

Lezione 11: Interfacciamento del software verso il RISC-V

• Cicli FOR, WHILE, DO..WHILE *Come rappresentare gli statement, i cicli*

Statement 1, viene eseguito sempre, comunque

for (k=0; k<100; ++k) { ... } ...

ST1 ST2 ST3 ST4 ST5

Statement iniziale in Modello UNIVERSALE per For

INIZIALE a k associato registro x8

add x8, x0, x0 **K=0** ST1 Sempre

ST2 CONTROLLO

ini_for:
 slt x5, x8, x10
 beq x5, x0, fin_for

eseguito sempre **ST2** controllo condizione

Corpo For

FINALE

addi x8, x8, 1 ++k

ST3

viene eseguito dopo il corpo del for

SALTO INCONDIZIONATO

beq x0, x0, ini_for

fin_for:

ST5

Cosa si trova

dopo il ciclo for, se non viene eseguito ciclo for, esegui

lo ST5, salto a fin_for

while (k<100) { ... } ...

ST1 ST2 ST3

CONDIZ dello WHILE eseguito se la condiz. è verificata

k è in x8, x100 variabile

ini_wh: come For

slt x5, x8, x10

beq x5, x0, fin_wh

ST1

ST2

beq x0, x0, ini_wh

Salto incondizionato Fine ciclo,

fin_wh:

ST3

do { ... } while (k<100); ...

ST1 ST2 ST3

do esegue sempre ST1

inizia all'inizio

ini_do:

...

ST1

corpo

slt x5, x8, x10

verifica, controllo condizione

bne x5, x0, ini_do

Se condiz. vera, x1=1

ST2

...

ripeti il ciclo

ST3

instr. Successiva allo branch

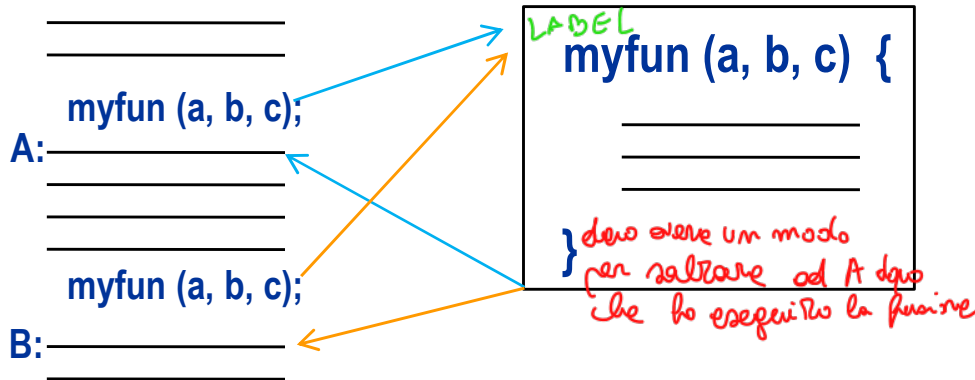
Se x5=1 cioè k<100, allora ritorna a ini_do, altrimenti continua

Chiamata a funzione

Raggruppo codice in una funzione che può essere rivista più volte

Patterson: 2.8,2.13,2.14

- In un programma C, la funzione rappresenta una porzione di codice richiamabile in uno o più punti del programma a, b, c parametri.



- La chiamata deve prevedere un meccanismo per saltare al codice della funzione + un meccanismo per tornare al chiamante

Nota: l'indirizzo di "myfun" è lo stesso nei due casi ma la funzione deve essere in grado di tornare in due punti diversi del programma (A e B)

→ La chiamata a funzione quindi

branch dovrebbe saltare una volta al punto A, e poi al punto B
 branch nella myfun non sa dove tornare

NON PUO' essere implementata con una semplice bne/beq → OCCORRE:

- memorizzare da qualche parte il punto di ritorno, es. $x1 = \&A$
 (prima di saltare)
 X1 usato per memorizzare indirizzo di ritorno della funzione
- saltare all'inizio della funzione "myfun" e al termine saltare a (x1)
 (indirizzo contenuto in x1)

PRIMA DI SALTARE, memorizzo punto successivo in cui devo saltare dopo la F, dove ritornare.

- Nel RISC-V si usa quindi la pseudo-istruzione **"jump and link"** o jal:
 $jal\ label\ 2op\ \# \ x1 \leftarrow PC+4; \ PC \leftarrow PC + OFFSET(\&label - PC)$
 LINK: memorizzo, collego l'indirizzo di ritorno
 Come BRANCH, JUMP, salto a MYFUN
 AGGIUNGO IL PROGRAM COUNTER
 jump = salto

→ salta a label (es. myfun) mettendo l'indirizzo di ritorno (es. A oppure B) nel registro x1. Nota: l'istruzione reale è jal x1, OFFSET (v. slide succ.)

X1: registro usato per memorizzare indirizzo di ritorno del codice

- Per tornare indietro uso x1 tramite la pseudo-istruzione "return" o ret:

ret # PC ← x1
 Pseudo istruzione di ritorno
 Preleva X1 e lo mette nel PC

Serve supporto HW per salto a 2 destinazioni
 PC+4 indirizzo istruzione successiva a jal

Istruzione "JAL rd,offset" e il formato J nuovo e ultimo formato

Come memorizzato LABEL? codifico con offset.

- 'JAL label' viene implementata con JAL rd,offset che è una istruzione che **memorizza nel registro generico rd l'indirizzo PC+4 mentre somma offset a PC** si spera offset non sia lontano piccolo
- Quindi 'JAL label' è un modo più rapido per scrivere 'JAL x1,offset' dove **offset=&label-PC**; per avere un intervallo di salto ampio uso il **formato 'J'**: simile a Q
- Il formato J è una variante del formato U in cui, sfruttando bene i 20 bit dell'immediato, posso coprire un offset di 512ki istruzioni* 21 bit istruzioni, uso solo indirizzi pari
- similmente alla BEQ i 20 bit indicano un offset in byte allineato alla half-word (quindi è come se avessi 21 bit in cui il bit0 è 0); contando 4B per istruz. -> 512ki istruz.

FORMATO U (già visto e usato per la LUI) 20 bit per codificare nuovo registro destino.

0000 0000 0001 0000 0000 0000 00001 0110011

es. LUI x1,0x00100
imm20

Potrebbe essere 256 ki di avanti o 256 ki di indietro

imm20 [19:0] come 20 bit per offset ma sparpagliati; vantaggio per implementazione, Rete più semplice e veloce nell'implementazione HW, risparmio energia

FORMATO J (anche chiamato FORMATO UJ) Solo avanti o indietro
0010 0000 0000 0000 0000 0000 00001 1101111

es. JAL 0x00200
imm21

OPCODE 6F

3 bit più successivo per vedere se solo avanti o indietro
[imm21[20], imm21[10:1], imm21[11], imm21[19:12]]

Completivamente sono i bit 20:1 di imm21, sparpagliati opportunamente all'interno dei 20 bit dell'istruzione

imm21 = 0x00200 = (in base2) 0 0000 0000 0010 0000 0000

SALTI A ISTRUZIONE SONO SEMPRE ALLINEATI A MULTIPLI DI 2 BYTE

Il bit meno significativo è sempre zero, quindi non occorre codificarlo

offset più a sono 20 bit
20 19:12 11 10:1

0010 0000 0000 0000 0000

passo scordato, indirizzi sempre a multipli di 2

- Il registro rd codifica l'indice del registro x1 (ovvero 00001); x1 conterrà PC+4 (punti «A» e «B» della slide precedente) Massimo salto

*Nota: 256 ki di avanti e 256 ki di indietro
512ki=512*1024=524288 istruzioni
o 1 Mb avanti o 1 Mb indietro

21 bit per istruz. ogni istruz. 4 byte
512 ki di istruzioni
 $2^{18} = 2^{10} \cdot 2^8$

Pseudo-istruzione JAL, istruzione JAL, istruzione JALR

- Abbiamo quindi due possibili sintassi per JAL:

• **JAL label** → denota la **pseudo-istruzione** (basata su JAL (X_1) $ra, offset$)
• **JAL $rd, offset$** → denota una **ISTRUZIONE** dove $rd \in \{x0, \dots, x31\}$ ora X_1 , riservato per le chiamate a funzione
 $label - PC$

- Normalmente noi faremo riferimento alla pseudo-istruzione JAL label

Se riesco a scrivere tutto codice C con Beq e Bne
BEQ/BNE e JAL permettono di generare codice «PIC» (Position Independent Code) che può essere caricato a qualsiasi indirizzo ovunque nella memoria, spostarlo dove voglio, non devo mai scrivere ad indirizzi fissi
MAGGIORE PORTABILITÀ

- Per usare offset di ampiezza maggiore di 2^{20} (e quindi poter chiamare funzioni o fare salti a qualsiasi indirizzo) occorre far riferimento a un registro → istruzione JALR Se codice molto grande? allora posso saltare ad un indirizzo in un registro di 264 bit di ampiezza.

