
Lezione 3

Assembly MIPS (3)

<http://www.dii.unisi.it/~giorgi/didattica/arcal1>

All figures from Computer Organization and Design: The Hardware/Software Approach, Second Edition, by David Patterson and John Hennessy, are copyrighted material. (COPYRIGHT 1998 MORGAN KAUFMANN PUBLISHES, INC. ALL RIGHTS RESERVED.) Figures may be reproduced only for classroom or personal educational use in conjunction with the book and only when the above copyright line is included. They may not be otherwise reproduced, distributed, or incorporated into other works without the prior written consent of the publisher.

Indirizzamento nelle istruzioni Branch e Jump

- Per variare il flusso sequenziale delle istruzioni si usa:

bne \$t4,\$t5,Label

l'istruzione successiva e' a Label se \$t4 != \$t5

beq \$t4,\$t5,Label

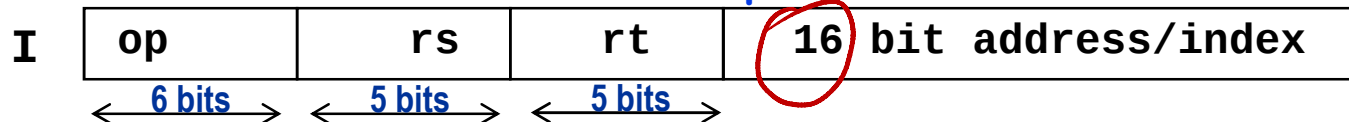
l'istruzione successiva e' a Label se \$t4 == \$t5

j Label

l'istruzione successiva e' a Label

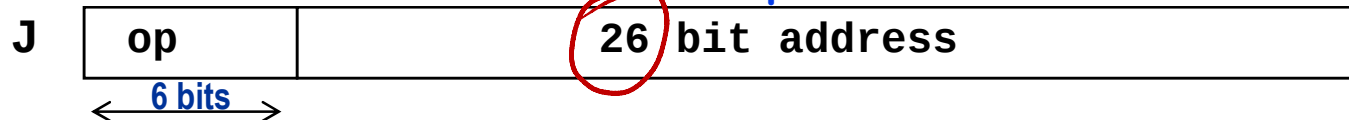
- Per la codifica dell'istruzione, nel caso di bne, beq dovendo inserire anche gli indici di due registri si usa

- Formato I → restano 16 bit per codificare DOVE saltare



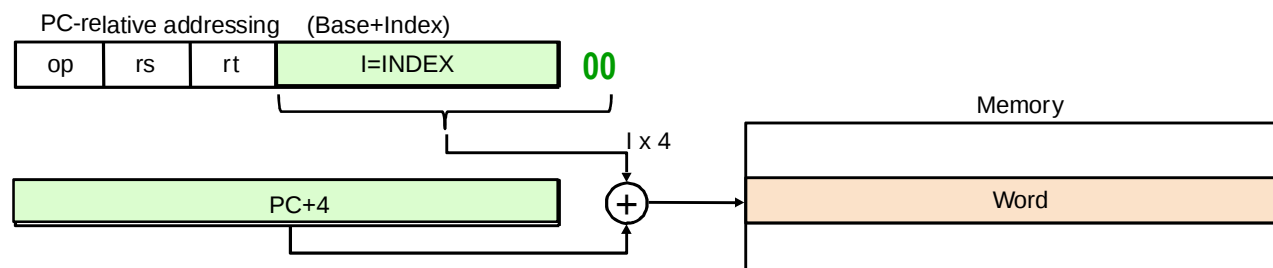
- Nel caso di j (jump) si usa

- Formato J → restano 26 bit per codificare DOVE saltare



- Come si possono sfruttare al massimo questi bit?

Indirizzamento relativo al PC (bne, beq)



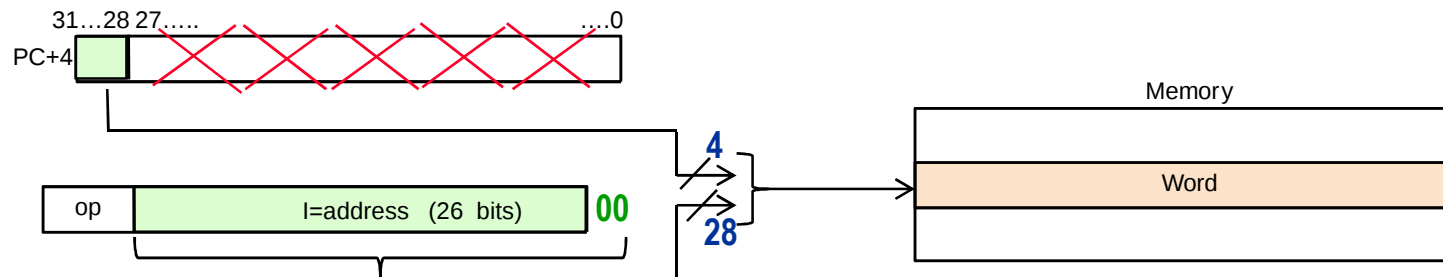
- E' anche noto come indirizzamento 'Base+Indice'
- I branch tipicamente saltano vicino (principio di localita': la distanza e' quasi sempre <20 istr.)

$$A = (PC+4) + 4 \cdot I$$

Nota: "PC+4" tiene conto del valore assunto da PC nel momento del calcolo dell'indirizzo effettivo A

- A= indirizzo effettivo utilizzato
 - PC=Program Counter DELL'ISTRUZIONE
 - I= indice (campo 'index' dell'istruzione), puo' essere negativo
- Poiche' l'indirizzo di una istruzione e' sempre multiplo di 4 il campo effettivo in cui posso spaziare e' 2^{18} (256KB)

Indirizzamento Pseudoindiretto



$A = \{ (PC+4)[31:28], I*4 \}$ ← Notazione Verilog: significa «concatenamento»

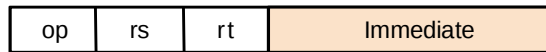
- A = indirizzo effettivo utilizzato
- PC = Program Counter
- I = indice (campo 'address' dell'istruzione)

