):
sequenza di byte
in memoria

000000000000010078 <scambia>:

```
10078:      00359313
1007c:      932a
1007e:      00033283
10082:      00833383
10086:      00733023
1008a:      00533423
1008e:      00008067
```



```
slli    x6,x11,0x3
c.add    x6,x10
ld       x5,0(x6)
ld       x7,8(x6)
sd       x7,0(x6)
sd       x5,8(x6)
jalr     x0,0(x1)
```

riscv64-linux-gnu-objdump -M no-aliases -M numeric -r -S INPUT.o

Formato binario

00000000000010078 <scambia>:

10078: 00359313

1007c: 932a

1007e: 00033283

10082: 00833383

10086: 00733023

1008a: 00533423

1008e: 00008067

slli x6, x11, 0x3

c.add x6, x10

ld x5, 0(x6)

ld x7, 8(x6)

sd x7, 0(x6)

sd x5, 8(x6)

jalr x0, 0(x1)

Livello 1 (ISA):
sequenza di byte
in memoria

0b 0000000 00111 00110 011 00000 0100011

Istruzione	Formato	immediato	rs2	rs1	funz3	immediato	codop
sd (memorizzazione di parola doppia)	S	indirizzo	reg	reg	011	indirizzo	0100011

riscv64-linux-gnu-objdump -M no-aliases -M numeric -r -S INPUT.o

Esempio: RISC-V ISA vs MIPS ISA

Registro-registro

	31	25	24	20	19	15	14	12	11	7	6	0										
RISC-V	funz7(7)					rs2(5)				rs1(5)				funz3(3)		rd(5)			codop(7)			
	31	26	25	21	20	16	15	11	10	6	5	0										
MIPS	Op(6)					Rs1(5)				Rs2(5)				Rd(5)			Cost(5)			Opx(6)		

Trasferimento dalla memoria

	31					20	19					15	14			12	11					7	6					0		
RISC-V	immed(12)												rs1(5)			funz3(3)			rd(5)			codop(7)								
	31					26	25					21	20					16	15									0		
MIPS	Op(6)					Rs1(5)					Rs2(5)					Cost(16)														

Trasferimento alla memoria

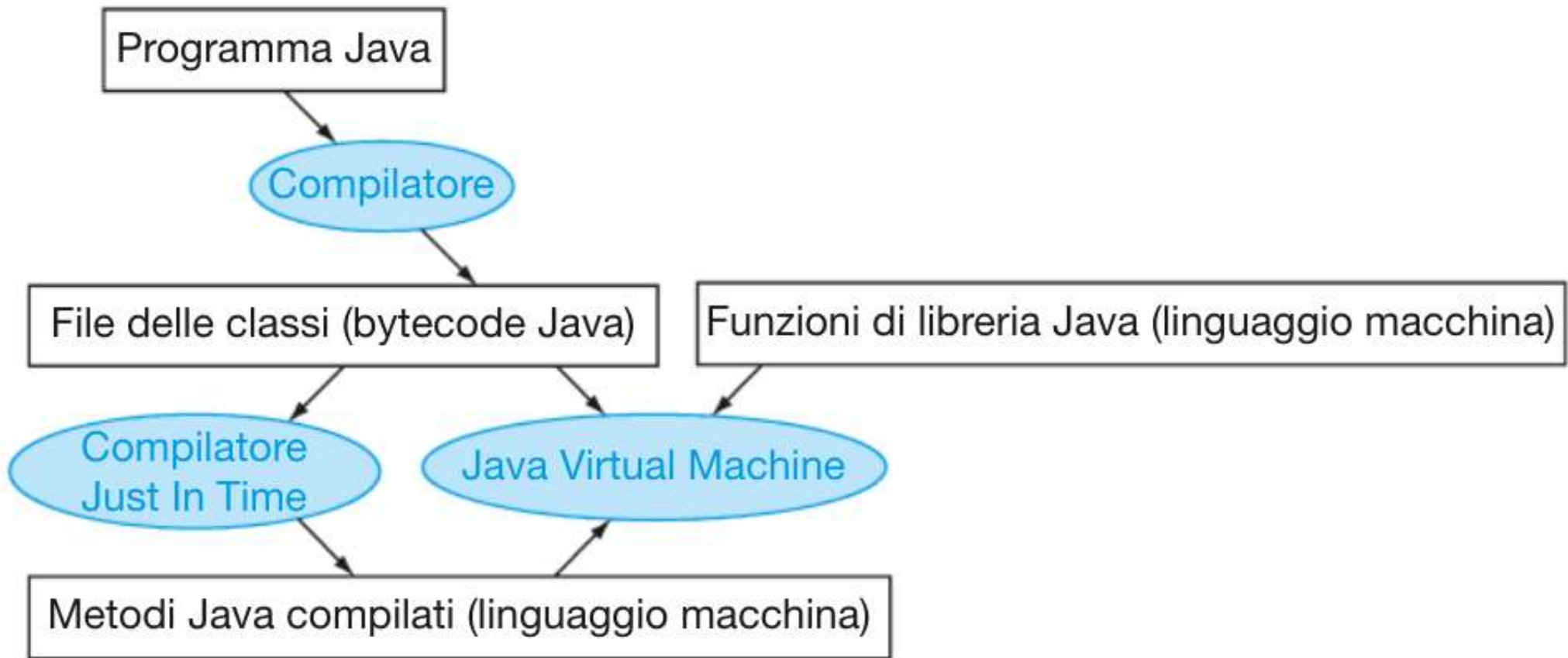
	31	25	24	20	19	15	14	12	11	7	6	0									
RISC-V	immed(7)					rs2(5)				rs1(5)				funz3(3)		immed(5)			codop(7)		
	31	26	25	21	20	16	15						0								
MIPS	Op(6)					Rs1(5)				Rs2(5)				Cost(16)							

Salto condizionato

	31	25	24	20	19	15	14	12	11	7	6	0											
RISC-V	immed(7)					rs2(5)				rs1(5)				funz3(3)			immed(5)			codop(7)			
	31	26	25	21	20	16	15						0										
MIPS	Op(6)					Rs1(5)				Opx/Rs2(5)				Cost(16)									

Il formato binario non è compatibile!

Traduzione del codice Java



- Il programma Java è eseguito da un interprete (Java Virtual Machine)
- **La JVM può invocare il compilatore Just In Time (JIT), che compila i metodi del linguaggio Java nel linguaggio macchina del calcolatore sul quale è in esecuzione**

Cos'è un linguaggio Assemblativo?

Quando si parla di **Linguaggio Assemblativo** si intende un linguaggio le cui istruzioni sono ottenute dalle istruzioni ISA sostituendo i *codici binari* con **codici mnemonici**; il linguaggio assemblativo è quindi molto vicino al linguaggio macchina: c'è sostanzialmente una **corrispondenza uno-uno** tra le istruzioni ISA e le istruzioni del linguaggio assemblativo.

In realtà, il linguaggio assemblativo fornisce altre facilitazioni al programmatore, quali l'uso di:

- **etichette** simboliche per variabili e indirizzi,
- primitive per **allocazione in memoria** di variabili,
- **costanti**,
- definizione di **macro**, ...

Cos'è un Linguaggio Assemblativo?

Per passare dal programma scritto in linguaggio **assemblativo** al programma **eseguibile** in linguaggio macchina (ISA) si utilizza un programma traduttore detto **assemblatore** (assembler) che traduce i *codici mnemonici* nei *codici numerici* corrispondenti alle istruzioni ISA.

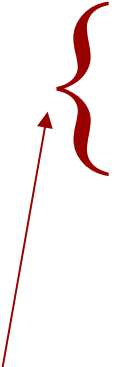
ATTENZIONE!! C'è un po' di confusione sulla nomenclatura....

Il nome **assembler** (ovvero il nome che indica il programma traduttore) viene da molti usato come sinonimo di linguaggio **assemblativo**.

Il termine **linguaggio macchina** viene talvolta usato per indicare il linguaggio **assemblativo**, altre volte per istruzioni ISA.

Linguaggio assembly: Intel x86


Computazione di $N = I + J$

Label	Opcode	Operands	Comments
FORMULA:	MOV	EAX,I	; register EAX = I
	ADD	EAX,J	; register EAX = I + J
	MOV	N,EAX	; N = I + J
 I	DD	3	; reserve 4 bytes initialized to 3
	DD	4	; reserve 4 bytes initialized to 4
	DD	0	; reserve 4 bytes initialized to 0

Comandi al programma assembly di riservare memoria per le variabili I, J, N

Linguaggio assembly: Motorola 680x0

Computazione di $N = I + J$

Label	Opcode	Operands	Comments
FORMULA	MOVE.L	I, D0	; register D0 = I
	ADD.L	J, D0	; register D0 = I + J
	MOVE.L	D0, N	; N = I + J
 I	DC.L	3	; reserve 4 bytes initialized to 3
	DC.L	4	; reserve 4 bytes initialized to 4
	DC.L	0	; reserve 4 bytes initialized to 0

Comandi al programma assembly di riservare memoria per le variabili I, J, N

Pseudo-istruzioni (1)

Il linguaggio assemblativo consente al programmatore di specificare informazioni indispensabili per la traduzione del programma sorgente in programma oggetto

Pseudo-istruzioni : non parte delle istruzioni del livello ISA ma alias per una o più istruzioni. Le pseudo-istruzioni quindi non compariranno come tali nel programma oggetto alla fine della fase di traduzione

Pseudo-istruzioni (2)