• **JALR $rd, offset(rs1)$** → $x[rd] = PC + 4$; $PC = x[rs1] + sext(offset) \& \sim 1$
che tiene 64 bit
• «**sext(offset)**» denota che offset viene esteso di segno a 64 bit
• « **$\& \sim 1$** » denota che il bit 0 viene azzerato
ultimo bit negativo **MASCHERA.**

Nota: la JALR viene codificata usando il formato I

Perché ho un offset

Nota2: la JALR ha bisogno di precaricare nel registro $x[rs1]$ l'indirizzo a cui saltare

JUMP RELATIVE RS1

- Con JALR $x0, 0(rs1)$ posso supportare la pseudo-istruzione **JR $rs1$** per fare un salto a un indirizzo qualsiasi (contenuto in $rs1$) Solo all'indirizzo contenuto in RS1
- Con JALR $x0, 0(ra)$ posso supportare la pseudo-istruzione **RET** per realizzare il ritorno al chiamante, al termine di una funzione contenuto in RS1

registro sempre dovrà contenere l'indirizzo del chiamante

1 = 000...01

~1 = 111...0

Prima parte viene mantenuta, messo a zero il bit meno sign.

AND

XXXXX...X
XX...0

allo bit meno sign.

JAL LABEL

PSEUDO ISTR.

RET

pseudo

salvo in

salvo a PC + offset.

Formato J

JAL LABEL → JAL X_1 , offset

X_1 registro

RET → JALR X_0 , $O(X_1)$ ← Formato I

↑

salvo all'indirizzo X_1

Salvo PC + 4 in " X_0 " da
valore zero, e
salvo ad X_1 con offset
di 0

non mi
interessa
cosa va qui dentro,
l'ho letto

è un salto ad un indirizzo qualunque contenuto in un registro

quando faccio una jal in una funzione,
implicitamente sto salvando

$RA = X_1 = PC + 4$, che ridimensiono
facendo un $RET = JALR X_0, O(RA)$

va quindi salvato ra quando invito chiamante
a fr. e poi ripristinato

salto

JAL LABEL = JAL X_1 , OFFSET

salvo in X_0
l'indir. ritorno

salvo in X_1 il ra
 $X_1 = PC + 4$

salto a
 $PC = PC + OFFSET$
dove
 $OFFSET = LABEL - PC$

RET = JALR X_0 , $O(X_1)$

salto al contenuto X_1

PC = X_1

con JALR salto
all'indirizzo
contenuto in
un registro

questioni: istruzioni

Ricapitolazione dei formati e istruzioni fin qui viste

opcode e val sulla stessa posizione

31	30	25	24	21	20	19	15	14	12	11	8	7	6	0				
funct7				rs2			rs1		funct3		rd			opcode		R-type		
imm[11:0]						rs1		funct3		rd			opcode		I-type			
imm[11:5]				rs2			rs1		funct3		imm[4:0]			opcode		S-type		
imm[12]		imm[10:5]			rs2			rs1		funct3		imm[4:1]		imm[11]		opcode		B-type
imm[31:12]										rd			opcode		U-type			
imm[20]		imm[10:1]			imm[11]		imm[19:12]			rd			opcode		J-type			

Istruzione	Significato	Variante	Significato
add x5,x6,x7	$x5 = x6 + x7$ <i>Somma</i>	addi x5,x6,10	$x5 = x6 + 10$ <i>immediato</i>
sub x5,x6,x7	$x5 = x6 - x7$ <i>Sottr.</i>		
slt x5,x6,x7	$x5 = (x6 < x7) ? 1 : 0$ <i>confronto se minore</i>	slti x5,x6,10	$x5 = (x6 < 10) ? 1 : 0$
lw x5,100(x6)	$x5 = \text{Memory}[x6+100]$ (32 bit)	ld x5,100(x6)	legge 64 bit
sw x5,100(x6)	$\text{Memory}[x6+100] = x5$ (32 bit)	sd x5,100(x6)	scrive 64 bit
bne x6,x7,L	salta a L se $x6 \neq x7$ (L si trova a offset imm13=(&L-PC))		
beq x6,x7,L	salta a L se $x5 == x7$ (L si trova a offset imm13=(&L-PC))		
lui x5,imm20	carica imm20 nei 20 bit alti <i>caricare parte alta di un registro</i>		
jal FUN	• esegue la funzione FUN (che si trova a offset imm21=(&FUN-PC))		
ret	• ritorna da funzione (jalr x0,0(x1))		

Regole di allocazione dei registri o ABI (Application Binary Interface)

CONVENZIONE, uso
dei registri
de abbiamo

Ra = Return Address
sp = Stack pointer

Num. registro	Scopo	Nome	invariato su "call"? ^{JAR}
x0	la costante 0	zero	SI
x1	return address (il chiamato DEVE salvarlo**) <i>indirizzo ritorno funzione</i>	ra <i>return address</i>	SI
x2	stack pointer (il chiamato DEVE salvarlo**) <i>indirizzo ritorno funzione</i>	sp	SI
x3	global pointer (il chiamato DEVE salvarlo**) <i>indirizzo ritorno funzione</i>	gp	SI
x4	thread pointer (il chiamato DEVE salvarlo**) <i>indirizzo ritorno funzione</i>	tp	SI
x5	temporaneo / return-address-alternativo <i>registri per i conti</i>	t0 <i>registri</i>	no
x6-x7	temporanei <i>registri per i conti</i>	t1-t2 <i>Temporanei</i>	no
x8	registro salvato o frame pointer (il chiamato DEVE salvarlo**) <i>registri</i>	s0/fp <i>registri</i>	SI
x9	registro salvato (il chiamato DEVE salvarlo**) <i>registri</i>	s1 <i>SALVATI</i>	SI
x10-x11	argomenti della funzione / valori di ritorno <i>registri Temporanei per</i>	a0-a1	no
x12-x17	argomenti della funzione <i>parametri argomenti FUNZIONI</i>	a2-a7	no
x18-x27	registro salvati (il chiamato DEVE salvarli**) <i>registri</i>	s2-s11	SI
x28-x31	temporanei (il chiamato li puo' sovrascrivere)	t3-t6	no

JAR
Se e' invariato
deve essere
salvato.
SALVA IL
CHIAMATO
PERMAN.
DENTRO UNA
FUNZIONE

** nel caso in cui tale registro sia utilizzato

** N.B.: «salvati» significa «salvati-prima», ovvero Se uso registri S, preliminarmente se chiamo una f, metto nello stack i registri S, che poi recupero
devono essere salvati nel frame i valori precedenti all'inizio della funzione e prima dell'uso all'interno della funzione

