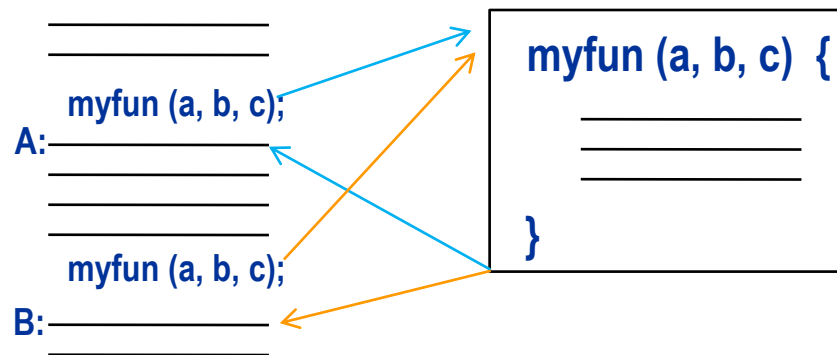


Il set di istruzioni RISC-V

Terza Parte

Autore Principale

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



- 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

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

i) memorizzare da qualche parte il punto di ritorno

(es. `ra` = indirizzo dell'etichetta `A`, in `C` si usa la notazione **&A**);

ii) saltare all'inizio della funzione "myfun" e al termine saltare al contenuto di `ra`

- Nel RISC-V si usa quindi la pseudo-istruzione "jump and link" o `jal`:
`jal label # ra ← PC+4; PC ← PC+OFFSET (OFFSET=&label-PC)`
→ salta a `label` (es. `myfun`) mettendo l'indirizzo di ritorno (es. `A` oppure `B`) nel registro `ra` (ovvero **x1**). Nota: l'istruzione reale è `jal ra,OFFSET` (v. slide succ.)
- Per tornare indietro uso `ra` tramite la pseudo-istruzione "return" o `ret`:
`ret # PC ← ra`

Istruzione "JAL rd,offset" e il formato J

- '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
- Quindi 'JAL label' è un modo più rapido per scrivere 'JAL ra,offset' dove $\text{offset} = \&\text{label} - \text{PC}$; per avere un intervallo di salto ampio si usa il formato 'J'

Istruzione "JAL rd,offset" e il formato J

- 'Il formato J è una variante del formato U in cui sfruttando bene in 20 bit dell'immediato posso coprire un offset di 512ki istruzioni*
 - similmente alla BEQ i 20 bit indicano un offset in byte allineato alla half-word (quindi e' come se avessi 21 bit con uno 0 finale); contando 4B per istruz. ->512ki istruz.

0000 0000 0001 0000 0000	00001	0110011
imm20[19:0]	rd	opcode

FORMATO "U" (gia' usato per la LUI)
es. LUI x1,0x00100

imm20

FORMATO J

0010 0000 0000 0000 0000	00001	1101111
{imm21[20],imm21[10:1],imm21[11],imm21[19:12]}	rd	opcode

Compressivamente sono i bit imm21[20:1]

FORMATO "J" (o formato "UJ")
es. JAL 0x00200

imm21

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

20 19:12 11 10:1
0010 0000 0000 0000 0000

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

- Il registro rd codifica l'indice del registro **ra (ovvero x1)** che conterrà PC+4 (punti «A» e «B» della slide precedente)

*Nota:

512ki=512*1024=524288 istruzioni

Pseudo-istruzione JAL, istruzione JAL, istruzione JALR

Abbiamo quindi due possibili sintassi per JAL:

- **JAL label** → denota la **pseudo-istruzione** (basata su JAL ra,offset)
- **JAL rd,offset** → denota una **ISTRUZIONE** dove $rd \in \{x0, \dots, x31\}$
- Normalmente noi faremo riferimento alla pseudo-istruzione JAL label
- BEQ/BNE e JAL permettono di generare codice «PIC»
- (Position Independent Code) che puo' essere caricato a qualsiasi indirizzo

Pseudo-istruzione JAL, istruzione JAL, istruzione JALR

- Per permettere di caricare del codice a qualsiasi indirizzo (e quindi poter chiamare funzioni o fare salti a qualsiasi indirizzo) occorre far riferimento a un registro → istruzione JALR

- `JALR rd,offset(rs1) → x[rd]=PC+4; PC=x[rs1]+sext(offset)&~1`
- «`sext(offset)`» denota che `offset` viene esteso di segno a 64 bit
- «`&~1`» denota che il bit 0 viene azzerato

Nota: la JALR viene codificata usando il formato I

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

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

Noi useremo solo la coppia: JAL etichetta + RET

Ricapitolazione dei formati e istruzioni fin qui viste

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			

<u>Istruzione</u>	<u>Significato</u>	<u>Variante</u>	<u>Significato</u>
<code>add s1,s2,s3</code>	<code>s1 = s2 + s3</code>	<code>addi s1,s2,10</code>	<code>s1 = s2 + 10</code>
<code>sub s1,s2,s3</code>	<code>s1 = s2 - s3</code>		
<code>slt t0,s1,s2</code>	<code>t0 = (s1 < s2) ? 1 : 0</code>	<code>slti t0,s1,10</code>	<code>t0 = (s1 < 10) ? 1 : 0</code>
<code>lw s1,100(s2)</code>	<code>s1 = Memory[s2+100] (32 bit)</code>	<code>ld s1,100(s2)</code>	<code>legge 64 bit</code>
<code>sw s1,100(s2)</code>	<code>Memory[s2+100] = s1 (32 bit)</code>	<code>sd s1,100(s2)</code>	<code>scrive 64 bit</code>
<code>bne s4,s5,L</code>	<code>salta a L se s4!=s5 (L si trova a offset imm13=(&L-PC))</code>		
<code>beq s4,s5,L</code>	<code>salta a L se s4==s5 (L si trova a offset imm13=(&L-PC))</code>		
<code>lui x5,imm20</code>	<code>carica imm20 nei 20 bit alti</code>		
<code>jal FUN</code>	<code>esegue la funzione FUN (che si trova a offset imm21=(&FUN-PC))</code>		
<code>ret</code>	<code>ritorna da funzione (jalr x0,0(ra))</code>		

Salti con offset grandi (istruzione `auipc`)

- Così come per le costanti, anche i salti possono essere fatti sia ad istruzioni "vicine" (entro i 2048 byte sopra o sotto l'istruzione di salto) oppure più lontane
 - Il RISC-V fornisce in alternativa un intervallo di salto pari a 2^{32}
- Per fare questo occorre però introdurre una nuova istruzione «`auipc`»
 - In modo simile alla `lui`, la `auipc` usa il formato U per memorizzare all'interno dell'istruzione un immediato a 20 bit (con segno) che chiameremo `imm20`
 - L'operazione svolta dalla `auipc` è "add upper immediate and PC" (essendo ad es. `x5` il registro di destinazione e PC il Program Counter):
$$x5 = PC + (imm20 \ll 12)$$

- Esempio:

```
Label: auipc x5,0x12345    #mette in x5 l'indirizzo di Label (PC) + 0x12345000
```

- A questo punto occorre però una istruzione che compia:

$$PC = x5 + imm12$$

- Questa istruzione non è altro che la `jalr`:

```
jalr x0,0x678(x5)    #  $x0 \leftarrow PC+4$  (operazione nulla) e  $PC \leftarrow x5+0x678$ 
```

- La coppia di istruzioni

```
auipc x5,0x12345      #  $x5 = PC + 0x12345000$ 
```

```
jalr x0,0x678(x5)     #  $newPC = x5 + 0x678 (=PC + 0x12345678)$ 
```

Realizza il salto (incondizionato) a `PC+0x12345678`

Operazioni connesse alla chiamata di funzione

1) eventuale salvataggio registri da preservare (es. t0-t6, a0-a7, <u>nello stack param. oltre l'ottavo</u>)	pre-chiamata (lato chiamante)
2) preparazione dei parametri di ingresso (nuovi a0-a7)	
3) chiamata della funzione	JAL MYFUN
4) allocazione del call-frame nello stack	prologo (lato chiamato)
5) eventuali salvataggi vari (es. (old) a0-a7, (old) ra, (old) fp/s0, (old) s1-s11)	
6) eventuale <u>inizializzazione nuovo fp</u>	
7) <u>esecuzione del codice della funzione</u>	corpo (lato chiamato)
8) preparazione dei parametri di uscita (a0-a1)	epilogo (lato chiamato)
9) ripristino dei parametri salvati (salvati al punto 5)	
10) ritorno al codice originario	RET
11) uso dei valori di uscita della funzione	post-chiamata (lato chiamante)
12) eventuale ripristino dei vecchi valori (t0-t6, a0-a7) salvati al punto 1	

- I vari "**salvataggi**" vengono fatti in una zona di memoria chiamata "record di attivazione" (o "call frame")
- I call-frame sono gestiti con politica **LIFO** (Last-In, First-Out), e quindi conviene gestirli con uno **STACK**

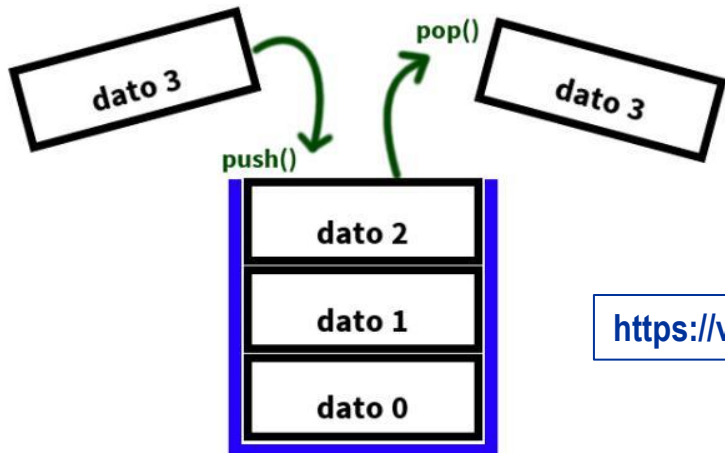
I registri

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

```
int f() {  
    ...  
    g()  
    ...  
}  
  
f: Caller  
g: Callee
```

Lo stack

- Lo stack è un'area di memoria contigua gestita in modalità Last In First Out (LIFO), cioè l'ultimo oggetto inserito è il primo ad essere rimosso.
- Le due operazioni principali sono push (aggiunge un elemento in cima allo stack) e pop (rimuove un elemento dalla cima dello stack).
- Un registro chiamato stack pointer (SP) serve per determinare dove si trovano i vari elementi. Vediamo tra poco.
- Lo stack consiste di un'insieme di segmenti logici (stack frame) che vengono inseriti nello stack quando viene chiamata una funzione ed eliminati quando la funzione ritorna.