Alcune delle pseudo-istruzioni presenti nel RISC-V

nop	addi x0, x0, 0	No operation
li rd, immediate	<i>Myriad sequences</i>	Load immediate
mv rd, rs	addi rd, rs, 0	Copy register
not rd, rs	xori rd, rs, -1	One's complement
neg rd, rs	sub rd, x0, rs	Two's complement
negw rd, rs	subw rd, x0, rs	Two's complement word
sext.w rd, rs	addiw rd, rs, 0	Sign extend word
seqz rd, rs	sltiu rd, rs, 1	Set if = zero
snez rd, rs	sltu rd, x0, rs	Set if \neq zero
sltz rd, rs	slt rd, rs, x0	Set if < zero
sgtz rd, rs	slt rd, x0, rs	Set if > zero
fmv.s rd, rs	fsgnj.s rd, rs, rs	Copy single-precision register
fabs.s rd, rs	fsgnjx.s rd, rs, rs	Single-precision absolute value
fneg.s rd, rs	fsgnjn.s rd, rs, rs	Single-precision negate
fmv.d rd, rs	fsgnj.d rd, rs, rs	Copy double-precision register
fabs.d rd, rs	fsgnjx.d rd, rs, rs	Double-precision absolute value
fneg.d rd, rs	fsgnjn.d rd, rs, rs	Double-precision negate
beqz rs, offset	beq rs, x0, offset	Branch if = zero
bnez rs, offset	bne rs, x0, offset	Branch if \neq zero
blez rs, offset	bge x0, rs, offset	Branch if \leq zero
bgez rs, offset	bge rs, x0, offset	Branch if \geq zero
bltz rs, offset	blt rs, x0, offset	Branch if < zero
bgtz rs, offset	blt x0, rs, offset	Branch if > zero
bgt rs, rt, offset	blt rt, rs, offset	Branch if >
ble rs, rt, offset	bge rt, rs, offset	Branch if \leq
bgtu rs, rt, offset	bltu rt, rs, offset	Branch if >, unsigned
bleu rs, rt, offset	bgeu rt, rs, offset	Branch if \leq , unsigned

Table 25.2: RISC-V pseudoinstructions.

Pseudo-istruzioni (3)

Alcune delle pseudo-istruzioni presenti nel RISC-V

pseudoinstruction	Base Instruction	Meaning
j offset	jal x0, offset	Jump
jal offset	jal x1, offset	Jump and link
jr rs	jalr x0, 0(rs)	Jump register
jalr rs	jalr x1, 0(rs)	Jump and link register
ret	jalr x0, 0(x1)	Return from subroutine
call offset	auipc x1, offset[31:12] + offset[11] jalr x1, offset[11:0] (x1)	Call far-away subroutine
tail offset	auipc x6, offset[31:12] + offset[11] jalr x0, offset[11:0] (x6)	Tail call far-away subroutine
fence	fence iorw, iorw	Fence on all memory and I/O
rdinstret[h] rd	csrrs rd, instret[h], x0	Read instructions-retired counter
rdcycle[h] rd	csrrs rd, cycle[h], x0	Read cycle counter
rdtime[h] rd	csrrs rd, time[h], x0	Read real-time clock
csrr rd, csr	csrrs rd, csr, x0	Read CSR
csrw csr, rs	csrrw x0, csr, rs	Write CSR
csrs csr, rs	csrrs x0, csr, rs	Set bits in CSR
csrc csr, rs	csrrc x0, csr, rs	Clear bits in CSR
csrwi csr, imm	csrrwi x0, csr, imm	Write CSR, immediate
csrsi csr, imm	csrrsi x0, csr, imm	Set bits in CSR, immediate
csrci csr, imm	csrrci x0, csr, imm	Clear bits in CSR, immediate
frcsr rd	csrrs rd, fcsr, x0	Read FP control/status register
fscsr rd, rs	csrrw rd, fcsr, rs	Swap FP control/status register
fscsr rs	csrrw x0, fcsr, rs	Write FP control/status register
frfm rd	csrrs rd, frm, x0	Read FP rounding mode
fsrm rd, rs	csrrw rd, frm, rs	Swap FP rounding mode
fsrm rs	csrrw x0, frm, rs	Write FP rounding mode
frflags rd	csrrs rd, fflags, x0	Read FP exception flags
fsflags rd, rs	csrrw rd, fflags, rs	Swap FP exception flags
fsflags rs	csrrw x0, fflags, rs	Write FP exception flags

Assembly Directives (1)

Directive	Arguments	Description
.align	integer	align to power of 2 (alias for .p2align)
.file	"filename"	emit filename FILE LOCAL symbol table
.globl	symbol_name	emit symbol_name to symbol table (scope GLOBAL)
.local	symbol_name	emit symbol_name to symbol table (scope LOCAL)
.comm	symbol_name,size,align	emit common object to .bss section
.common	symbol_name,size,align	emit common object to .bss section
.ident	"string"	accepted for source compatibility
.section	[{.text,.data,.rodata,.bss}]	emit section (if not present, default .text) and make current
.size	symbol, symbol	accepted for source compatibility
.text		emit .text section (if not present) and make current
.data		emit .data section (if not present) and make current
.rodata		emit .rodata section (if not present) and make current
.bss		emit .bss section (if not present) and make current
.string	"string"	emit string
.asciz	"string"	emit string (alias for .string)

<https://github.com/riscv-non-isa/riscv-asm-manual/blob/master/riscv-asm.md>

Assembly Directives (2)

<code>.macro</code>	<code>name arg1 [, argn]</code>	begin macro definition \argname to substitute
<code>.endm</code>		end macro definition
<code>.type</code>	<code>symbol, @function</code>	accepted for source compatibility
<code>.option</code>	<code>{rvc,norvc,pic,nopic,relax,norelax,push,pop}</code>	RISC-V options. Refer to .option for a more detailed description.
<code>.byte</code>	<code>expression [, expression]*</code>	8-bit comma separated words
<code>.2byte</code>	<code>expression [, expression]*</code>	16-bit comma separated words
<code>.half</code>	<code>expression [, expression]*</code>	16-bit comma separated words
<code>.short</code>	<code>expression [, expression]*</code>	16-bit comma separated words
<code>.4byte</code>	<code>expression [, expression]*</code>	32-bit comma separated words
<code>.word</code>	<code>expression [, expression]*</code>	32-bit comma separated words
<code>.long</code>	<code>expression [, expression]*</code>	32-bit comma separated words
<code>.8byte</code>	<code>expression [, expression]*</code>	64-bit comma separated words
<code>.dword</code>	<code>expression [, expression]*</code>	64-bit comma separated words
<code>.quad</code>	<code>expression [, expression]*</code>	64-bit comma separated words
<code>.dtprelword</code>	<code>expression [, expression]*</code>	32-bit thread local word
<code>.dtpreldword</code>	<code>expression [, expression]*</code>	64-bit thread local word
<code>.sleb128</code>	<code>expression</code>	signed little endian base 128, DWARF
<code>.uleb128</code>	<code>expression</code>	unsigned little endian base 128, DWARF
<code>.p2align</code>	<code>p2,[pad_val=0],max</code>	align to power of 2
<code>.balign</code>	<code>b,[pad_val=0]</code>	byte align
<code>.zero</code>	<code>integer</code>	zero bytes
<code>.variant_cc</code>	<code>symbol_name</code>	annotate the symbol with variant calling convention

n.md

Esempio: Le macro

Una **definizione di macro** (*macro definition*) è un modo per assegnare un nome ad una sequenza di istruzioni.

Dopo aver definito una macro il programmatore può scrivere il nome al posto della sequenza di istruzioni

Per la definizione di una macro occorre:

- un header della macro che indica il nome della macro da definire
- il testo che comprende il corpo della macro
- una “Assembly Directive” che indica la fine della definizione

Esempio: Le macro

Codice in assembly per scambiare 2 registri, usando un terzo registro come appoggio

```
# swap macro
```

```
.macro swap reg1, reg2, reg3
```

```
    add \reg3, \reg2, zero
```

```
    add \reg2, \reg1, zero
```

```
    add \reg1, \reg3, zero
```

```
.endm
```

```
li    s1, 10
```

```
li    s2, 20
```

```
li    s3, 30
```

```
swap  s1, s2, t0
```

```
swap  s2, s3, t0
```

tradotto come

```
add t0, s2, zero
```

```
add s2, s1, zero
```

```
add s1, t0, zero
```

```
add t0, s3, zero
```

```
add s3, s2, zero
```

```
add s2, t0, zero
```

Esempio: Le macro

Confronto tra uso di **macro** e uso di **procedure**.

Item	Macro call	Procedure call
When is the call made?	During assembly	During program execution
Is the body inserted into the object program every place the call is made?	Yes	No
Is a procedure call instruction inserted into the object program and later executed?	No	Yes
Must a return instruction be used after the call is done?	No	Yes
How many copies of the body appear in the object program?	One per macro call	One

