

---

**Linguaggio macchina/assembly  
Interprete/Compilatore/Assemblatore/  
Linker  
SPIM simulator**

**(vedi Capitolo 2 e Appendice A)**

**Salvatore Orlando**

# Sommario

---

- **Assembly MIPS**
- **Compilazione**
  - Traduzione in linguaggio assembly/macchina delle principali strutture di controllo
  - Implementazione dei tipi semplici e dei puntatori
  - Funzioni (gestione dello stack - salvataggio dei registri - funzioni ricorsive)
  - Strutture dati
- **Compilazione/Linker/Loader**
- **Uso del simulatore SPIM**
- **Esercitazioni**

# Linguaggio macchina / assembler

---

- È noioso **Ehm, forse sì**
- È difficile **Mito**
- Non serve a niente **Assolutamente no**
  
- **PRO:**
  - Propedeuticità: aiuta a far capire come funziona veramente un computer
  - Alto/basso livello: aiuta a scrivere software ad alto livello più efficiente, perché si capisce come poi viene eseguito
  - Cybersecurity: per difendersi da molti attacchi informatici è necessario far riferimento al livello assembly/macchina
  - È il PIÙ POTENTE dei linguaggi di programmazione:
    - Nel senso che consente l'accesso completo a TUTTE le risorse del computer
    - Il linguaggio di alto livello “astrae” dalla specifica piattaforma, fornendo costrutti più semplici e generali
    - Cioè, è una “interfaccia generica” buona per ogni computer (con tutti i vantaggi che questo comporta)
    - Ma proprio perché è uguale per tutte le macchine, NON può fornire accesso alle funzionalità specifiche di una determinata macchina

# Linguaggi di programmazione

---

- È possibile programmare il computer usando vari linguaggi di programmazione
- Ogni linguaggio ha i suoi pro e contro

MA. . .

- L'unico linguaggio che la CPU è in grado di interpretare ed eseguire il **linguaggio macchina**
- Per usare altri linguaggi, occorre usare un **traduttore** in grado di “far capire” al computer questi nuovi linguaggi

– INTERPRETAZIONE vs. COMPILAZIONE

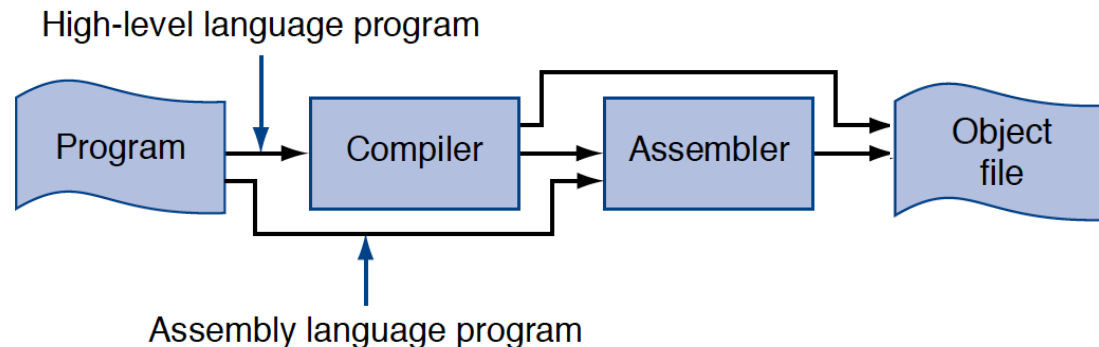
# Interprete

---

- Un **INTERPRETE** per un certo **linguaggio L** è un programma (scritto in linguaggio macchina) che “interpreta” linea per linea il linguaggio L ed esegue il codice macchina corrispondente
- **LENTEZZA DI ESECUZIONE**
  - Per eseguire un programma, ho sempre bisogno dell'interprete

# Compilatore

- Il **COMPILATORE** è un programma che prende in input un programma in un **linguaggio L** e lo traduce in un programma in un **altro linguaggio**
  - Il SOURCE LANGUAGE (“sorgente”) viene tradotto nel TARGET LANGUAGE (“obiettivo”)
- Tipicamente il TARGET LANGUAGE è il LINGUAGGIO MACCHINA, anche se e’ possibile compilare in linguaggio ASSEMBLY
- **ASSEMBLATORE**: è un semplice compilatore, per tradurre da LINGUAGGIO ASSEMBLY a LINGUAGGIO MACCHINA



- Java: Linguaggio **interpretato** a BYTE-CODE
  - il linguaggio viene compilato in un LINGUAGGIO INTERMEDIO (bytecode) che poi viene INTERPRETATO

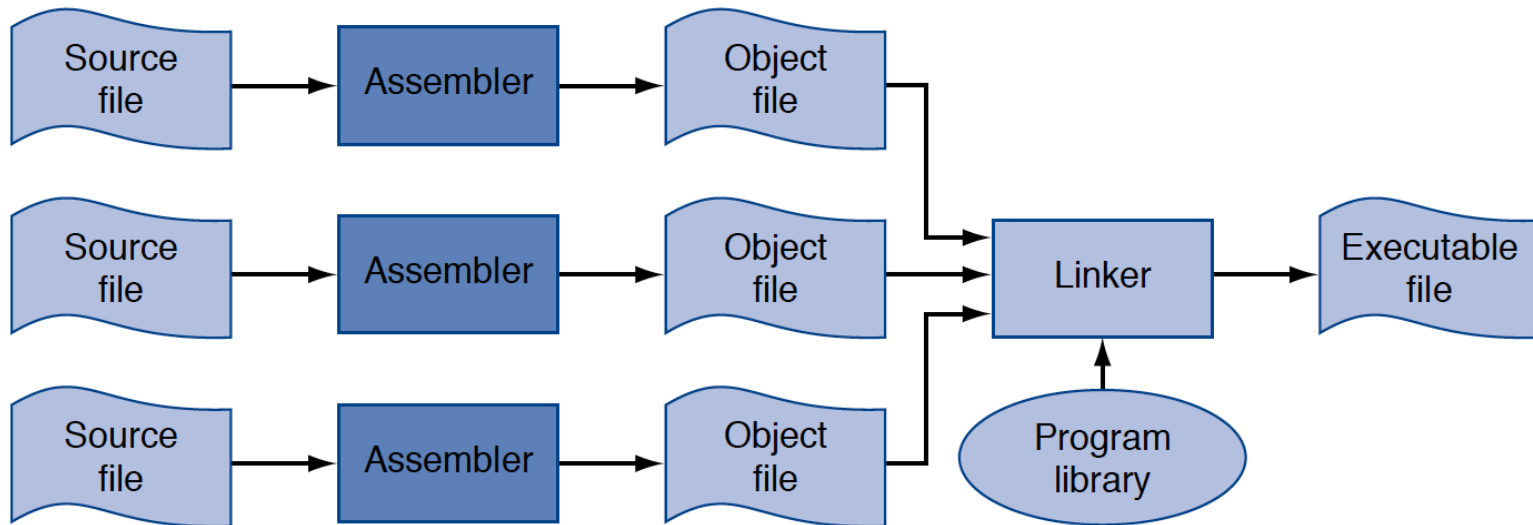
# Cos'è un «object file»

---

- Un **file oggetto** è il risultato della compilazione, ed è composto da:
  - Istruzioni in formato binario (linguaggio macchina)
  - Dati in formato binario
  - Informazioni necessarie per modificare il codice (istruzioni e dati) e allocarlo in memoria ad un certo indirizzo (*rilocalizzazione del codice*)
- NOTA:
  - Per produrre l'istruzione macchina di ogni istruzione assembly, l'assemblatore deve conoscere l'*indirizzo* in memoria corrispondente ad ogni *etichetta (label)* utilizzata nei *jump* e nei *conditional branch*
  - Le etichette utilizzate sono mantenute in una **symbol table**.
    - **Coppie etichette - indirizzi**

# Il processo di compilazione/esecuzione

- Il passaggio dal codice sorgente alla esecuzione del programma passa per 3 fasi, che include anche il **LINKER** e **LOADER**
  1. Compilazione / Assembling per produrre i **file oggetto**
  2. Linking per produrre il **file eseguibile**
  3. Caricamento ed Esecuzione



- Torneremo su questo processo in seguito**



# Cosa sono “programmi e dati” caricati per l'esecuzione?

---

- La CPU vede tutto come numeri, che rappresentano
  - Dati (interi, caratteri, floating point)
  - Istruzioni

- Un programma (istruzioni + dati) è quindi una sequenza di numeri binari

```
001001111011110111111111111100000
10101111101111110000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
10001111101110000000000000011000
00000001110011100000000000011001
00100101110010000000000000000001
00101001000000010000000001100101
10101111101010000000000000011100
00000000000000000111100000010010
00000011000011111100100000100001
0001010000100000111111111110111
10101111101110010000000000011000
00111100000001000001000000000000
10001111101001010000000000011000
00001100000100000000000011101100
00100100100001000000010000110000
10001111101111110000000000010100
00100111101111010000000000100000
0000001111100000000000000001000
00000000000000000001000000100001
```

- La CPU *interpreta* certi numeri come istruzioni (il cosiddetto LINGUAGGIO MACCHINA)

# Assembly: Linguaggio Macchina per umani

---

- Leggere “numeri” non è molto facile
- Per noi umani, conviene tradurre questi codici numerici in qualcosa di più leggibile, ovvero stringhe alfanumeriche, mnemoniche e più leggibili
  - LINGUAGGIO ASSEMBLY
- Il LINGUAGGIO ASSEMBLY è un linguaggio di programmazione, che rispecchia fedelmente le istruzioni del linguaggio macchina
  - USO: più leggibile, più flessibile
  - Ha bisogno di un semplice compilatore (ASSEMBLER) per tradurre il codice in LINGUAGGIO MACCHINA
  - Linguaggi ASSEMBLY differenti per CPU caratterizzate da diversi *Instruction Set*
- Spesso, vista la stretta corrispondenza, si fa confusione, o si considerano i linguaggi MACCHINA e ASSEMBLY come termini equivalenti

# Assembly: Linguaggio Macchina per umani

- Usando gli opcode e i formati delle istruzioni, possiamo tradurre le istruzioni macchina in un corrispondente programma simbolico

- **disassemblatore**

- Più semplice da leggere

- Istruzioni e operandi scritti con simboli

- Ma ancora difficile

- Le locazioni di memoria sono espresse con indirizzi numerici, invece che con simboli/etichette

addiu	\$29, \$29, -32
sw	\$31, 20(\$29)
sw	\$4, 32(\$29)
sw	\$5, 36(\$29)
sw	\$0, 24(\$29)
sw	\$0, 28(\$29)
lw	\$14, 28(\$29)
lw	\$24, 24(\$29)
multu	\$14, \$14
addiu	\$8, \$14, 1
slti	\$1, \$8, 101
sw	\$8, 28(\$29)
mflo	\$15
addu	\$25, \$24, \$15
bne	\$1, \$0, -9
sw	\$25, 24(\$29)
lui	\$4, 4096
lw	\$5, 24(\$29)
jal	1048812
addiu	\$4, \$4, 1072
lw	\$31, 20(\$29)
addiu	\$29, \$29, 32
jr	\$31
move	\$2, \$0

# Assembly: Linguaggio Macchina per umani

- La routine scritta finalmente in **linguaggio assembly** con le **etichette mnemoniche (label)** per le varie locazioni di memoria
- I comandi che iniziano con i punti sono direttive assembler

- **.text**    **.data**
  - Indica che le linee successive contengono istruzioni o dati
- **.align n**
  - Indica l'allineamento su  $2^n$
- **.globl main**
  - L'etichetta **main** è globale, ovvero visibile da codici in altri file
- **.ascii**
  - Area di memoria che memorizza una stringa terminata da zero.


```
.text
.align    2
.globl    main

main:
    subu   $sp, $sp, 32
    sw     $ra, 20($sp)
    sd     $a0, 32($sp)
    sw     $0, 24($sp)
    sw     $0, 28($sp)

loop:
    lw     $t6, 28($sp)
    mul    $t7, $t6, $t6
    lw     $t8, 24($sp)
    addu   $t9, $t8, $t7
    sw     $t9, 24($sp)
    addu   $t0, $t6, 1
    sw     $t0, 28($sp)
    ble    $t0, 100, loop
    la     $a0, str
    lw     $a1, 24($sp)
    jal    printf
    move   $v0, $0
    lw     $ra, 20($sp)
    addu   $sp, $sp, 32
    jr     $ra

.data
.align    0

str:
.asciiiz  "The sum from 0 .. 100 is %d\n"
```