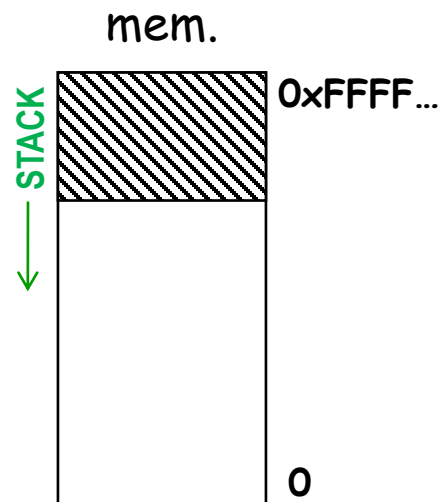
<https://vitolavecchia.altervista.org/che-cose-e-come-funziona-lo-stack-in-informatica/>

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

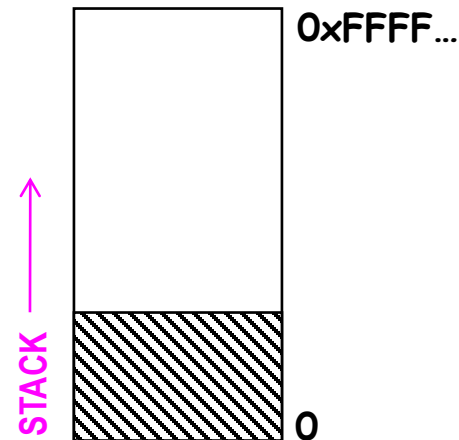
- Assumendo che nella rappresentazione della memoria gli indirizzi alti siano in alto e quelli bassi in basso...
 - Con '0xFFFF...' indico gli indirizzi alti in memoria
 - Con '0' indico gli indirizzi bassi in memoria

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

Convenzione grow down



Convenzione grow up



RISC-V



Convenzione sugli stack (2): “a cosa punta sp”

- 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

Operazioni sullo stack:

Grow-down + Next-Empty

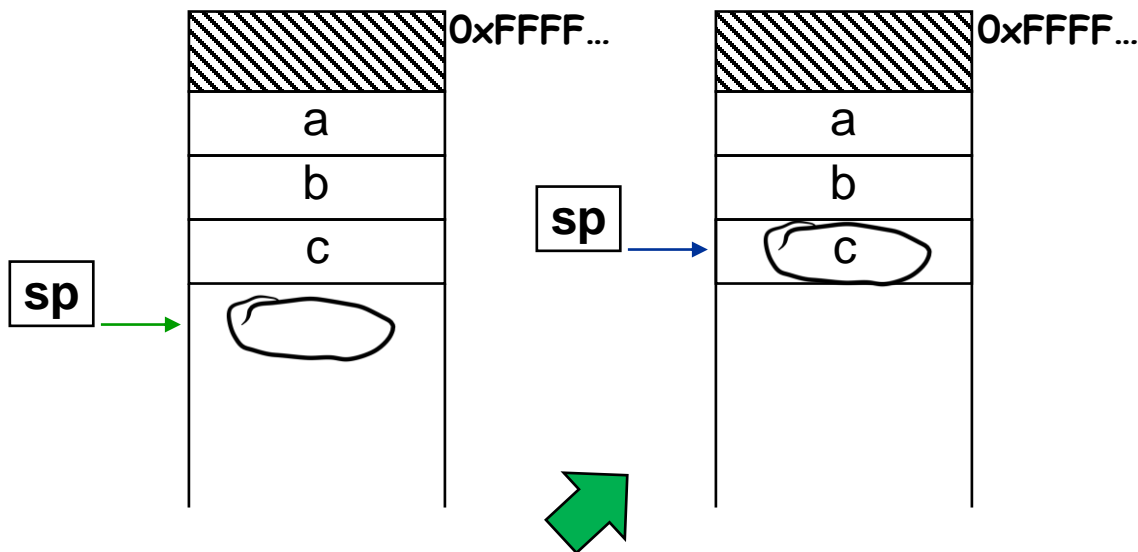
POP: Incrementa sp
Legge da Mem[sp]

PUSH: Scrive in Mem[sp]
Decrementa sp

Grow-down + Last-Full

POP: Legge da Mem[sp]
Incrementa sp

PUSH: Decrementa sp
Scriva in Mem[sp]

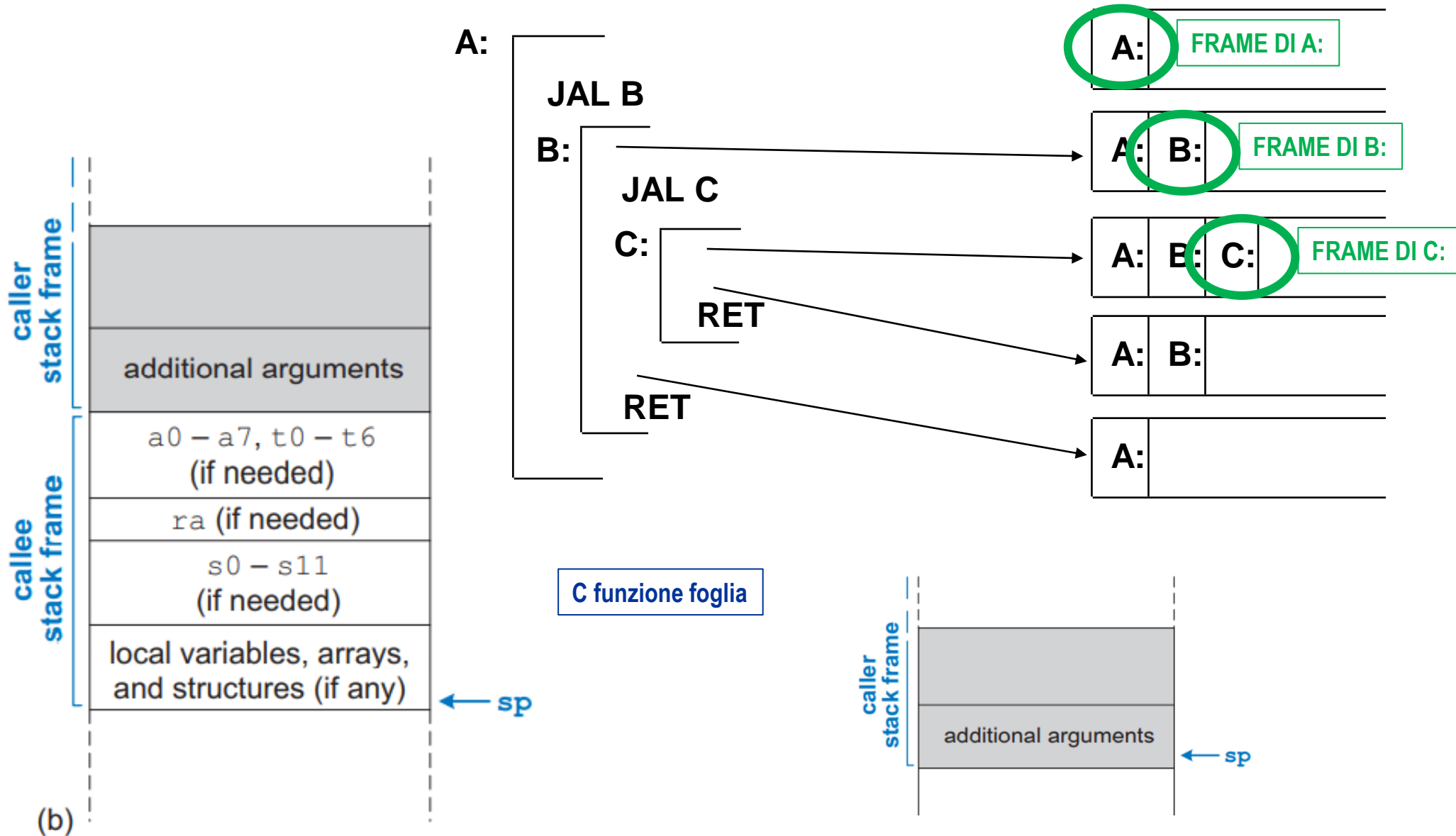


- nel caso RISC-V la convenzione è:
 - **Grow-Down+Last-Full**

Nota: lo stack viene usato non solo per i call-frame ma anche per:

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

Uso dello stack nelle chiamate a funzione



Un esempio, pp. 91 ss

Sistema operativo	Puntatori	int	long int	long long int
Windows Microsoft	64 bit	32 bit	32 bit	64 bit
Linux, la maggior parte di Unix	64 bit	32 bit	64 bit	64 bit

```
long long int esempio_foglia(long long int g, long long
int h, long long int i, long long int j)
{
    long long int f;
    f = (g + h) - (i + j);
    return f;
}
```

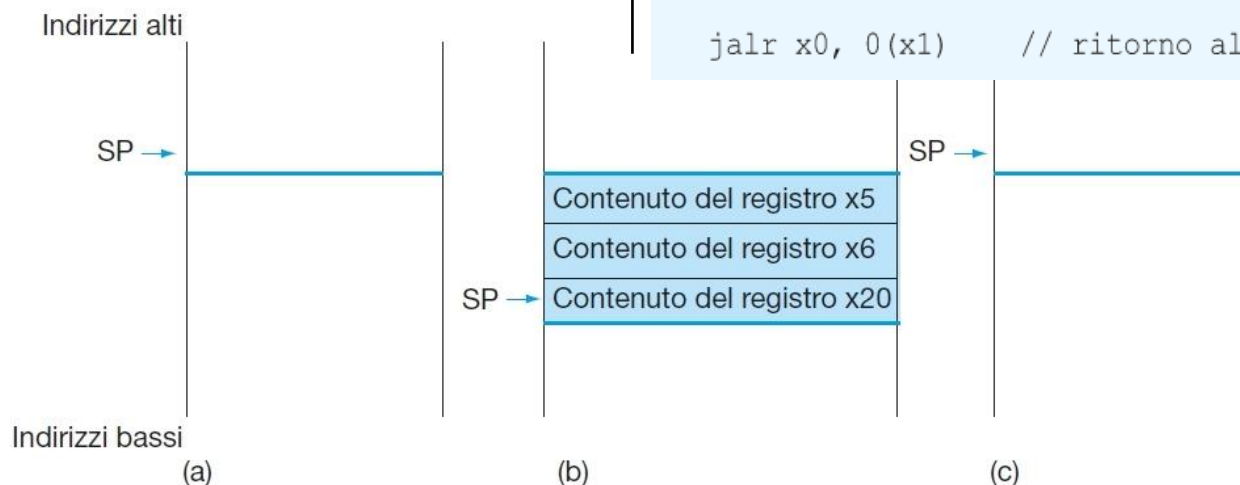
```
add x5, x10, x11 // il registro x5 contiene g + h
add x6, x12, x13 // il registro x6 contiene i + j
sub x20, x5, x6  // f = x5 - x6, cioè (g + h) - (i + j)
```

Per restituire il valore di `f`, occorre copiarlo in un registro dei parametri:

```
addi x10, x20, 0 // restituzione di f (x10 = x20 + 0)
```

La procedura termina con un'istruzione di salto a registro, utilizzando l'indirizzo di ritorno:

```
jalr x0, 0(x1) // ritorno al programma chiamante
```

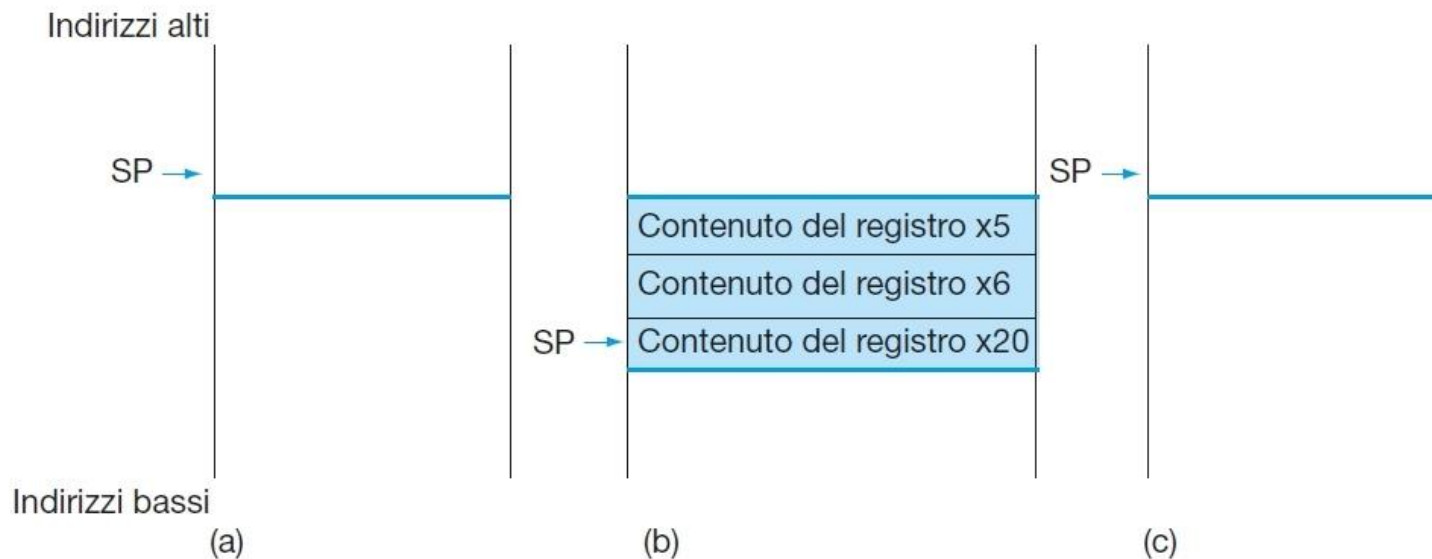


PASSO i parametri
g,h,i,j in x10-x13 (a0-a3)
METTO la var automatica
f in x20 (s4)

Figura 2.10 Contenuto dello stack pointer e situazione dello stack prima (a), durante (b) e dopo (c) la chiamata a procedura. Lo stack pointer punta sempre alla cima (top) dello stack, cioè all'ultima parola doppia presente nello stack.

Push e Pop

```
addi sp, sp, -24 // aggiornamento dello stack pointer
// per fare posto a 3 elementi,
sd x5, 16(sp) // salvataggio del registro x5 per poter
// essere utilizzato successivamente
sd x6, 8(sp) // salvataggio del registro x6, per poter
// essere utilizzato successivamente
sd x20, 0(sp) // salvataggio del registro x20, per poter
// essere utilizzato successivamente
```



```
ld x20, 0(sp) // ripristino del registro x20 per il chiamante
ld x6, 8(sp) // ripristino del registro x6 per il chiamante
ld x5, 16(sp) // ripristino del registro x5 per il chiamante
addi sp, sp, 24 // aggiornamento dello stack pointer con
// l'eliminazione di 3 elementi
```

Però

Conservato	Non conservato
Registri da salvare: x8-x9, x18-x27	Registri temporanei: x5-x7, x28-x31
Registro stack pointer: x2 (sp)	Registri argomento/risultato: x10-x17
Frame pointer: x8 (fp)	
Registro dell'indirizzo di ritorno: x1 (ra)	
Stack al di sopra dello stack pointer	Stack al di sotto dello stack pointer

Figura 2.11 Che cosa viene e che cosa non viene conservato in una chiamata a procedura. Se il software utilizza il registro frame pointer o il registro global pointer, che esamineremo in seguito, anche il contenuto di questi registri deve essere preservato.

```
addi sp, sp, -24 // aggiornamento dello stack pointer
// per fare posto a 3 elementi,
sd x5, 16(sp) // salvataggio del registro x5 per poter
// essere utilizzato successivamente
sd x6, 8(sp) // salvataggio del registro x6, per poter
// essere utilizzato successivamente
sd x20, 0(sp) // salvataggio del registro x20, per poter
// essere utilizzato successivamente
```

Ergo basta `addi sp, sp, -8`

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

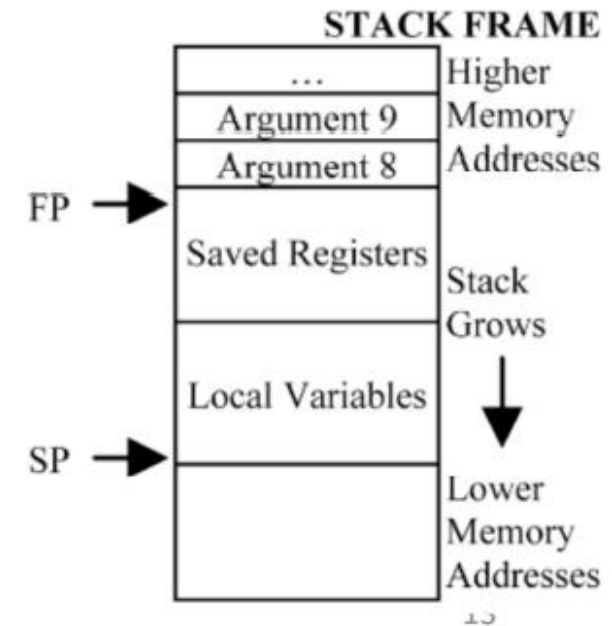
PRE-CHIAMATA (LATO CHIAMANTE)

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 in caso di altra chiamata)

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), 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