- Così l'ampiezza massima di salto è 2^{28} (256 MB)
 - La memoria è quindi suddivisa in 16 «fette» ampie 256MB
 - Il loader e il linker controllano che non si scavalchino "confini" di 256MB

Nota: per saltare ad un indirizzo qualunque a 32 bit si usa l'istruzione `jr` (jump relative) ovvero la coppia `la $x, addr; jr $x`

Tabella riassuntiva modi indirizzamento MIPS:

1. Immediate addressing



addi, andi, ori
slti, lui
addiu, sltiu

2. Register addressing: (direct addressing)

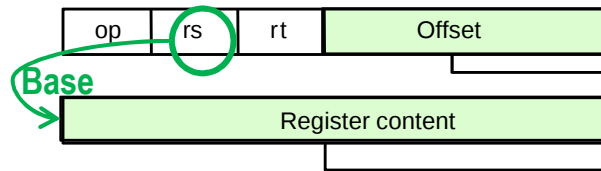


Registers

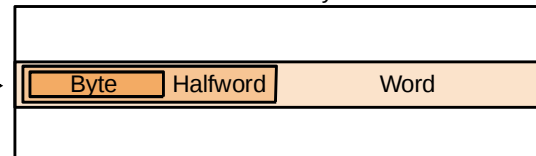
Register content

add, sub, mult, div, slt
addu, subu, multu, divu, sltu
and, or, xor, nor,
sll, srl, sra
mfhi, mflo, mthi, mtlo
jr

3. Base addressing (Base+Offset)

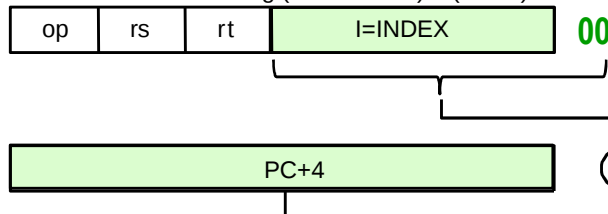


Memory

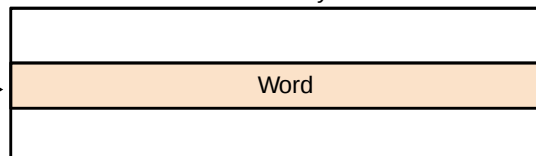


lw, lh, lb
lhu, lbu
sw, sh, sb

4. PC-relative addressing (Base+Index) $(PC+4) + 4 \cdot I$

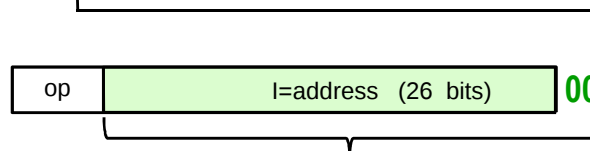
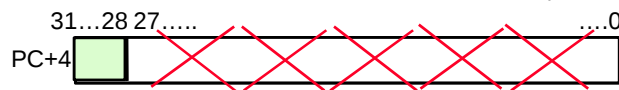


Memory

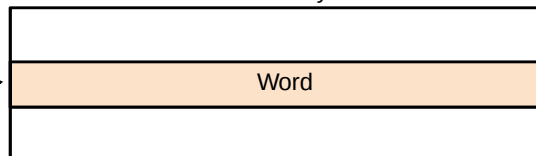


bne, beq

5. Pseudodirect addressing: $PC = (PC+4) [31..28] \mid \text{address} \times 4$



Memory



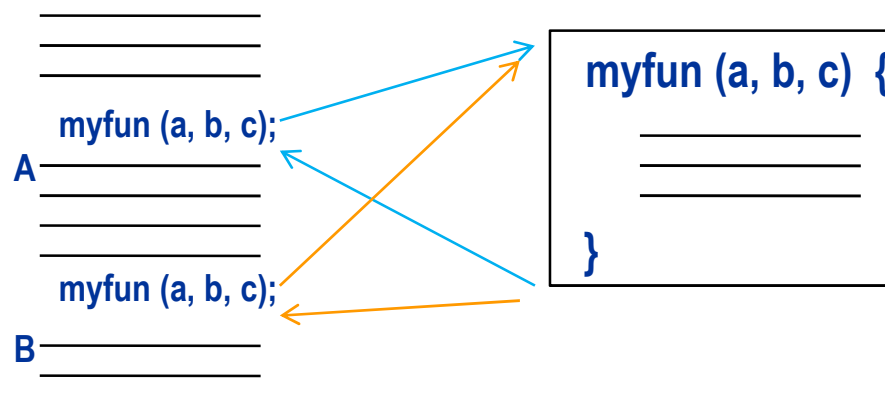
j, jal

jal=jump-and-link

Chiamata a funzione (1)

Patterson 2.7

- 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" e' 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 **NON PUO'** essere implementata con una semplice `j` o `bne/beq` → OCCORRE:

- i) memorizzare da qualche parte il punto di ritorno (es. "A");
- ii) saltare all'inizio della funzione "myfun"

- In Assembly MIPS questi due passi sono effettuati dalla istruzione `jal` → `jal myfun` salta a myfun mettendo l'indirizzo di ritorno nel registro `$ra`



Operazioni connesse alla chiamata di funzione

- 1) eventuale **salvataggio** registri da preservare
(es. \$t..., \$a..., nello stack param. oltre il quarto)
- 2) preparazione dei parametri di ingresso (nuovi \$a...)
- 3) chiamata della funzione

pre-chiamata
(lato chiamante)

jal MYFUN

- 4) allocazione del call-frame nello stack
- 5) eventuali **salvataggi** vari
(es. \$a0-\$a3, \$ra, \$fp, \$s0-\$s7)
- 6) eventuale inizializzazione nuovo \$fp

prologo (lato chiamato)

- 7) esecuzione del codice della funzione

corpo (lato chiamato)

- 8) preparazione dei parametri di uscita (\$v0-\$v1)
- 9) ripristino dei parametri salvati (salvati al punto 5)
- 10) ritorno al codice originario

epilogo (lato chiamato)