x0...x4, usati

Registri t, temporanei

Registri a, registri temporanei per argomenti f, non solo in stack

Permanenti: nella
Funzione

PUSEMO Parte dello STACK, ci metteremo prima
i registri S

• a x, 8 reg. a

Sx -> 12 12 registri statici

Registri S => permanenti dentro una
Funzione, Salvati

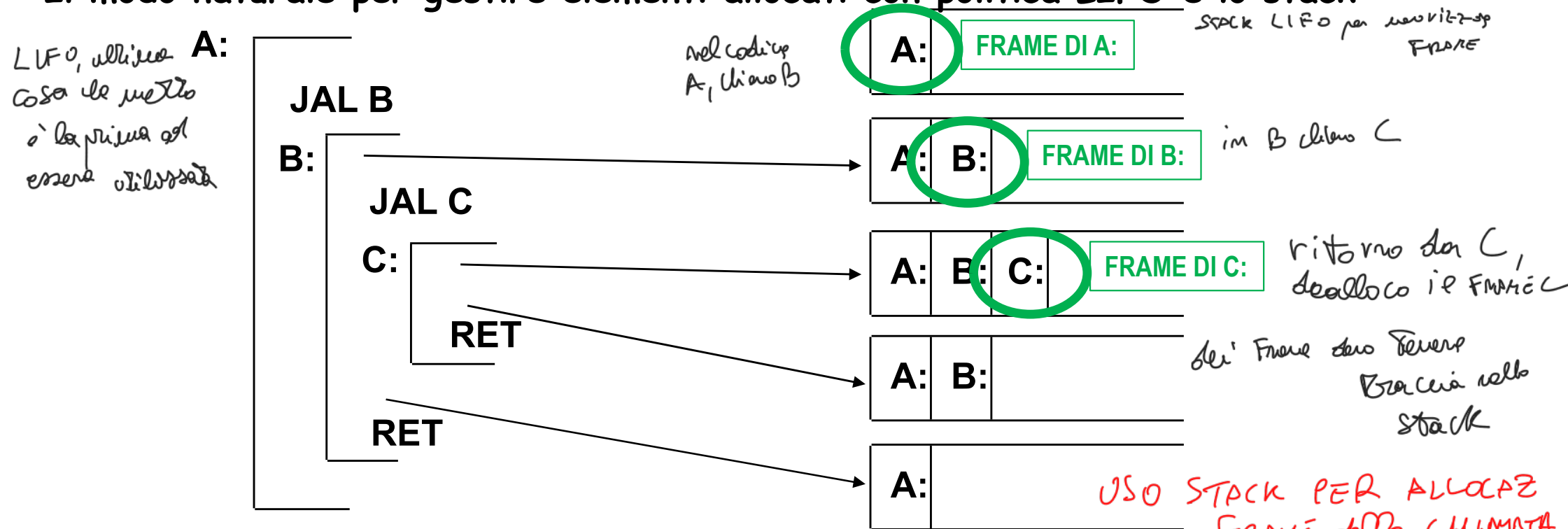
12 Parz:

- Template per usare registri come da Standard

aggiungo un frame sopra l'altro nello STACK

Chiamate annidate, frame vengono messi nello stack proprio perché le chiamate si chiuderanno con politica LIFO

- Per ogni chiamata alloco un nuovo frame (i parametri passati possono essere diversi)
- Per chiamate annidate, il frame precedente non viene buttato e quindi si ha necessità di avere una politica **LIFO** (Last-In First-Out) per gestire i frame
- Il modo naturale per gestire elementi allocati con politica LIFO è lo stack



- Alcune macchine forniscono le istruzioni per la gestione di stack come parte della architettura stessa (e.g. VAX, processori Intel)
 - Istruzioni PUSH e POP *ho istruzioni specifiche HW per gestione STACK*
- Gli stack possono però anche essere implementati via software (e.g. MIPS, RISC-V)
HW più semplice possibile

come facciamo crescere lo stack?

Convenzione sugli stack (1): "verso di crescita"

- Assumendo che nella rappresentazione della memoria gli indirizzi alti siano in alto e quelli bassi in basso...
 - Con '0xFFFF...' indico gli indirizzi alti in memoria
 - Con '0' indico gli indirizzi bassi in memoria

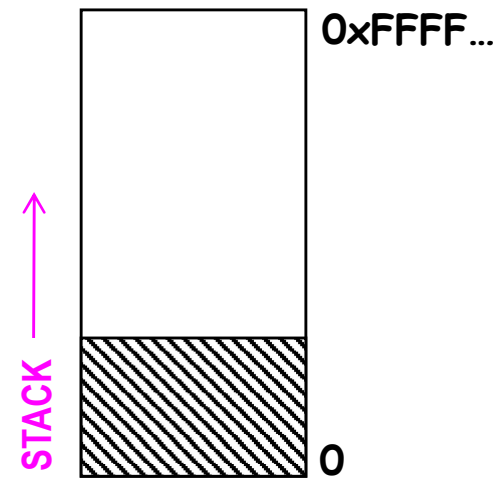
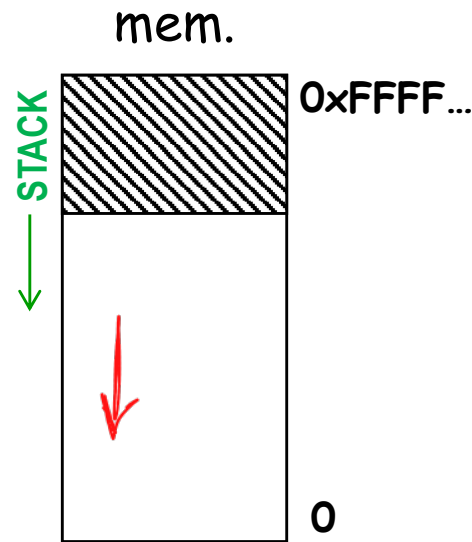
stack lo abbiamo ordinato ab2.

...gli stack possono crescere verso l'alto oppure verso il basso

Convenzione
grow down

Convenzione
grow up

STACK INIZIA
DA INDIRIZZO
ALTO, e cresce
verso basso
addi sp, sp, -32



alloca stack sull'indirizzo
basso

RISC-V

Convenzione sugli stack (2): "a cosa punta sp" *Step pointer*

- Il punto di inserimento dello stack viene puntato dal registro sp
- Ci sono due opzioni per ciò a cui punta sp: **Next-Empty** vs. **Last-Full**

Convenzione
(**grow-down** +)
next-empty

Convenzione
(**grow-down** +)
last-full *ultimo el. inserito è pieno*

Operazioni sullo stack: *struttura dati con oper. di PUSH, POP*

Grow-down + Next-Empty *operazione di gestione della memoria*

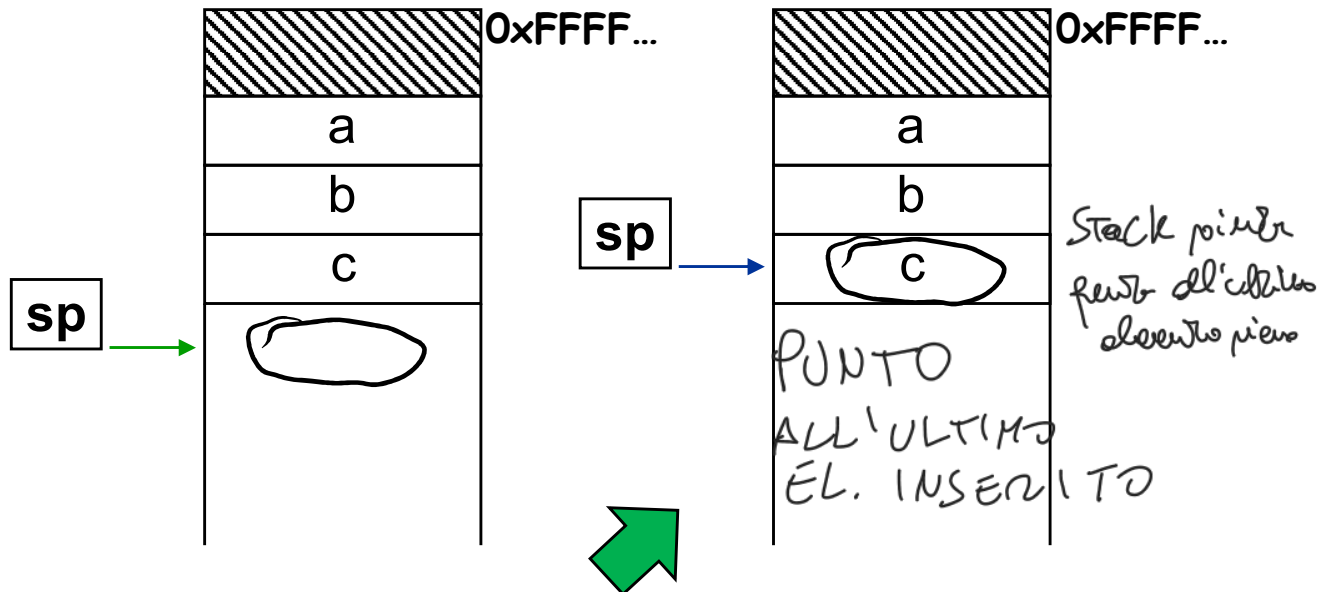
POP: Incrementa sp *inserisco elemento stack*
Legge da Mem[sp]

PUSH: Scrive in Mem[sp] *prelavo*
Decrementa sp *il prelavo*

Grow-down + Last-Full

POP: Legge da Mem[sp]
Incrementa sp

PUSH: Decrementa sp
Scriva in Mem[sp]



- nel caso RISC-V la convenzione è:

- **Grow-Down+Last-Full**

Per inserire altro, prima INCREMENTO, poi INSERISCO

sp punta all'ultimo el. dello stack

Nota: lo stack viene usato non solo per i call-frame ma anche per: *ho finito Registrari*

- allocazione VARIABILI LOCALI della funzione
- MEMORIZZAZIONE TEMPORANEA di valori in "esubero" dai registri (es. nella valutazione di espressioni)