Argomenti: esempio

```
int sum10(int a, int b, int c, int d, int e,  
         int f, int g, int h, int i, int j);
```

```
# sum10(10,20,30,40,50,60,70,80,90,100);
```

```
main:
```

```
    li a0, 10      # 1st parameter  
    li a1, 20      # 2nd parameter  
    li a2, 30      # 3rd parameter  
    li a3, 40      # 4th parameter  
    li a4, 50      # 5th parameter  
    li a5, 60      # 6th parameter  
    li a6, 70      # 7th parameter  
    li a7, 80      # 8th parameter  
    addi sp, sp, -8 # Allocate stack space  
    li t1, 100     # Push the 10th parameter  
    sw t1, 4(sp)  
    li t1, 90      # Push the 9th parameter  
    sw t1, 0(sp)  
    jal sum10      # Invoke sum10  
    addi sp, sp, 8  # Deallocate the parameters from stack  
    ret
```

```
sum10:
```

```
    lw t1, 0(sp)   # Loads the 9th parameter into t1  
    lw t2, 4(sp)   # Loads the 10th parameter into t2  
    add a0, a0, a1  # Sums all parameters  
    add a0, a0, a2  
    add a0, a0, a3  
    add a0, a0, a4  
    add a0, a0, a5  
    add a0, a0, a6  
    add a0, a0, a7  
    add a0, a0, t1  
    add a0, a0, t2  # Place return value on a0  
    ret            # Returns
```

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

PROLOGO (LATO CHIAMATO)

4) Eventuale allocazione del call-frame sullo stack
(→ aggiornare sp)

5) Eventuale salvataggio registri che si intende sovrascrivere

- Salvataggio degli argomenti a0-a7 solo se la funzione ha necessita' di riusarli nel corpo di questa funzione, successivamente a ulteriori chiamate a funzione che usino tali registri,
(nota: negli altri casi a0-a7 possono essere sovrascritti)
- Devo salvare il vecchio ra, solo se la funzione chiama altre funzioni...
- 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)

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

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)

9) I registri (se salvati) devono essere ripristinati

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

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

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

Perché a0 ed a1?

- The RISC-V ilp32 ABI defines that values should be returned in register a0.
- In case the value being returned is 64-bit long, then the least significant 32 bits must be returned in register a0 and the most significant 32 bits must be returned in register a1.

Integer ABIs

ilp32

- int, long, pointers are 32bit
- long long is 64bit
- char is 8bit
- short is 16bit

ilp32 is currently only supported for 32-bit targets.

lp64

- int is 32bit
- long and pointers are 64bit
- long long is 64bit
- char is 8bit
- short is 16bit

lp64 is only supported for 64-bit targets.

Floating Point ABIs

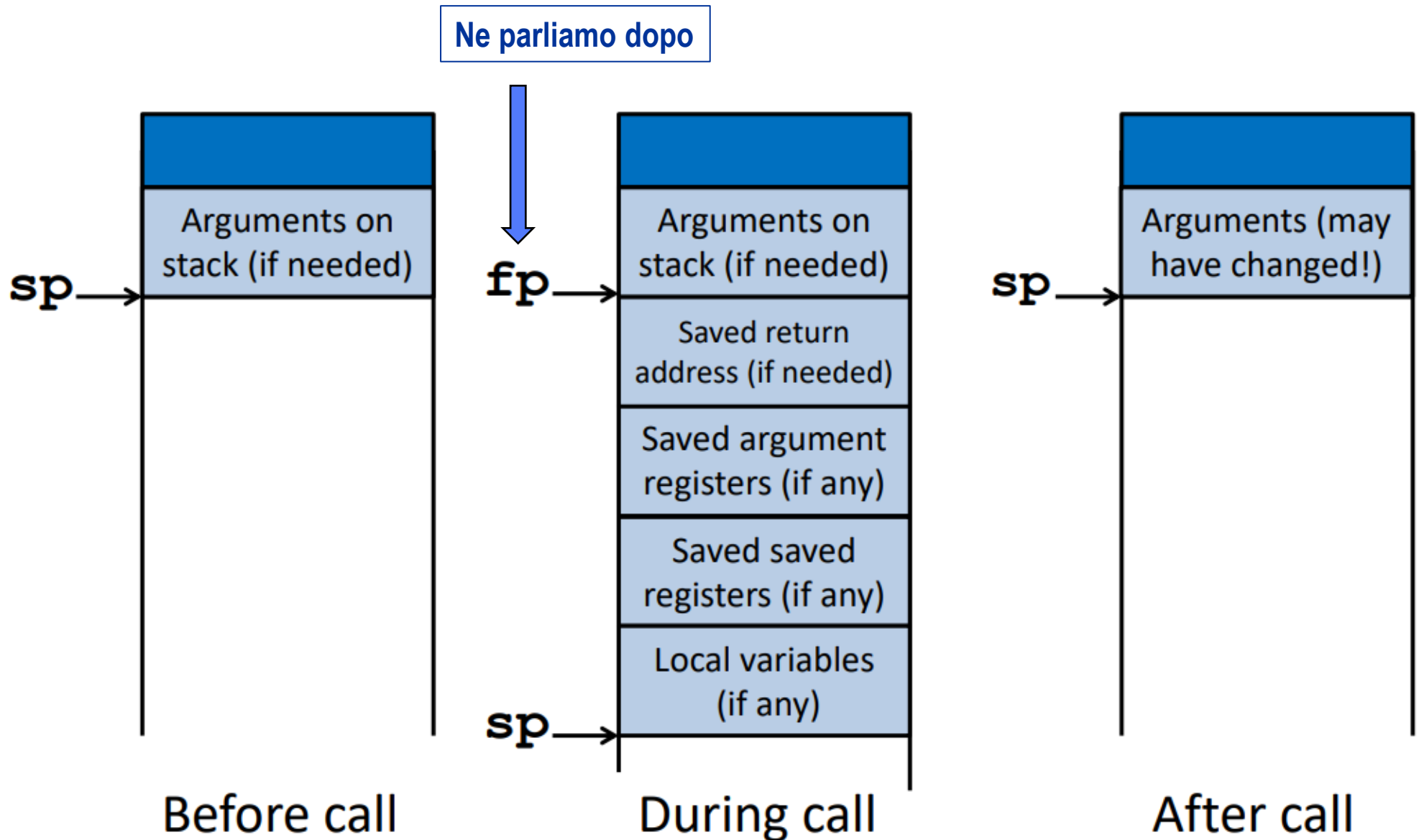
f

- 32bit and smaller floating point types are passed in registers.
- Requires F type floating point registers and instructions.

d

- 64bit and smaller floating point types are passed in registers.
- Requires D type floating point registers and instructions.

Riepilogo



Esempio

High-Level Code

```
int main(){
    int y;
    . . .

    y = diffofsums(2, 3, 4, 5);
    . . .
}

int diffofsums(int f, int g, int h, int i){
    int result;

    result = (f + g) - (h + i);

    return result;
}
```

RISC-V Assembly Code

```
# s7 = y
main:
    . . .
    addi a0, zero, 2    # argument 0 = 2
    addi a1, zero, 3    # argument 1 = 3
    addi a2, zero, 4    # argument 2 = 4
    addi a3, zero, 5    # argument 3 = 5
    jal  diffofsums     # call function
    add  s7, a0, zero   # y = returned value

# s3 = result
diffofsums:
    addi sp, sp, -4     # make space on stack to store one register
    sw   s3, 0(sp)      # save s3 on stack
    add  t0, a0, a1     # t0 = f + g
    add  t1, a2, a3     # t1 = h + i
    sub  s3, t0, t1     # result = (f + g) - (h + i)
    add  a0, s3, zero   # put return value in a0
    lw   s3, 0(sp)      # restore s3 from stack
    addi sp, sp, 4      # deallocate stack space
    jr   ra             # return to caller
```

Value vs Reference

```
int pow2(int v)
{
    return v*v;
}
```

```
pow2:
    mul a0, a0, a0 # a0 = a0 * a0
    ret           # return
```

```
void inc(int* v)
{
    *v = *v + 1;
}
```

```
inc:
    lw  a1, (a0) # a1 = *v
    addi a1, a1, 1 # a1 = a1 + 1
    sw  a1, (a0) # *v = a1
    ret
```

Sia int abc = 5;

Passaggio per valore

la t0, abc

lw a0, 0(t0)

jal pow2

abc = 5

Passaggio per riferimento

la a0, abc

jal inc

Alla fine

abc = 6

Esempio ricorsivo

High-Level Code

```
int factorial(int n) {  
    if (n <= 1)  
        return 1;  
  
    else  
        return (n * factorial(n - 1));  
}
```

RISC-V Assembly Code

```
0x8500 factorial: addi sp, sp, -8           # make room for a0, ra  
0x8504             sw  a0, 4(sp)  
0x8508             sw  ra, 0(sp)  
0x850C             addi t0, zero, 1        # temporary = 1  
0x8510             bgt  a0, t0, else       # if n > 1, go to else  
0x8514             addi a0, zero, 1        # otherwise, return 1  
0x8518             addi sp, sp, 8          # restore sp  
0x851C             jr   ra                # return  
0x8520 else:       addi a0, a0, -1        # n = n - 1  
0x8524             jal  factorial          # recursive call  
0x8528             lw   t1, 4(sp)         # restore n into t1  
0x852C             lw   ra, 0(sp)         # restore ra  
0x8530             addi sp, sp, 8          # restore sp  
0x8534             mul  a0, t1, a0        # a0 = n * factorial(n - 1)  
0x8538             jr   ra                # return
```