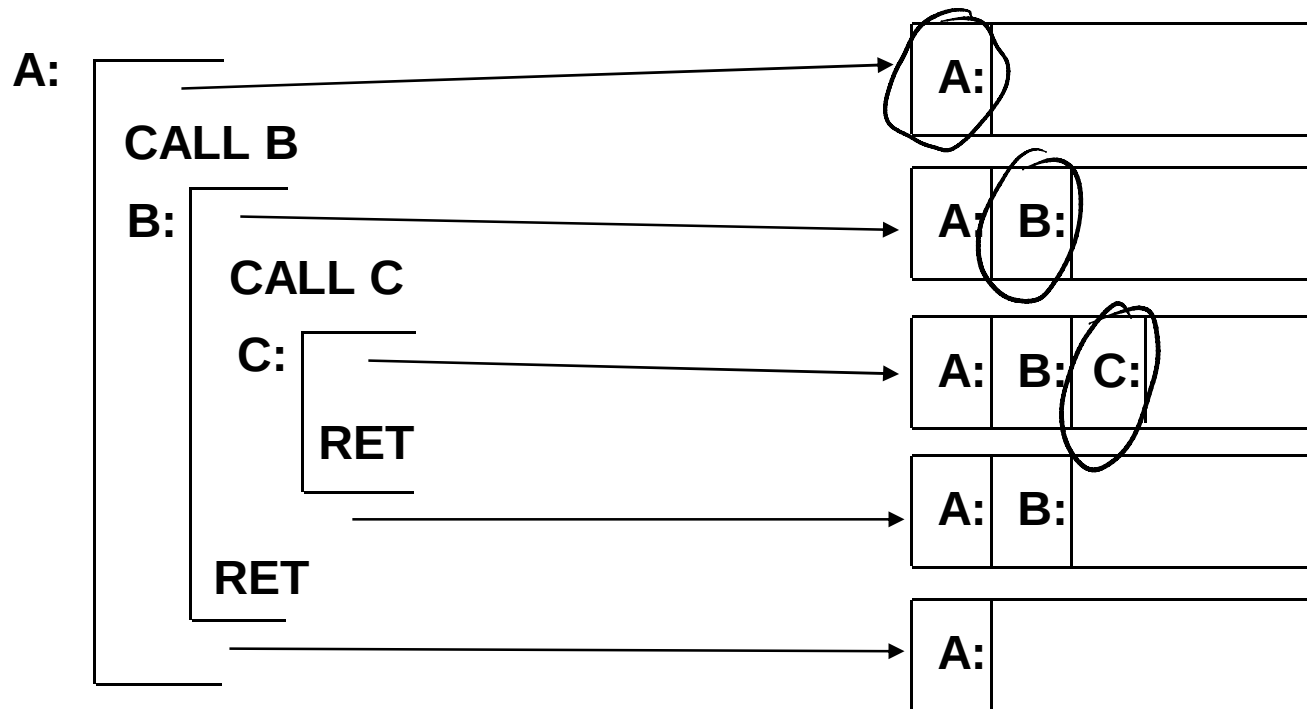
jr \$ra

- 11) eventuale ripristino dei vecchi valori (es. \$t... \$a...)
- 12) uso dei valori di uscita della funzione

post-chiamata
(lato chiamante)

- 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**

Uso dello stack nelle chiamate a funzione

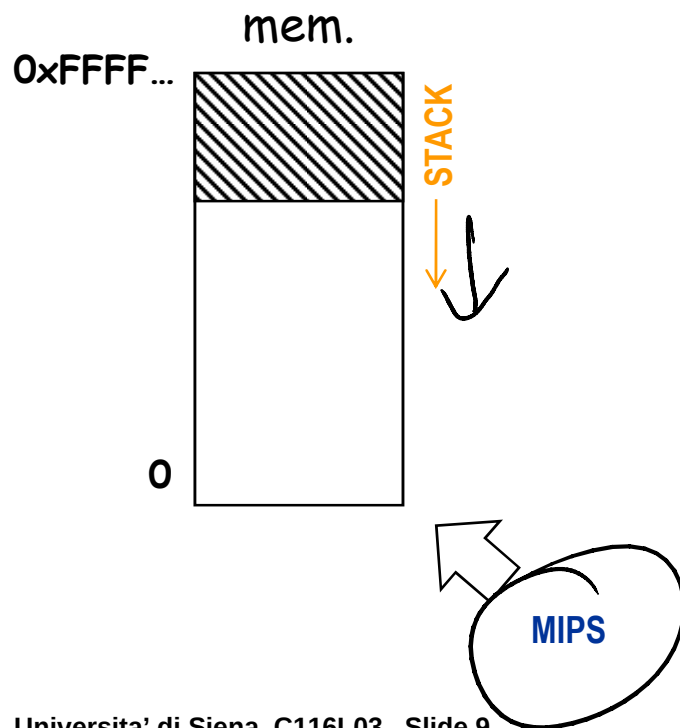


- Alcune macchine forniscono la gestione di stack come parte della architettura stessa (e.g. VAX, processori Intel)
 - Istruzioni PUSH e POP
- Gli stack possono comunque anche essere implementati via software (e.g. MIPS)

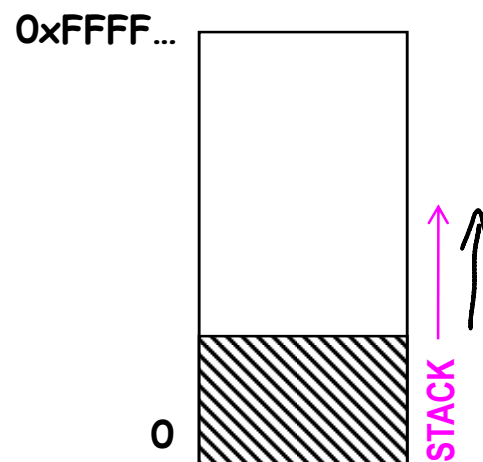
Convenzione sugli stack (1): “verso di crescita”

- Gli stack possono crescere verso l'alto oppure verso il basso
- Sto assumendo che gli indirizzi di memoria crescano verso l'alto
 - Con '0xFFFF...' indico gli indirizzi alti in memoria
 - Con '0' indico gli indirizzi bassi in memoria