ASSEMBLATORE

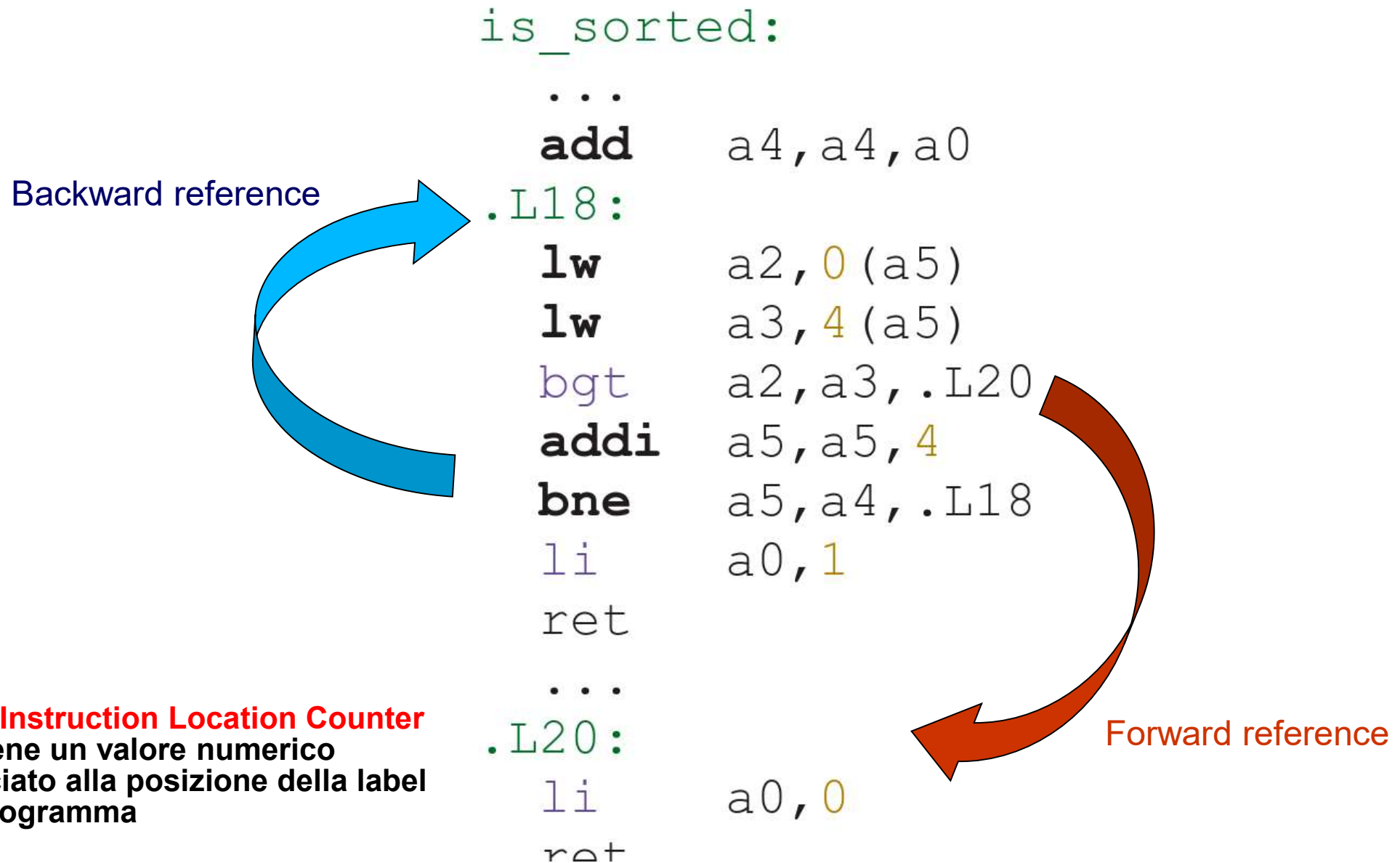
L'assemblatore **traduce** un programma scritto in linguaggio assembly nel corrispondente programma in linguaggio macchina eseguibile.

L'assemblatore legge tutte le istruzioni del programma assembly, ne traduce in linguaggio macchina i codici operativi, i dati e le label, controllandone la correttezza sintattica, e restituisce in output il file "oggetto".

Livello implementato tramite **compilazione** e non **interpretazione**



Dal sorgente al modulo oggetto



ASSEMBLATORE a due PASSI

Problema delle forward reference !

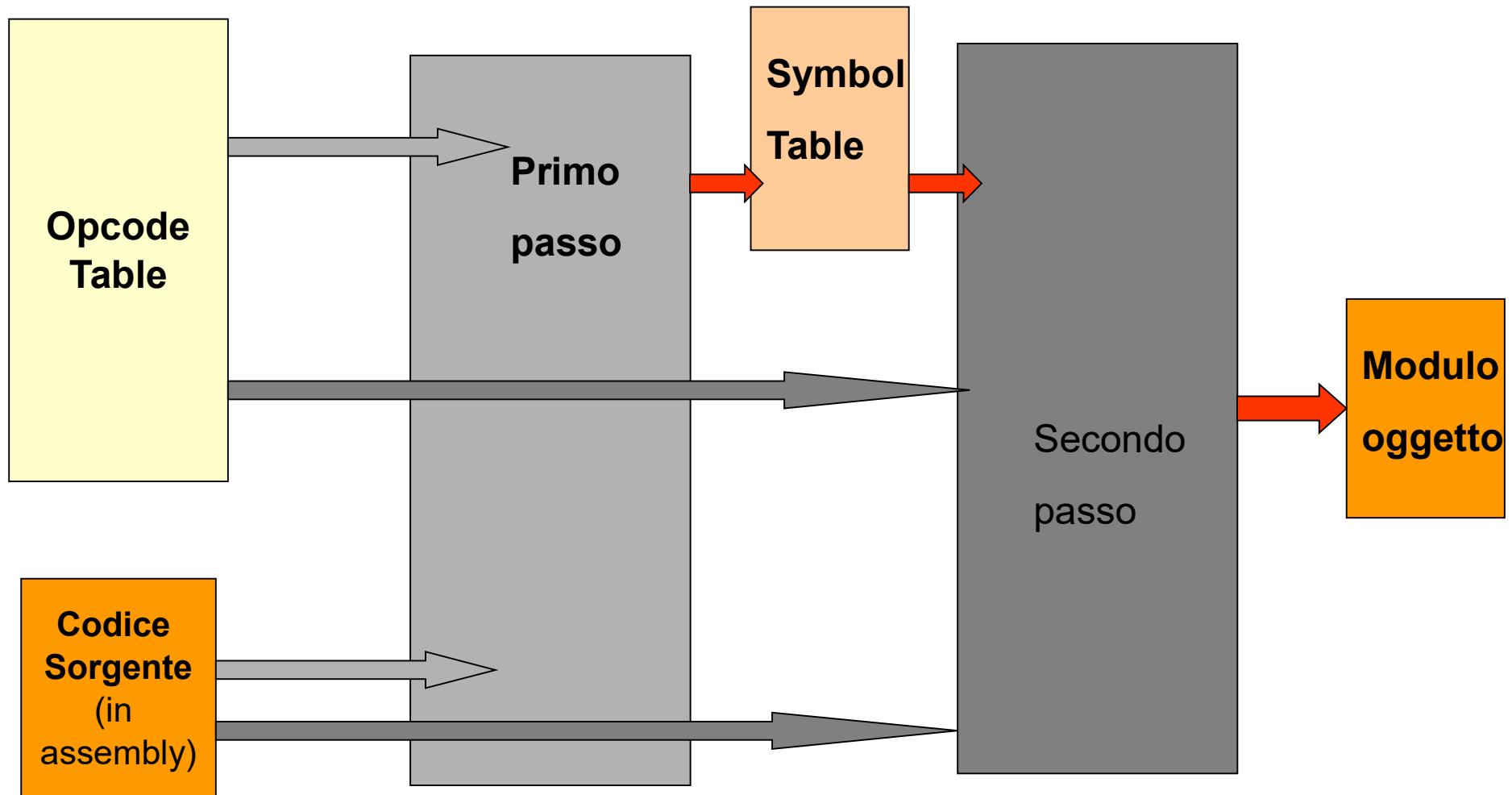
1° Passo :

- individuazione di tutti i **nomi** (le etichette) che compaiono come riferimento simbolico di dati o di istruzioni
- creazione di una **Symbol Table** che contiene le etichette con la loro posizione relativa all'interno del programma

2° Passo :

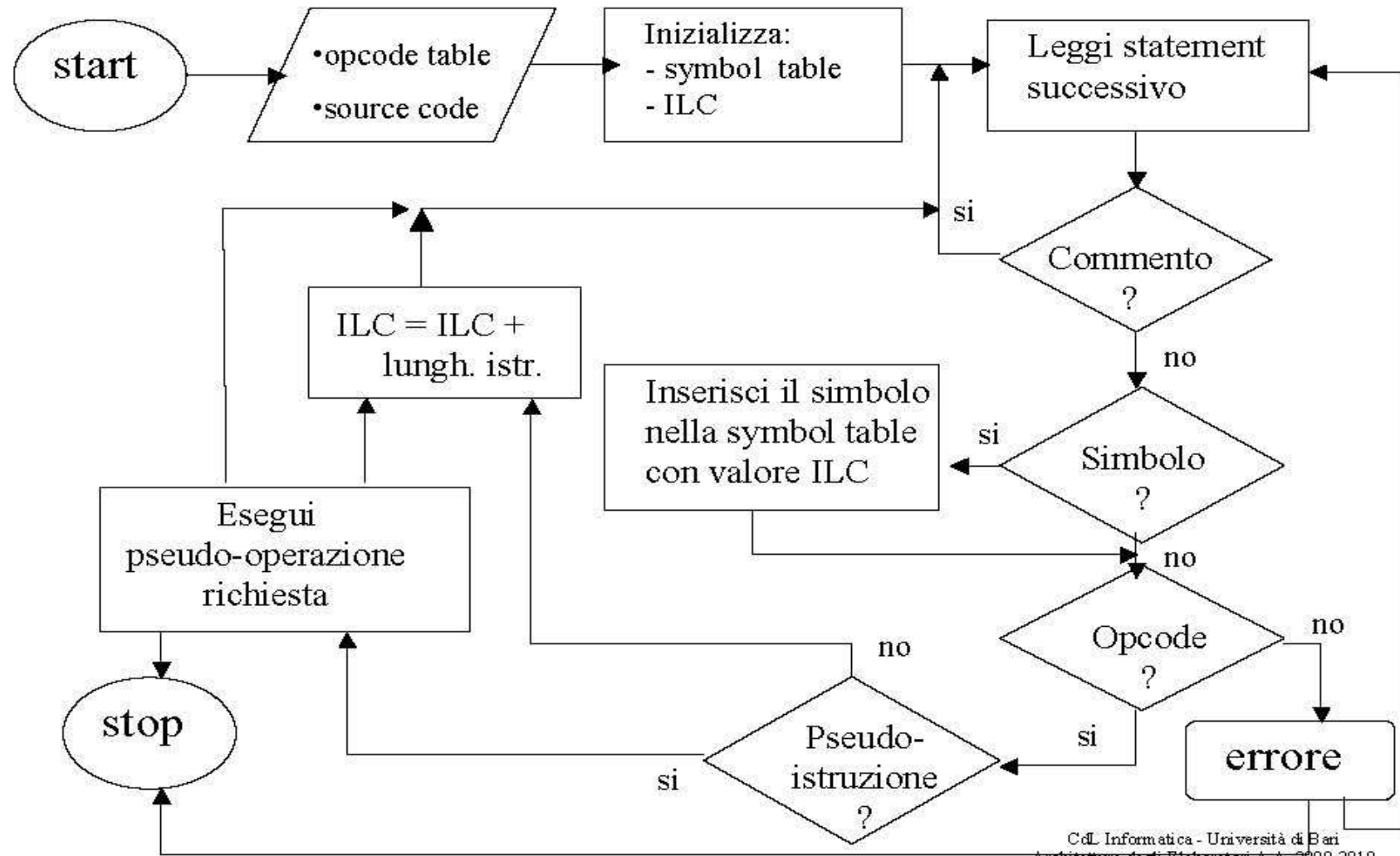
- traduzione dei codici mnemonici delle istruzioni, degli operandi e delle etichette, mediante la consultazione della **Symbol Table** costruita nel 1° passo.