Stack sia per Callframe che
per registri Temporari

in C se scrivo

```
int myfun(...)
```

comp
Finita

int a, b, c se c'è spazio nei registri, li salvo lì, a, b, c
altrimenti li salvo nello STACK di chiore
nella Finita

Convenzione RISC-V per le chiamate a funzioni (1)

PRE-CHIAMATA (LATO CHIAMANTE)

1) Eventuale salvataggio registri da preservare nel chiamante

- Si assume che a0-a7, t0-t6, possano essere sovrascritti
 - se li si vuole preservare vanno salvati nello stack (dal chiamante)
(in particolare i vecchi valori di a0-a7)

Salvataggio di ciò
che uso di Temporaneo

parametri nell'ordine a0, a1, ... a7
oltre l'ottavo 8 valore
vengono salvati nello stack

2) Preparazione degli argomenti della funzione

- I primi 8 argomenti vengono posti in a0-a7 (nuovi valori)
- Gli eventuali altri argomenti oltre l'ottavo vanno salvati nello stack (EXTRA_ARGS), *estensione del Frame a carico del chiamante*
così che si trovino subito sopra il frame della funzione chiamata

CHIAMATA

3) jal MYFUN

NOTA: la jal mette innanzitutto in ra l'indirizzo di ritorno
(ovvero l'indirizzo dell'istruzione successiva alla jal stessa);
Dopodiché salta all'indirizzo specificato da MYFUN

Convenzione RISC-V per le chiamate a funzioni (2)

PROLOGO (LATO CHIAMATO) ALLOCO CALL FRAME nella F decremento SP

4) Eventuale allocazione del call-frame sullo stack *decremento SP di una quantità precisa
(→ aggiornare SP) prima scrivo corpo e poi scelgo di quanto decrementare SP, non so a priori*

5) Eventuale salvataggio registri che si intende sovrascrivere

- Salvataggio degli argomenti a0-a7 solo se la funzione ha necessità di riusarli nel corpo di questa funzione, successivamente a ulteriori chiamate a funzione che usino tali registri, es. ricorsione

(nota: negli altri casi a0-a7 possono essere sovrascritti)

- Devo salvare il vecchio ra, solo se la funzione chiama altre funzioni... *va salvato anche ra*

- Devo salvare il vecchio fp, solo se ho bisogno effettivamente del call-frame pointer (e devo quindi sovrascriverlo)

- Devo infine SEMPRE salvare i vecchi s0-s11 se intendo usare tali registri (il chiamante si aspetta di trovarli intatti) *nel prologo*

6) Eventuale inizializzazione di fp : punta al nuovo call-frame

prima non posso, devo prima salvare il precedente FP, poi posso allocare uno nuovo

punta nuovo frame allocato

CORPO DELLA FUNZIONE

7) CODICE EFFETTIVO DELLA FUNZIONE

Convenzione RISC-V per le chiamate a funzioni (3)

EPILOGO (LATO CHIAMATO)

8) Se deve essere restituito un valore dalla funzione

- Tale valore viene posto in a0 (e a1) *risultati in a0 e a1*

9) I registri (se salvati) devono essere ripristinati

- a0-a7 (nel caso siano stati salvati all'interno della funzione) *non sovrascrivere a0*
- s0-s11
- ra *←*
- fp *←*

Inoltre, notare che sp deve solo essere aumentato di opportuno offset *dealloco il frame*
(lo stesso sottratto nel punto 4)

*allocare frame, aggiungere
quantità con cui sono decrementato
fp*

RITORNO AL CHIAMANTE

10) ret

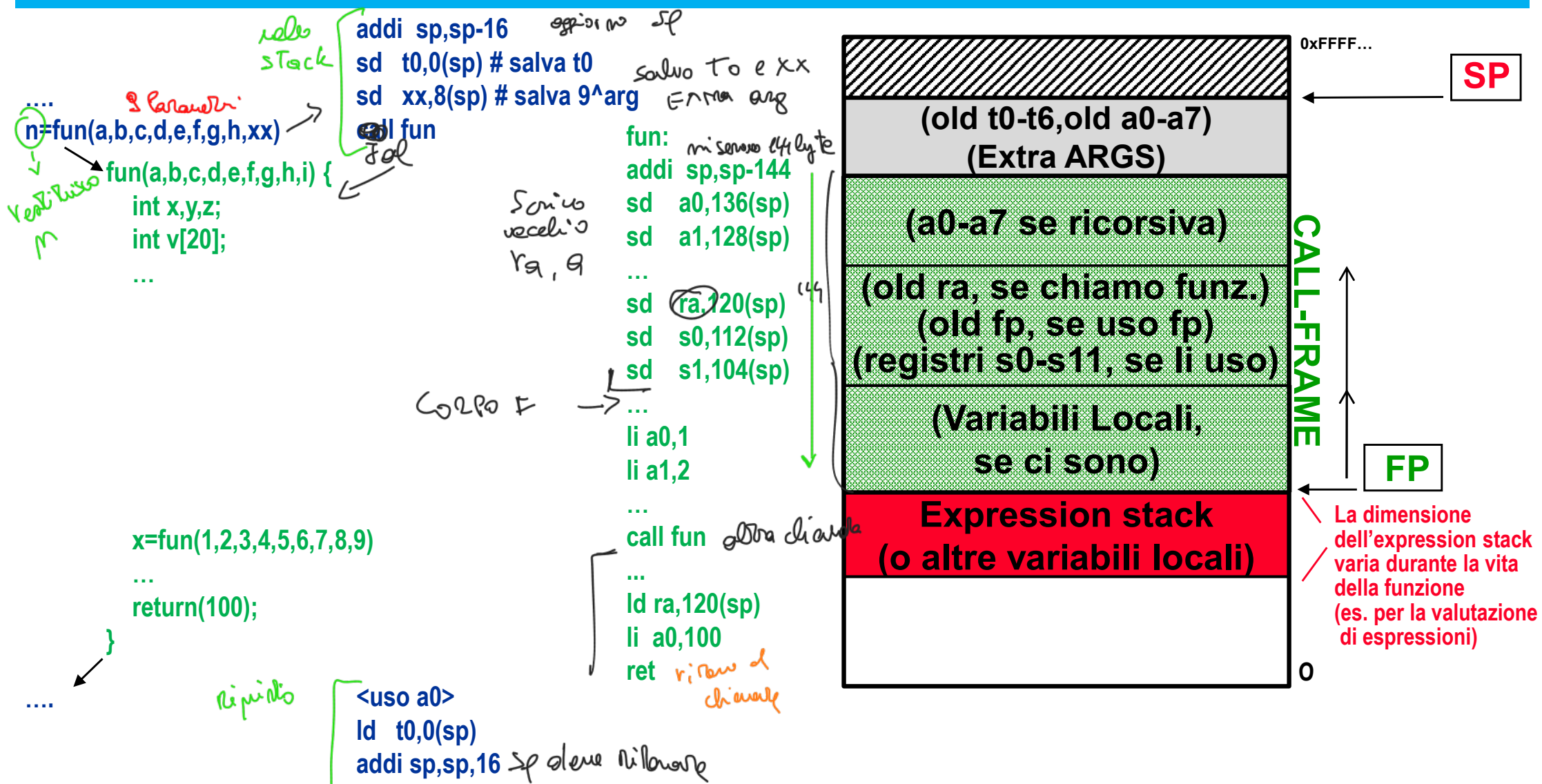
POST-CHIAMATA (LATO CHIAMANTE)

11) Eventuale uso del risultato della funzione (in a0 (e a1))