Convenzione grow down

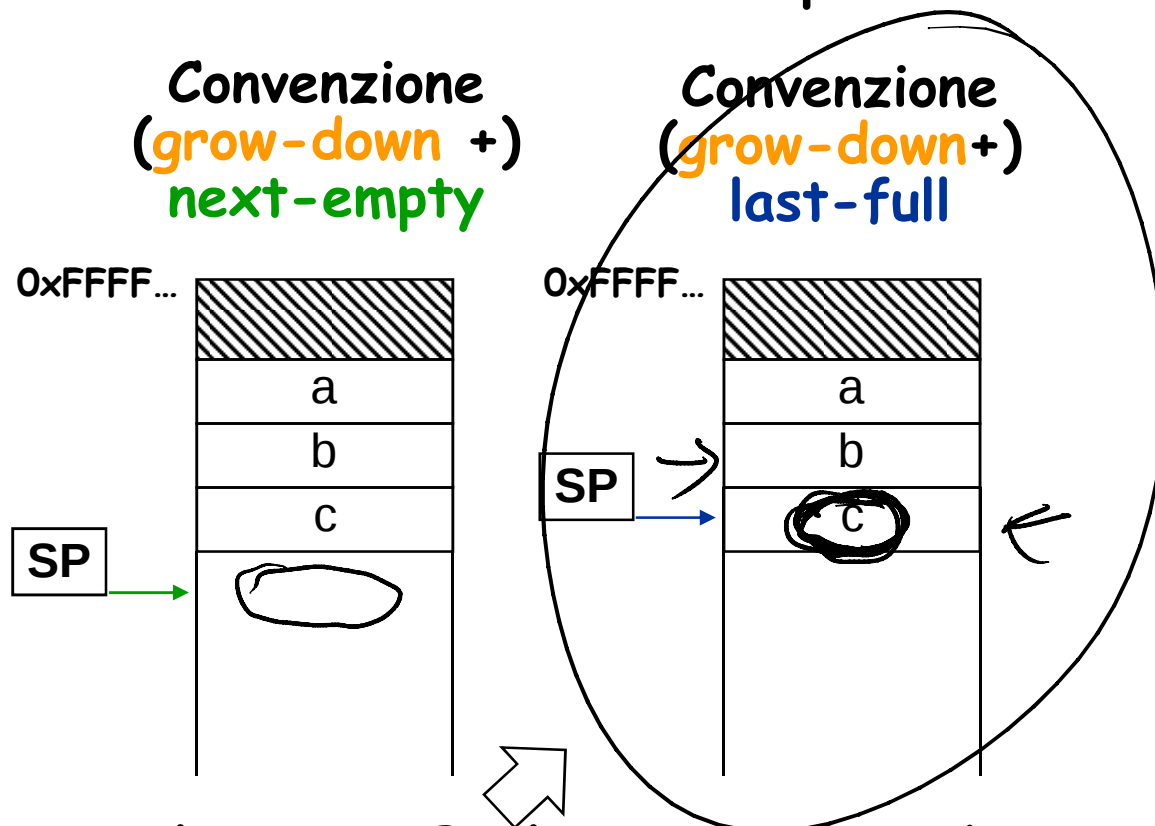


Convenzione grow up



Convenzione sugli stack (2): "a cosa punta SP"

- Convenzione su 'a cosa punta SP': **Next-Empty** vs. **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
Scrive in Mem[SP]

- nel caso MIPS la convenzione e':
 - Grow-down + Last-Full**
 - SP e' nel registro \$sp**

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)

Convenzione MIPS per le chiamate a funzioni (1)

PRE-CHIAMATA (LATO CHIAMANTE)

1) Eventuale salvataggio registri da preservare nel chiamante

- Si assume che \$a0-\$a3 e \$t0-\$t9 possano essere sovrascritti
 - se li si vuole preservare vanno salvati nello stack (dal chiamante)
 - in particolare (i vecchi valori di) \$a0-\$a3

2) Preparazione degli argomenti della funzione

- I primi 4 argomenti vengono posti in \$a0-\$a3 (nuovi valori)
- Gli eventuali altri argomenti oltre il quarto 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

Convenzione MIPS 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-a3 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-a3 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 salvare \$s0-\$s7 se intendo sovrascrivere tali registri (il chiamante si aspetta di trovarli intatti)

6) Eventuale inizializzazione di \$fp al nuovo call-frame

CORPO DELLA FUNZIONE

7) CODICE EFFETTIVO DELLA FUNZIONE

Convenzione MIPS per le chiamate a funzioni (3)

EPILOGO (LATO CHIAMATO)

8) Se deve essere restituito un valore dalla funzione

- Tale valore viene posto in \$v0 (e \$v1)

9) I registri (se salvati) devono essere ripristinati

- \$s0-\$s7
- \$ra
- \$fp

Notare che \$sp deve solo essere aumentato di opportuno offset (lo stesso sottratto nel punto 4)