ASSEMBLATORE a due PASSI



Primo passo ASSEMBLATORE

1° passo dell'assemblatore



CdL Informatica - Università di Bari
Architettura degli Elaboratori A.A. 2009-2010

Primo passo ASSEMBLATORE

```
is_sorted:
```

```
...
```

```
add    a4, a4, a0
```

```
.L18:
```

```
lw     a2, 0(a5)
```

```
lw     a3, 4(a5)
```

```
bgt    a2, a3, .L20
```

```
addi   a5, a5, 4
```

```
bne    a5, a4, .L18
```

```
li     a0, 1
```

```
ret
```

```
...
```

```
.L20:
```

```
li     a0, 0
```

```
ret
```

Symbol Table

SIMBOLO	Valore ILC	altro...
....
is_sorted	0x420
.L18	0x432
.L20	0x462
.....

1 istruzione RISC-V = 32 bit

Primo passo ASSEMBLATORE

(altre ISA – esempio x86)

Label	Opcode	Operands	Comments	Length	ILC
MARIA:	MOV	EAX, I	EAX = I	5	100
	MOV	EBX, J	EBX = J	6	105
ROBERTA:	MOV	ECX, K	ECX = K	6	111
	IMUL	EAX, EAX	EAX = I * I	2	117
	IMUL	EBX, EBX	EBX = J * J	3	119
	IMUL	ECX, ECX	ECX = K * K	3	122
MARILYN:	ADD	EAX, EBX	EAX = I * I + J * J	2	125
	ADD	EAX, ECX	EAX = I * I + J * J + K * K	2	127
STEPHANY:	JMP	DONE	branch to DONE	5	129

Il contatore (**ILC**)
 tiene traccia degli
 indirizzi di memoria
 delle istruzioni
 In questo esempio si
 suppone che le
 istruzioni precedenti
 alla etichetta **MARIA**
 occupino 100 bytes.

Symbol Table

Symbol	Value	Other information
MARIA	100	
ROBERTA	111	
MARILYN	125	
STEPHANY	129	

Primo passo ASSEMBLATORE

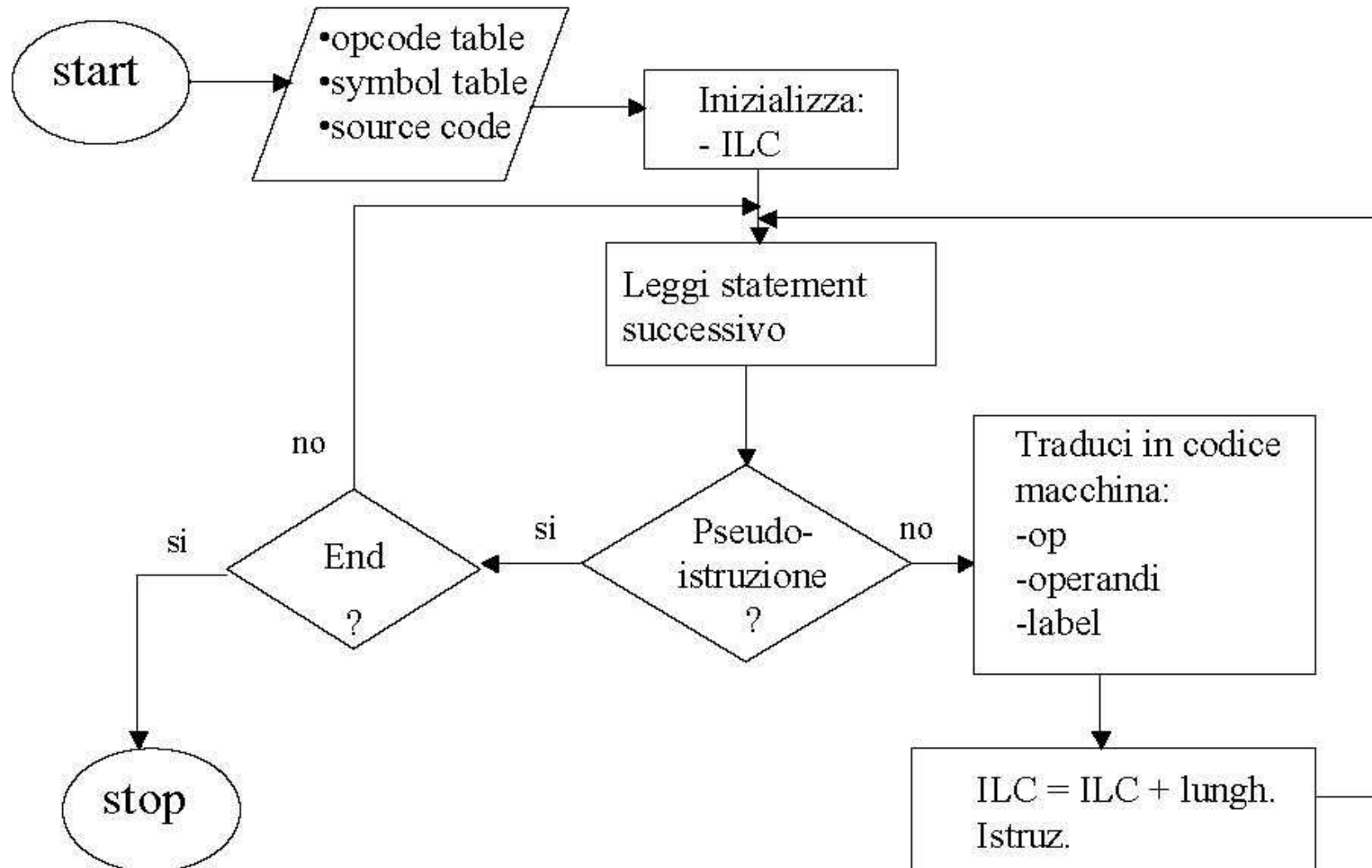
(altre ISA – esempio x86)

Un frammento della opcode table per la x86

Opcode	First operand	Second operand	Hexadecimal opcode	Instruction length	Instruction class
AAA	—	—	37	1	6
ADD	EAX	immed32	05	5	4
ADD	reg	reg	01	2	19
AND	EAX	immed32	25	5	4
AND	reg	reg	21	2	19

Secondo passo ASSEMBLATORE

2° passo dell'assemblatore



ASSEMBLATORE a due PASSI

is_sorted:

```
...  
add    a4, a4, a0  
.L18:  
lw     a2, 0(a5)  
lw     a3, 4(a5)  
bgt    a2, a3, .L20  
addi   a5, a5, 4  
bne    a5, a4, .L18  
li     a0, 1  
ret  
...  
.L20:  
li     a0, 0  
ret
```

ILC

0x432: lw a2, 0(a5)

0x43a: bgt a2, a3, 0x28

0x462: li a0, 0
0x464: ret

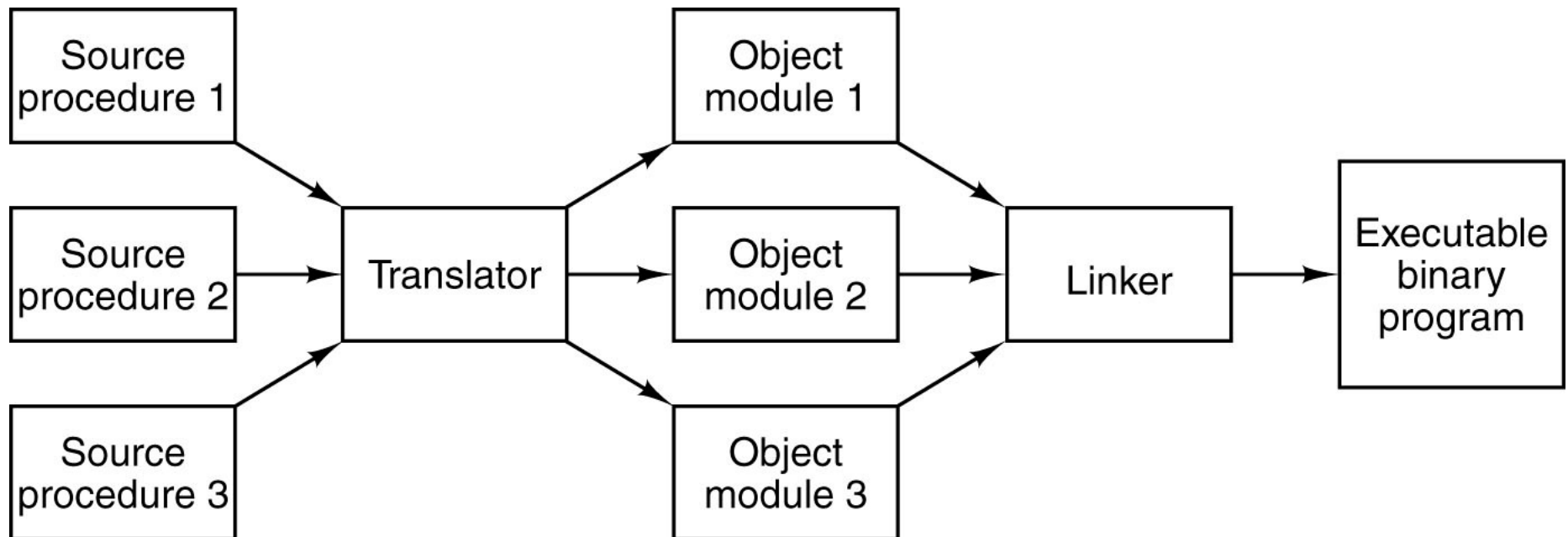
Symbol Table

SIMBOLO	Valore ILC	altro...
....
is_sorted	0x420
.L18	0x432
.L20	0x462
.....