12) Ripristino dei valori t0-t6, a0-a7 (vecchi) eventualmente salvati

Riepilogo: struttura del Call-Frame

- Grow-Down + Last-Full Stack
- FP=Frame Pointer (i.e., s0)
- SP=Stack Pointer



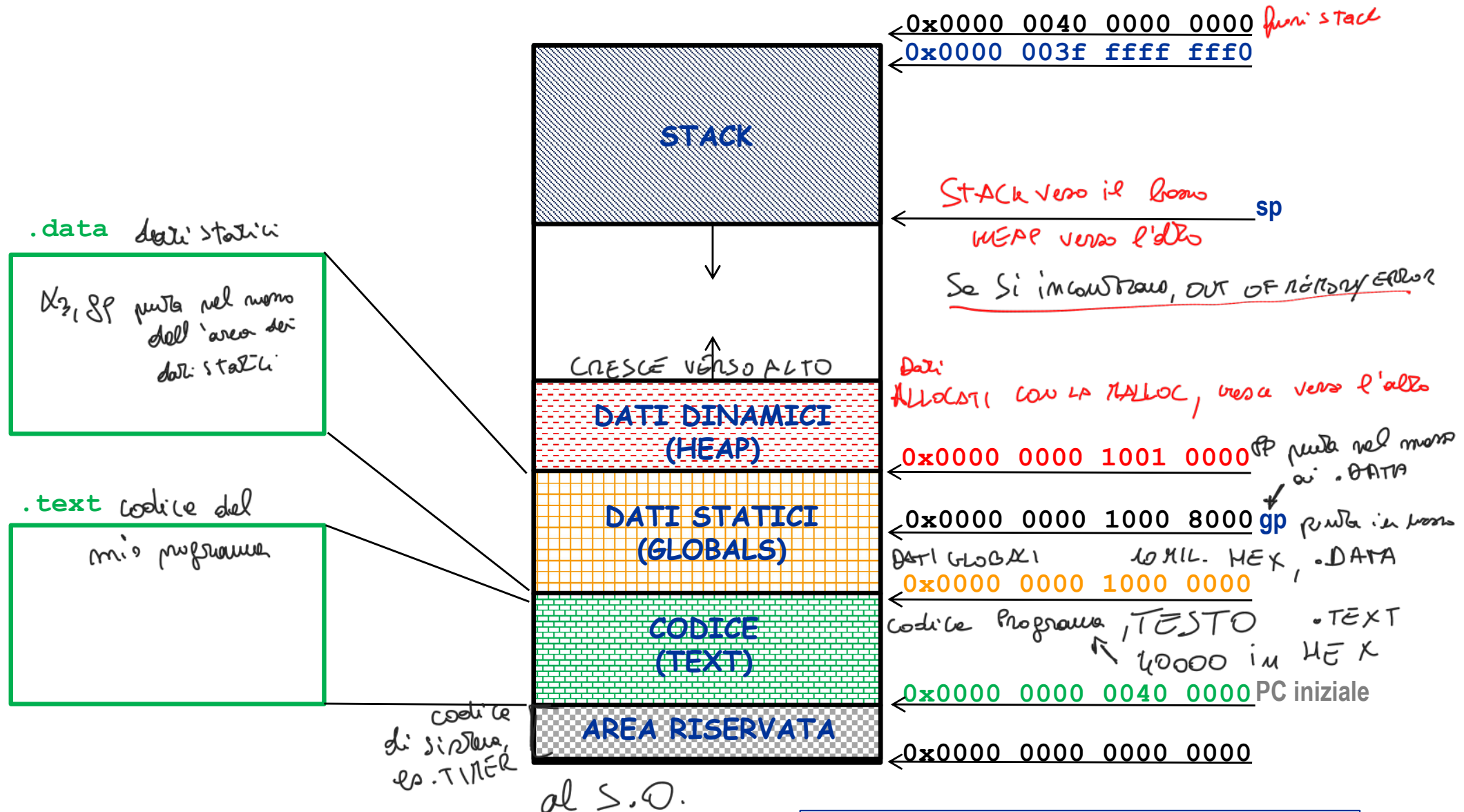
Nota1: FP è comodo per codificare nel corpo della funzione gli accessi a offset fissi all'interno del call-frame (non possibile con SP perché varia)

Nota2: I compilatori normalmente cercano di allocare le variabili locali prima di tutto nei registri, non in memoria (o stack)!

Mappa di memoria (Memory Layout)

Nella memoria dell'elaboratore

- Dove si trova lo stack? il programma? i dati globali?



Principali direttive per l'assemblatore

- .data [<addr>]** marca l'inizio di una zona dati;
se <addr> è specificato, i dati sono memorizzati a partire da <addr>
- .text [<addr>]** marca l'inizio del codice assembly; *Assembly*
se <addr> è specificato, il codice è memorizzato a partire da <addr>

.globl <symb> *codice inizia all'indirizzo symb. es. -globl main* dichiara che il simbolo <symb> visibile è dall'esterno (gli altri simboli sono locali per default) in modo che possa essere usato da altri file *main: es. LIBRARY*

.extern <symb> [<size>] *se lo cose in altri file* dichiara che il dato memorizzato all'indirizzo <symb> è un simbolo globale ed occupa <size> byte. *es. in libreria o in altri file*
Questo consente all'assemblatore di:

- 1) memorizzare il dato in una porzione del segmento dati (quello indirizzato tramite il registro gp)
- 2) dichiarare che i riferimenti al simbolo <symb> riguardano un oggetto esterno ad un modulo (torneremo su questo fra poco)

stringa memorizzata in memoria
.asciz "str" mette in memoria la stringa "str", seguita da un byte '0'
.ascii "str" mette in memoria la stringa "str", SENZA lo '0' finale

carattere con terminazione /n

Programma fattoriale_riscv.s (1)

albertusa compilato

$fact(n) = n \cdot fact(n-1)$ se $n \neq 0$
1 se $n = 0$

definisco zona dati

```
.data
messaggio: .asciz "Calcolo di n!\n"
insdati: .asciz "Inserisci n = "
visris: .asciz "n! = "
RTN: .asciz "\n"
```

} ZONA DATI

.globl main *può di ingresso, prima istruzione che esegue si trova dopo etichetta main*
 .text *definisco codice*
 fact: FUNZIONE **Fact**

lo stack cresce verso il basso **Prologo**, allorché il frame
 addi sp, sp, -24 *24 = 3 * 8* # allocazione del call frame nello stack 3 by 16, 24 di 8, da che cresce di 24
 swd a0, 16(sp) # salvataggio di n nel call frame *parametro di ricorsione*
 swd ra, 8(sp) # salvataggio dell'indirizzo di ritorno
 swd s0, 0(sp) # salvataggio del precedente frame pointer *per fare aware di davanti del frame*
 add s0, sp, zero # aggiornamento del frame pointer punta al nuovo frame pointer *con valore attuale dello stack pointer*

calcolo del fattoriale
 bne a0, zero, Ric # test fine ricorsione *se $n \neq 0$, faccio l'ordine di ricorsione*
 addi a0, zero, 1 # $0! = 1$ *se $a0 \neq 0$ vado alla funzione altrimenti $a0 = 1$*
 j Fine *Salvo a Fine*

Ric: *qui tiene n*
 addi a0, a0, -1 *recupero* # chiamata ricorsiva per il calcolo di (n-1)!
 jal fact # $a0 \leftarrow (n-1)$ passaggio del parametro in a0 per fact(n-1)
 lw t0, 16(s0) *parametro ricorsione* # chiama fact(n-1) -> risultato in **a0 della fact**
 mul a0, a0, t0 # $n! = (n-1)! \times n$ *in t0 ho il valore q_0*

uscita dalla funzione
 Fine:

$a = n \cdot fact(n-1)$

Epilogo
 lw s0, 0(sp) # recupera il frame pointer
 lw ra, 8(sp) # recupera l'indirizzo di ritorno
 addi sp, sp, 24 # elimina il call frame dallo stack
 ret # ritorna al chiamante
destrutto il frame

*queste parti possono non essere
 coinvolte prima scrivere corpo*

*Scrivo il corpo, vedo quello di cui
 ho bisogno*

Programma fattoriale_riscv.s (2) *comincia per usare funzione*

```
# Programma principale
#
```

```
main:
```

```
# Stampa intestazione
la a0, messaggio
addi a7, zero, 4
ecall
```

```
# Stampa richiesta
la a0, insdati
addi a7, zero, 4
ecall
```

```
# legge n (valore in a0)
addi a7, zero, 5
ecall
```

```
# chiama fact(n)
call fact
add s0, a0, zero
```

```
# stampa messaggio per il risultato
la a0, visris
addi a7, zero, 4
ecall
```

```
# stampa n!
add a0, s0, zero
addi a7, zero, 1
ecall
```

```
# stampa \n
la a0, RTN
addi a7, zero, 4
ecall
```

```
# exit
addi a7, zero, 10
ecall
```

*servizio sistema
operativ, ecall, environment call,*

*istruzione che chiama
un servizio*

servizi in a7

legge intero con servizio n 5 da tastiera

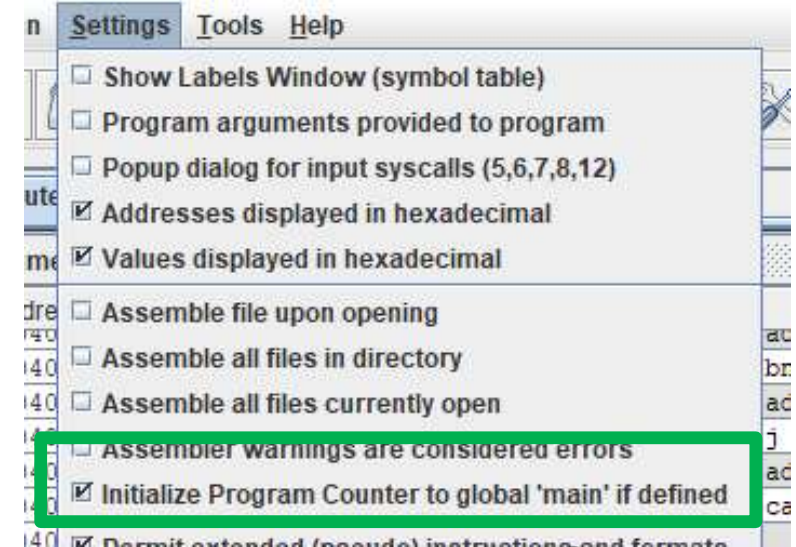
*parametro n in a0
salva il risultato in s0 solo a0 in s0
salva in s0 il valore restituito*

-> stampa il fattoriale

Stampa intero

servizio 10, solo del programma

Nota: NEL SIMULATORE RARS, selezionare Settings/Initialize_Program_Counter_to_global_'main'



USO MAIN per inizio programma.