RITORNO AL CHIAMANTE

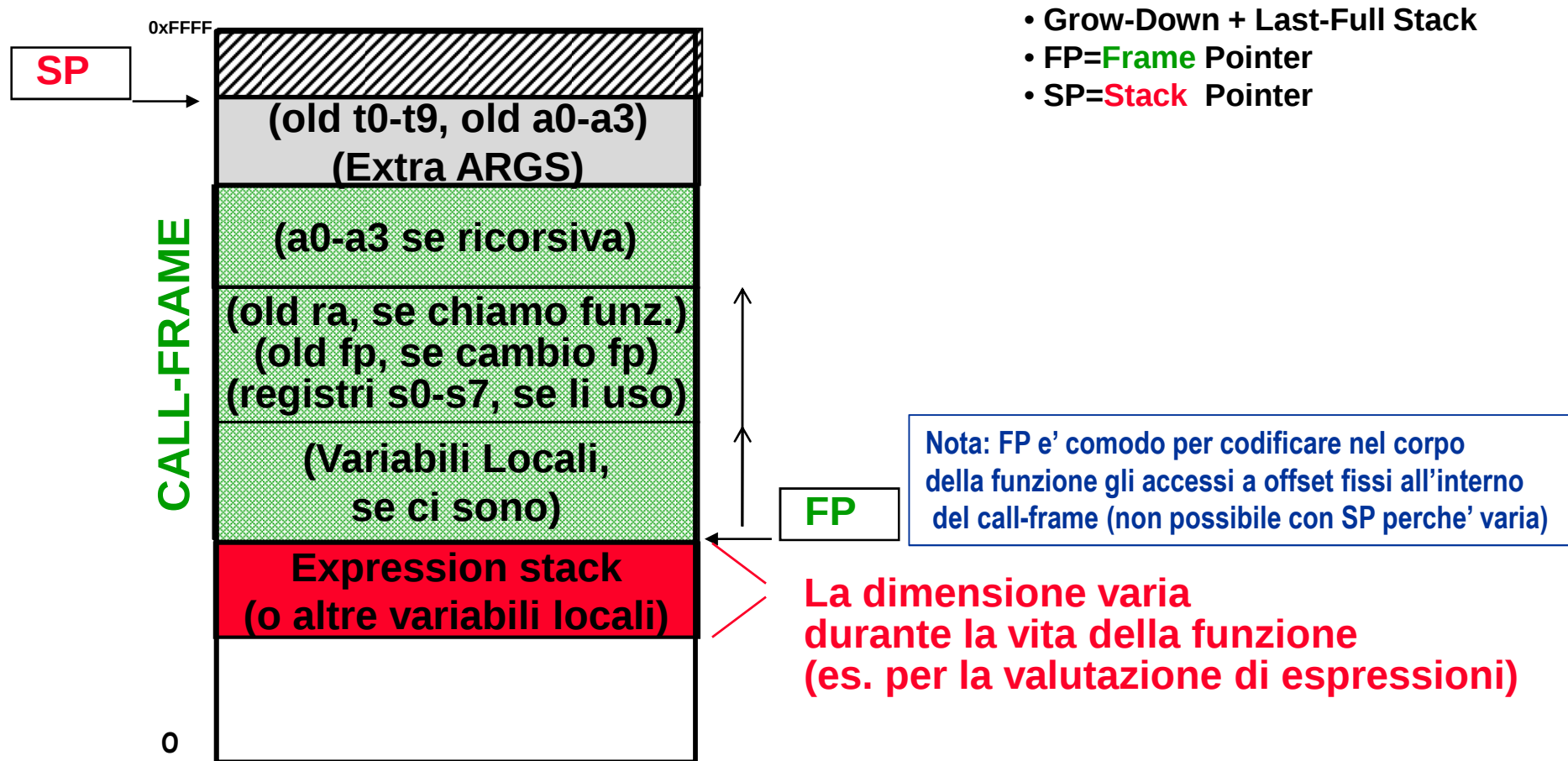
10) jr \$ra

POST-CHIAMATA (LATO CHIAMANTE)

11) Ripristino dei valori \$t... \$a... (vecchi) eventualmente salvati

12) Eventuale uso del risultato della funzione (in \$v0 (e \$v1))