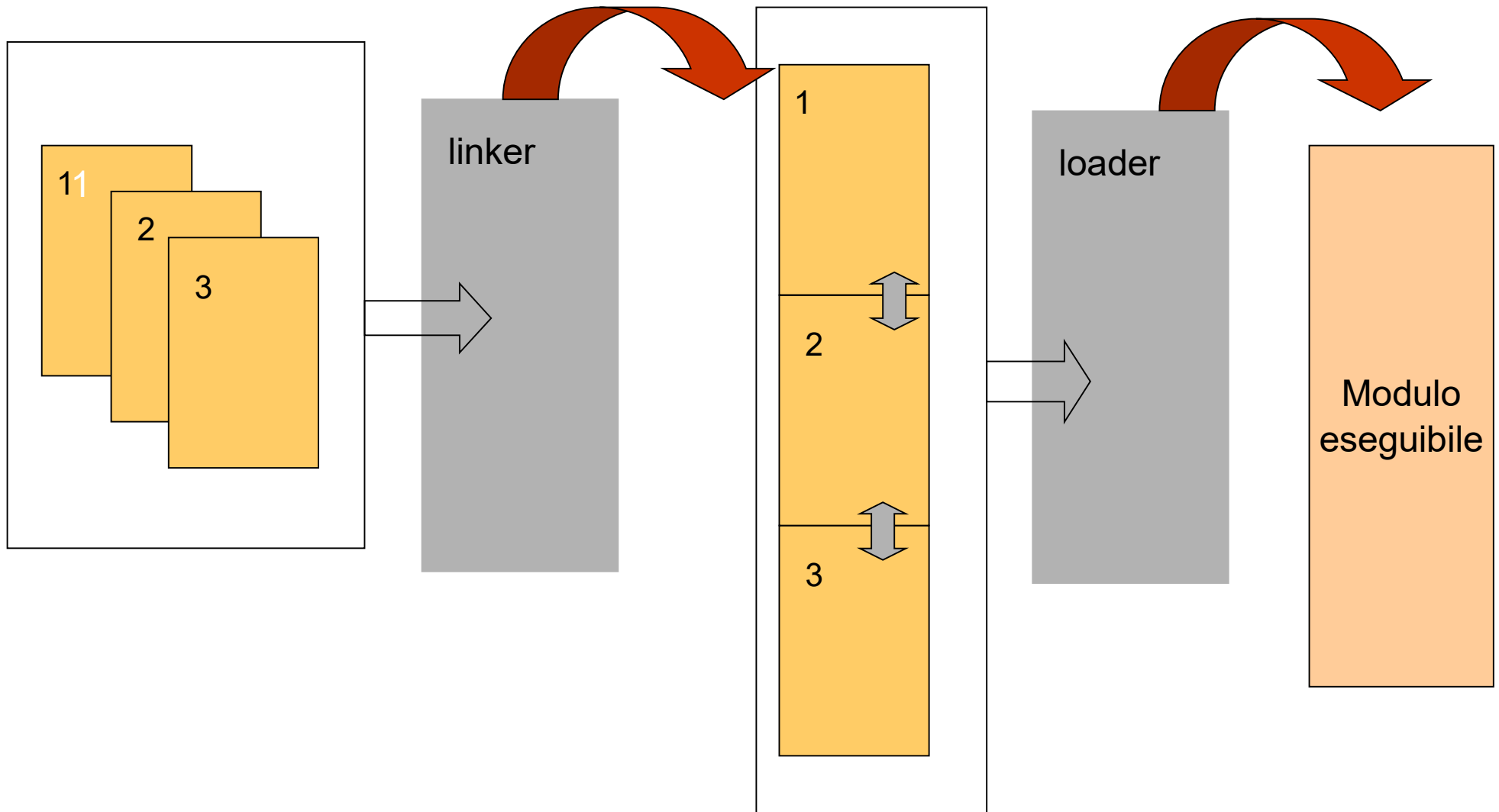
Linking e Loading

Programmi: insieme di procedure (**moduli**) tradotti separatamente dall'assemblatore (o compilatore). Ogni modulo oggetto ha il suo **spazio di indirizzamento** separato

Linker: è un programma che esegue la funzione di **collegamento** dei **moduli oggetto** in modo da formare un unico **modulo eseguibile**