NOTA: prerequisito: occorre installare JAVA SE

1.5

Output del simulatore RARS

<https://github.com/TheThirdOne/rars>

```
Run I/O
Calcolo di n!
Inserisci n = 5
n! = 120
-- program is finished running --
```


Ulteriori direttive assembler

.byte <b1>, ..., <bn>

Mette in memoria gli n byte che sono specificati da <b1>, ... , <bn>

.word <w1>, ..., <wn>

Mette in memoria le n word a 32-bit che sono specificate da <w1>, ... , <wn>

.float <f1>, ..., <fn>

Mette in memoria gli n numeri floating point a singola precisione (32 bit) (in locazioni di memoria contigue)

.double <d1>, ..., <dn>

Mette in memoria gli n numeri floating point a doppia precisione (64 bit) (in locazioni di memoria contigue)

.half <h1>, ..., <hn>

Mette in memoria le n quantità a 16 bit (in locazioni di memoria contigue)

.space <n>

space 100 riserva 100 byte nella regione dati statici

Mette in memoria n spazi (ovvero riserva n byte di memoria)

.align <n>

importante allineare dati; il dato non deve essere spezzato allineare dati anche se spreco spazio

Allinea il dato successivo a un indirizzo multiplo di 2^n

- es. «.align 2» allinea il valore successivo "alla word"

- «.align 0» disattiva l'allineamento generato dalle direttive .half, .word, .float, e .double fino alla successiva direttiva .data o .kdata

=> programma che prende in ingresso il codice assembly e produce i binari a 32 bit

VA UNO IN .DATA

sto dicendo dati statici.

Per tipi di dati predefiniti

NELLA PARTE .DATA

*riserva 1000 byte nella regione di dati locali
dati statici*

Per tipi di dati strutturati

(es. Array, strutture, ...)

Importante allineare dati per migliorare prestazioni

Servizi di sistema (system call) - istruzione 'ecall' *environment call*

- 'ecall' serve per chiedere un servizio al sistema operativo (operazione nota come 'system call')
 - Il numero del servizio è indicato in **a7***
 - 1: stampa un intero a video
(**a0*** in ingresso, contiene l'intero in binario da stampare) *Qo il parametro di uscita f.*
 - 4: stampa un messaggio a video *leggo in a0, che punta a una stringa*
(**a0*** in ingresso, indirizzo della stringa da stampare) *puntatore stringa mette il puntatore alla stringa da stampare*
 - 5: leggi un intero dalla tastiera *metto in a0 un intero*
(**a0*** in uscita, conterrà l'intero letto in binario)
 - 10: termina il programma *= exit in C.*
(il sistema operativo riprende il controllo del calcolatore)

DSN TOCCA I M(E) NESSUN

Nota: si può assumere che la ecall si comporti come una chiamata a funzione speciale in cui non deve essere alterato alcun registro (salvo quelli utilizzati in ingresso o uscita naturalmente). Questo è possibile grazie al meccanismo di eccezione (che vedremo in dettagli più avanti nella lezione 9). *ecall, un interrupt, meccanismo che solo automaticamente tutto*

* Con riferimento al simulatore «RARS»; altri simulatori usano rispettivamente a0 e a1 anziché a7 e a0 *puero servizi e paramet- possono essere diversi*

3 passi per mandare in esecuzione un programma che ho fatto io

Compilazione di un programma strutturato a moduli

- Nel precedente esempio non compaiono esplicitamente alcuni passi logici per arrivare ad eseguire un programma

- Compilazione** pura dei file *traduce le istruzioni da alto livello in BINARIO, produce la CODIFICA delle istruzioni (FORMATO I, R, B, S, U)*
→ per controllare di aver scritto bene ciascun modulo

- Collegamento** di più moduli *collega i moduli in modo che siano in memoria coerenti e compatibili*
→ per creare un unico file da essere eseguito (il programma) *es. libreria*

- Caricamento** del programma in memoria e sua esecuzione
→ per consentire al processore di eseguire il nostro programma

Compilazione separata

```
cc -c main1.c
```

```
cc -c queue.c
```

- vengono così creati (salvo errori) i file `main1.o` `queue.o` *programma viene caricato in memoria a partire da un certo indirizzo, il processore prende il PC dell'inizio della sezione di quel programma e inizia ad eseguire le istruzioni*

- Collegamento (Linking)** → viene invocato un altro stadio di cc

```
cc -o main main1.o queue.o
```

- Caricamento ed esecuzione**

```
./main
```

PREPROCESSORE (#) => COMPILAZIONE (da FILE ASSEMBLY) => ASSEMBLER (genera codice binario)

Passi principali della compilazione di un programma C compilatore più avanzato

1) Preprocessore (stadio 'cpp') C pre-processor

- PREPROCESSOR: FRONT END
- espande le macro e le costanti (#define) viene sostituito dove necessario
 - inserisce nell'elaborato i file inclusi (#include)
 - processa le direttive condizionali (#ifdef, #ifndef, ...)

Compilazione ha processi in front-end e back-end

2) Compilazione vera e propria

- si svolge in uno o più stadi (cc1, cc2) fino ad ottenere il livello di ottimizzazione desiderato

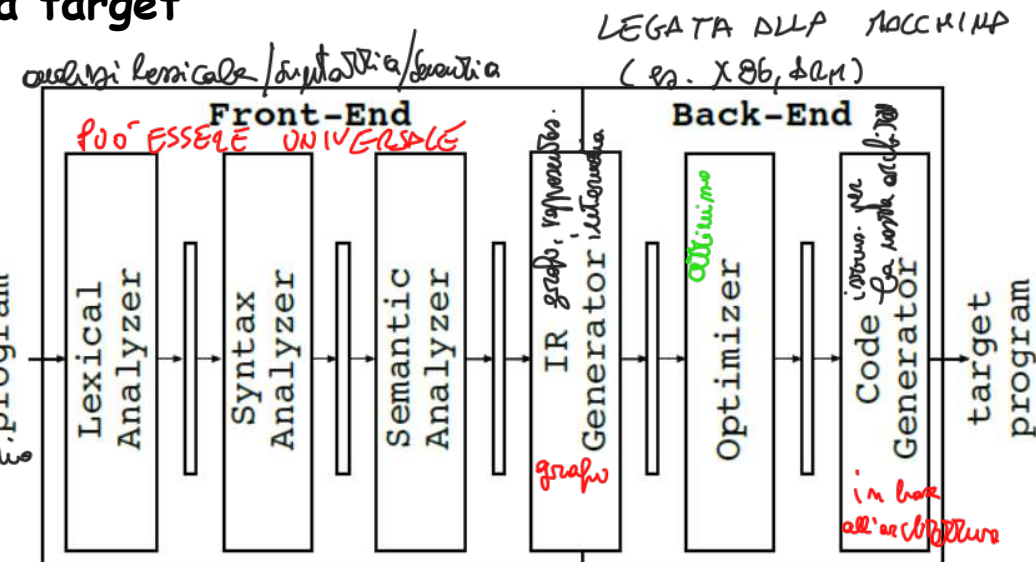
- il risultato intermedio è un programma in ASSEMBLY passo di mezzo di compilazione di generazione di assembly, che viene dato all'assembler
- il codice ASSEMBLY è legato all'architettura target

• Creazione del file oggetto viene esportato dall'assembler binario con Metadata

- questo passo è eseguito dall'assemblatore
- il file oggetto contiene un binario