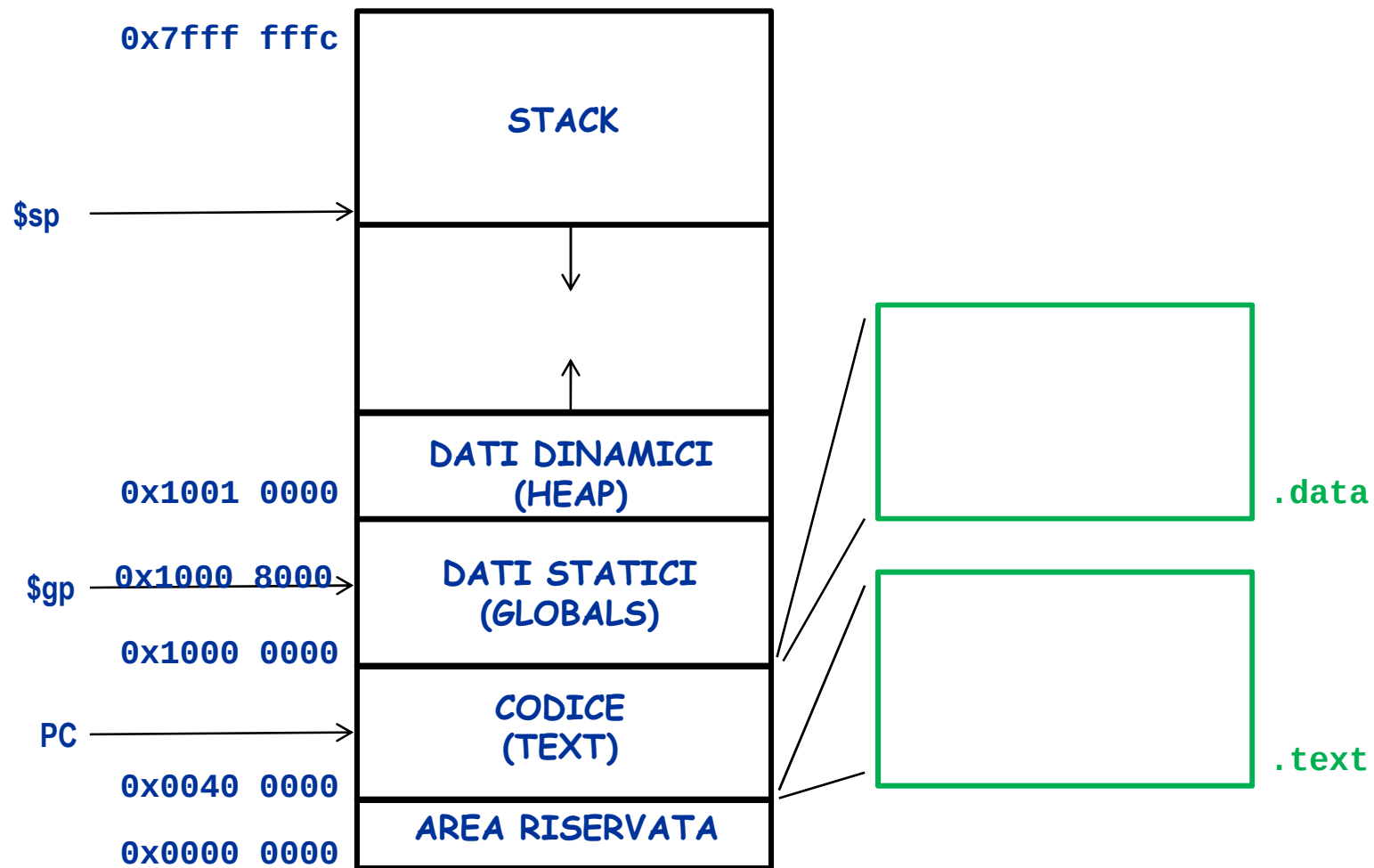
Riepilogo: struttura del Call-Frame



- I linguaggi strutturati inseriscono un'ulteriore informazione: (e.g. Pascal) un link al frame di attivazione 'padre'
- I compilatori normalmente cercano di allocare le variabili locali prima di tutto nei registri, non in memoria (o stack)!

Memory Layout

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



Programma fattoriale.s (1)

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

.text
.globl main

# Funzione per il calcolo del fattoriale
# Parametri:    n : $a0 ($4)
# Risultato:    n! : $v0 ($2)

fact:

# crea il call frame sullo stack (12 byte)
# lo stack cresce verso il basso
    addi $sp, $sp, -12      # allocazione del call frame nello stack
    sw    $a0, 8($sp)       # salvataggio di n nel call frame
    sw    $ra, 4($sp)       # salvataggio dell'indirizzo di ritorno
    sw    $fp, 0($sp)       # salvataggio del precedente frame pointer
    add   $fp, $sp, $0       # aggiornamento del frame pointer

# calcolo del fattoriale
    bne   $a0, $0, Ric      # test fine ricorsione n!=0
    addi  $v0, $0, 1         # 0! = 1
    j     Fine

Ric:
    addi  $a0, $a0, -1       # chiamata ricorsiva per il calcolo di (n-1)!
    jal   fact              # $a0 ← (n - 1) passaggio del parametro in $a0 per fact(n-1)
    lw    $t0, 8($fp)       # chiama fact(n-1) → risultato in $v0
    mult  $v0, $t0          # $t0 ← n
    mflo  $v0              # n! = (n-1)! x n

# uscita dalla funzione
Fine:
    lw    $fp, 0($sp)       # recupera il frame pointer
    lw    $ra, 4($sp)       # recupera l'indirizzo di ritorno
    addi  $sp, $sp, 12      # elimina il call frame dallo stack
    jr    $ra              # ritorna al chiamante
```

Programma fattoriale.s (2)

```
# Programma principale
#
main:
# Stampa intestazione
    la    $a0, messaggio
    addi $v0, $0, 4
    syscall

# Stampa richiesta
    la    $a0, insdati
    addi $v0, $0, 4
    syscall

# legge n (valore in $v0)
    addi $v0, $0, 5
    syscall

# chiama fact(n)
    add $a0, $v0, $0      # parametro n in a0
    jal fact
    add $s0, $v0, $0      # salva il risultato in s0