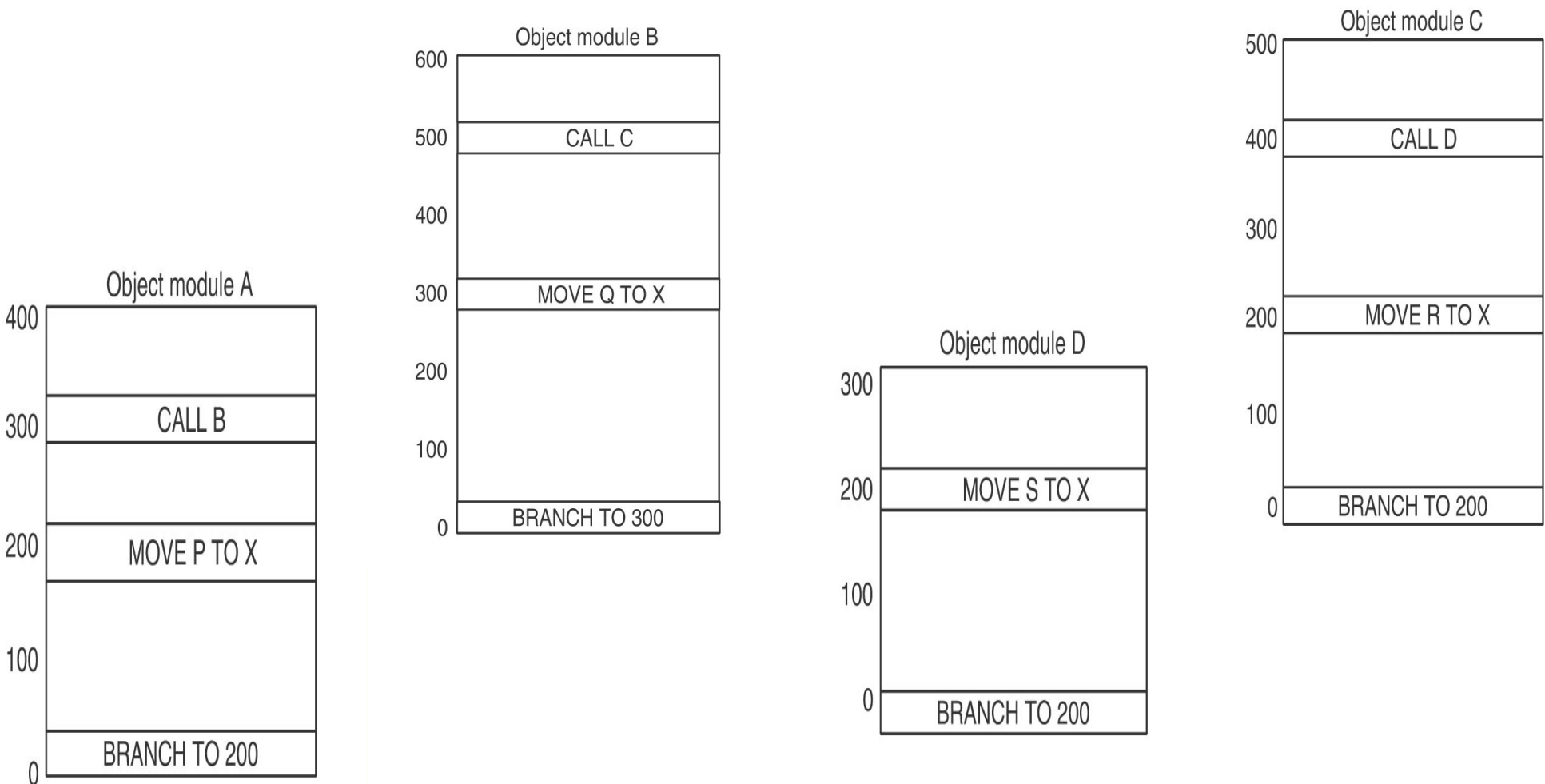


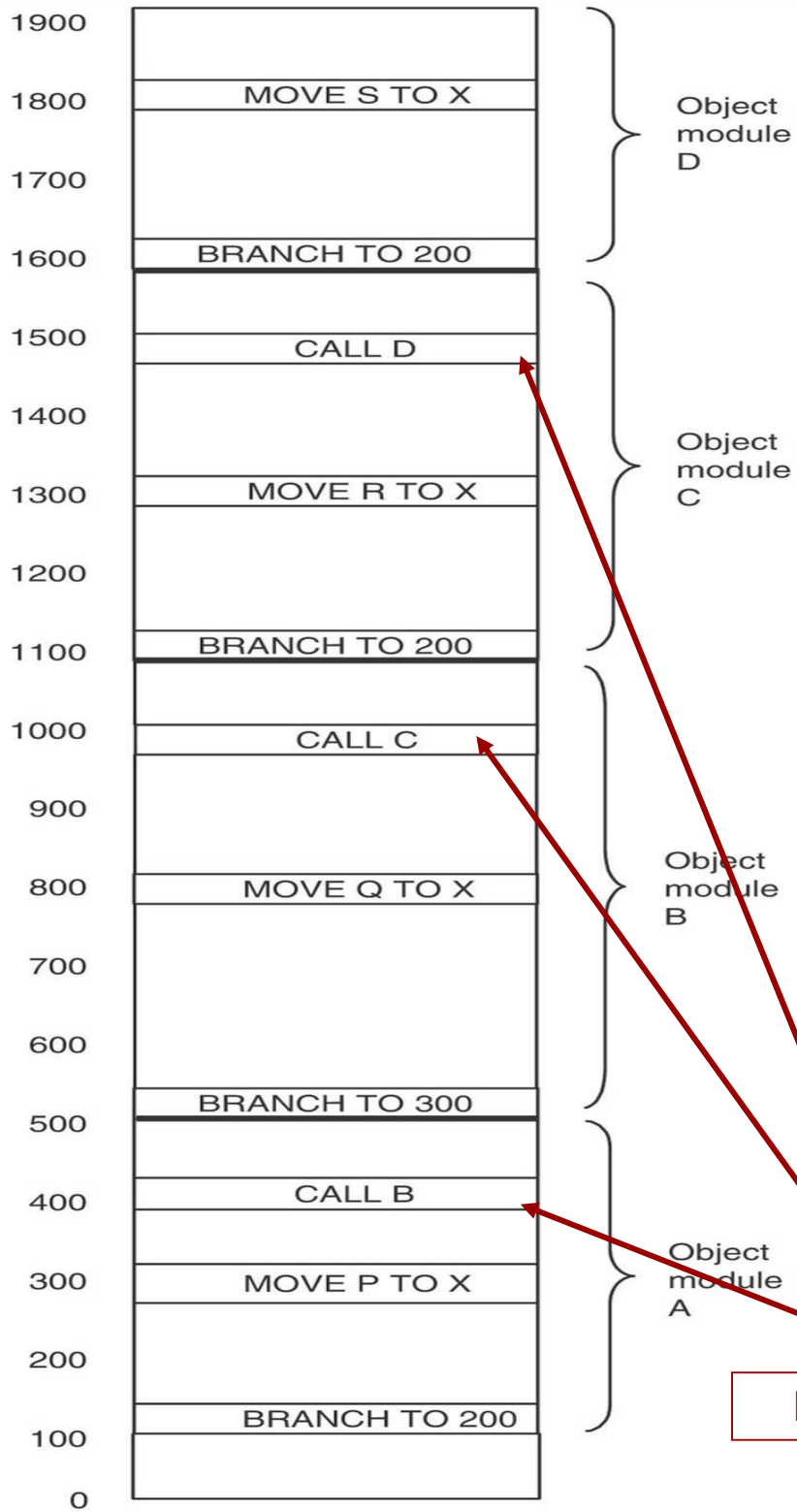
Linking e Loading



Compiti del Linker

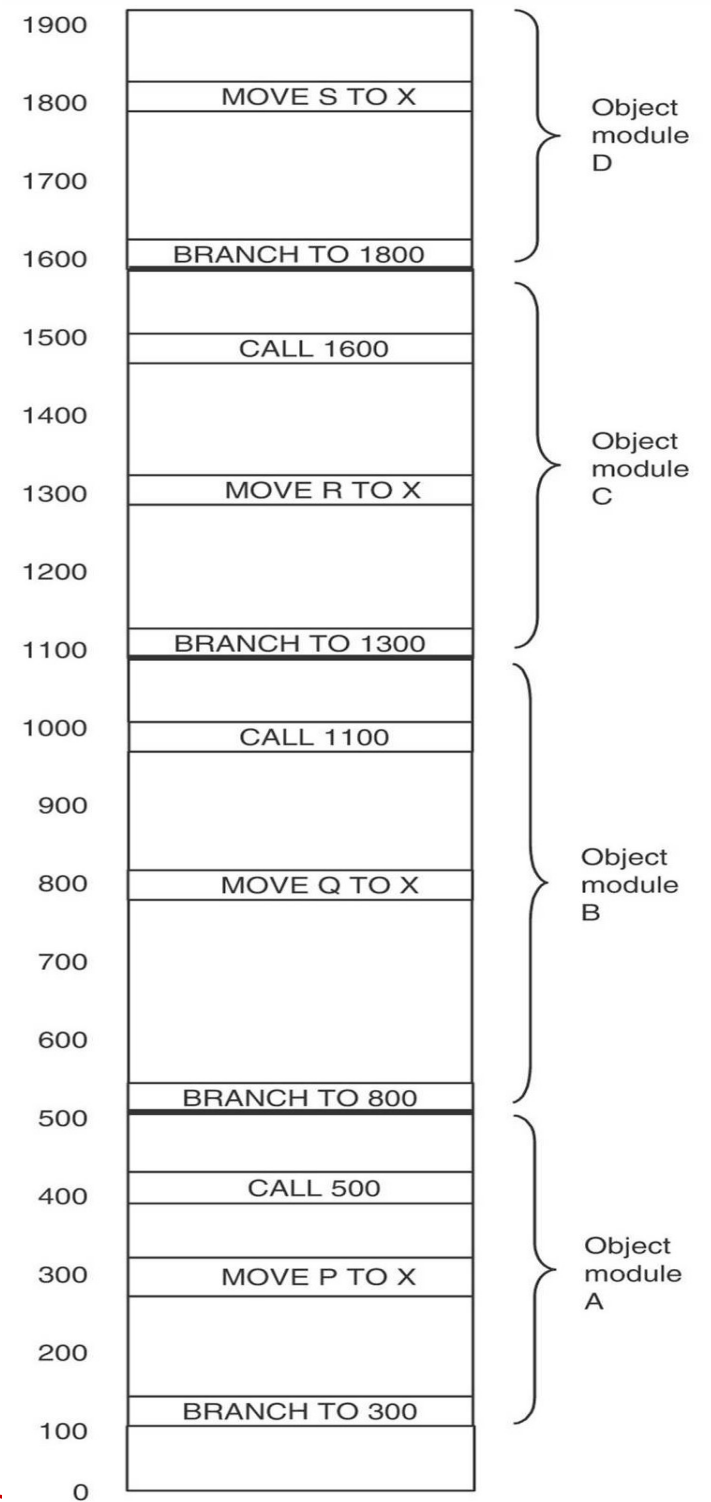
Ogni modulo ha il suo spazio di indirizzamento,
che parte dall'indirizzo 0





External reference

Architettura degli Elabc



Compiti del Linker

Il linker **fonde** gli spazi di indirizzamento separati dei moduli oggetto in uno **spazio lineare unico** nel modo seguente:

- Costruisce una tabella di tutti i moduli oggetto e le loro lunghezze
- Assegna un indirizzo di inizio ad ogni modulo oggetto
- Trova tutte le istruzioni che accedono alla memoria e aggiunge a ciascun indirizzo una **relocation constant** corrispondente all'indirizzo di partenza del suo modulo
- Trova tutte le istruzioni che fanno riferimento ad altri moduli e le aggiorna con l'indirizzo corretto

Modulo	Lunghezza	Indirizzo partenza
A	400	100
B	600	500
C	500	1100
D	300	1600

Linking - Esempio

- **Eseguire il link di due file oggetto**
- **Trovare gli indirizzi aggiornati delle prime istruzioni del file eseguibile completo**
- **Procedure A e B, fornite da 2 moduli distinti**
- **Indirizzi delle parole doppie X e Y**

SP → 0000 003f ffff fff0_{esa}

0000 0000 1000 0000_{esa}





PC → 0000 0000 0040 0000_{esa}

0







**Indirizzi
all'interno di
ogni file oggetto**

File oggetto con Procedura A

	Nome	Procedura A		
	Dimensione del testo	100 _{esa}		 dimensioni
	Dimensione dei dati	20 _{esa}		
Segmento testo	Indirizzo	Istruzione		
	0	ld x10, 0(x3)		
	4	jal x1, 0		
		
Segmento dati	0	(X)		 variabile
		
Informazioni di rilocalizzazione	Indirizzo	Tipo di istruzione	Dipendenza	
	0	ld	X	 dipendenze
	4	jal	B	
Tabella dei simboli	Etichetta	Indirizzo		
	X	—		 tabella dei simboli incompleta
	B	—		

File oggetto con Procedura B

	Nome	Procedura B		
	Dimensione del testo	200 _{esa}	 dimensioni	
	Dimensione dei dati	30 _{esa}		
Segmento testo	Indirizzo	Istruzione		
	0	sd x11, 0 (x3)		
	4	jal x1, 0		
		
Segmento dati	0	(Y)	 variabile	
		
Informazioni di rilocalizzazione	Indirizzo	Tipo di istruzione	Dipendenza	
	0	sd	Y	 dipendenze
	4	jal	A	
Tabella dei simboli	Etichetta	Indirizzo		
	Y	—		 tabella dei simboli incompleta
	A	—		

File eseguibile

Intestazione del file eseguibile		
Dimensione del testo		300 _{esa}
Dimensione dei dati		50 _{esa}
Segmento testo	Indirizzo	Istruzione
indirizzi aggiornati	0000 0000 0040 0000 _{esa}	ld x10, 0(x3)
	0000 0000 0040 0004 _{esa}	jal x1, 252 _{dec}

	0000 0000 0040 0100 _{esa}	sd x11, 32(x3)
	0000 0000 0040 0104 _{esa}	jal x1, -260 _{dec}

Segmento dati	Indirizzo	
indirizzi delle variabili	0000 0000 1000 0000 _{esa}	(X)

	0000 0000 1000 0020 _{esa}	(Y)

dimensione aggiornata

File eseguibile

Intestazione del file eseguibile		
Dimensione del testo		300 _{esa}
Dimensione dei dati		50 _{esa}
Segmento testo	Indirizzo	Istruzione
	0000 0000 0040 0000 _{esa}	ld x10, 0(x3)
	0000 0000 0040 0004 _{esa}	jal x1, 252 _{dec}

	0000 0000 0040 0100 _{esa}	sd x11, 32(x3)
	0000 0000 0040 0104 _{esa}	jal x1, -260 _{dec}



jal alla
procedura B

- jal utilizza l'indirizzamento relativo al PC
- salta all'indirizzo 40 0100_{esa} (l'indirizzo della procedura B)
- **40 0100_{esa} – 40 0004_{esa} = 252_{dec}**