Stack e Frame Pointer

- Le variabili **automatiche** sono variabili locali in un blocco di istruzioni (e quindi anche in una funzione). Altrimenti sono statiche.
- Esse sono automaticamente allocate sullo **stack** quando si entra in quel blocco di codice **SE** non ho abbastanza registri disponibili
 - Vettori e strutture invece sono direttamente allocate nello stack
- Le variabili automatiche vengono distrutte quando si esce dal blocco stesso.
- SP quindi potrebbe cambiare nel corso della procedura, per cui si usa un nuovo registro
- FP frame pointer: punto alla prima DW di un record di attivazione
 - NON è obbligatorio

```
int s = 2; //Variabile statica

int main(void)
{
    {
        int a; //Variabile automatica
        a = 10;
    }
    {
        int b; //Variabile automatica
        //printf("a = %d\n", a);
        printf("b = %d\n", b); //Cosa ottengo?
        printf("s = %d\n", s); //2
    }
}
```

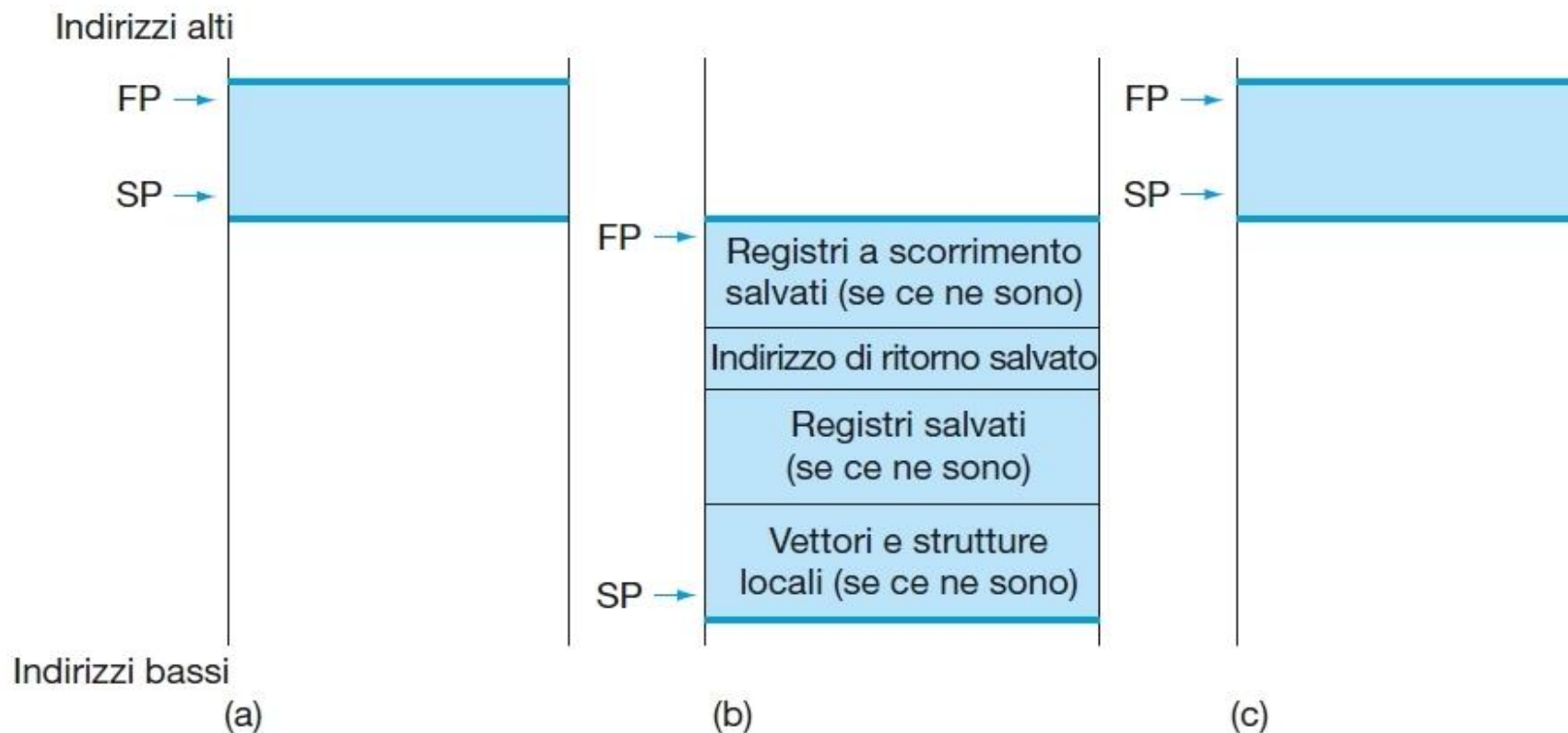


Figura 2.12 Allocazione dello stack prima (a), durante (b) e dopo (c) la chiamata a procedura. Il frame pointer (fp o x8) punta alla prima parola doppia del frame, spesso contenente un registro da preservare, e lo stack pointer (sp) punta alla cima dello stack. Lo stack viene aggiornato per fare spazio a tutti i registri da preservare e alle variabili locali che devono essere salvate in memoria. Dal momento che lo stack pointer può cambiare durante l'esecuzione del programma, è più facile per i programmatori indirizzare le variabili in stack tramite il frame pointer, che invece rimane stabile (anche se ciò potrebbe essere fatto con il solo stack pointer e un po' di aritmetica sugli indirizzi). Se lo stack non contiene variabili locali alla procedura, il compilatore risparmia tempo di esecuzione evitando di impostare e ripristinare il frame. Quando viene utilizzato, il frame pointer viene inizializzato con l'indirizzo che ha sp all'atto della chiamata della procedura e sp viene ripristinato al termine della procedura utilizzando il valore di fp . Si possono trovare queste informazioni anche nella quarta colonna della scheda tecnica riassuntiva del RISC-V in fondo al libro.

gcc option `-fomit-frame-pointer`

Most smaller functions don't need a frame pointer - larger functions MAY need one.

It's really about how well the compiler manages to track how the stack is used, and where things are on the stack (local variables, arguments passed to the current function and arguments being prepared for a function about to be called). I don't think it's easy to characterize the functions that need or don't need a frame pointer (technically, NO function HAS to have a frame pointer - it's more a case of "if the compiler deems it necessary to reduce the complexity of other code").

I don't think you should "attempt to make functions not have a frame pointer" as part of your strategy for coding - like I said, simple functions don't need them, so use `-fomit-frame-pointer`, and you'll get one more register available for the register allocator, and save 1-3 instructions on entry/exit to functions. If your function needs a frame pointer, it's because the compiler decides that's a better option than not using a frame pointer. It's not a goal to have functions without a frame pointer, it's a goal to have code that works both correctly and fast.

Note that "not having a frame pointer" should give better performance, but it's not some magic bullet that gives enormous improvements - particularly not on x86-64, which already has 16 registers to start with. On 32-bit x86, since it only has 8 registers, one of which is the stack pointer, and taking up another as the frame pointer means 25% of register-space is taken. To change that to 12.5% is quite an improvement. Of course, compiling for 64-bit will help quite a lot too.

Function inlining

- L'uso delle funzioni offre molti vantaggi
- Però le funzioni hanno un costo relativo alla creazione del record di attivazione.
- Il compilatore può decidere di sostituire il codice all'interno della definizione della funzione al posto di ogni chiamata a tale funzione.

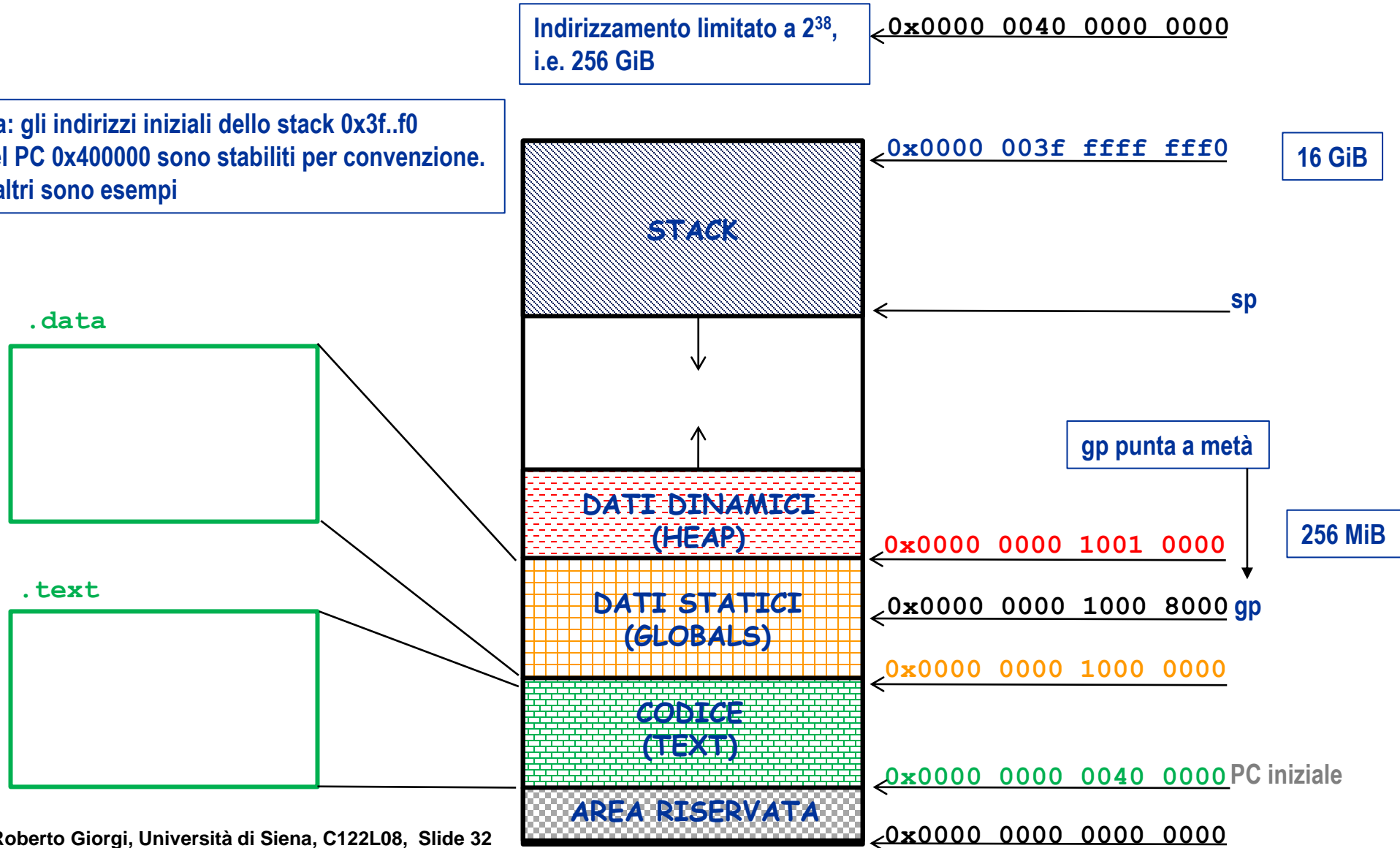
```
int max(int a, int b) { return a < b ? b : a; }
```