# stampa messaggio per il risultato
    la    $a0, visris
    addi $v0, $0, 4
    syscall

# stampa n!
    add $a0, $s0, $0
    addi $v0, $0, 1
    syscall

# stampa \n
    la    $a0, RTN
    addi $v0, $0, 4
    syscall

# exit
    addi $v0, $0, 10
    syscall
```

Ulteriore materiale di riferimento:
Appendice A, Patterson/Hennessy

Principali direttive per l'assemblatore

<code>.data [<addr>]</code>	marca l'inizio di una zona dati; se <addr> viene specificato i dati sono memorizzati a partire da tale indirizzo
<code>.text [<addr>]</code>	marca l'inizio del codice assembly; se <addr> viene specificato i dati sono memorizzati a partire da tale indirizzo
<code>.globl <symb></code>	dichiara un simbolo <symb> visibile dall'esterno (gli altri sono locali per default) in modo che possa essere usato da altri file
<code>.ascii <str></code>	mette in memoria la stringa <str>, seguita da un byte '0'

Ulteriori direttive assemblatore (1)

.ascii <str>

Mette in memoria la stringa <str> e non inserisce lo '0' finale

.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 quantita' a 16 bit (in locazioni di memoria contigue)

.space <n>

Mette in memoria n spazi

.align <n>

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`

Ulteriori direttive assemblatore (2)

.extern <symbol> [<size>]

Dichiara che il dato memorizzato all'indirizzo <symbol> e' un simbolo globale ed occupa <size> byte di memoria. 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 <symbol> riguardano un oggetto esterno ad un modulo (torneremo su questo fra poco)

.kdata [<addr>]

Marca l'inizio di una zona dati pertinente al kernel. Se e' presente il parametro opzionale <addr>, tali dati sono memorizzati a partire da tale indirizzo.

.ktext [<addr>]

Marca l'inizio di una zona contenente codice del kernel. Se e' presente il parametro opzionale <addr>, tali dati sono memorizzati a partire da tale indirizzo.

Alcune syscall

- 'syscall' serve per chiedere un servizio al sistema operativo (letteralmente 'system call')
 - Il numero del servizio e' indicato in \$v0
 - 1: stampa un intero a video
(\$a0 contiene l'intero in binario da stampare)
 - 4: stampa un messaggio a video
(\$a0 indirizzo della stringa da stampare)
 - 5: leggi un intero dalla tastiera
(\$v0 conterra' l'intero letto in binario)
 - 10: termina il programma
(il sistema operativo riprende il controllo del calcolatore)

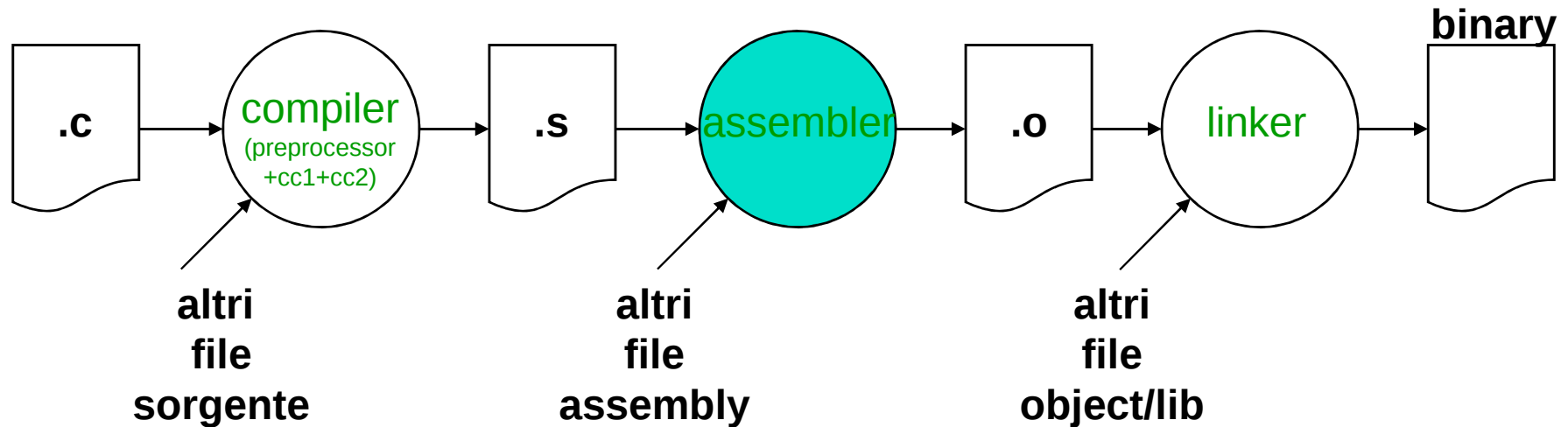
Compilazione di un programma strutturato a moduli

- 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 piu' 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
 - `cc -c main1.c`
 - `cc -c queue.c`
 - vengono cosi' 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 piu' stadi (cc1, cc2) fino ad ottenere il livello di ottimizzazione desiderato
 - il risultato intermedio e' un programma in **ASSEMBLY**
 - il codice **ASSEMBLY** e' legato all'architettura target
- **Creazione del file oggetto**
 - questo passo e' eseguito dall'assemblatore
 - il file oggetto contiene
 - la codifica delle istruzioni assembly
 - i dati statici
 - informazioni di rilocazione
 - tabella dei simboli
 - altre informazioni di utilita'

Assemblatore



- **UNIX:**
E' possibile produrre l'output in assembly del compilatore,

`cc -S main.c`

il risultato si trova nel file **main.s**

Assemblatore a due passate (1)

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

```

    add  $18, $18, $0      LC=0
    addi $19, $19, 0       LC=4
loop: add  $4, $18, $0      LC=8
    add  $19, $19, $4      LC=12
    add  $18, $18, $19     LC=16
    bne  $18, $5, loop     LC=20
    la   $18, myvar        LC=24
    .....
    .....
etic1: .....              LC=1000
    .....
    .....
    j    etic1             LC=3000
    .....
myvar: .....              LC=7000
```

Symbol Table (provvisoria)

symbol	Value of Location Counter (LC)
loop	8
etic1	1000
myvar	7000

Assemblatore a due passate (2)

- Seconda passata:

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

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

$$\text{offset} = \{LC(\text{SYMBOL}) - [LC(\text{BNE}) + 4]\} / 4$$

$$\{8 - [20 + 4]\} / 4 = -4 \Rightarrow \text{bne } \$18, \$5, -4$$

Nota: il "+4" tiene conto dalla posizione del PC a tempo di esecuzione

- Per ogni etichetta irrisolta, si controlla che non sia definita come simbolo esterno (direttiva .globl)
 - se non e' un simbolo esterno, puo' essere un errore
 - se e' un simbolo esterno o un riferimento ad un indirizzo assoluto (es. nelle istruzioni jal, j, la) **deve restare in symbol table**

symbol	LC
etic1	1000
myvar	7000

Creazione della Tabella di Rilocalizzazione

	-----	LC=0	Relocation Table		
	la \$18, myvar	LC=24			
etic1:	-----	LC=1000	LC=LOCAZIONE ISTRUZIONE	TIPO ISTRUZIONE	SIMBOLO DIPENDENTE
	-----		24	la	myvar
	-----		3000	j	etic1
	j etic1	LC=3000			

- 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)

```

0x0040'0000      add  $18, $18, $0
0x0040'0004      addi $19, $19, 0
0x0040'0008 (loop:) add  $4, $18, $0
0x0040'000C      add  $19, $19, $4
0x0040'0010      add  $18, $18, $19
0x0040'0014      bne  $18, $5, -4      # gia' stabilito dall'assembler
0x0040'0018      la   $18, 0x1000'0000 # diviso in due parti da 16 bit
0x0040'001C      .....
0x0040'001C      .....
0x0040'03E8 (etic1:).....
0x0040'03EC      .....
0x0040'03EC      .....
0x0040'0BB8      j     0x010'00FA      # (26 bit) 1000FA*4=4003E8

0x1000'0000 (myvar: ).....

```

Simboli esterni

- Un simbolo e' esterno se viene esportato (es. da una libreria) verso altri moduli

```
.globl cubic
```

```
cubic:
```

```
mult $4, $4      LC=0
```

```
mflo $5
```

```
mult $4, $5
```

```
mflo $5
```

```
jr $31
```

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

Symbol Table modulo B

symbol	LC
cubic	0

- In qualche altro modulo esistera' un riferimento alla funzione "cubic":