File eseguibile

Intestazione del file eseguibile		
Dimensione del testo		300 _{esa}
Dimensione dei dati		50 _{esa}
Segmento testo	Indirizzo	Istruzione
	0000 0000 0040 0000 _{esa}	ld x10, 0(x3)
	0000 0000 0040 0004 _{esa}	jal x1, 252 _{dec}

	0000 0000 0040 0100 _{esa}	sd x11, 32(x3)
	0000 0000 0040 0104 _{esa}	jal x1, -260 _{dec}



jal alla
procedura A

- jal utilizza l'indirizzamento relativo al PC
- salta all'indirizzo 40 0000_{esa} (l'indirizzo della procedura A)
- $40\ 0000_{\text{esa}} - 40\ 0104_{\text{esa}} = -260_{\text{dec}}$

File eseguibile

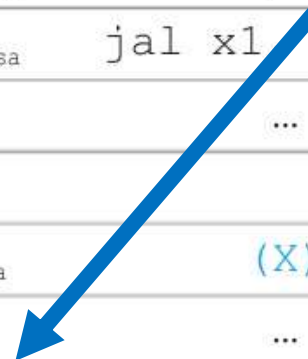
Intestazione del file eseguibile		
Dimensione del testo		300 _{esa}
Dimensione dei dati		50 _{esa}
Segmento testo	Indirizzo	Istruzione
	0000 0000 0040 0000 _{esa}	ld x10, 0(x3)
	0000 0000 0040 0004 _{esa}	jal x1, 252 _{dec}

	0000 0000 0040 0100 _{esa}	sd x11, 32(x3)
	0000 0000 0040 0104 _{esa}	jal x1, -260 _{dec}

Segmento dati	Indirizzo	
	0000 0000 1000 0000 _{esa}	(X)

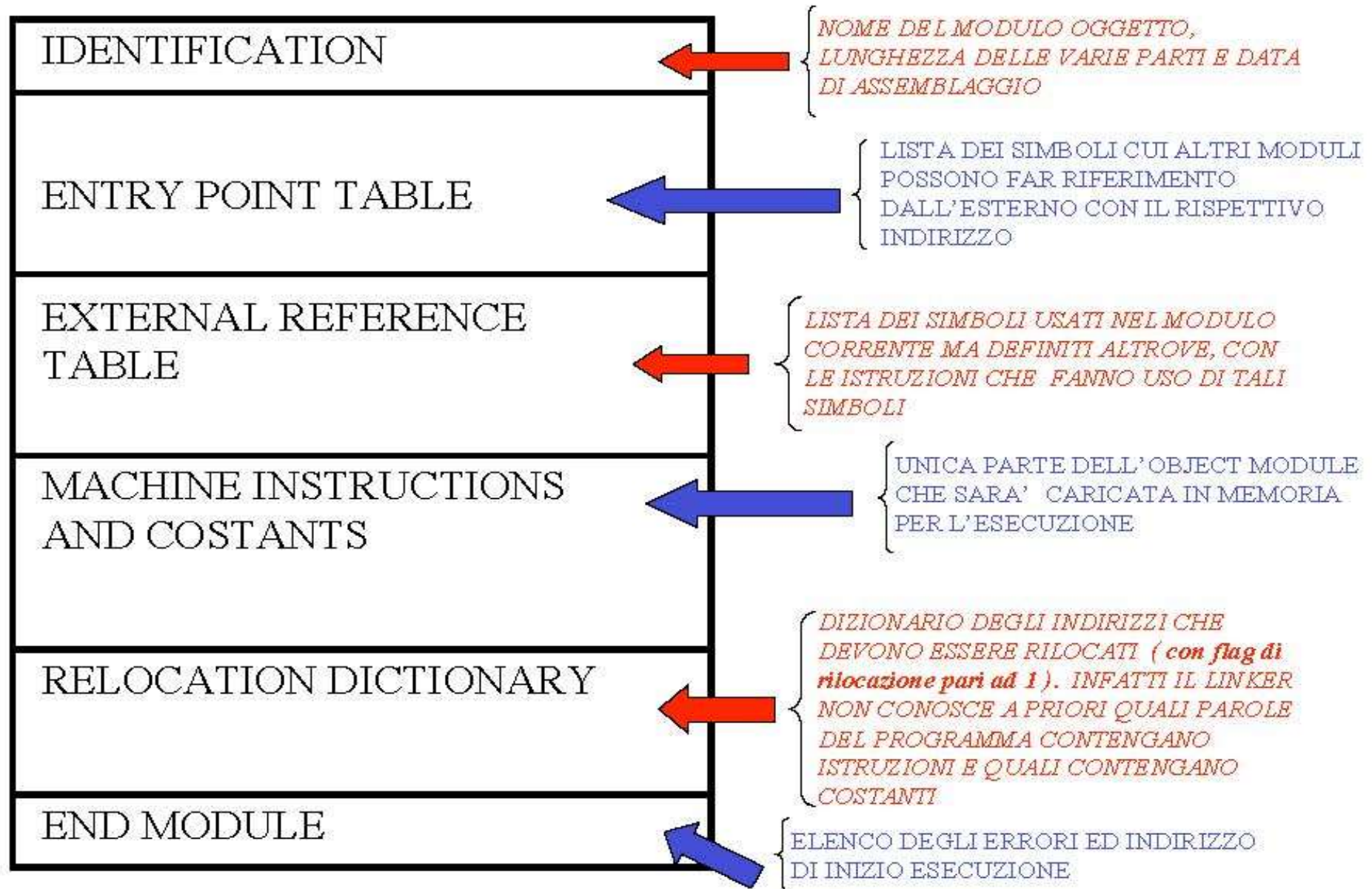
	0000 0000 1000 0020 _{esa}	(Y)

indirizzo di Y
aggiornato



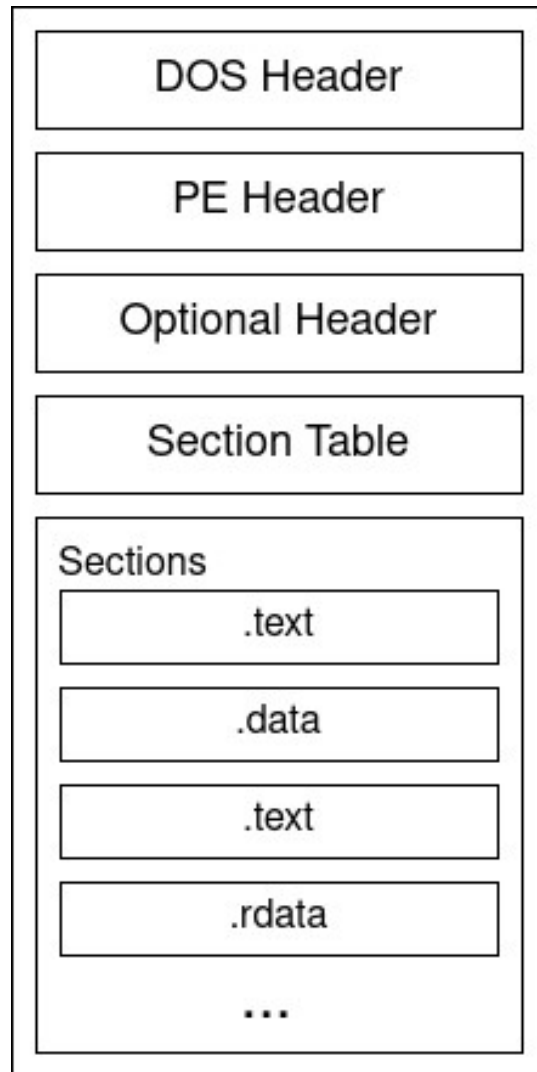
registro base x3, supponiamo abbia 0000 0000 1000 0000_{esa}

Struttura modulo oggetto

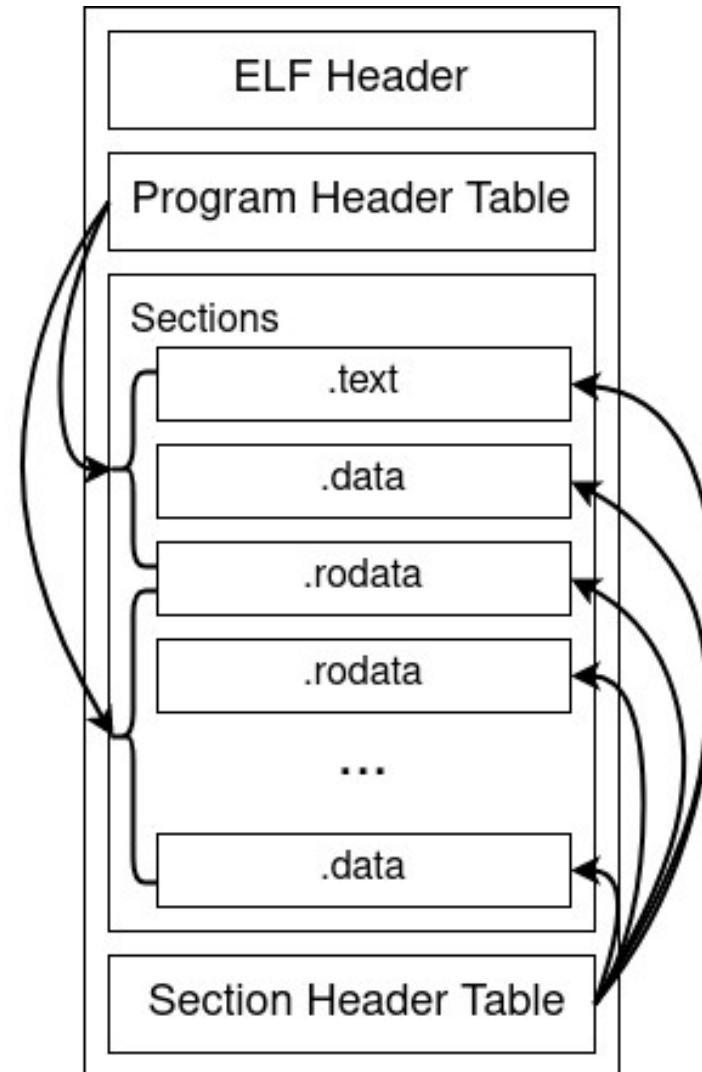


Formato Binario

Windows PE



ELF



ELF- File oggetto nei sistemi UNIX

- **object file header:** descrive la dimensione e la posizione degli altri segmenti del file oggetto stesso
- **text segment:** contiene il codice in linguaggio macchina
- **static data segment:** contiene i dati allocati per tutta la durata del programma - sia dati statici che dinamici (che possono crescere o diminuire di dimensione)