- La sostituzione del codice inline viene eseguita a discrezione del compilatore. Ad esempio, il compilatore non effettuerà questa modifica ad esempio se decide che è troppo grande.
- Quindi voi continuate ad usarle, alle ottimizzazioni pensa il compilatore.

Mappa di memoria (Memory Layout)

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

Nota: gli indirizzi iniziali dello stack 0x3f..f0 e del PC 0x400000 sono stabiliti per convenzione. Gli altri sono esempi



Record di attivazione

http://www.diit.unict.it/users/scava/dispense/Fdl_270/RecordAttivazioneC.pdf

- Il return address è PC+4 rispetto a $y = \text{fact}(x)$
- Il link serve a ripristinare lo stato del main (record di attivazione)
- Non del tutto consistente con quanto visto finora ma lettura interessante

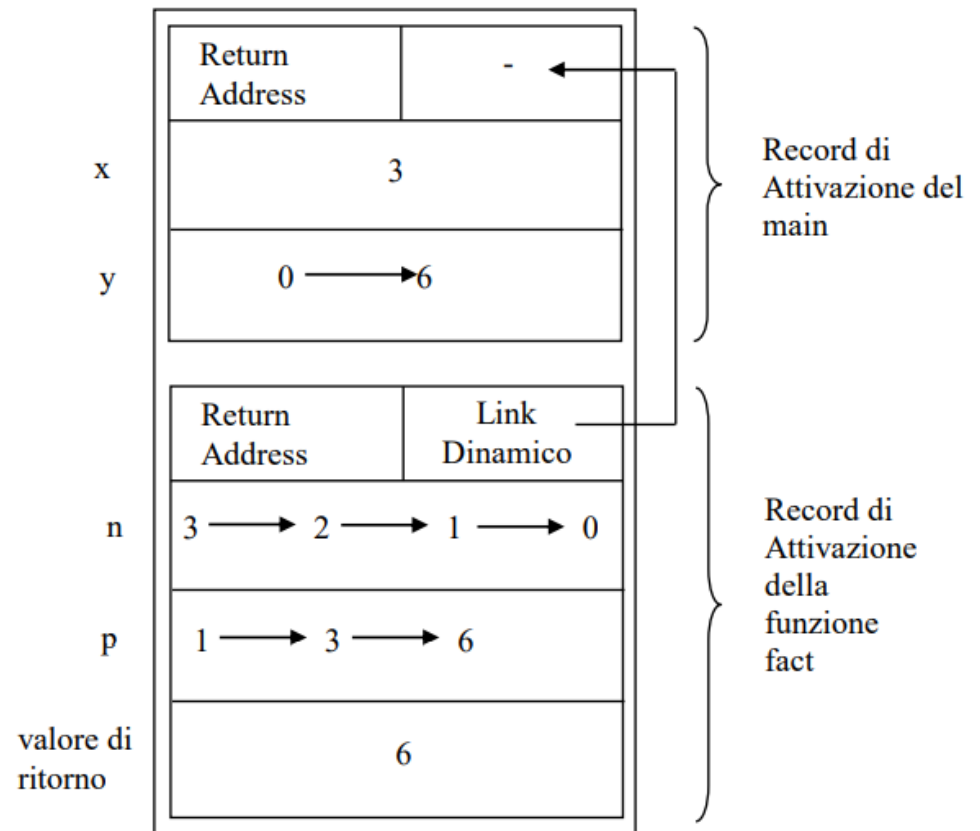
```
#include <stdio.h>

int fact ( int n)
{
    int p;

    p=1;
    while (n>0) p*=n--;
    return p;
}

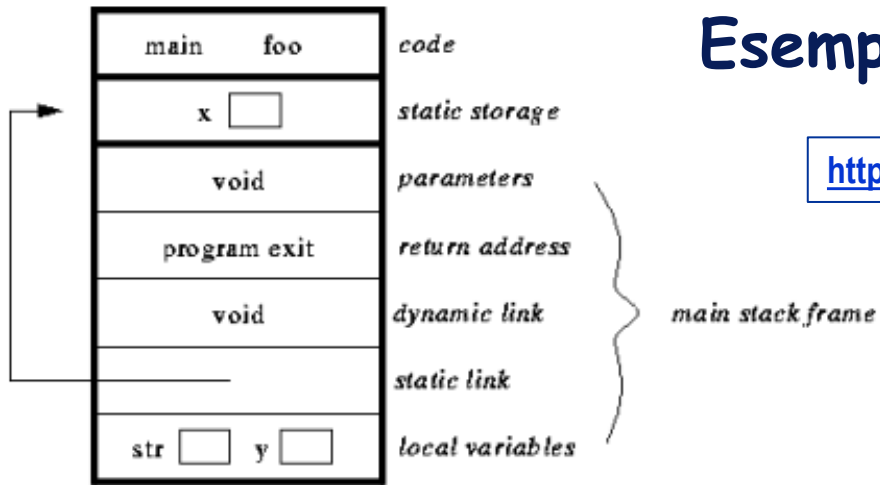
int main(void)
{
    int x, y;

    x=3;
    y=fact (x);
}
```



Esempio

<https://web.archive.org/web/20170829060314/http://www.inf.udec.cl/~leo/teoX.pdf>



```
int x;                                /* static storage */

void main() {
    int y;                            /* dynamic stack storage */
    char *str;                        /* dynamic stack storage */

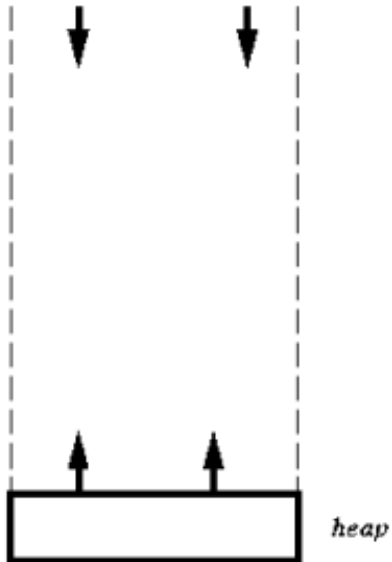
    str = malloc(100); /* allocates 100 bytes of dynamic heap storage */

    y = foo(23);
    free(str);                    /* deallocates 100 bytes of dynamic heap storage */
    /* y and str deallocated as stack frame is popped */
}

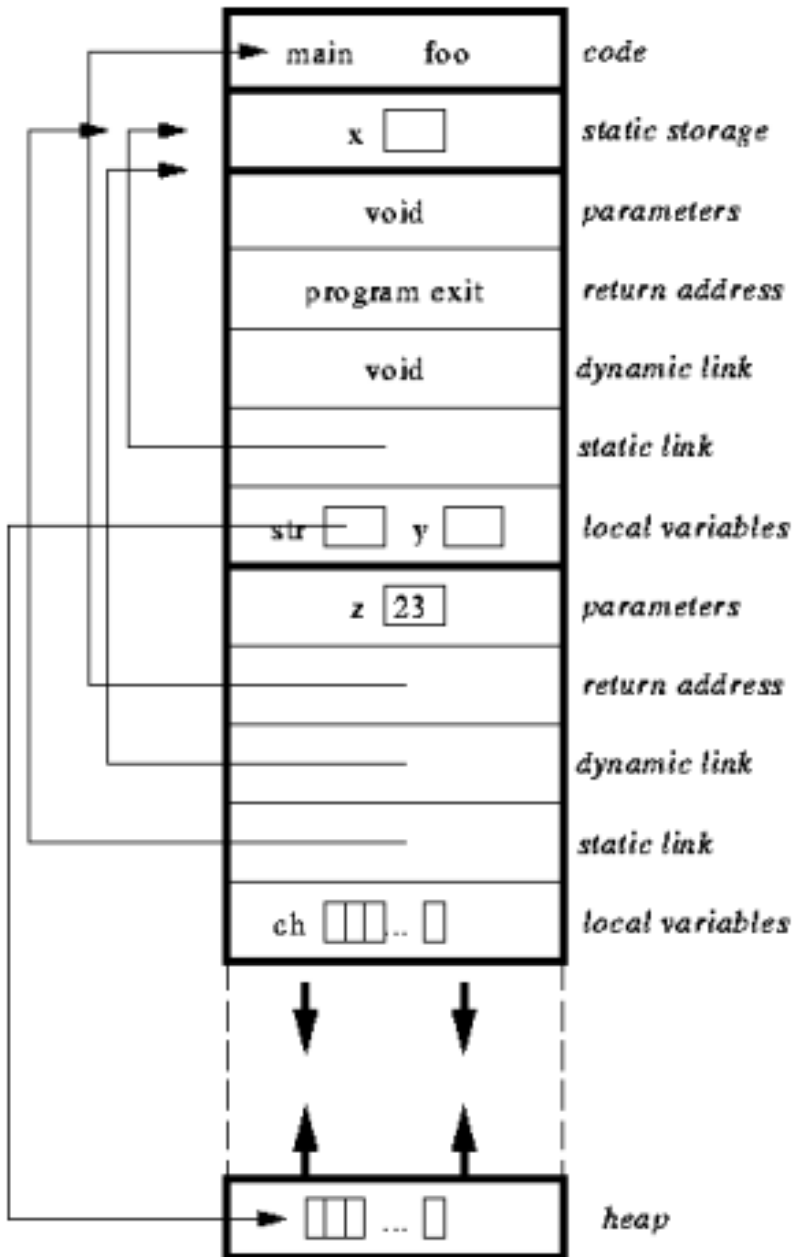
int foo(int z) {                     /* z is dynamic stack storage */
    char ch[100];                   /* ch is dynamic stack storage */

    if (z == 23) foo(7);

    return 3;                       /* z and ch are deallocated as stack frame is popped,
                                     3 put on top of stack */
}
```