- METADATA
- la codifica delle istruzioni assembly
 - i dati statici per spostare programma in memoria
 - informazioni di rilocazione
 - tabella dei simboli o variabili in cella memoria
 - altre informazioni di utilità

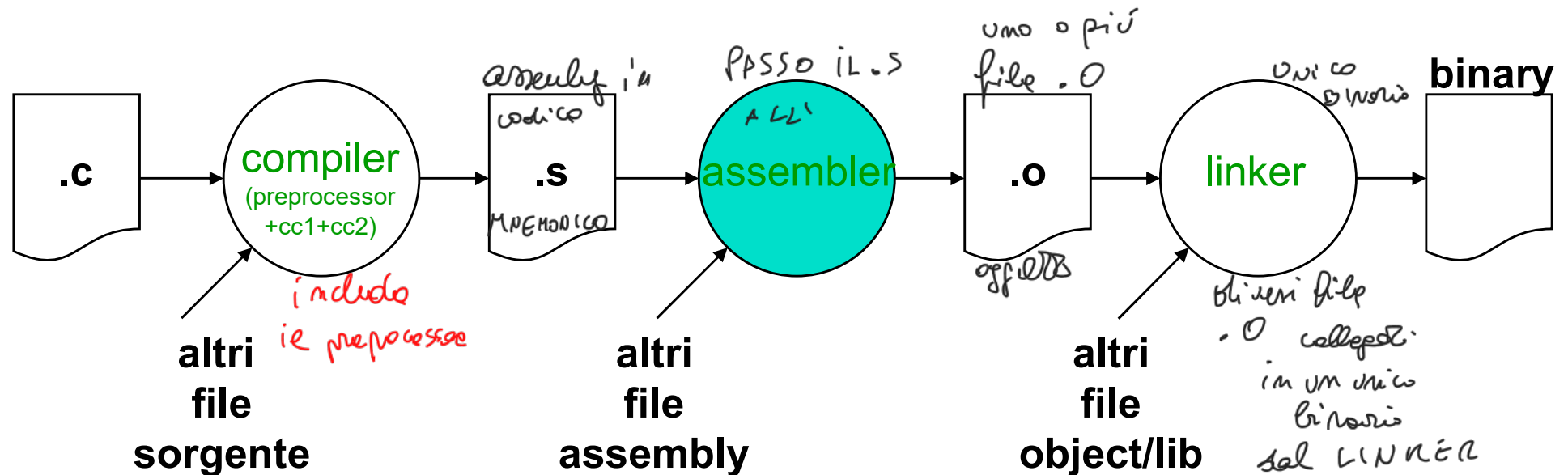
→ codice PIC, INDIPENDENTE dalla locazione di MEMORIA ma in generale DIPENDE dalla MEM



Visione semplificata di un compilatore

[https://www.researchgate.net/publication/332241231 Principles Techniques and Tools for Explicit and Automatic Parallelization](https://www.researchgate.net/publication/332241231_Principles_Techniques_and_Tools_for_Explicit_and_Automatic_Parallelization)

Assemblatore



- È possibile produrre l'output in assembly del compilatore (UNIX/LINUX),

`cc -S main.c`

il risultato si trova nel file **main.s** File che contiene l'assembly

CI SONO DEI SIMBOLI che vengono definiti prima

ASPETTO ALLA LORO DEFINIZIONE

Assemblatore a due passate (1)

Assemblatore legge riga dopo riga, se include un simbolo che non sa a cosa si riferisce, non può produrre, allora tiene prima tutto una volta

- Prima passata: determinazione delle locazioni di memoria etichettate (creazione della tabella dei simboli):

ho myvar legg prima testo. es. dov'è il my var? e' sotto, pensa legg per determinare la loc. di memoria myvar

SIMBOLI: LOC. di memoria etichettate

```
.text
add x18, x18, x0
addi x19, x19, 0
loop: add x4, x18, x0
add x19, x19, x4
add x18, x18, x19
bne x18, x5, loop
la x18, myvar
.....
jal fun1
.....

fun1:
.....
.....
.....
```

CONTATORE, LOCATION COUNTER

LC=0
LC=4
LC=8
LC=12
LC=16
LC=20
LC=24
LC=3000

vedo loop, prendo nota che e' a Loc. Counter N. 8

e' piu' avanti

LC=20'000'000

```
.data
myvar: .....
```

LC della

mem. DATA

LC=0

sta all'inizio 0 e lo segno in tabella

NELLA 2ª PASSATA SO SONO SONO TUTTI I SIMBOLI

devo solo produrre la istruzione con i relativi offset

Symbol Table (provvisoria)

symbol	Symbol Location Counter (Symbol-LC)
loop	8
myvar	0
fun1	20'000'000

Tabella simboli, ci servirà nella 2ª passata,

es. jal fun1 => jal x1, OFFSET

Assemblatore a due passate (2)

one 2 flow new loop, myvar, funt

• Seconda passata:

- traduzione delle istruzioni nei rispettivi codici operativi (opcode)
- determinazione dei numeri dei registri (rs1, rs2, rd)
- sostituzione delle etichette** nelle istruzioni con indirizzamento relativo al PC (beq, bne) e produzione symbol table finale
- creazione della **Tabella di Rilocazione**

es. Jal funt = Jal x1, offset

• ** Es. Per la bne a LC=20 che usa il simbolo 'loop'

$$\text{offset} = \text{LC}(\text{SYMBOL}) - [\text{LC}(\text{BNE})]$$

loop era a LC[8]

$$8 - [20] = -12 \quad \Rightarrow \quad \text{bne x18, x5, -12}$$

loop:	add	x4, x18, x0	LC=8
	add	x19, x19, x4	LC=12
	add	x18, x18, x19	LC=16
	bne	x18, x5, loop	LC=20

Nota: nella codifica si scarta il bit meno significativo dell'offset:
-12 diventa -6 *indiviso 2 byte / 2*
→ memorizzo nel campo «imm» dell'istruzione il valore -6

- Per ogni etichetta irrisolta, si controlla che non sia definita come "simbolo esterno" tramite .globl *direttiva* (ovvero un simbolo che deve essere visibile esternamente)
 - se non è un simbolo esterno, può essere un errore
 - se è un simbolo esterno o un riferimento ad un indirizzo assoluto, *da altri file* (es. nelle istruzioni jal, j, la)

→ **deve restare in symbol table** *nel dubbio, dobbiamo risolvere myvar*

symbol	Symbol-LC
myvar	0
fun1	20'000'000

Simbolo irrisolto, magari viene chiamato risolvere myvar
Altri programmi, file ecc, lascio in symbol table

Creazione e uso della Tabella di Rilocalizzazione

 la x18, myvar

 jal fun1

 fun1: -----
 LC=0
 LC=24
 LC=3000
 LC=20'000'000

Simbolo myvar all LC 24
 viene chiamato all'istruzione la

Relocation Table

Referring Location Counter (Referring-LC)	TIPO ISTRUZIONE	SIMBOLO DIPENDENTE
24	la	myvar
3000	jal	fun1

importante sapere da che tipo di instr. sono riferite

- **USO DELLA RELOCATION TABLE:** in fase di linking le informazioni in Symbol Table e Relocation Table vengono combinate con gli indirizzi effettivi: in questo caso il linker assume che il programma venga caricato ad un indirizzo fisso prestabilito (es. 0x0040'0000) mentre i dati risiedono ad un altro indirizzo fisso (es. 0x1000'0000):

Codice
 del
 programma

0x0040'0000 add x18, x18, x0
 0x0040'0004 addi x19, x19, 0
 0x0040'0008 (loop:) add x4, x18, x0
 0x0040'000C add x19, x19, x4
 0x0040'0010 add x18, x18, x19
 0x0040'0014 bne x18, x5, -12
 0x0040'0018 la x18, 0x1000'0000
 0x0040'001C

la x18 0x1000'0000
 lui x18 0x10000
 ori x18, x18, 0

X18
 1 0 0 0 0 ... 0

già stabilito dall'assembler
 # diviso in due parti da 20 bit e 12 bit
 # la seconda parte è zero, per cui basta
 # lui x18, 0x10000

codice caricato
 a 0x0040'0000

lui, e ori parte della lui, 5 byte

0xBB8=3000

0x0040'0BB8 jal fun1
 0x1000'0000 (myvar:)