ELF- File oggetto nei sistemi UNIX

- **relocation information**: identificano le istruzioni e i dati che, quando il programma è posto in memoria, dipendono da indirizzi assoluti
- **symbol table**: contiene le etichette di cui non è stata trovata una definizione (e.g., moduli esterni)
- **debugging information**: informazioni per il debugger, che permette di associare le istruzioni in linguaggio macchina al codice sorgente C

LOADER

- Una volta creato l'eseguibile (ad opera del linker) esso viene memorizzato su un supporto di memoria secondaria
- Al momento dell'esecuzione il sistema operativo lo carica in memoria centrale e ne avvia l'esecuzione
- Il loader (che è un programma del sistema operativo) si occupa di:
 1. Leggere l'intestazione per determinare la dimensione del programma e dei dati
 2. Riservare uno spazio in memoria sufficiente per contenerli
 3. Copiare programma e dati nello spazio riservato
 4. Copiare nello stack i parametri (se presenti) passati al main
 5. Inizializzare tutti i registri e lo stack pointer (ma anche gli altri del modello di memoria)
 6. Saltare ad una procedura che copia i parametri dallo stack ai registri e che poi invoca il main

Binding e rilocalizzazione dinamica

Se si **spostano** in memoria programmi per cui è già stato fatto il collegamento e il calcolo degli indirizzi, tutti gli **indirizzi** di memoria risultano sbagliati e le informazioni di rilocalizzazioni sono state scartate da tempo. Il problema di spostare in memoria programmi è connesso con la scelta del momento in cui effettuare il collegamento (**binding**) tra nomi simbolici e indirizzi fisici (**binding time**).

Quando fare il collegamento? Ci sono alcune scelte possibili:

- Al momento della **scrittura** del programma
- Al momento della **traduzione** del programma
- Al momento del **linking** (ma prima del loading)
- Al momento del **loading**
- Al momento dell'**esecuzione** (uso di un registro di base)

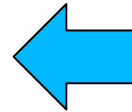
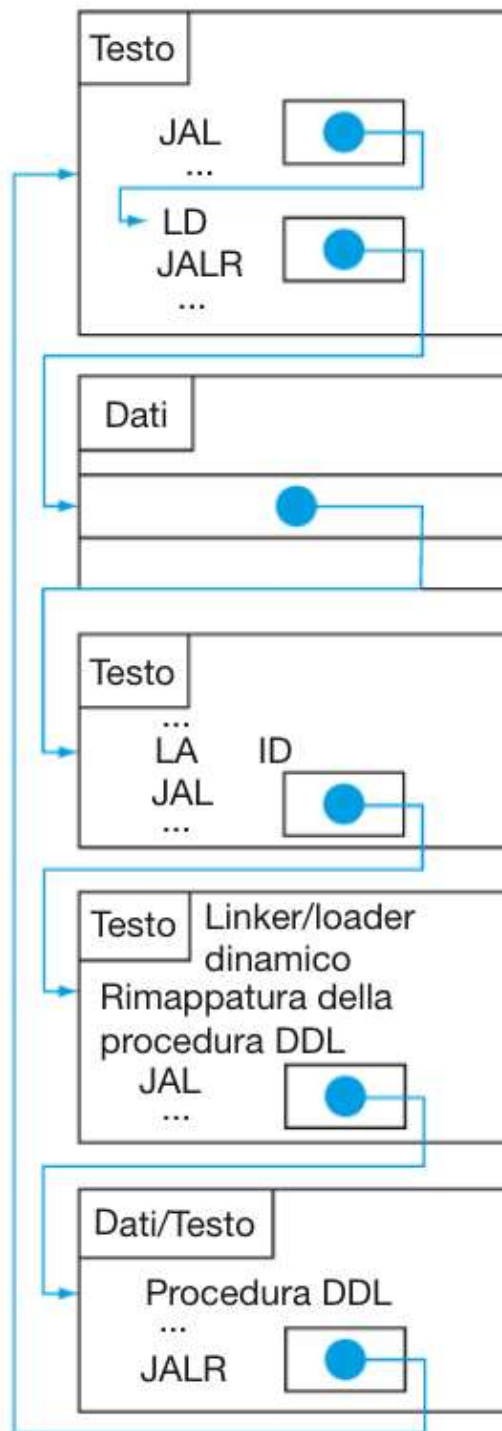
Binding e rilocalizzazione dinamica

Collegamento statico:

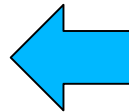
- **Le funzioni di libreria diventano parte del codice eseguibile**
- Se viene rilasciata una nuova versione, un programma che carica staticamente le librerie continua a utilizzare la vecchia versione
- La libreria può essere molto più grande del programma; i file binari diventano eccessivamente grossi

Collegamento dinamico:

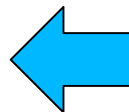
- DLL, Dynamically Linked Libraries
- Le funzioni di libreria non vengono collegate e caricate finché non si inizia l'esecuzione del programma
- DLL con collegamento lazy: ogni procedura viene caricata solo dopo la sua prima chiamata



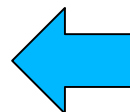
Programma che utilizza una libreria dinamica



Dummy: contiene un'istruzione di salto indiretto che porta al linker/loader dinamico (Sistema Operativo)

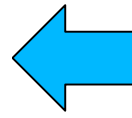
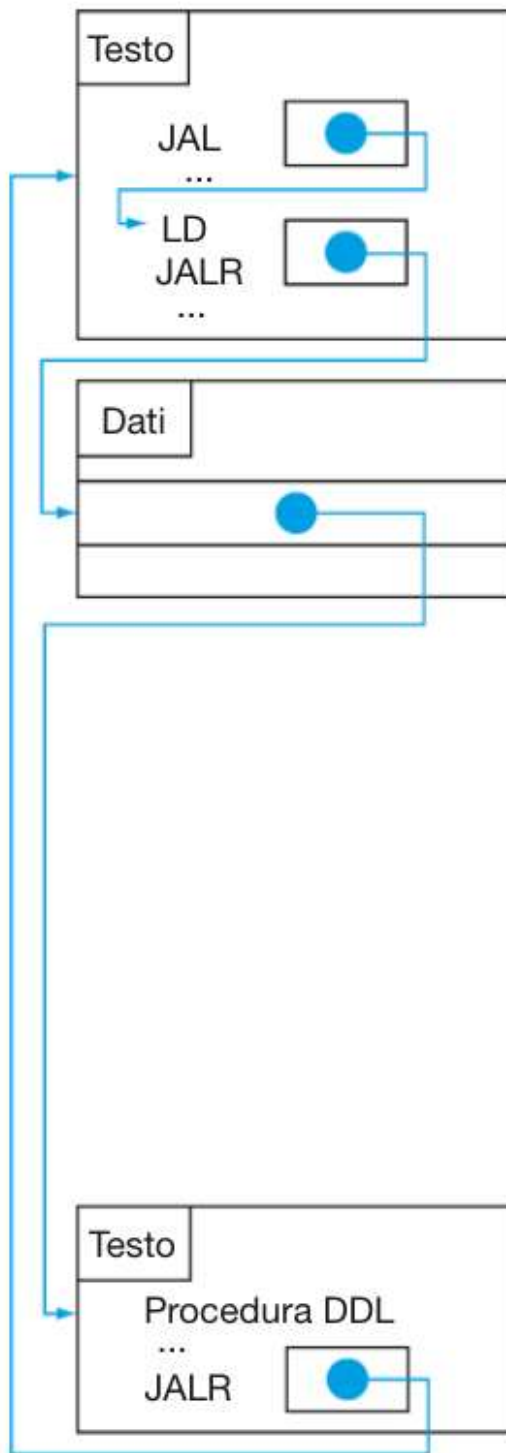


Il linker-loader dinamico trova la funzione di libreria, la rimappa in memoria e cambia l'indirizzo contenuto nell'istruzione di salto indiretto (sopra)



La funzione è adesso disponibile e si ritornerà al punto originale di chiamata una volta completata

(a) Prima chiamata a una procedura di una DDL



In seguito la chiamata alla funzione di libreria prevederà un salto indiretto unico alla sua prima istruzione, senza salti addizionali

(b) Chiamate successive alla procedura della DDL


```
#include <stdio.h>
```

```
int main(void) {
```

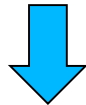
```
...
```

```
printf("%d\n", is_sorted(v, s));
```



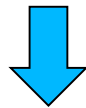
```
49c: 00000517
4a0: 05c50513
4a4: f0dff0ef
4a8: 45a9
```

```
auipc  a0,0x0
addi   a0,a0,92 # 4f8 <main+0x8e>
jal    ra,3b0 <printf@plt>
li     a1,10
```



```
000000000000003b0 <printf@plt>:
3b0: 00002e17
3b4: c68e3e03
3b8: 000e0367
3bc: 00000013
```

```
auipc  t3,0x2
ld     t3,-920(t3) # 2018 <printf@GLIBC_2.27>
jalr   t1,t3
nop
```



```
selection.o:          file format elf64-littleriscv
```

```
DYNAMIC SYMBOL TABLE:
```

```
000000000000003c0 l
0000000000000000
0000000000000000
```

```
d  .text 0000000000000000 .text
DF *UND* 0000000000000000 GLIBC_2.27 putchar
DF *UND* 0000000000000000 GLIBC_2.27 printf
```

```
riscv64-unknown-elf-objdump --reloc -T selection.o
```