# Assembly: Linguaggio Macchina per umani

---

- **Ma qual è il corrispondente programma scritto in linguaggio C?**

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i;
    int sum = 0;

    for (i=0; i <= 100; i++)
        sum = sum + i*i
    printf("The sum from 0 .. 100 is %d\n", sum);
}
```

# Linguaggio assembly

---

- Le tipiche operazioni di un linguaggio macchina/assembly, incluso quello del MIPS, riguardano:
  - La manipolazione di dati (istruzioni aritmetiche)
  - Lo spostamento di dati (load / store)
  - Il controllo del flusso di istruzioni (salti)
- Le istruzioni che movimentano i dati trasferiscono dati dalla memoria ai REGISTRI (memorie interne veloci) e viceversa
- Nel MIPS ci sono 32 REGISTRI generali
  - Nell'ASSEMBLY MIPS li possiamo nominare come: `$0, $1, ..., $31`
  - Li possiamo anche nominare come:
    - `$zero, $v0, $v1, $k0, $k1`
    - `$sp, $ra, $at, $gp, $fp`
    - `$s0, $s1, $s2, . . . , $s7`
    - `$t0, $t1, $t2, . . . , $t9`
    - `$a0, $a1, $a2, $a3`

# Istruzioni MIPS per manipolare i dati

## ARITMETICHE

---

- **add** *reg\_dest, reg\_source1, reg\_source2*  
*reg\_dest = reg\_source1 + reg\_source2*
- **sub** *reg\_dest, reg\_source1, reg\_source2*  
*reg\_dest = reg\_source1 - reg\_source2*
- **mult** *reg1, reg2*  
*? = reg1 \* reg2*
  - Non c'è il registro destinatario perché stiamo moltiplicando due registri (reg1 e reg2) da 32 bits, ma il risultato può essere più grande di 32 bits
  - Il risultato nei *registri speciali* \$Hi e \$Lo (i primi 32 bits in \$Hi, gli altri 32 in \$Lo)
- **div** *reg1, reg2*  
*? = reg1 / reg2*
  - Anche qui non c'è il registro destinazione !!
  - Il risultato va nel registro \$Lo
  - Viene anche calcolato il RESTO della divisione, che va nel registro \$Hi

# Istruzioni MIPS per manipolare i dati

## ARITMETICHE

---

- Variante IMMEDIATA delle istruzioni
  - Per tutte queste operazioni, è possibile specificare direttamente un numero costante (di 16 bits) al posto del secondo registro source  
*reg\_dest = reg\_source1 OP costante*
- La corrispondente istruzione *immediata* si ottiene in assembly aggiungendo una “i” al nome:
  - **addi**, **mult**i, **div**i
- Non esiste **subi**, perché?



# Istruzioni MIPS per manipolare i dati LOGICHE

---

- **Bitwise manipulation**

Operation	C	Java	MIPS
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR			or, ori
Bitwise NOT	~	~	nor

- **Istruzioni utili, tra l'altro, per inserire/estrarre gruppi di bit in/da una word**

# Istruzioni MIPS per manipolare i dati LOGICHE

---

- Interpretazione funzionale delle operazioni logiche
  - Tramite AND, OR e XOR possiamo modificare un bit.
  - La modifica dipende dal valore del secondo operando: 0 o 1
- AND
  - $\text{Dest\_bit} = \text{Source\_bit} \text{ AND } 0 \rightarrow \text{Spegni il Source\_bit}$
  - $\text{Dest\_bit} = \text{Source\_bit} \text{ AND } 1 \rightarrow \text{Copia il Source\_bit}$
  - In ASSEMBLY l'istruzione **and** agisce su words, il che significa che possiamo agire **contemporaneamente sui 32 bit** di una word, con le due operazioni sopra dette (SPEGNI/CANCELLA o COPIA)
  - `andi $t0, $t1, 0000000011111111` (**zero-extended immediate**)  
prendi \$t1, copia gli 8 bits più a destra, annulla gli altri bits più a sinistra, e metti il risultato in \$t0
  - Serve anche a verificare quali bit sono accesi
    - `andi $t0, $t1, 000000000100000`
    - \$t0 è diverso da 0 se e solo se il sesto bit di \$t1 è acceso.

# Istruzioni MIPS per manipolare i dati LOGICHE

---

- OR

- Dest\_bit = Source\_bit **OR 0** → **Copia** il Source\_bit
- Dest\_bit = Source\_bit **OR 1** → **Accendi** il Source\_bit
- In ASSEMBLY l'istruzione **or** agisce su words, il che significa che possiamo agire contemporaneamente sui 32 bits di una word, con le due operazioni sopra dette (ACCENDI o COPIA)
- `ori $t0, $t1, 0000000011111111` (**zero-extended immediate**)  
prendi \$t1, poni a 1 (accendi) gli 8 bits più a destra, copia i bit più a sinistra, e metti il risultato in \$t0

# Istruzioni MIPS per manipolare i dati LOGICHE

0	0	0
1	0	1
0	1	1
1	1	0

## XOR

- Dest\_bit = Source\_bit **XOR 0** → **Copia** il Source\_bit
- Dest\_bit = Source\_bit **XOR 1** → **Invertire** il Source\_bit
- In ASSEMBLY l'istruzione **xor** agisce su words, il che significa che possiamo agire contemporaneamente sui 32 bits di una word, con le due operazioni sopra dette (INVERTI o COPIA)
- **xori \$t0,\$t1, 0000000011111111** (**zero-extended immediate**)  
prendi \$t1, inverti gli 8 bits più a destra, copia i bit più a sinistra, e metti il risultato in \$t0
- INVERTIRE è un'operazione reversibile
  - **A = INVERT ( INVERT (A) )**

# Istruzioni MIPS per spostare i dati

---

- Occorrono istruzioni anche per spostare i dati. . .
  - Da registri a memoria (STORE)
  - Da memoria a registri (LOAD)
  - Da registri/costanti a registri (MOVE)
- **sw reg2, indirizzo(reg1)**
  - Copia il valore di `reg2` alla locazione di memoria `indirizzo+reg1`
- **sb reg2, indirizzo(reg1)**
  - Copia il byte più basso di `reg2` alla locazione di memoria `indirizzo+reg1`
- **lw reg2, indirizzo(reg1)**
  - Copia in `reg2` il valore presente nella locazione di memoria `indirizzo+reg1`
- **lbu reg2, indirizzo(reg1)**
  - Copia nel byte più basso di `reg2` il byte presente nella locazione di memoria `indirizzo+reg1`
- **lui reg, numero**
  - Copia nei 16 bit PIÙ ALTI (*UPPER*) di `reg` il valore `numero`, i 16 più bassi posti a 0
- **ori \$reg, \$zero, numero**
  - Copia nei 16 bit PIÙ BASSI (*LOWER*) di `reg` il valore `numero`

# Istruzioni MIPS per spostare i dati

---

- **Importante**
  - Tutte le istruzioni di LOAD e STORE, in generale, funzionano rispettando il cosiddetto principio fondamentale di **ALLINEAMENTO** della memoria
  - Quando si sposta un dato di **taglia n**, si può operare solo su **indirizzi di memoria multipli di n**
  - Nel caso di istruzioni che manipolano un byte ( $s_b, l_b$ ), il principio di allineamento dice che ogni indirizzo deve essere multiplo di 1 byte, e quindi qualsiasi indirizzo va bene: nessuna restrizione!

# Istruzioni MIPS per spostare i dati

---

- Infine, ci sono istruzioni che permettono di spostare dati internamente, da certi registri “nascosti” ad altri registri
- Servono per recuperare i valori dei registri interni  $\$Hi$  e  $\$Lo$  (quelli usati per moltiplicare e dividere) che non sono direttamente riferibili
- **mfhi reg\_dest**
  - *Move From Hi*: muove il contenuto di  $\$Hi$  nel registro `reg_dest`
- **mflo reg\_dest**
  - *Move From Lo*: muove il contenuto di  $\$Lo$  nel registro `reg_dest`

# Istruzioni MIPS per modificare il flusso di controllo

---

- Normalmente, l'esecuzione è sequenziale
- Il program counter (contatore di programma), o PC, dice al computer dove si trova la prossima istruzione da eseguire
- Il flusso standard è: esegui l'istruzione all'indirizzo indicato da PC, e avanza all'istruzione successiva ( $PC = PC + 4$ )
- Le istruzioni per la modifica del flusso servono a forzare la modifica del PC rispetto al flusso standard sequenziale
  - possiamo ridirigere l'esecuzione del programma in ogni punto, facendo salti in avanti e indietro
- Si può modificare il flusso con istruzioni di
  - salto condizionato (branch)
  - salto assoluto (jump)



# Istruzioni MIPS per modificare il flusso di controllo

---

- I salti condizionati hanno tutti la forma  
`branch_condizione reg1, reg2, indirizzo_salto`
- Il significato è:
  - Se confrontando `reg1` e `reg2` viene soddisfatta la condizione, allora salta
- Branch Equal  
`beq reg1, reg2, indirizzo`  
(salta se `reg1==reg2`)
- Branch Not Equal  
`bne reg1, reg2, indirizzo`  
(salta se `reg1!=reg2`)
- Anche se trattasi di un'istruzione per manipolare dati, la seguente istruzione di **set less than** è usata per realizzare salti condizionati  
`slt reg1, reg2, reg3`  
(if (`reg2<reg3`) then `reg1=1` else `reg1=0`)

# Istruzioni MIPS per modificare il flusso di controllo

---

- Le istruzioni di salto assoluto specificano solo l'indirizzo a cui saltare
- Tre possibili salti assoluti:
  1. Jump: `j indirizzo`
  2. Jump and link `jal indirizzo`
    - serve per saltare all'indirizzo iniziale di una funzione / procedura
    - salta, ma si ricorda come tornare indietro, salvando PC+4 (l'istruzione successiva) nel registro speciale `$ra`
  3. Jump register `jr reg`
    - salta all'indirizzo specificato nel registro: `PC = reg`

# PSEUDO-ISTRUZIONI

---

- Se devo copiare un indirizzo in un registro, dove l'indirizzo in assembly è un'ETICHETTA?
  - **la reg, indirizzo**
    - Copia la “costante speciale” indirizzo a 32 bit in reg
    - E' una **PSEUDO-ISTRUZIONE**
    - Non esiste una corrispondente istruzione macchina
    - Sarà compito dell'assemblatore tradurla in una sequenza di istruzioni macchina!!!
      - es.: **lui** (16 bit più significativi) + **ori** (16 bit meno significativi)
- **PSEUDO-ISTRUZIONI**
  - Istruzioni fornite dal linguaggio assembly, ma che non hanno corrispondenza nel linguaggio macchina
  - **SCOPO**: semplificare la programmazione assembly senza complicare l'hardware
  - Ci sono tanti esempi, oltre 1a
    - Ad esempio la pseudo istruzione blt (branch if less than), tradotta usando le istruzioni: slt e bne.

# PSEUDO ISTRUZIONI

- Alcune istruzioni assembly non possono essere tradotte direttamente in una istruzione macchina, a causa della limitazione del *campo immediate* delle istruzioni macchina
  - diventano quindi pseudo istruzioni
- Esempio: branch in cui indirizzo target di salto è molto lontano dal PC corrente (indirizzamento PC-relative)

```
beq    $s0, $s1, L1
```

diventa

```
bne    $s0, $s1, L2
```

```
j      L1
```

```
L2:
```

- Esempio: load il cui *displacement costante* da sommare al registro è troppo grande

```
lw     $s0, 0x00FFFFFF($s1)
```

diventa

```
lui    $at, 0x00FF          # registro $at ($1)
```

```
ori    $at, $at, 0xFFFF     # riservato per l'assemblatore
```

```
add    $at, $s1, $at
```

```
lw     $s0, 0($at)
```

# Istruzioni MIPS: riassunto

---

- **Manipolazione di dati:**

`add, sub, mult, div, slt`

`sll, srl`

`and, or, nor, xor`

`addi, subi, slti, andi, ori, xori`

- **Movimento di dati:**

`sw, sb, lw, lbu, lui, mfhi, mflo`

- **Flusso di istruzioni:**

`beq, bne, j, jal, jr`

- **Questo set di istruzioni è molto ridotto. Inoltre l'assembly language mette a disposizione moltissime pseudo-istruzioni**

- **Vedi Appendice A**

# Compilazione

---

- **Come si passa da un linguaggio di alto livello a uno di basso livello?**
- **Ovvero, come funziona un compilatore, che effettivamente traduce un linguaggio a più alto livello in uno a più basso livello?**
- **Come esempio di passaggio da alto a basso livello**
  - **Linguaggio C come alto livello**
  - **Linguaggio assembly MIPS come basso livello**
- **Discuteremo una possibile traduzione/compilazione dei vari costrutti del linguaggio C**
  - ⇒ **utile concettualmente**
  - ⇒ **evidenzia man mano le differenze tra alto e basso livello, cosa avviene veramente nella macchina**

# Compilazione

---

- **Mentre il C ha costrutti di programmazione e variabili con tipo. . .**
  - **Il linguaggio assembly ha istruzioni, memoria interna e memoria esterna (indirizzabile al byte)**
    - **Interna alla CPU: registri**
    - **Esterna alla CPU: RAM**
- ⇒ dovremo tradurre i costrutti attraverso particolari strutturazione delle semplici istruzioni MIPS**
- ⇒ dovremo trovare una adeguata rappresentazione delle variabili in memoria**

# Variabili e Memoria

---

- La memoria si distingue in interna (registri) ed esterna (semplicemente “memoria”)
- I registri sono in numero fisso, mentre la memoria è potenzialmente illimitata
- Siccome anche il numero di variabili in un programma, in generale, non è fisso
  - ⇒ la scelta naturale è di mettere i valori delle variabili in memoria
- Quindi, per ogni variabile del programma, ad esempio  
“a”, “b”, “c”
- Avremo una corrispondente posto/indirizzo in memoria:
  - “a” ⇒ 200
  - “b” ⇒ 360
  - “c” ⇒ 380



# Variabili e Memoria

---

- **Tipi di dato**
  - **I linguaggi ad alto livello come il C permettono di definire variabili con associato un tipo:**
    - caratteri, interi (normali corti lunghi), stringhe, reali (floating point a singola o doppia precisione), ecc.
  - **Ognuno di questi tipi ha una certa taglia: `sizeof(tipo)`**
    - caratteri: 8 bit;      interi corti: 16 bit;      interi: 32 bit;  
  reali: 32 o 64 bit;      stringhe:  $8 \times (\text{lunghezza della stringa} + 1)$  bit
  - **Mentre la “taglia” di un dato è trasparente per il programmatore in un linguaggio ad alto livello . . . . in linguaggio macchina questo non è vero**
    - **bisogna conoscere la taglia di ogni tipo di dato (numero di byte), per allocare il dato in memoria**

# Semplice compilazione di un'espressione C

---

- In C: `a = b+c;`
- Mappatura delle variabili intere (4 B) in memoria:
  - `a`  $\Rightarrow$  100, . . . , 103
  - `b`  $\Rightarrow$  104, . . . , 107
  - `c`  $\Rightarrow$  108, . . . , 111
- In assembly MIPS ci manca l'istruzione:  
`add mem100, mem104, mem108`
- La “add” del MIPS funziona con registri, non con memoria esterna  
 $\Rightarrow$  occorre prima passare i valori delle variabili da memoria a registri (LOAD), fare le operazioni che vogliamo, e poi rimetterle in memoria (STORE)

# Semplice compilazione di un'espressione C

- In C: `a = b+c;`

- Mappatura delle variabili intere (4 B) in memoria:

`a`  $\Rightarrow$  100, . . . , 103

`b`  $\Rightarrow$  104, . . . , 107

`c`  $\Rightarrow$  108, . . . , 111

- Traduzione:

```
lw $t0, 104           # carica ``b`` in t0
lw $t1, 108           # carica ``c`` in t1
add $t2,$t0,$t1        # mette la somma in t2
sw $t2, 100           # mette t2 in ``a``
```

**Pseudo-istruzione.** In linguaggio macchina l'istruzione sarà tradotta usando anche il registro \$0 (\$zero), che è in sola lettura e ha sempre il valore zero:

```
sw $t2, 100($0)
```

# Semplice compilazione di un'espressione C

---

- In C: `a = 24;`
- Usiamo ancora il registro speciale, `$zero` (`$0`), che ha sempre il valore zero
- In assembly:  

```
addi $t0, $zero, 24      # metti 0+24 in t0
sw $t0, 100              # metti t0 in ``a``
```
- Naturalmente, ci sono molti altri modi per inserire la costante 24 in `$t0`
  - Es: `ori $t0, $zero, 24`
  - C'è anche una pseudo-istruzione (*load immediate*):  
`li $t0, 24`

# Array in C

- In C: `int A[20];`
- E' un "modo compatto" per dichiarare e allocare 20 variabili `int` in un colpo solo: `A[0], A[1], . . . , A[19]`
- Il vantaggio è che possiamo usare un'espressione come indice (variabile parametrica): `A[x]`
- `int A[20]` può essere mappato in memoria come 20 variabili `int` consecutive:  
$$A[0] \Rightarrow 200, . . . , 203 \quad (\&A[i] \Rightarrow \text{primo indirizzo al byte in cui è mappato l'intero } A[i])$$
$$A[1] \Rightarrow 204, . . . , 207$$
$$. . .$$
- L'indirizzo in memoria di `A[x]` è allora
  - $200 + 4 * x$  equivalente a:  $200 + \text{sizeof(int)} * x$
  - Dove 200 è l'indirizzo di `A[0]`, la base, ovvero `&A[0]`

# Array in C

- In C: `b = A[c];`
- Mappatura variabili:
  - `b`  $\Rightarrow$  104, . . ., 107
  - `c`  $\Rightarrow$  108, . . ., 111
  - `A[0]`  $\Rightarrow$  200, . . ., 203
  - `A[1]`  $\Rightarrow$  204, . . ., 207
  - . . .
- Per tradurre dobbiamo spostare dati da e alla memoria, e dobbiamo calcolare l'indirizzo in memoria di `A[c]`
  - `Reg0`  $\leftarrow$  `Mem[108]` # leggo variabile `c`
  - `Reg1`  $\leftarrow$  `200 + 4 * Reg0` # calcolo indirizzo `&A[c]`
  - `Reg2`  $\leftarrow$  `Mem[Reg1]` # leggo `A[c]` lw
  - `Mem[104]`  $\leftarrow$  `Reg2` # copio valore `A[c]` in `b`

# Array in C

- In C: `b = A[c];`
- Mappatura variabili:
  - `b`  $\Rightarrow$  104, . . ., 107
  - `c`  $\Rightarrow$  108, . . ., 111
  - `A[0]`  $\Rightarrow$  200, . . ., 203
  - `A[1]`  $\Rightarrow$  204, . . ., 207
  - . . .
- Seguendo lo schema precedente, una possibile traduzione assembly è la seguente:

```
lw  $t0,108($zero)    # carica ``c`` in t0
ori  $t1,$zero,4       # metti 4 in t1
mult $t0,$t1           # calcola 4*c
mflo $t2               # mettilo in t2
ori  $t3,$zero,200     # indir. di A[0] in t3
add  $t3,$t3,$t2       # indir. di A[c] in t3
lw   $t4,0($t3)        # A[c] in t4
sw   $t4,104($zero)    # t4 = A[c] in ``b``
```

# Modularità della traduzione vs. ottimizzazione

---

- Traduciamo ogni accesso ad una variabile (lato destro di un assegnamento) con una LOAD e ogni scrittura con una STORE (lato sinistro di un assegnamento)
- Questo modo di compilare è modulare, nel senso che possiamo eseguire la traduzione di ogni istruzione in modo indipendente
- **Vantaggi:** semplicità
- **Svantaggi:** efficienza
  
- Nella pratica, il compilatore esegue le cosiddette ‘ottimizzazioni del codice’
- Tra le ottimizzazioni più importanti:
  - Evitiamo di copiare i dati (le variabili) dai registri in memoria, a meno che non sia strettamente necessario
  - Uso dei registri per memorizzare variabili
    - I registri molto più veloci della memoria esterna
  - Ad esempio, una variabile (solo tipi semplici) che viene usata spesso si può tenere in un registro
    - In C, c’è addirittura la dichiarazione “register” proprio per forzare questo.
  - I registri sono pochi però...



# Esempio di ottimizzazione nell'allocazione delle variabili

---

- In C:

$a = b + d + a * 3 + (3/a - a/2 + a * a * 3.14)$

$b = a/2$

$c = a + a * a + 1/(a - 2) \dots$

- La variabile  $a$  è molto usata nelle espressioni precedenti. Se possibile, conviene mappare “ $a$ ” in un registro:

$a \Rightarrow \$s0$

$b \Rightarrow 200, \dots, 203$

$d \Rightarrow 204, \dots, 207$

- I compilatori C sono ora bravissimi nell'eseguire traduzioni efficienti e ottimizzate
- Il problema generale che i compilatore sono in grado di affrontare
  - massimizzare il numero di variabili mantenute nei registri (veloci)
  - minimizzare gli accessi alla memoria/cache (relativamente più lenta)

# Etichette

- Nelle istruzioni in linguaggio macchina si usano gli indirizzi:
  - Per un programmatore assembly non è conveniente/possibile calcolare/conoscere esplicitamente l'indirizzo numerico di dati/istruzioni
  - si usano etichette mnemoniche (**labels**), che si associano a specifici punti del programma (istruzioni o dati)
  - l'assemblatore poi traduce le etichette nei corrispondenti indirizzi

- Istruzioni:

```
label_name:  addi $4, $5, 10
              sub  $4, $4, $6
```

- Dati (variabili a, b, c, da 4 bytes ciascuna)

```
.data
a: .space 4      # lascia uno spazio di 4 bytes
b: .space 4      # lascia uno spazio di 4 bytes
c: .space 4      # lascia uno spazio di 4 bytes
```

## Etichette (cont.)

---

- **Possiamo tradurre agevolmente l'espressione C:**

`a = b + c;`

**con:**

`lw $t1, b`

`lw $t2, c`

`add $t0, $t1, $t2`

`sw $t0, a`

# Direttive assembler per l'allocazione dei dati

---

- **Direttive per l'allocazione di dati inizializzati**

```
.half    h1,...,hn    # dich. di una sequenza di n half-word;  
.word    w1,...,wn    # dich. di una sequenza di n word (interi)  
.byte    b1,...,bn    # dich. di una sequenza di byte  
.float    f1,...,fn    # dich. di una sequenza di float  
.double   d1,...,dn    # dich. di una sequenza di double  
.asciiz   str          # dich. di una stringa cost. (terminata da 0)
```

- **Gli indirizzi dei vari elementi sono scelti in modo da rispettare degli allineamenti prefissati**
  - **.half** alloca i vari elementi su indirizzi multipli di 2, mentre **.word** e **.float** su indirizzi multipli di 4, e **.double** su indirizzi multipli di 8

# Direttive assembler per l'allocazione dei dati

---

- **Per allocare array monodimensionali (vettori):**

```
int a[10];  
char b[5];
```

**In assembly traduciamo come segue:**

```
.data  
.align 2  
a:      .space 40  
b:      .space 5
```

- **Per inizializzare la memoria con una costante:**

```
.word numero
```

- **Esempio:**

```
.data  
a:  .word 5    #  fai spazio per la variabile "a"  
                #  intera, e ponila uguale a 5
```

# Traduzione assembly dei principali costrutti di controllo C

---

- `if (condizione) . . . ; else . . . ;`
- `while (condizione) . . . ;`
- `do {  
    . . .  
} while (condizione);`
- `for (. . . ; . . . ; . . . )  
    . . . ;`
- `switch (expression) {  
    case i: . . . ; break;  
    case j: . . . ; break;  
    . . .  
}`

# if (condizione) . . . ; else . . . ;

---

- In assembly esistono solo salti:
  - salti condizionati: `beq, bne`
  - salti incondizionati: `j`
- . . . questi equivalgono in C all'uso di costrutti “deprecati”:
  - `if (condizione) goto label;`
  - `goto label;`
- Usando i *goto* ad un *etichetta*, il costrutto *if* può essere riscritto in C come:

```
if (!condizione) goto else-label;
...;    // ramo then
goto exit-label;

else-label:
...;    // ramo else

exit-label:
```

# if (condizione) . . . ; else . . . ;

---

- In assembly il costrutto *if* può essere tradotto come segue:

```
    # calcola condizione booleana e ponila in $t0
    beq $t0, $zero, else-label
    . . .      # traduzione ramo then
    j exit-label
else-label:
    . . .      # traduzione ramo else
exit-label:
```



# while (condizione) . . . ;

---

- Usando i *goto*, il costrutto *while* può essere riscritto in C:

```
init-while:
    if (!condizione) goto exit-while;
    ...;    // corpo del while
    goto init-while;
exit-while:
    ...
```

- Compilazione in assembly:

```
init-while:
    # calcola condizione in $t0
    beq $t0, $zero, exit-while
    ...    # corpo del while
    j init-while
exit-while:
    ...
```

# while ottimizzata

---

- Usando i *goto*, il costrutto *while* può essere riscritto in C:

```
    if (!condizione) goto exit-while;
init-while:
    ...;    // corpo del while
    if (condizione) goto init-while;
exit-while:
    ...
```

- Compilazione in assembly:

```
    # calcola condizione in $t0
    beq $t0, $zero, exit-while
init-while:
    ...    # corpo del while
    # calcola condizione in $t0
    bne $t0, $zero, init-while
exit-while:
    ...
```

- Viene eseguito solo un salto condizionato per ciclo, invece di un salto condizionato ed uno incondizionato

# do ... ; while (condizione);

---

- Usando i *goto*, il costrutto *do* può essere riscritto in C:

```
init-do:
    ...;    // corpo del do
    if (condizione) goto init-do;
    ...
```

- Compilazione in assembly:

```
init-do:
    ...    # corpo del do
    # calcola condizione in $t0
    bne $t0, $zero, init-do
    ...
```

# for (<init>; <cond>; <incr>;) ...;

---

- Possiamo esprimere il *for* con il *while*, per il quale possiamo usare la traduzione assembly precedentemente introdotta:

```
<init>;  
while (<cond>) {  
    ...      // corpo del for  
    <incr>;  
}  
...
```

# switch (expr) { case i: ...; break; ... }

---

- Possiamo esprimere lo *switch* con una cascata di *if then else*, per il quale possiamo usare la traduzione assembly precedentemente introdotta.

- **Esempio di switch**

```
switch (expression) {  
    case i: . . . ; break;  
    case j: . . . ; break;  
    case k: . . . ; break;  
}
```

- **Traduzione**

```
if (expression == i)  
    ...;    // corpo case i  
else if (expression == j)  
    ...;    // corpo case j  
else if (expression == k)  
    ...;    // corpo case k
```

# Procedure e funzioni

---

- Le procedure/funzioni sono delle porzioni di programmi di particolare utilità che possono essere richiamate per eseguire uno specifico compito
- In genere sono «generalizzate» per poter essere richiamate più volte al fine di eseguire compiti «parametrizzati»
- Le funzioni ritornano un valore, le procedure no
  - in C queste ultime hanno tipo `void`
- Nome alternativo: subroutine
- Una procedura può ancora richiamare un'altra procedura, e così via ....
  - Il programma principale chiama la procedura A
  - A inizia l'esecuzione
  - A chiama la procedura B
  - B inizia e completa l'esecuzione
  - B ritorna il controllo al *chiamante* (procedura A)
  - A completa l'esecuzione
  - A ritorna il controllo al *chiamante* (programma principale)

# Procedure e funzioni (cont.)

---

- Per **chiamare** una procedura, abbiamo bisogno di dare il controllo al codice della procedura stessa
  - Dobbiamo **saltare** alla **prima istruzione** del codice della procedura
- Abbiamo già visto istruzioni per saltare
  - In particolare, l'istruzione **jal** (jump and link, salta e collega)
  - **jal indirizzo**
  - salta all'indirizzo, e mette nel registro speciale **\$ra** (return address - \$31) la locazione di memoria successiva alla **jal**
- Quindi se la procedura A vuole chiamare B ... e la prima istruzione di B è memorizzata all'indirizzo **indirizzodiB**
  - ➔ A può usare: **jal indirizzodiB**
- Una volta saltati in B (da A), al termine di B il controllo deve essere restituito ad A .... per tornare a eseguire quel che resta di A
  - ➔ B può usare **jr \$ra**

# Passaggio di parametri e ritorno dei risultati

---

- Ogni procedura può essere vista come un piccolo programma a sé stante:
  - Riceve dei dati in input
  - Fa qualcosa
  - Dà dei risultati in output
- Sono necessari **due contenitori (input e output)** per scambiare dati tra programma chiamante e procedura chiamata
  - Locazioni in memoria dove mettere i dati di input/output
  - Che memoria usiamo ?
  - Cosa è più conveniente?
- Siccome nella programmazione assembler le chiamate di procedura sono molto frequenti, conviene usare come **contenitori di input/output dei registri**, che sono più veloci
  - Soluzione ottimale solo se i dati in input/output sono pochi e sufficientemente piccoli



# Passaggio di parametri e ritorno dei risultati (cont.)

- **Esempio:**
  - A chiama B
  - A e B si accordano per avere l'input in \$t3, e restituire l'output in \$t4
  - B processa il dato fornitogli in \$t3 (lo moltiplica per 4 con due somme)
  - Infine restituisce il risultato in \$t4

B:

```
add $t3,$t3,$t3
add $t4,$t3,$t3
jr $ra
```

```
int B(int x) {
    return x*4;
}
```

- **NECESSARIO UN ACCORDO TRA CHIAMATO E CHIAMANTE:**
  - In generale ci sono MOLTE PROCEDURE
  - Per ricordarci meglio dove dobbiamo mettere i dati in input e dove li ritroviamo i dati in output
    - è meglio usare la stessa convenzione per tutte le procedure/funzioni

# Side Effect

---

- Il problema dei SIDE EFFECT (effetto collaterale):
  - Una volta eseguito il programma della procedura B, non solo il registro di ritorno `$t4` è stato modificato, ma anche `$t3` è stato modificato:

B:

```
add $t3,$t3,$t3
```

```
add $t4,$t3,$t3
```

```
jr $ra
```

- Il problema è che una procedura in esecuzione può aver bisogno di risorse di memoria (in questo caso, il registro `$t3`), per i suoi calcoli
- Queste risorse verranno quindi modificate rispetto ai valori originali
  - l'invocazione di una procedura può causare alcune modifiche dello stato (registri) non richieste
    - ➔ SIDE EFFECT (*effetto collaterale*)

# Side Effect (cont.)

---

- Bisogna fare attenzione ai side-effects di una procedura, perché si possono modificare/distruggere in modo irreparabile i valori di qualche registro  
→ registri “modificati” da una procedura, in aggiunta a quello/quelli di output
- Quando si chiama una procedura/funzione con side-effects, abbiamo due alternative:
  1. Il chiamante si assicura che i registri, **potenzialmente modificabili** dal chiamato come *side effect*, **NON CONTENGANO** valori da ri-utilizzare dopo il ritorno della procedura/funzione (soluzione non sempre possibile, dipende dal numero di registri coinvolti)
    - Necessario documentare, approccio poco generale (vedi next slide)
  2. Qualcuno **salva i registri** importanti per il prosieguo della computazione, modificabili come *side-effect* di una chiamata, in un'**area di memoria** (store), viene quindi eseguita la procedura, e poi si **ripristinano** dalla memoria i **registri salvati** (load)
    - Approccio generale, possibile stabilire «convenzioni di chiamata» per dare ruoli e responsabilità su attività di salvataggio/ripristino

# Quindi?

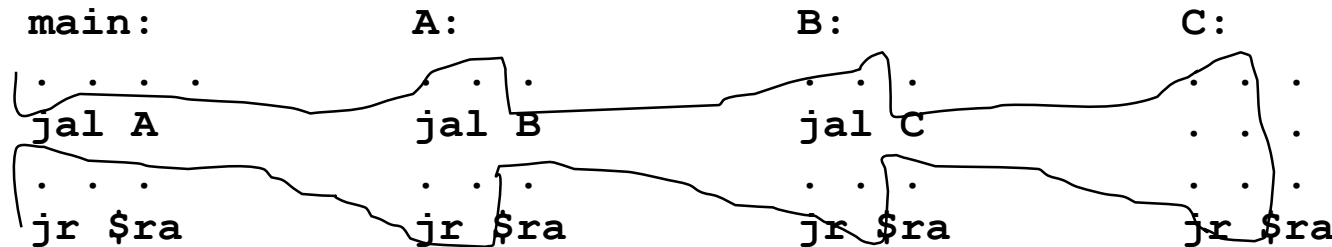
- Un possibile approccio potrebbe essere quello di documentare chiaramente quali sono i **registri potenzialmente modificabili** come *side-effect*:
  - Non solo dov'è l'input e dov'è l'output, **ma anche se ci sono altri registri modificati**

```
# INPUT: $t3
# OUTPUT: $t4 (conterrà 4*$t3)
# MODIFICA: $t3
B:
    add $t3,$t3,$t3
    add $t4,$t3,$t3
    jr $ra
```

- Questo tipo di approccio **non è compatibile** con il tipico approccio alla programmazione “**black box**”:
  - chiamante e chiamata non dovrebbero conoscere nulla dei dettagli interni delle implementazioni
  - funzioni scritte anche da persone diverse, sostituibili una volta che la relativa interfaccia è nota

# Più procedure

- Visto che ogni procedura è un programma a sé stante, nulla vieta di chiamare altre procedure
- Esempio, `main` può chiamare la procedura `A`, `A` può chiamare `B`, e `B` può chiamare `C`



- Qui c'è un problema che va al di là dei *side effects*
    - Ogni volta che invochiamo `jal`, il registro `$ra` viene riscritto !!
    - Se non lo **salviamo**, perdiamo il valore precedente, che serve per effettuare il *return della procedura*
- ➔ **Soluzione: si usano delle convenzioni standard per le chiamate di procedura, con ruoli ben definiti su salvataggio/ripristino dei registri**

# Registri MIPS e convenzioni d'uso

Register name	Number	Usage
\$zero	0	constant 0
\$at	1	reserved for assembler
\$v0	2	expression evaluation and results of a function
\$v1	3	expression evaluation and results of a function
\$a0	4	argument 1
\$a1	5	argument 2
\$a2	6	argument 3
\$a3	7	argument 4
\$t0	8	temporary (not preserved across call)
\$t1	9	temporary (not preserved across call)
\$t2	10	temporary (not preserved across call)
\$t3	11	temporary (not preserved across call)
\$t4	12	temporary (not preserved across call)
\$t5	13	temporary (not preserved across call)
\$t6	14	temporary (not preserved across call)
\$t7	15	temporary (not preserved across call)

# Registri MIPS e convenzioni d'uso (cont.)

Register name	Number	Usage
\$s0	16	saved temporary (preserved across call)
\$s1	17	saved temporary (preserved across call)
\$s2	18	saved temporary (preserved across call)
\$s3	19	saved temporary (preserved across call)
\$s4	20	saved temporary (preserved across call)
\$s5	21	saved temporary (preserved across call)
\$s6	22	saved temporary (preserved across call)
\$s7	23	saved temporary (preserved across call)
\$t8	24	temporary (not preserved across call)
\$t9	25	temporary (not preserved across call)
\$k0	26	reserved for OS kernel
\$k1	27	reserved for OS kernel
\$gp	28	pointer to global area
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address (used by function call)

# Salvataggio e stack

- Dove salvo i registri che devono essere preservati (*preserved across calls*)?
- Soluzione che non funziona in generale:
  - Ogni procedura: spazio di memoria statico/privato per salvare i registri
  - Ma se la procedura è ricorsiva, abbiamo bisogno di un numero arbitrario di questi spazi di memoria
- Soluzione generale:
  - Nota che le chiamate di procedura e i ritorni seguono una politica stretta di tipo **LIFO** (last-in, first-out)

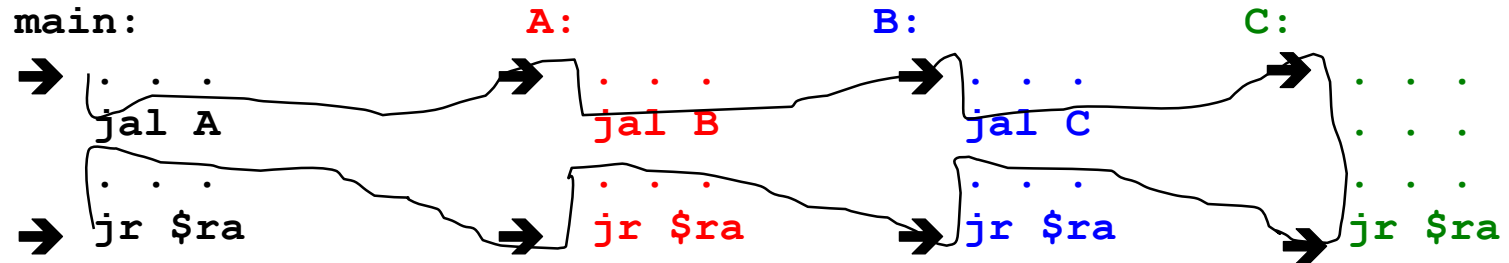


- Quindi la **memoria dove salviamo i registri** può essere allocata e deallocata su una struttura dati **STACK** (Pila)
- I blocchi di memoria allocati per ciascuna invocazione di procedura sono chiamati **STACK FRAMES**

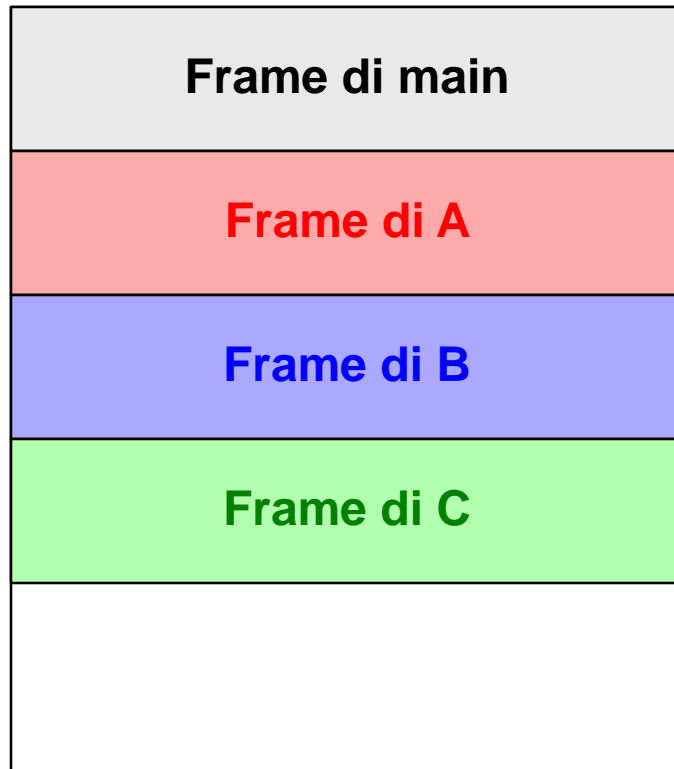


# Allocazione/Deallocazione degli stack frame

- Esempio animato di allocazione dei frame sullo stack:



## STACK



Indirizzi alti



Direzione di  
crescita dello  
stack

# Stack e operazioni

---

- **PUSH(registro)** mette un registro in cima allo stack

```
addi $sp,$sp,-4      # fa spazio sullo stack
sw  registro,0($sp)   # mette il registro nello stack
```

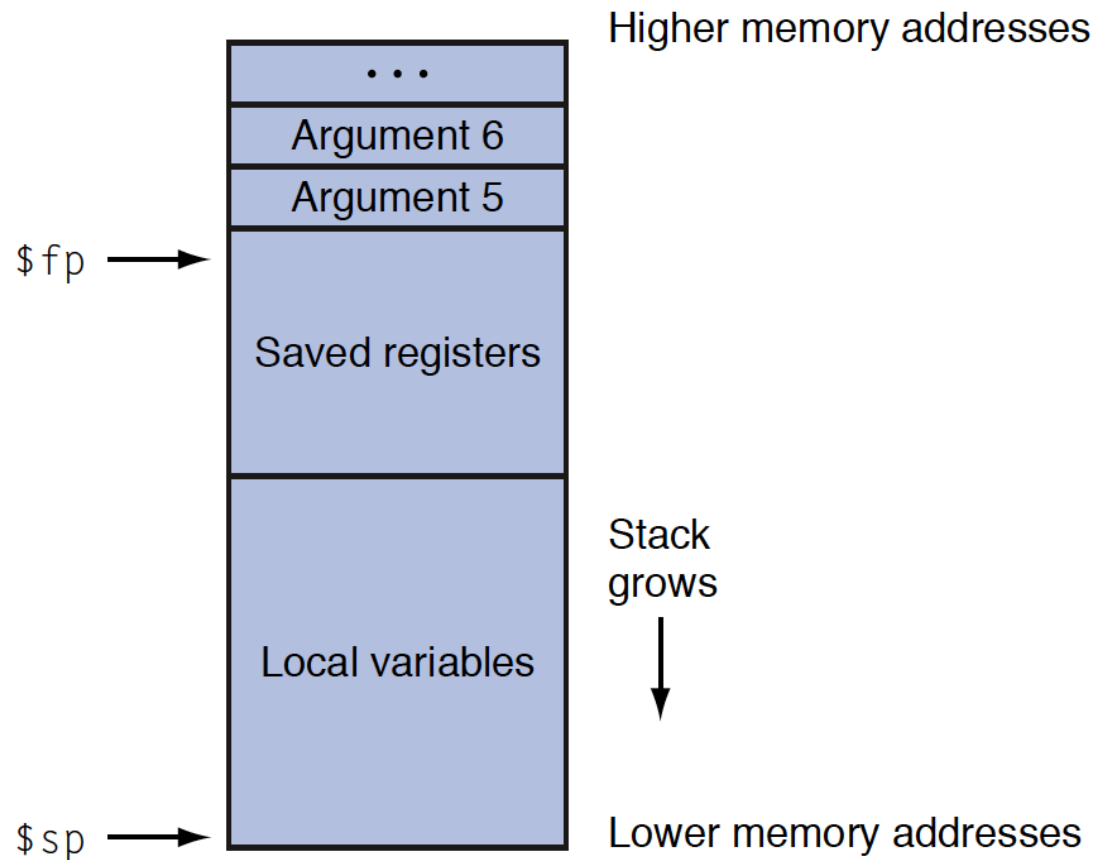
- **POP** toglie l'elemento in cima allo stack (e lo mette in un registro)