Esempio



```

int x;                                /* static storage */

void main() {
    int y;                             /* dynamic stack storage */
    char *str;                         /* dynamic stack storage */

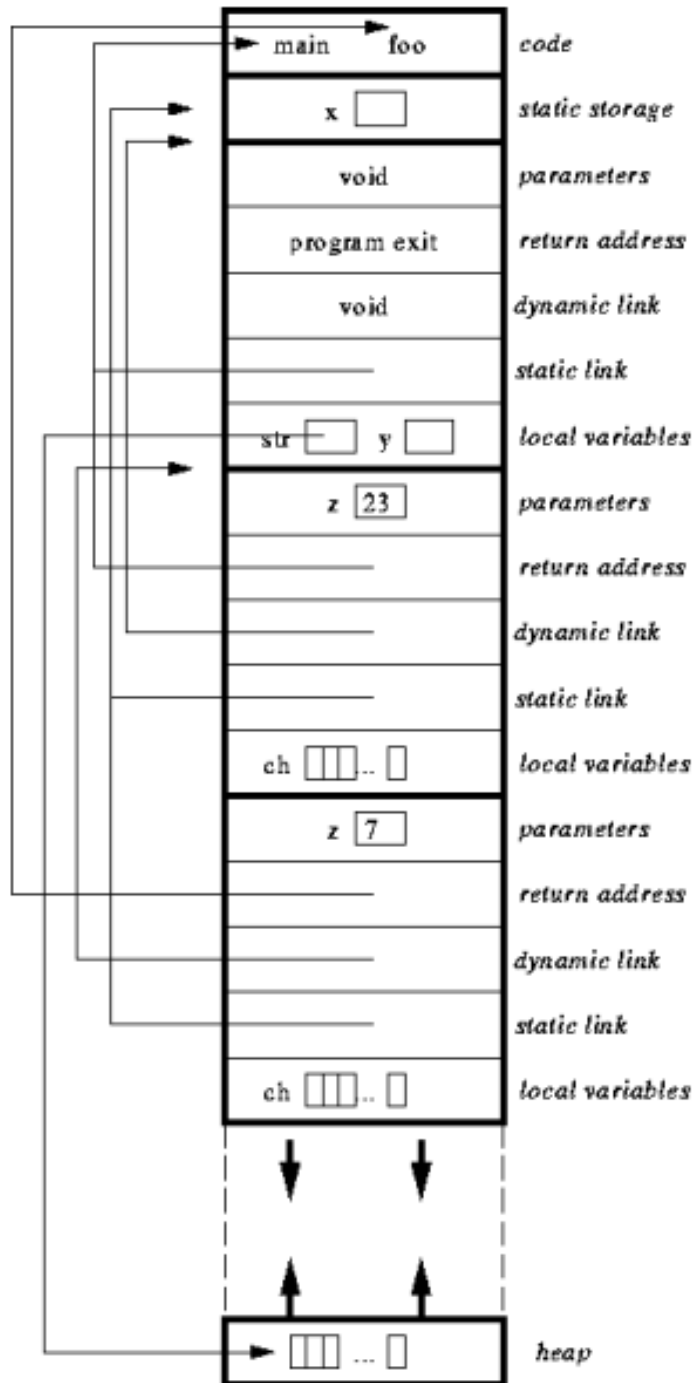
    str = malloc(100); /* allocates 100 bytes of dynamic heap storage */

    y = foo(23);
    free(str); /* deallocates 100 bytes of dynamic heap storage */
    /* y and str deallocated as stack frame is popped */
}

int foo(int z) { /* z is dynamic stack storage */
    char ch[100]; /* ch is dynamic stack storage */

    if (z == 23) foo(7);

    return 3; /* z and ch are deallocated as stack frame is popped,
               3 put on top of stack */
}
    
```

[illegible]

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
→ per controllare di aver scritto bene ciascun modulo
 - **Collegamento** di più moduli
→ per creare un unico file da essere eseguito (il programma)
 - **Caricamento** del programma in memoria e sua esecuzione
→ per consentire al processore di eseguire il nostro programma

- **Compilazione** separata

A che serve?

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

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

- vengono così creati (salvo errori) i file `main1.o` `queue.o`

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

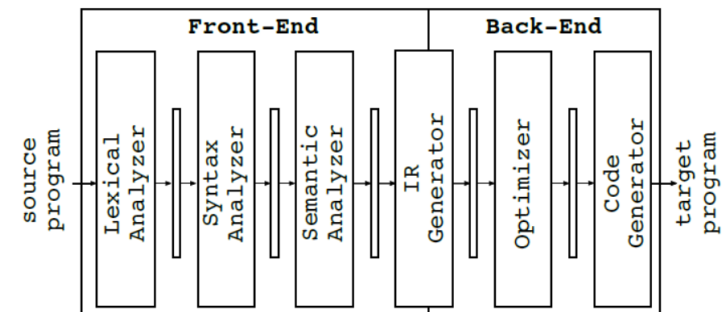
```
cc -o main main1.o queue.o
```

- **Caricamento** ed esecuzione

```
./main
```

Passi principali della compilazione di un programma C

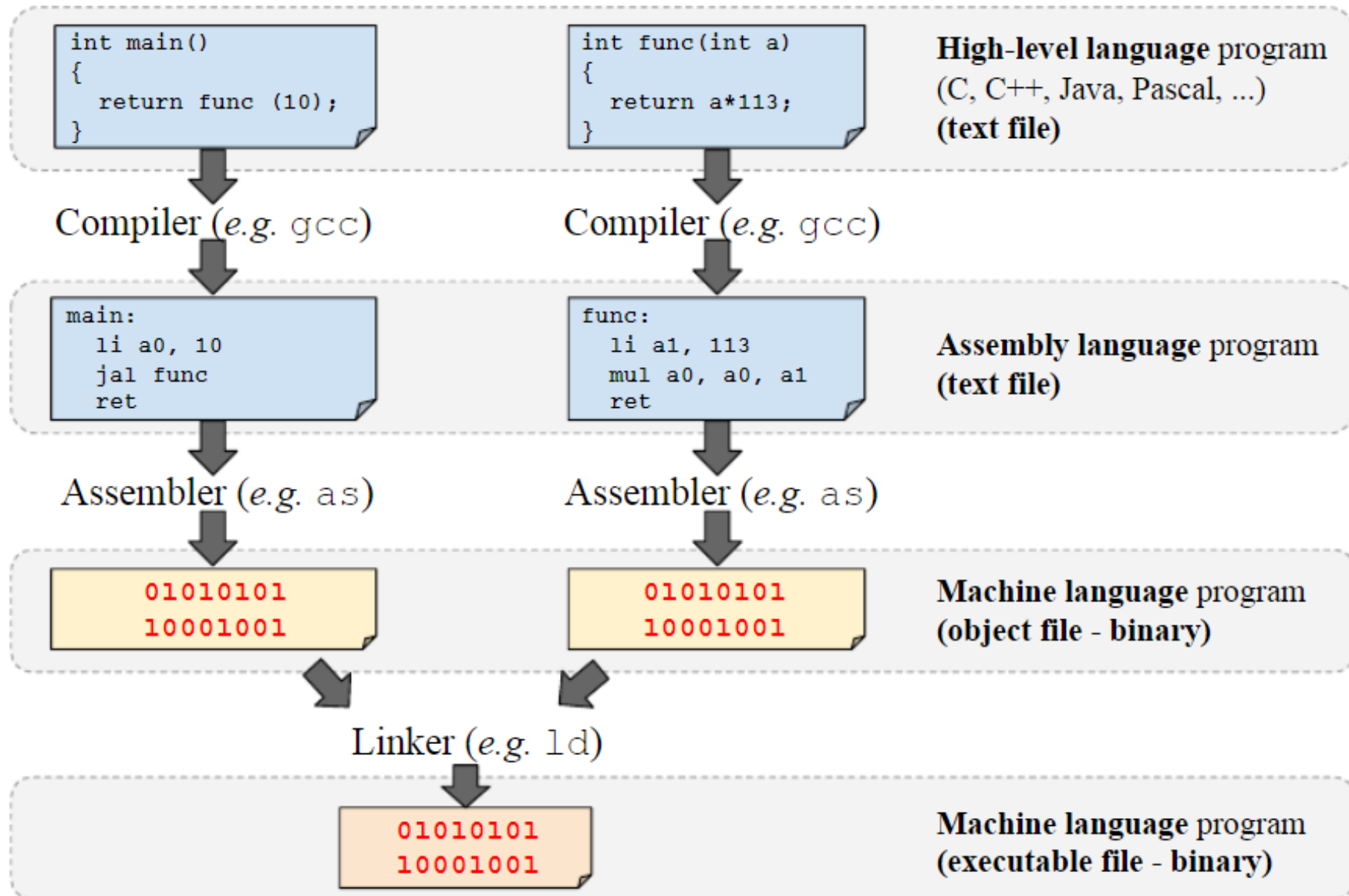
- **Preprocessore (stadio 'cpp')**
 - espande le macro e le costanti (`#define`)
 - inserisce nell'elaborato i file inclusi (`#include`)
 - processa le direttive condizionali (`#ifdef`, `#ifndef`, ...)
- **Compilazione vera e propria**
 - si svolge in uno o più stadi (cc1, cc2) fino ad ottenere il livello di **ottimizzazione** desiderato
 - il risultato intermedio è un programma in **ASSEMBLY**
 - il codice **ASSEMBLY** è legato all'architettura target
- **Creazione del file oggetto**
 - questo passo è eseguito dall'assemblatore
 - il file oggetto contiene
 - la codifica delle istruzioni assembly
 - i dati statici
 - informazioni di rilocazione
 - tabella dei simboli
 - altre informazioni di utilità