0x131'2D00=20'000'000

0x0171'2D00 (fun1:)
 0x0171'2D04

non si può tradurre con un offset da 20 bit
 # lui x5, 0x01713 # 171'2D00=0x40000+20'000'000
 # jalr ra, 0xD00(x5) # 0xD00 è negativo quindi
 # aumento di 1 l'immediato della lui

per myvar l'indirizzo LC=0 corrisponde all'inizio
 # della zona di dati globali 0x1000'0000

non entra in 20 bit l'indirizzo di jump

$X_1 = R_0$

Jal 0x01712D00

lui

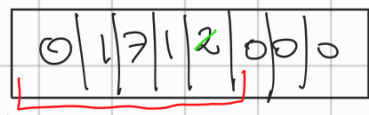
X_5

0x01712D00

73

2 in da moia

X_5 3



Faccio rif. col X_5

Jalr

X_1

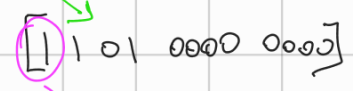
0x00000000 (X_5)

offset

la lui mettere in X_5

costante $X_5 + 0x100$

$X_5 + 0x100$



Dato aggiungiamo la prima parte che mi manca, come offset

è negativo! vera sottrazione se aggiungo un 1 a X_5 si trova

0x0000 è negativo, se lo aggiungo a 1712, si trova

a 1711, aggiungo 1 alla 5 cifra significat in Hex per far tornare conto

aggiungo +1 al X_5 in Hex

così se vedo a sottrarre, vedo a sommare valore desiderato

0x40000
indirizzo base

20000000 → di non bra LC=0 e 20000000

verso la Jalr, con ripulito X_5 salvo in X_5 l'indirizzo in cui ritornare

Simboli esterni

- Un simbolo è **esterno**, se viene esportato (es. da una libreria) verso altri moduli

.globl <symbol> dichiara il simbolo <symbol> come un'etichetta visibile all'esterno del modulo

Modulo B *es. Funzione libreria*

```
.globl cubic può di ingresso dal codice
cubic:
mul t0, a0, a0 # LC=0
mul a0, t0, a0 ritorna a^3
ret
```

Symbol Table modulo B

symbol	Symbol-LC
cubic	0

- In qualche altro modulo esisterà un riferimento alla funzione "cubic":
.extern <symbol> dichiara che ogni riferimento a <symbol> (es. jal cubic) si riferisce ad un simbolo esterno a questo modulo *si trova in un altro file o libreria*

Modulo A *Simbolo si trova in un altro file*

```
.extern cubic si trova in file esterno
----
jal cubic # LC=1500
----
----
```

Symbol Table modulo A *sarà vuota*

symbol	Symbol-LC

Relocation Table modulo A

Referring-LC	TIPO ISTRUZIONE	SIMBOLO DIPENDENTE
1500	jal	cubic

Nota: il simbolo "cubic" verrà quindi risolto in fase di linking

Nota2: nel simulatore RARS, si possono editare i due moduli separatamente (anche senza la dichiarazione .extern) ma con la dichiarazione .globl nel modulo «B» e abilitare l'assemblaggio simultaneo di tutti i moduli (il collegamento viene effettuato generando un'unica symble table)

Formato del file oggetto (.o) →

NO NECESSITA' DI PIU' INFO

HEADER	INTERPRETARE
TEXT segment	codice da RILOCARE
DATA segment	Dati Statici
RELOCATION info	TABLE
SYMBOL table	
DEBUGGING info	

- Dimensione del file, delle parti (text, data) e CRC FORMATO
- Codifica delle istruzioni (linguaggio macchina DA RILOCARE)
- Dati Statici
- Lista delle istruzioni che fanno riferimento ad indirizzi assoluti
- Lista dei simboli locali (etichette, variabili) e etichette visibili all'esterno (assembly .globl)
- Parte opzionale per facilitare il debugging del programma

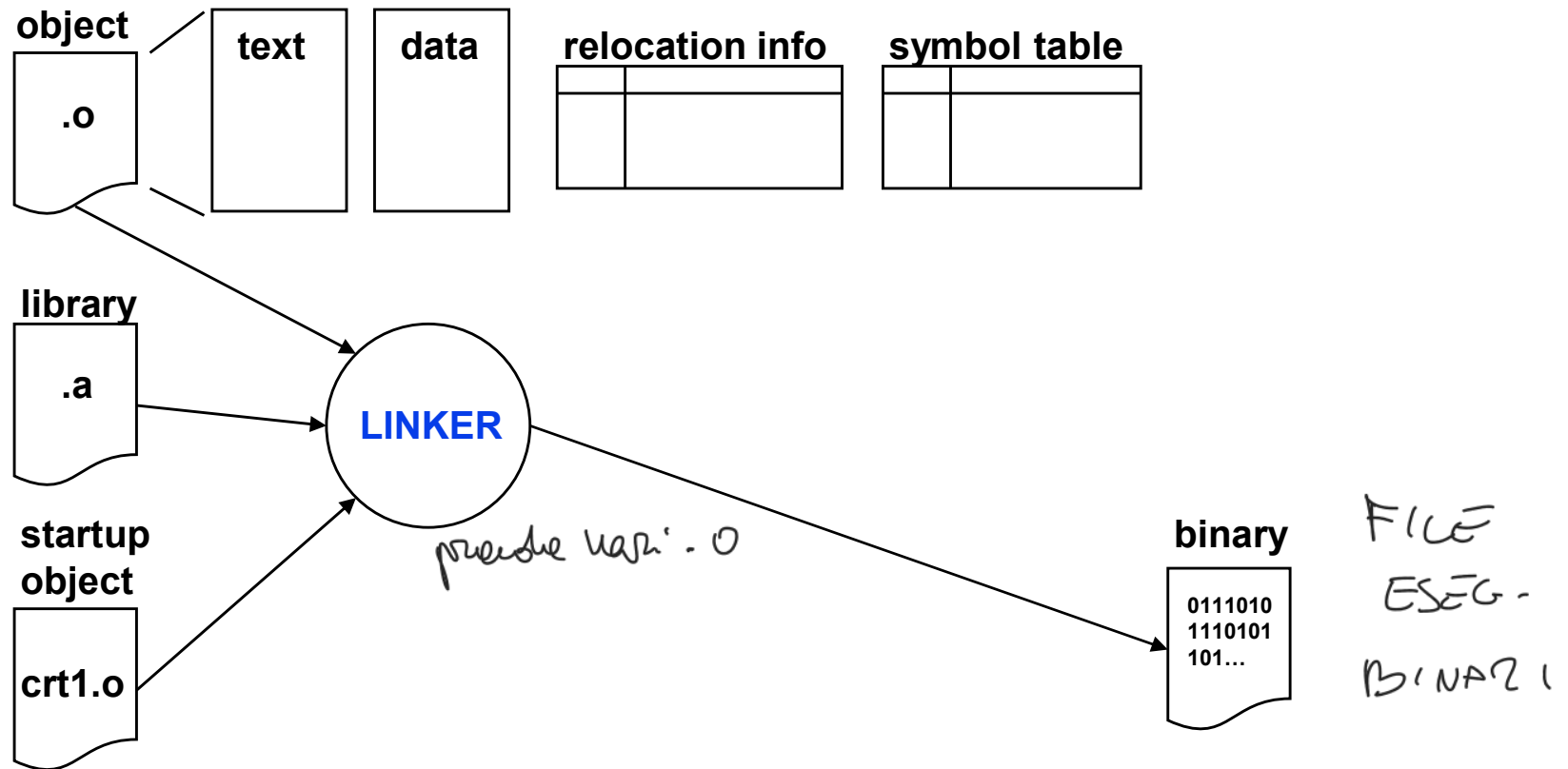
• UNIX/LINUX:

il comando **nm** consente di produrre informazioni sul layout del file oggetto

File eseguibile (binary)

vede vari .o

- è prodotto dal linker



- ~~I riferimenti assoluti sono rilocati e i simboli sostituiti con indirizzi~~

Nota: un formato molto usato per gli eseguibili è detto ELF.

Per approfondimenti: <http://www.caldera.com/developers/devspecs/mipsabi.pdf>

Caricamento (loader) *S.O. deve fare spazio in memoria, deve decodificare*

- Il Sistema Operativo gestisce la richiesta fatta alla shell *dinamic. Memoria* di mandare in esecuzione il mio programma *e ci carica il programma*
 - Il sistema operativo effettua le seguenti operazioni
 - Lettura del file header
 - Creazione dello spazio di indirizzi allocato al programma
 - Copia di TEXT e DATA nello spazio indirizzi
 - Inizializzazione dei registri
 - Salto alla routine di startup *Faccio partire il programma*
- Fatto spazio, carica il FILE BINARIO, eventualmente inizializza i registri, modifica le info di ribobol.*

Nota:

è possibile generare il file binario in un colpo solo con

```
cc -o main main.c queue.c
```

BINARIO DI NOME MAIN