```
lw  registro,0($sp)   # copia il primo elemento dello stack
                                # nel registro
addi $sp,$sp,4        # libera lo stack
```

- `$sp ($29)` = stack pointer
- Lo stack permette di allocare uno spazio di memoria privato (FRAME) per ogni istanza/chiamata di procedura
  - più istanze attive nel caso di ricorsione
- Lo stack è dinamico
  - la dimensione si adatta alle necessità (ne uso di più o di meno a seconda delle necessità)

# Stack frame

- Viene usato per salvare e poi ripristinare i registri
- Viene anche usato per
  - Passare parametri
  - Allocare memoria per le variabili locali della procedura/funzione
- Lo stack pointer ( $\$sp$ ) punta all'ultima word nel frame
- I primi 4 argomenti sono passati nei registri ( $\$a0, \dots, \$a3$ )



# Chi salva i registri?

- Chi è responsabile per il salvataggio/ripristino dei registri quando si effettuano chiamate di funzioni?
  - La funzione *chiamante* (**caller**) conosce quali registri sono importanti per sé e che dovrebbero essere salvati
  - La funzione *chiamata* (**callee**) conosce quali registri userà e che dovrebbero essere salvati prima di modificarli
- Entrambi i metodi potrebbero portare a salvare più registri del necessario, se si accetta l'approccio "**black-box**"
  - La funzione *chiamante* (**caller**) salverebbe tutti i registri che sono importanti per sé, anche se la procedura chiamata non li modificherà
  - La funzione *chiamata* (**callee**) salverebbe tutti i registri che si appresta a modificare, anche quelli che non verranno poi utilizzati dalla procedura chiamante una volta che la procedura chiamata le avrà restituito il controllo (return: jr \$ra)

# Cooperazione tra callee e caller per salvare i registri

---

- Chi deve *salvare* i valori dei registri nello stack?
  - la procedura *chiamante* (**caller**) o quella *chiamata* (**callee**)?
  - In realtà, esistono delle **convenzioni di chiamata**, sulla base delle quali le **procedure cooperano** nel salvataggio dei registri
- I registri (\$s0, . . . , \$s7, \$ra) si presuppone che siano **importanti per la procedura chiamante**, e debbano essere salvati dal chiamato (**callee**) prima di procedere a modificarli
- \$s0, . . . , \$s7      (***callee-saved registers***)
  - devono essere salvati dalla procedura *chiamata* prima di modificarli
- \$ra      (***callee-saved registers***)
  - La procedura *chiamata* salva il registro \$ra prima di modificarlo, ovvero prima di eseguire la *jal* (e diventare a sua volta **caller**) !!!

# Cooperazione tra callee e caller per salvare i registri

---

- Riserviamoci un certo numero di registri che ogni procedura può **modificare liberamente**:
  - `$t0 . . . $t9`Infatti, “t” sta per temporaneo
- Inoltre, si presuppone che ogni procedura possa liberamente modificare i registri `$a0, . . . $a3` e `$v0, $v1`
  - Questo succede se la procedura dovesse invocare a sua volta altre procedure, che a loro volta restituiscono risultati
- Quindi i registri `$t0 . . . $t9, $a0, . . . $a3, $v0, $v1` devono essere preservati, se necessario, dal **chiamante**
  - La procedura **chiamante** assume che la procedura **chiamata** sia libera di modificarli liberamente !!

# Cooperazione tra callee e caller per salvare i registri?

---

- $\$a0, \dots, \$a3$  (*caller-saved registers*)
  - Registri usati per passare i primi 4 parametri
  - La procedura *chiamante* assume che la procedura *chiamata* li modifichi, ad esempio per invocare un'altra procedura
  - Li salva solo se ha necessità di preservarli
- $\$t0, \dots, \$t9$  (*caller-saved registers*)
  - Poiché la procedura chiamata potrebbe modificare questi registri, se il *chiamante* ha proprio la necessità di preservare qualcuno di essi, sarà suo compito salvarli !!
- $\$v0, \$v1$  (*caller-saved registers*)
  - Registri usati per ritornare i risultati
  - La procedura *chiamante* assume che la procedura *chiamata* li possa modificar, e li salva solo se ha necessità di preservarli

# Minimo salvataggio di registri

---

- La funzione *chiamata* non deve salvare nulla se
  - non chiama nessun'altra funzione (non modifica \$ra)e se scrive solo
  - nei registri temporanei (\$t0, ... \$t9)
  - nei registri usati per gli argomenti della funzione (\$a0, ... \$a3)
  - in quelli usati convenzionalmente per i risultati (\$v0, \$v1)
  - in quelli non indirizzabili (\$Hi, \$Lo)
- La funzione *chiamante* non deve salvare nulla se non necessita che certi registri vengano preservati al ritorno della chiamata
  - (\$t0, ... \$t9, \$v0, \$v1, \$a0, ... \$a3 )



# Minimo salvataggio di registri (esempio)

- La funzione `main`, in qualità di funzione *chiamata*, invoca a sua volta la funzione `quadrato`, modificando quindi `$ra` che deve quindi salvare
  - `$ra` (*callee-saved*)

```
.data
x:          .word 5
result:     .space 4
```

```
.text
main:
# salvo ra
# PUSH($ra)
    addi $sp,$sp,-4
    sw $ra,0($sp)
```

```
# chiamo la procedura
# quadrato
lw $a0, x
jal quadrato
sw $v0,result
```

```
# ripristino ra
# POP($ra)
lw $ra, 0($sp)
addi $sp,$sp,4

# return dal main
jr $ra
```

```
# procedura che
# calcola il quadrato
# dell'argomento
# intero passato
# in $a0
```

```
quadrato:
    mult $a0,$a0
    mflo $v0
    jr $ra # return
```

# Minimo salvataggio di registri (esempio)

---

- La funzione *chiamata* **quadrato** non deve salvare nulla perché
  - non chiama nessun'altra funzionee scrive solo
  - nei registri `$a0, v0`
  - Nei registri non indirizzabili (`$Hi, $Lo`)

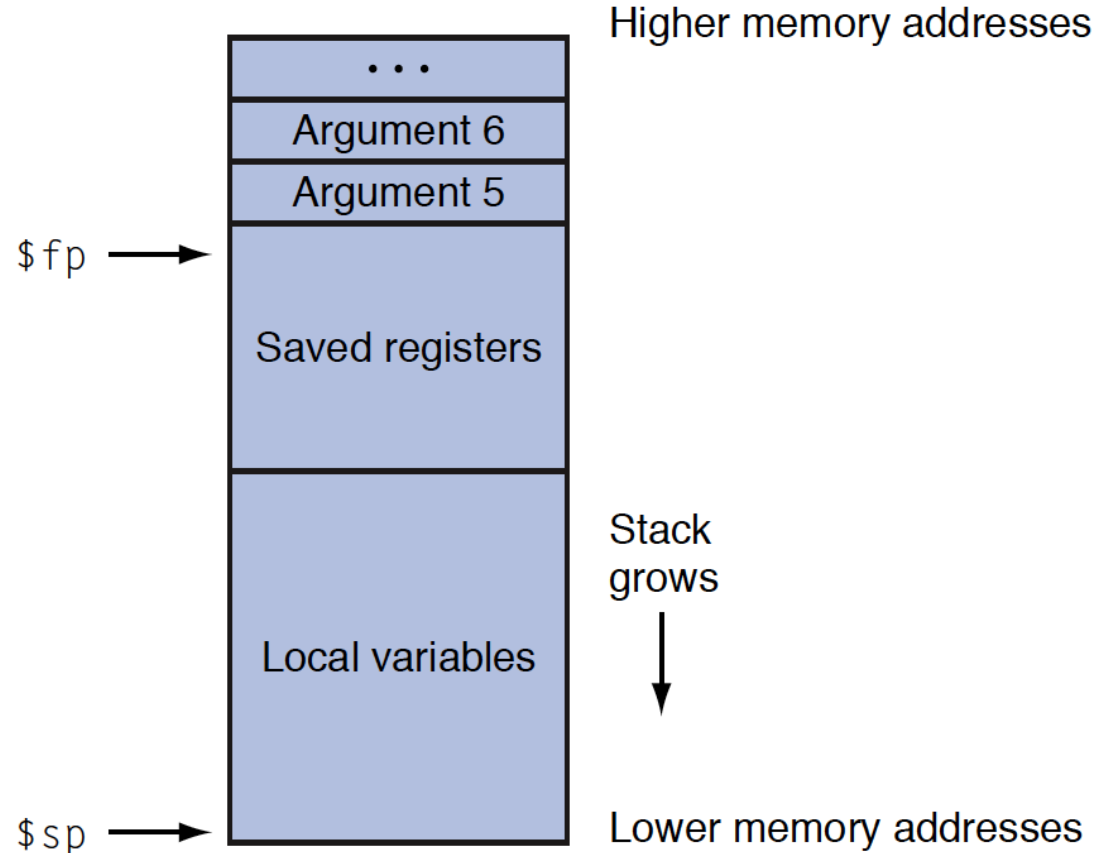
```
# procedura che calcola il quadrato  
# dell'argomento intero passato in $a0
```

**quadrato:**

```
mult $a0, $a0  
mflo $v0  
jr $ra # return
```

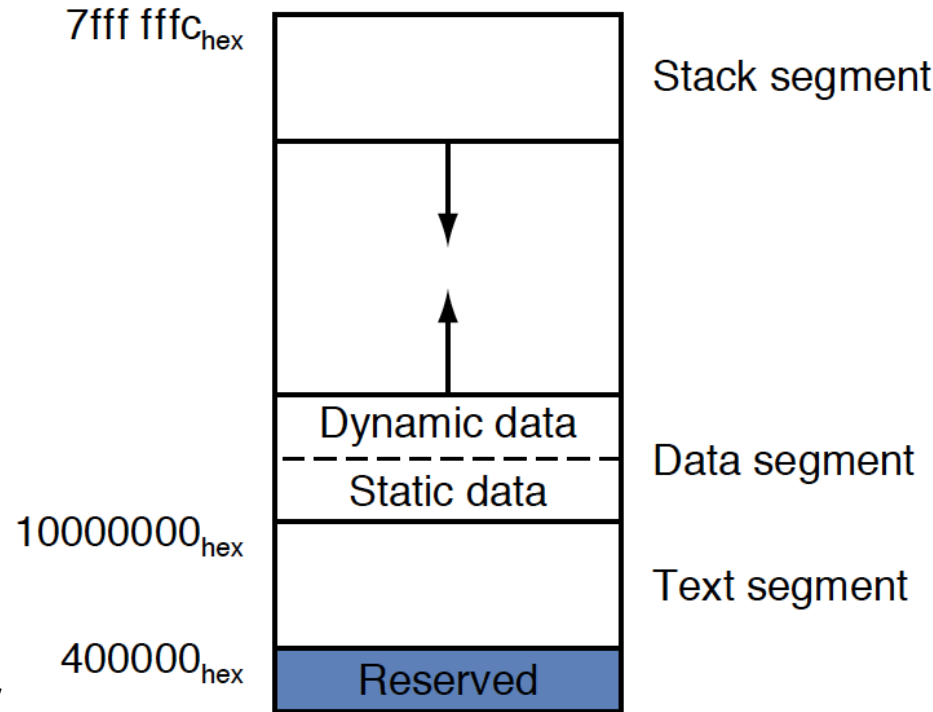
# Frame pointer

- Se ad ogni operazione di **PUSH/POP** modifichiamo dinamicamente **\$sp**
  - Utile utilizzare il registro **\$fp** (frame pointer) per “ricordare” l’inizio del frame
  - Questo serve a cancellare velocemente il frame al ritorno della procedura
- L’uso di questo registro è opzionale
- Il nuovo valore di **\$fp** deve essere assegnato all’inizio della procedura (dopo averne salvato il vecchio valore sullo stack)



# Layout della memoria

- I sistemi basati sul MIPS dividono la memoria in 3 parti
- **Text segment**
  - Istruzioni del programma, poste in memoria virtuale a partire da una locazione vicina all'indirizzo 0: 0x400000
- **Data segment**
  - **Static data** (a partire dall'indirizzo 0x10000000)
    - Contiene variabili globali, attive per tutta l'esecuzione del programma
  - **Dynamic data**
    - Immediatamente dopo i dati statici
    - Dati allocati dal programma durante l'esecuzione (malloc() in C)
- **Stack segment**
  - Inizia in cima all'indirizzamento virtuale (0x7fffffff) e cresce in senso contrario rispetto al data segment



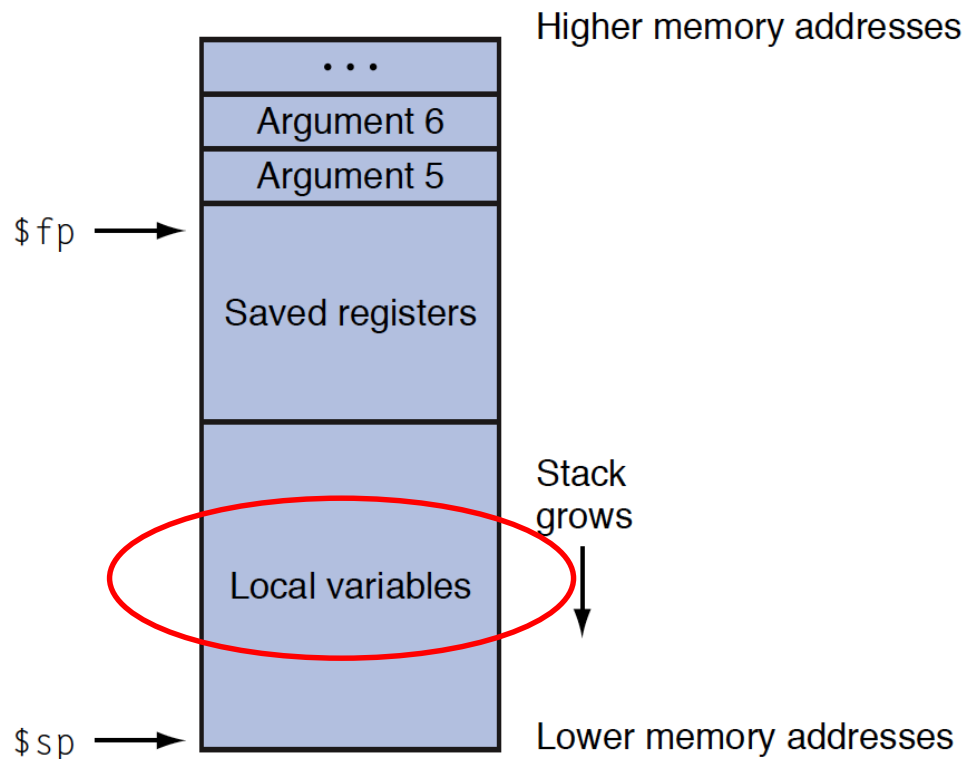
# Variabili globali e locali

---

- In C le variabili possono essere definite
  - Nel corpo delle funzioni (variabili locali o automatiche)
  - Esternamente alle varie funzioni (variabili globali)
- Scoping delle variabili
  - Le variabili globali possono essere lette e scritte da tutte le funzioni
  - Le variabili locali sono private delle funzioni in cui sono definite
- Per ora abbiamo visto solo come implementare le variabili globali
  - Direttiva `.data` per le variabili globali
- Come facciamo a implementare le variabili locali?

# Variabili locali

- Il problema è lo stesso che abbiamo visto per creare spazio privato dove salvare i registri
- Anche la soluzione è la stessa: usare lo Stack
  - ➔ le variabili vengono assegnate allo stack e  $\$sp$  viene spostato di conseguenza



# Variabili locali (esempio)