```
.extern cubic
```

```
----
```

```
jal cubic      LC=1500
```

```
-----
```

```
-----
```

```
-----
```

`.extern <symbol>` dichiara che i riferimenti a <symbol> riguardano un oggetto esterno al modulo

Symbol Table modulo A

symbol	LC

Relocation Table
modulo A

LC=LOCAZIONE ISTRUZIONE	TIPO ISTRUZIONE	SIMBOLO DIPENDENTE
1500	jal	cubic

Nota: il simbolo "cubic" verra' risolto in fase di linking

Formato del file oggetto (.o)

HEADER
TEXT segment
DATA segment
RELOCATION info
SYMBOL table
DEBUGGING info

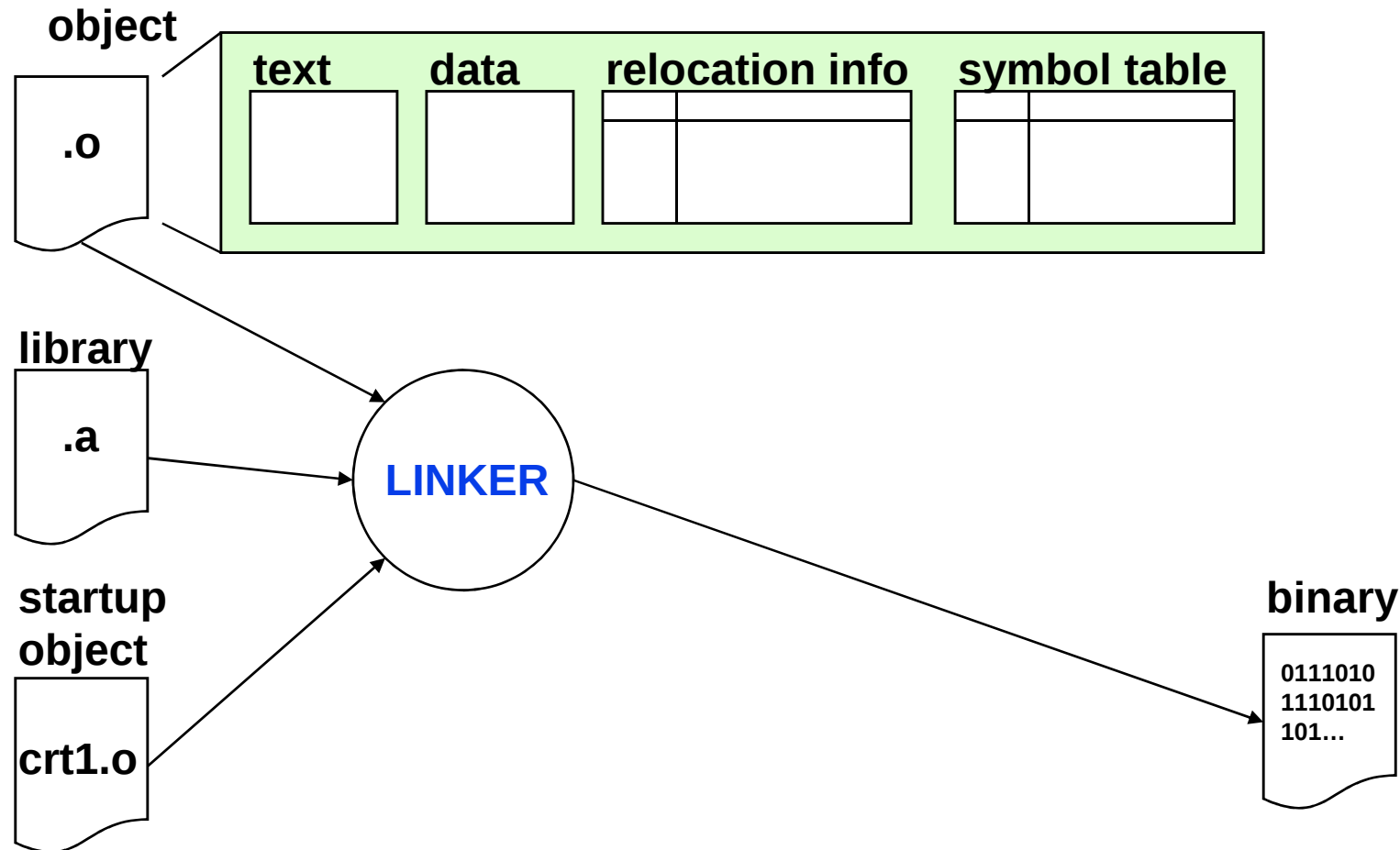
- Dimensione del file, delle parti (text, data) e CRC
- Codifica delle istruzioni (linguaggio macchina DA RILOCARE)
- Dati
- Lista delle istruzioni che fanno riferimento ad indirizzi assoluti
- Lista dei simboli locali (etichette, variabili) e etichette visibili all'esterno (assembly .globl)
- Parte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

- **UNIX:**

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

File eseguibile (binary)

- E' prodotto dal linker



I riferimenti assoluti sono rilocati e i simboli sostituiti con indirizzi

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

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

Caricamento (loader)

- Il Sistema Operativo gestisce la richiesta fatta alla shell di mandare in esecuzione il mio 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

Nota:

e' possibile generare il file binario in un colpo solo con

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

in questo caso vengono automaticamente fatte tutte le scelte di default

SLIDE AGGIUNTIVE

Regole di allocazione dei registri

Num. registro	Scopo	Nome	invariato su "call"?
0	la costante 0	\$zero	n.a.
1	riservato per l'assemblatore	\$at	n.a.
2-3	risultati di funzioni ed espressioni	\$v0-\$v1	no
4-7	argomenti della funzione	\$a0-\$a3	no
8-15	temporanei (il chiamato li puo' sovrascrivere)	\$t0-\$t7	no
16-23	temporanei salvati (il chiamato DEVE salvarli**)	\$s0-\$s7	SI
24-25	ulteriori temporanei (come sopra)	\$t8-\$t9	no
26-27	riservati per il kernel	\$k0-\$k1	n.a.
28	global pointer (il chiamato DEVE salvarlo**)	\$gp	SI
29	stack pointer (il chiamato DEVE salvarlo**)	\$sp	SI
30	frame pointer (il chiamato DEVE salvarlo**)	\$fp	SI
31	return address (il chiamato DEVE salvarlo**)	\$ra	SI

** nel caso in cui tale registro sia utilizzato