Visione semplificata di un compilatore

https://www.researchgate.net/publication/332241231_Principles_Techniques_and_Tools_for_Explicit_and_Automatic_Parallelization

La compilazione



Il linker

- `gcc -c main.c`
- `gcc -c func.c`
- `nm main.o`

U(ndefined) func

000000000000 T main

- `nm func.o`
- 000000000000 T func
- `gcc main.o func.o`
- `nm a.out`

UNIX/LINUX: il comando **nm** consente di produrre informazioni sul layout del file oggetto

```
0000000000004000 W data_start
0000000000001070 t deregister_tm_clones
00000000000010e0 t __do_global_dtors_aux
0000000000003df8 d __do_global_dtors_aux_fini_array_entry
0000000000004008 D __dso_handle
0000000000003e00 d _DYNAMIC
0000000000004010 D _edata
0000000000004018 B _end
00000000000011d8 T _fini
0000000000001120 t frame_dummy
0000000000003df0 d __frame_dummy_init_array_entry
0000000000002154 r __FRAME_END__
                                U func
0000000000003fc0 d _GLOBAL_OFFSET_TABLE_
                                w __gmon_start__
0000000000002004 r __GNU_EH_FRAME_HDR
0000000000001000 t _init
0000000000003df8 d __init_array_end
0000000000003df0 d __init_array_start
0000000000002000 R _IO_stdin_used
                                w _ITM_deregisterTMCloneTable
                                w _ITM_registerTMCloneTable
00000000000011d0 T __libc_csu_fini
0000000000001160 T __libc_csu_init
                                U __libc_start_main@@GLIBC_2.2.5
0000000000001129 T main
00000000000010a0 t register_tm_clones
0000000000001040 T _start
0000000000004010 D __TMC_END__
```


Assemblatore a due passate (1)

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

```
.text
    add    x18, x18, x0          LC=0
    addi   x19, x19, 0          LC=4
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
    la     x18, myvar            LC=24
    .....
    jal    fun1                  LC=3000
    .....

fun1: .....                     LC=20' 000' 000
    .....
    .....

.data
myvar: .....                     LC=0
```

Symbol Table (provvisoria)

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

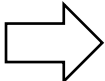
Assemblatore a due passate (2)

- **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. Per la bne con LC=20 che usa il simbolo 'loop'**

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

$8 - [20] = -12$  `bne x18, x5, -12`

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

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

- Per ogni etichetta irrisolta, si controlla che non sia definita come "simbolo esterno" tramite `.globl` (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 (es. nelle istruzioni `jal`, `j`, `la`)
→ **deve restare in symbol table**

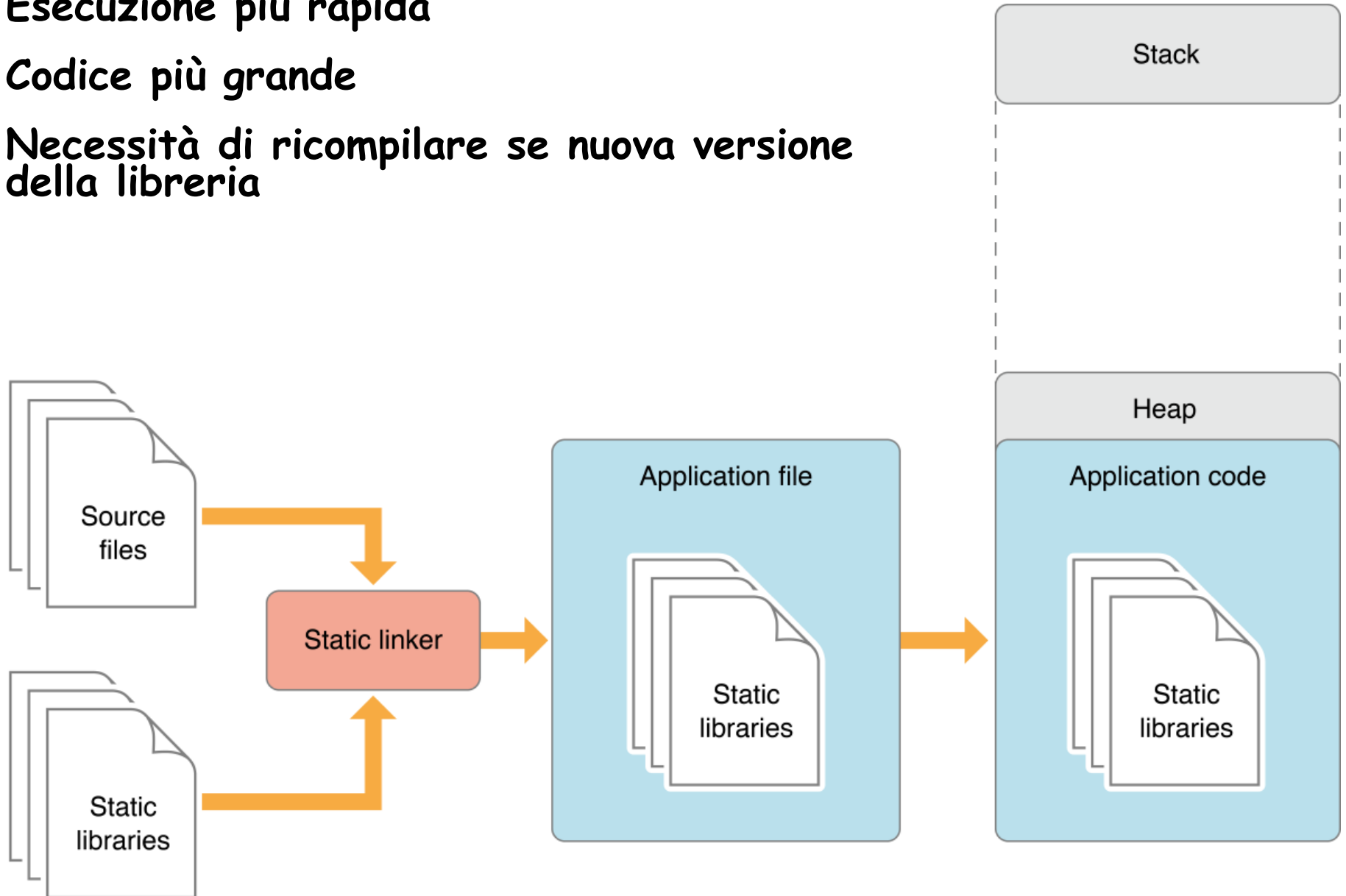
symbol	Symbol-LC
myvar	0
fun1	20'000'000

Formato del file oggetto (.o)

HEADER	• Dimensione del file, delle parti (text, data) e CRC
TEXT segment	• Codifica delle istruzioni (linguaggio macchina DA RILOCARE)
DATA segment	• Dati
RELOCATION info	• Lista delle istruzioni che fanno riferimento ad indirizzi assoluti
SYMBOL table	• Lista dei simboli locali (etichette, variabili) e etichette visibili all'esterno (assembly .globl)
DEBUGGING info	• Parte opzionale per facilitare il debugging del programma

Librerie statiche e dinamiche

- Esecuzione più rapida
- Codice più grande
- Necessità di ricompilare se nuova versione della libreria



Librerie statiche e dinamiche

- Necessito di un linker/loader dinamico che carica la libreria solo quando serve (se lazy) e inserisce gli indirizzi corretti nei salti