```
/* Funzione C incr
*/
int foo(int x) {
    int temp;
    temp=x+x;
    temp=temp+100;
    return temp;
}
```

**# Traduzione MIPS modulare,  
# senza ottimizzazione**

```
foo:
    addi $sp,$sp,-4    # alloco stack frame per
                        # temp

    add  $t0,$a0,$a0
    sw   $t0, 0($sp)   # memorizza temp

    lw   $t1, 0($sp)
    addi $t1, $t1, 100
    sw   $t1, 0($sp)   # memorizza temp

    lw   $v0, 0($sp)   # metto in $v0 (ritorno)
                        # la variabile temp
    addi $sp,$sp,4     # disalloco stack frame

    jr   $ra
```

# Word e Byte

---

- Finora, abbiamo solo agito con words (4 bytes)
- Però, abbiamo visto che ci sono istruzioni anche per manipolare i singoli bytes:
  - **sb reg2, indirizzo(reg1)**
    - Copia il byte più basso di reg2 alla locazione di memoria indirizzo+reg1
  - **lbu reg2, indirizzo(reg1)**
    - Copia nel byte più basso di reg2 il byte (senza estendere il segno) presente nella locazione di memoria indirizzo+reg1
    - Mette a 0 i 3 byte più significativi
    - Esiste versione con estensione di segno **lb reg2, indirizzo(reg1)**
- Motivo principale: manipolare stringhe
  - Le stringhe sono composte di caratteri, dove ogni carattere viene memorizzato in 1 byte
  - *“Questa è una stringa”*
- Nota che i vari Byte in memoria possono essere interpretati in vario modo:
  - Numeri interi o float (4 Byte per volta)
  - Istruzioni (4 Byte per volta)
  - e ... come **caratteri**



# Corrispondenza byte-caratteri

- **Codifica standard dei caratteri:**
  - ASCII: American Standard Code for Information Interchange
  - Codici per rappresentare 128 caratteri (sufficienti 7 bit)

- **Codici**

- Da 0 a 31: caratteri di controllo
  - Es.: 0 : \0 (null)
  - 10 : \n (line feed)
  - 13 : \r (carriage return)
- Da 32 a 126: caratteri stampabili
- 127: carattere di controllo (DEL)

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	<b>NUL</b> (null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	<b>SOH</b> (start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	<b>STX</b> (start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	<b>ETX</b> (end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	<b>EOT</b> (end of transmission)	36	24	044	&#36;	&	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	<b>ENQ</b> (enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	<b>ACK</b> (acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	<b>BEL</b> (bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	<b>BS</b> (backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	<b>TAB</b> (horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	<b>LF</b> (NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	<b>VT</b> (vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	<b>FF</b> (NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	<b>CR</b> (carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	<b>SO</b> (shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	<b>SI</b> (shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	<b>DLE</b> (data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	<b>DC1</b> (device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	<b>DC2</b> (device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	<b>DC3</b> (device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	<b>DC4</b> (device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	<b>NAK</b> (negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	<b>SYN</b> (synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	<b>ETB</b> (end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	<b>CAN</b> (cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	<b>EM</b> (end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	<b>SUB</b> (substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	<b>ESC</b> (escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	<b>FS</b> (file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	
29	1D	035	<b>GS</b> (group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	<b>RS</b> (record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	<b>US</b> (unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: [www.LookupTables.com](http://www.LookupTables.com)

- **UTF-8 (codifica moderna dei caratteri)**
  - Codifica a formato variabile, compatibile backward con ASCII

# ASCII

---

- 32 → ' ' (spazio)      33 → '!'  
34 → '"' (virgolette)    35 → '#'
- I **caratteri alfabetici** corrispondono a **codici consecutivi**
- I **caratteri numerici** corrispondono a **codici consecutivi**
  - Da 65 a 90: 'A', ... , 'Z'
  - Da 97 a 122: 'a', ... , 'z'
  - Da 48 a 57: '0', ... , '9'
- **Nota:**
  - Car + 32 trasforma il carattere Maiuscolo in Minuscolo
  - Car - 32 trasforma il carattere Minuscolo in Maiuscolo
  - Possiamo facilmente controllare se un carattere è una lettera
  - Possiamo facilmente calcolare il “*numero*” rappresentato da un carattere numerico, sottraendo la codifica di '0' (48)
    - Esempio: il *numero* del carattere '2' (ASCII 50) è  $50-48=2$

# Stringhe

---

- **Rappresentazione standard MIPS (come in C):**
  - Sequenza di codici numerici dei caratteri, terminata da un carattere speciale Null (`'\0'`)
  - Come si carica in memoria la stringa **“CIAO”** ?
    1. `.byte 67, 73, 65, 79, 0`
    2. `.asciiz "CIAO"`
    3. `.space 5`
- La terza opzione richiede di memorizzare i vari codici dei singoli caratteri con un serie di istruzioni `sb` (store byte)

# Esempio di uso di stringhe: strcpy()

---

```
void strcpy(char x[], char y[])
{
    int i = 0;
    char tmp;

    do {
        tmp = y[i];
        x[i] = tmp;
        i++;
    } while (tmp != '\\0');
}
```

```
int i = 0;
char tmp;

DOLOOP:
    tmp = y[i];
    x[i] = tmp;
    i++;
    if (tmp != 0) goto DOLOOP;
```

# Esempio di uso di stringhe: strcpy()

```
strcpy:          # $a0 <- &x[0]      $a1 <- &y[0]
    addi $sp, $sp, -8
    sw    $s0, 0($sp)      # PUSH $s0  (i -> $s0)
    sw    $s1, 4($sp)      # PUSH $s1  (tmp -> $s1)

    or     $s0, $zero, $zero    # i=0
DOLOOP:
    add    $t1, $a1, $s0        # $t1 <- &y[i]
    lbu    $s1, 0($t1)          # tmp = y[i]
    add    $t3, $a0, $s0        # $t3 <- &x[i]
    sb     $s1, 0($t3)          # x[i] = tmp
    addi   $s0, $s0, 1          # i++
    bne    $s1, $zero, DOLOOP   # if (tmp != 0) goto DOLOOP

    lw     $s0, 0($sp)          # POP $s0
    lw     $s1, 4($sp)          # POP $s1
    addi   $sp, $sp, 8
    jr     $ra
```

```
int i = 0;
char tmp;

DOLOOP:
    tmp = y[i];
    x[i] = tmp;
    i++;
    if (tmp != 0) goto
    DOLOOP;
```

## Registri modificati:

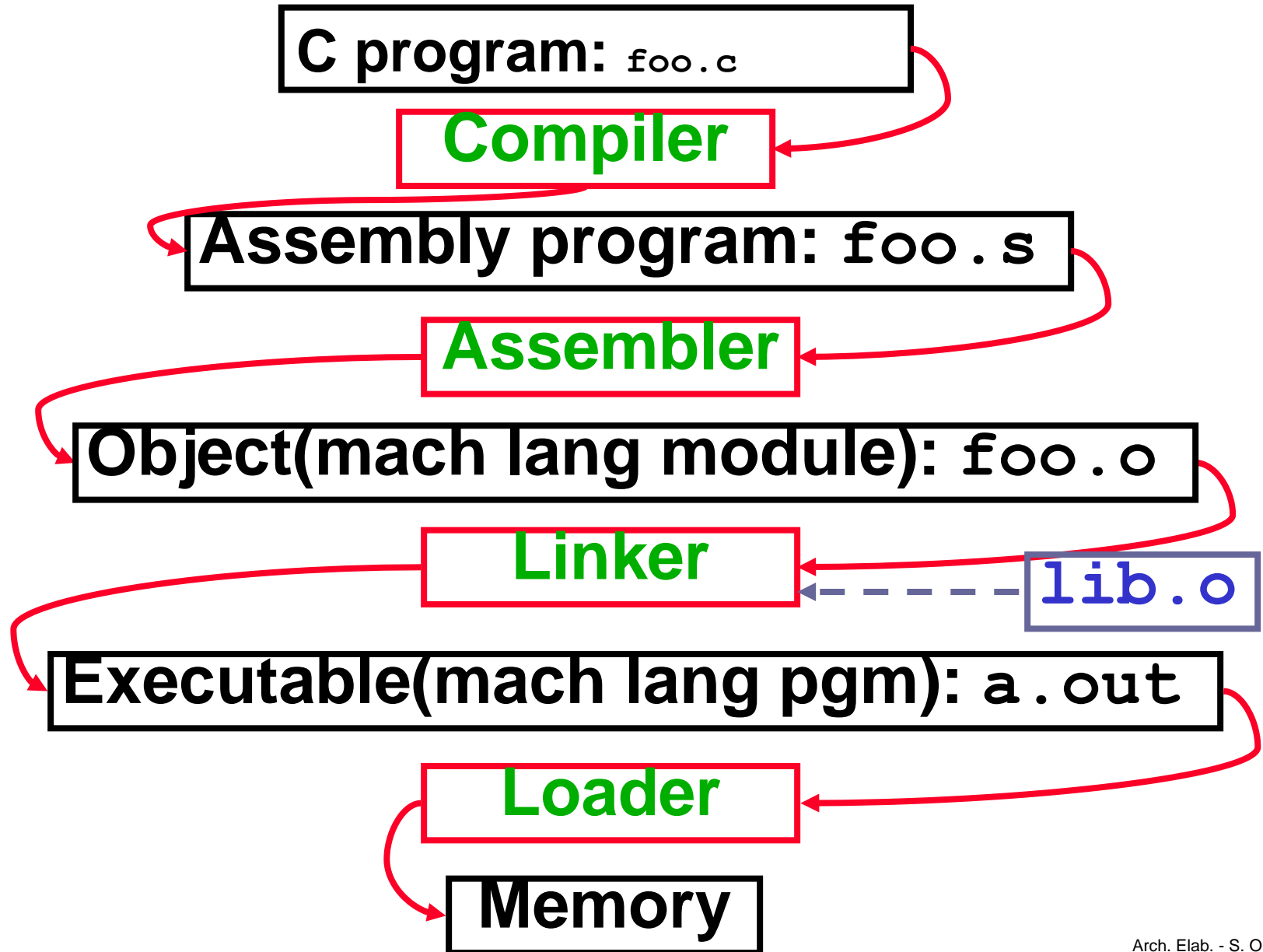
**\$s0 e \$s1: callee save**

**\$t1 e \$t3: temporanei  
da non salvare**

---

**Come viene eseguito un programma?**

# Passi per eseguire un programma C



# Compilatore

---

- Input: High-Level Language Code (es., C)
- Output: Assembly Language Code (es., MIPS)
- Nota: l'output *può* contenere **pseudoistruzioni**



# gcc

*When you invoke GCC, it normally does preprocessing, **compilation**, **assembly** and **linking**. The overall options allow you to stop this process at an intermediate stage. For example, the **-c** option says not to run the linker. Then the output consists of object files output by the assembler*

```
gcc -S foo.c
```

– **invoca il compilatore e produce l'assembly file:** `foo.s`

```
gcc -c foo.c
```

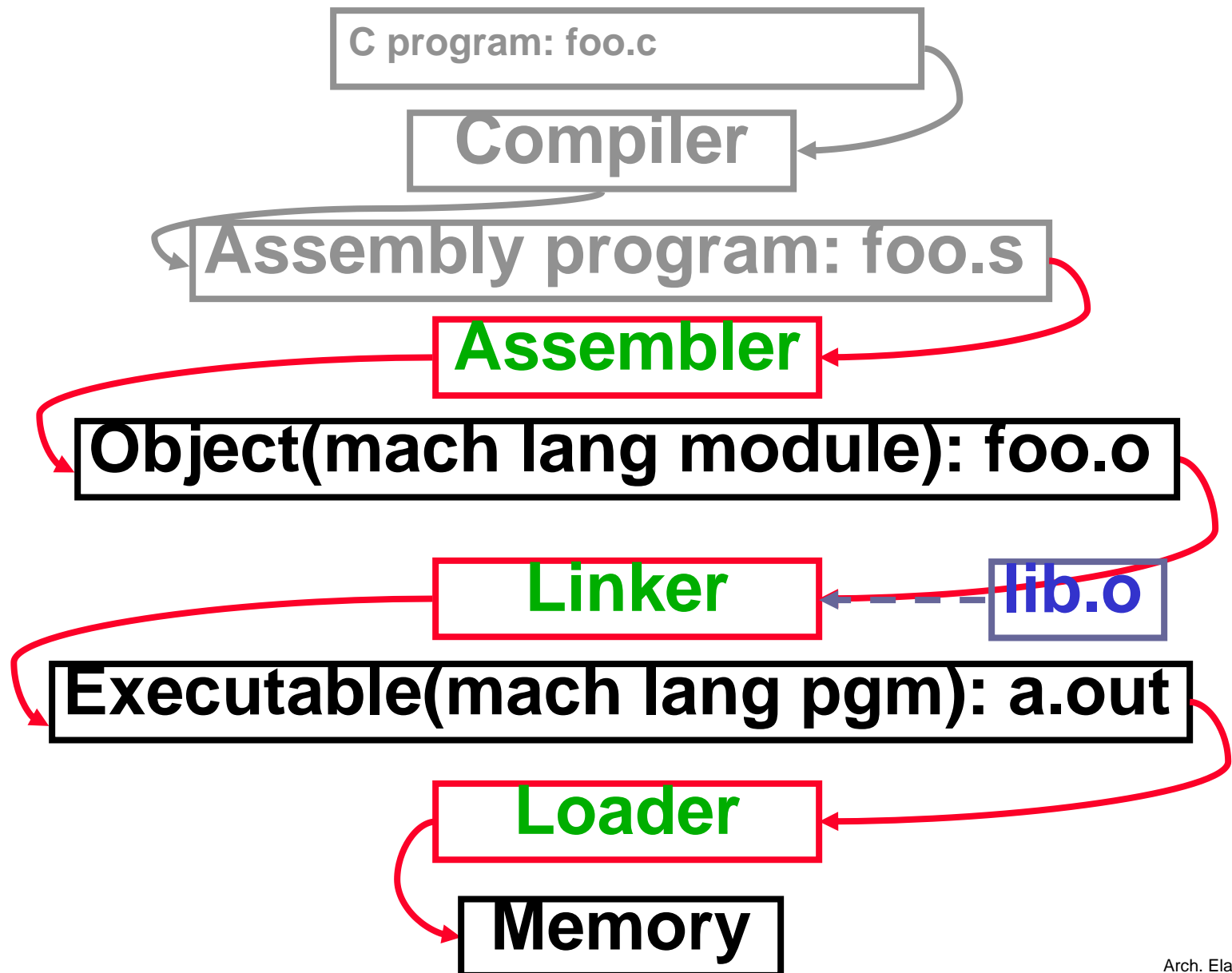
**Possiamo anche invocare `gcc -c foo.c`  
sul file `.c` per produrre il file oggetto `.o`**

– **invoca l'assemblatore e produce un *object file*:** `foo.o`

```
gcc -o foo.out foo.o
```

– **invoca il linker e produce un *executable file*:** `foo.out`

# Assembler



# Assembler

---

- Legge e usa le **Direttive**
- Rimpiazza pseudo-istruzioni
- Produce Machine Code
- Crea un **Object File**

# Produrre Linguaggio Machina (1/2)

---

- **Casi semplici da tradurre**
  - Istruzioni aritmetiche, logiche, ecc.
  - Tutte le informazioni necessarie sono contenute nell'istruzione assembly
- **E i branch?**
  - PC-relative (semplice, si tratta di calcolare dei **displacement** rispetto all'istruzione del **branch**)
  - Difficoltà potrebbero derivare dalle pseudo-istruzioni, che richiedono più istruzioni macchina
    - una volta effettuata il rimpiazzo delle pseudo-istruzioni, sappiamo quante istruzioni saltare
    - necessaria la conoscenza dell'indirizzo corrispondente alla *label* riferita nel branch

# Produrre Linguaggio Machina (2/2)

---

- Cosa succede sui jump (j e jal)?
  - I jump richiedono indirizzi assoluti
  - Se **rilochiamo** il programma durante il linking, gli indirizzi assoluti cambiano
- E i riferimenti ai dati?
  - Esempio:     la label
  - la (load address) è una pseudo-istruzione spezzata in lui e ori
  - Necessario conoscere l'indirizzo completo a 32-bit
  - Ancora, se rilochiamo il programma, gli indirizzi assoluti dei dati possono cambiare
- Dobbiamo creare due tabelle di supporto per le traduzioni ...

# Symbol Table

---

- Lista di “etichette” nominati nel file, che sono usati all’interno del file (e possono essere da altri file) per:
  - Chiamate di funzioni, salti (nella sezione `.text`)
  - Variabili (nella sezione `.data`)
- **First Pass:** registra coppie etichette-indirizzi
- **Second Pass:** produce il codice macchina, sulla conoscenza degli indirizzi corrispondenti a etichette
  - Nota: posso saltare ad una *etichetta* che occorre successivamente senza prima dichiararla

# Relocation Table

---

- La Relocation Table contiene la lista di istruzioni che necessitano di indirizzi assoluti, da rilocare durante il linking
  - per ottenere un unico eseguibile a partire da diversi file oggetto
- Quali sono queste istruzioni?
  - `j` e `jal` che saltano a *label* (=indirizzo)
  - etichetta:
    - interna
    - esterna (anche a file di libreria esterno da link-are)
  - Altre istruzioni che manipolano indirizzi
    - Come l'istruzione `la`

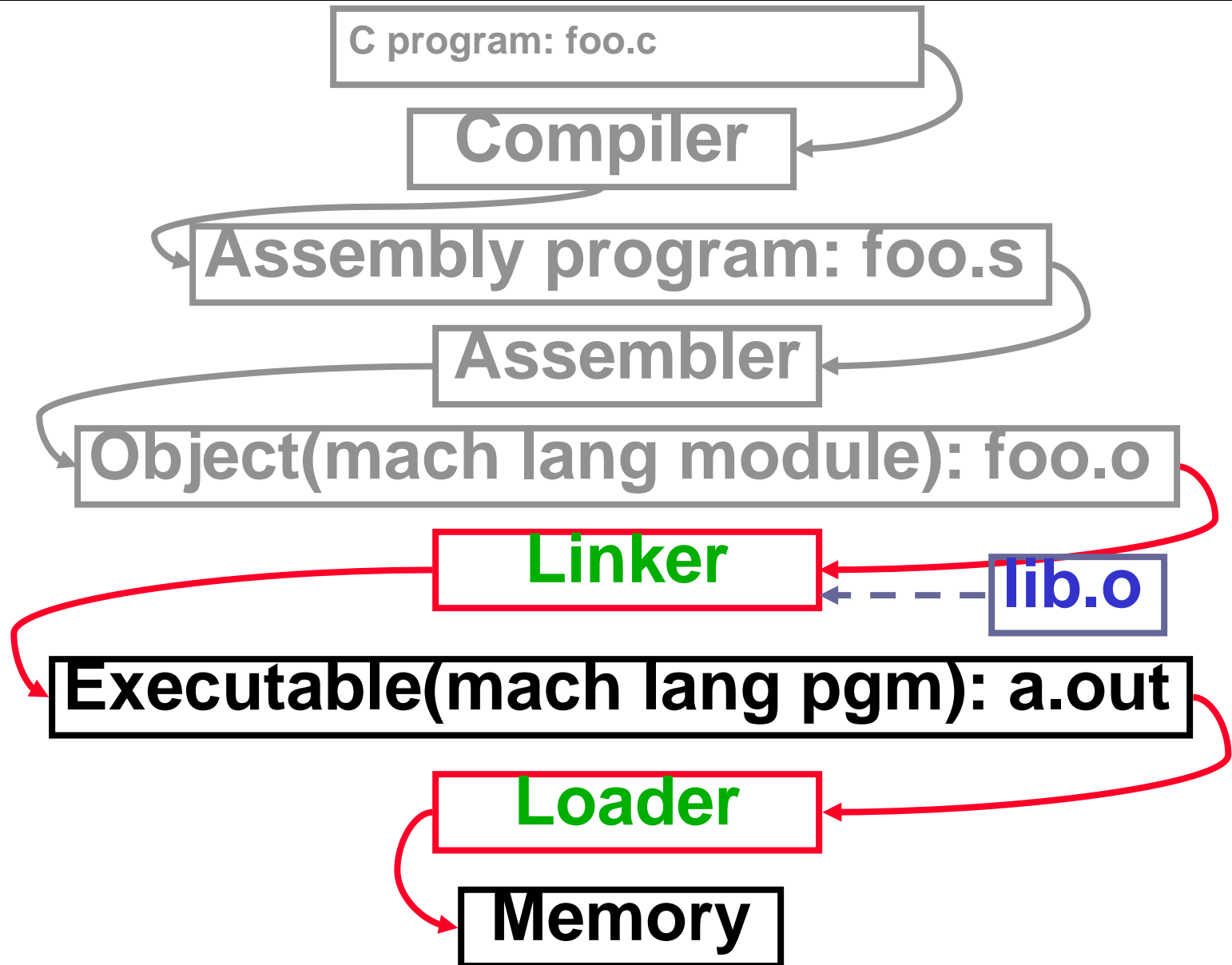
# Object File Format

---

- object file header: dimensione e posizione delle altre porzioni dell' object file
- text segment: contiene il machine code (binario) specificate in assembly tramite la direttiva `.text`
- data segment: rappresentazione (binaria) dei dati allocati in assembly tramite la direttiva `.data`
- relocation information: identifica le istruzioni e i dati che dipendono dall'indirizzo assoluto dove il programma verrà caricato in memoria
- symbol table: lista delle etichette di questo file che sono riferite nel codice



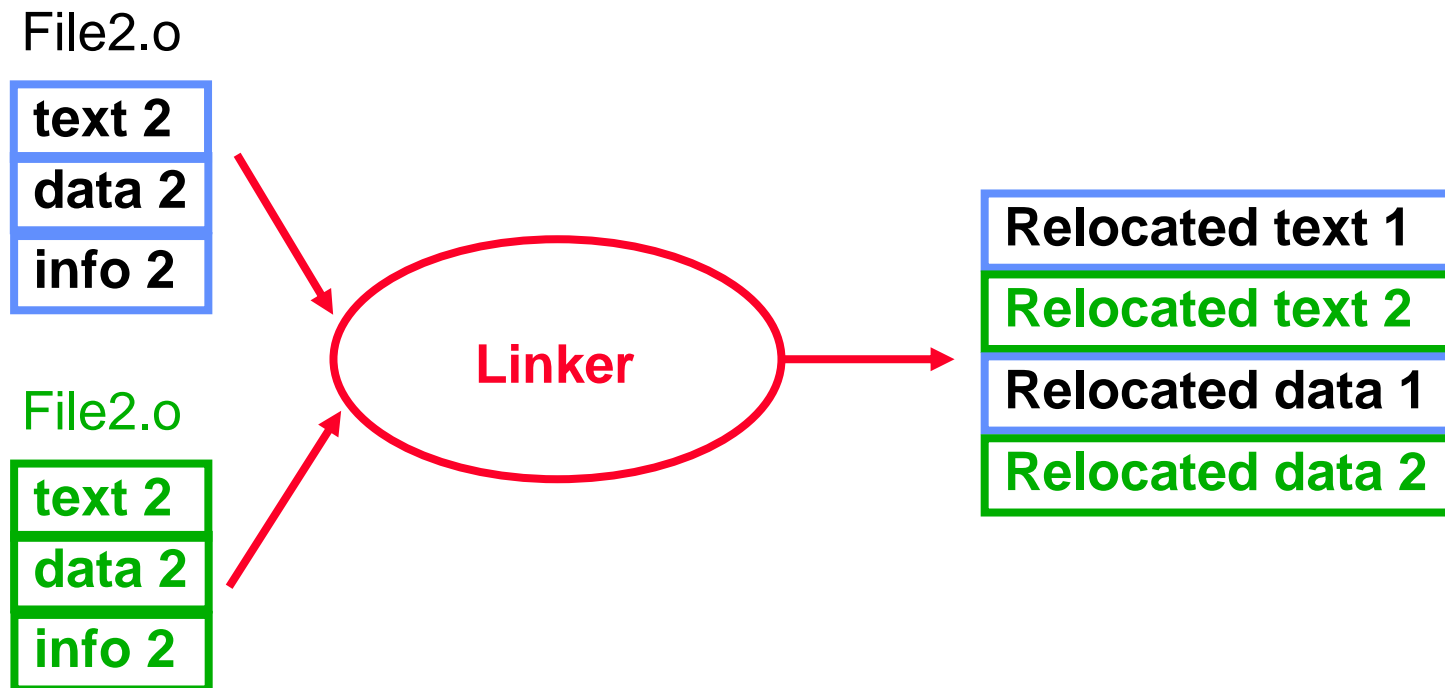
# Linker



# Link Editor/Linker (1/2)

---

- Combina diversi file oggetto (.o) in un singolo file eseguibile (“[linking](#)”)
  - Abilita la compilazione separata



# Link Editor/Linker (2/2)

---

- **Step 1:** Prende i *text segment* da ciascun file .o e li giustappone in un unico segmento.
- **Step 2:** Prende i *data segment* da ciascun file .o e li giustappone in un unico segmento. Infine concatena questo segmento alla fine del segmento text
- **Step 3:** Risolve i riferimenti sulla base della rilocalizzazione dei vari segmenti
  - Giustapporre file rispetto ad un indirizzamento virtuale, modifica potenzialmente gli indirizzi di istruzioni/dati
  - Vai nella tavola di rilocalizzazione e gestisci ciascun ingresso, sistemando gli **indirizzi assoluti**

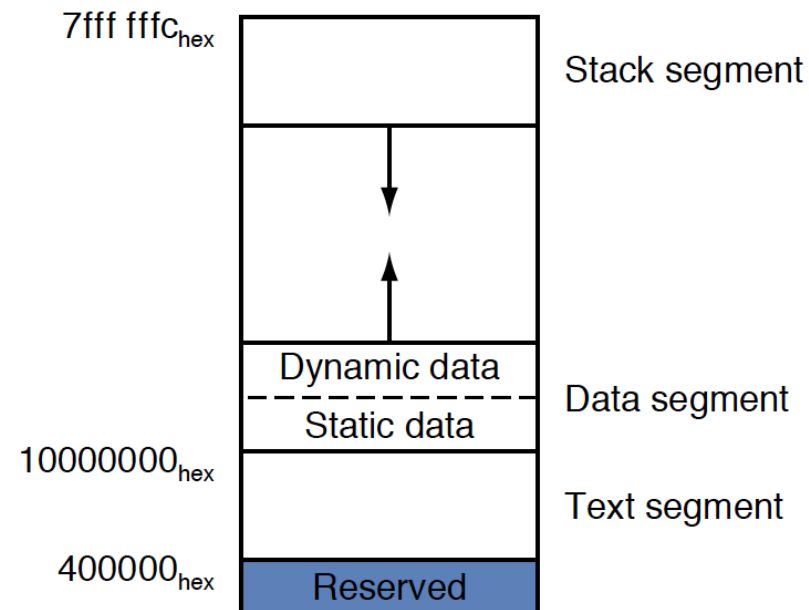
# Quattro tipo di indirizzi

---

- **PC-Relative Addressing (beq, bne):** non riloca mai
- **Absolute Address (j, jal):** riloca sempre
- **External Reference (usually jal):** riloca sempre
- **Data Reference:** riloca sempre

# Risolvere i riferimenti (1/2)

- Il linker *assume* che la prima word del *text segment* è all'indirizzo 0x400000 mentre la prima word del data segment è all'indirizzo 0x10000000.
- Il linker conosce:
  - La lunghezza di ciascun segmento text e data dei vari objects
  - L'ordine dei vari segmenti
- Il linker calcola:
  - L'indirizzo assoluto di ciascuna label a cui saltare (interno o esterno) e ciascun dato riferito



# Risolvere i riferimenti (2/2)

---

- **Per risolvere i riferimenti:**
  - **Cerca i riferimenti in tutte le symbol tables**
  - **Se non trovato, cerca nei file di libreria (per esempio `printf`)**
  - **Una volta che l'indirizzo assoluto è determinato, lo inserisce nel codice macchina in modo appropriato**
- **Output del linker: file eseguibile che contiene text e data (oltre all'header)**

# Esempio di due file oggetto da link-are

Object file header			
	Name	Procedure A	
	Text size	100 <sub>hex</sub>	
	Data size	20 <sub>hex</sub>	
Text segment	Address	Instruction	
	0	lw \$a0, 0(\$gp)	
	4	jal 0	
	...	...	
Data segment	0	(X)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	lw	X
	4	jal	B
Symbol table	Label	Address	
	X	—	
	B	—	
Object file header			
	Name	Procedure B	
	Text size	200 <sub>hex</sub>	
	Data size	30 <sub>hex</sub>	
Text segment	Address	Instruction	
	0	sw \$a1, 0(\$gp)	
	4	jal 0	
	...	...	
Data segment	0	(Y)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	sw	Y
	4	jal	A
Symbol table	Label	Address	
	Y	—	
	A	—	

In **blu** indirizzi e simboli che devono essere aggiornati dal linker

- Istruzioni che fanno riferimento agli indirizzi delle procedure **A B**
- Istruzioni che fanno riferimento agli indirizzi delle parole **X e Y**

# Esempio file eseguibile ottenuto dal linker

Executable file header		
	Text size	300 <sub>hex</sub>
	Data size	50 <sub>hex</sub>
Text segment	Address	Instruction
	0040 0000 <sub>hex</sub>	lw \$a0, 8000 <sub>hex</sub> (\$gp)
	0040 0004 <sub>hex</sub>	jal 40 0100 <sub>hex</sub>
	...	...
	0040 0100 <sub>hex</sub>	sw \$a1, 8020 <sub>hex</sub> (\$gp)
	0040 0104 <sub>hex</sub>	jal 40 0000 <sub>hex</sub>
	...	...
Data segment	Address	
	1000 0000 <sub>hex</sub>	(X)
	...	...
	1000 0020 <sub>hex</sub>	(Y)
	...	...

**Text segment** inizia da 0x400000  
e **Data segment** da 0x10000000.

Le porzioni **text** dei due oggetti, cioè le procedure **A** e **B** (**size<sub>A</sub>** = 0x100 e **size<sub>B</sub>** = 0x200) sono giustapposti e posizionati a partire da 0x400000

Le porzioni **data** dei due oggetti (**size<sub>A</sub>** = 0x20 e **size<sub>B</sub>** = 0x30) sono giustapposti e posizionati a partire da 0x10000000



# Esempio file eseguibile ottenuto dal linker

Executable file header		
	Text size	300 <sub>hex</sub>
	Data size	50 <sub>hex</sub>
Text segment	Address	Instruction
	0040 0000 <sub>hex</sub>	lw \$a0, 8000 <sub>hex</sub> (\$gp)
	0040 0004 <sub>hex</sub>	jal 40 0100 <sub>hex</sub>
	...	...
	0040 0100 <sub>hex</sub>	sw \$a1, 8020 <sub>hex</sub> (\$gp)
	0040 0104 <sub>hex</sub>	jal 40 0000 <sub>hex</sub>
	...	...
Data segment	Address	
	1000 0000 <sub>hex</sub>	(X)
	...	...
	1000 0020 <sub>hex</sub>	(Y)
	...	...

Il linker aggiorna i campi indirizzi delle istruzioni

L'istruzione `jal` all'indirizzo 0x400004 viene aggiornata con **0x400100** (l'indirizzo ora noto della procedure B)

L'istruzione `jal` all'indirizzo 0x400104 viene aggiornata con **0x400000** (l'indirizzo ora noto della procedure A)

Le `lw` / `sw` hanno bisogno di un registro per essere tradotte. Usiamo il `$gp` (global pointer) che immaginiamo essere stato inizializzato a 0x10008000.

- Inizializzando `$gp` (\$28) ad un indirizzo adatto nel data segment, evitiamo di usare sempre `ori` e `lui` prima della `lw` (usando ad esempio `$at`),

Per ottenere l'indirizzo 0x10000000 (indirizzo di X), inseriamo 0x8000 (valore negativo) nel campo immediato della `lw` all'indirizzo 0x400000.

Agiamo analogamente per la `sw` all'indirizzo 0x400100, dove il valore inserito nel campo immediato dell'istruzione è 0x8020

# Esempio file eseguibile ottenuto dal linker

Executable file header		
	Text size	300 <sub>hex</sub>
	Data size	50 <sub>hex</sub>
Text segment	Address	Instruction
	0040 0000 <sub>hex</sub>	lw \$a0, 8000 <sub>hex</sub> (\$gp)
	0040 0004 <sub>hex</sub>	jal 40 0100 <sub>hex</sub>
	...	...
	0040 0100 <sub>hex</sub>	sw \$a1, 8020 <sub>hex</sub> (\$gp)
	0040 0104 <sub>hex</sub>	jal 40 0000 <sub>hex</sub>
	...	...
Data segment		

Il linker aggiorna i campi indirizzi delle istruzioni

L'istruzione `jal` all'indirizzo `0x400004` viene aggiornata con **0x400100** (l'indirizzo ora noto della procedura `B`)

Le istruzioni MIPS sono allineate alla word (poste in memoria ad indirizzi multipli di 4, quindi i 2 bit meno significativi uguali a zero)

L'istruzione `jal` (come `beq/bne`) non codifica i 2 bit a destra per aumentare il range degli indirizzi esprimibili, usando quindi 26 b per esprimere un indirizzo di 28 b.

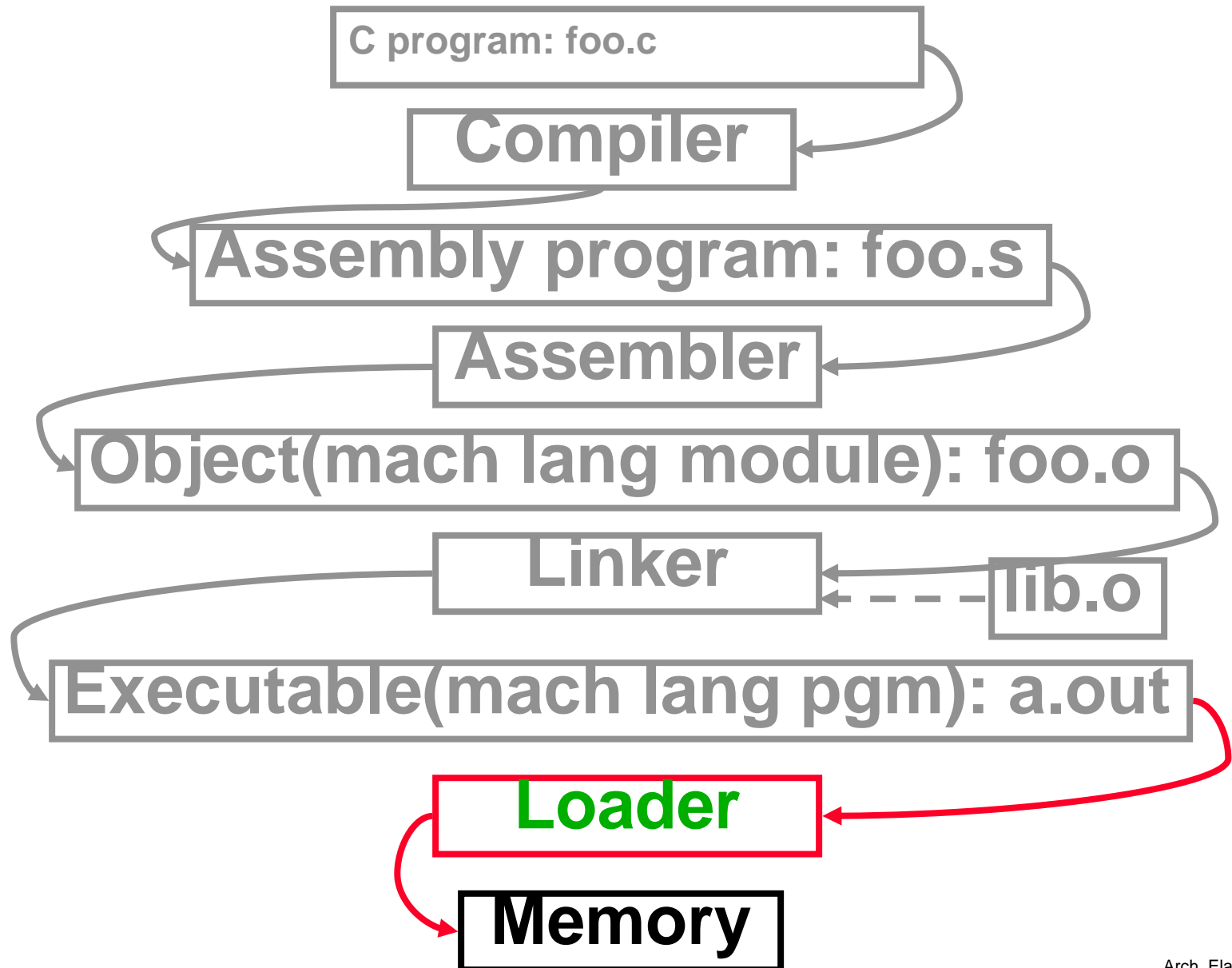
Quindi i valori finali inseriti nelle due `jal` saranno:

**0x100040** =  $0x400100 \gg 2$

**0x100000** =  $0x400000 \gg 2$

Agiamo analogamente per la `sw` all'indirizzo `0x400100`, dove il valore inserito nell'istruzione è `0x8020`

# Loader



# Loader

---

- Legge l'header del file per determinare la dimensione dei segmenti text e data
- Crea un nuovo *spazio di indirizzamento* per il programma, grande abbastanza per contenere i due segmenti, assieme allo stack
- Copia le istruzioni e i dati dal file eseguibile nel nuovo spazio di indirizzamento virtuale
- Copia i parametri del programma sullo stack.
- Inizializza i registri macchina (importanti `$sp` e `PC`).
- Salta ad una routine di start-up che chiama la routine `main()`
- Quando la routine `main()` ritorna, the routine chiamante di start-up porta a compimento il programma con una `exit system call`

# Dynamic Linked Libraries

---

- Approccio tradizionale: **STATIC LINKED LIBRARIES**
  - linking delle librerie prima dell'esecuzione dei programmi
  - NOTA: la traduzione corretta di **library** sarebbe **biblioteca** (di funzioni)
- **Vantaggi** delle SLL:
  - Chiamata veloce della funzione di libreria, con un salto diretto (`jal`)

# Dynamic Linked Libraries

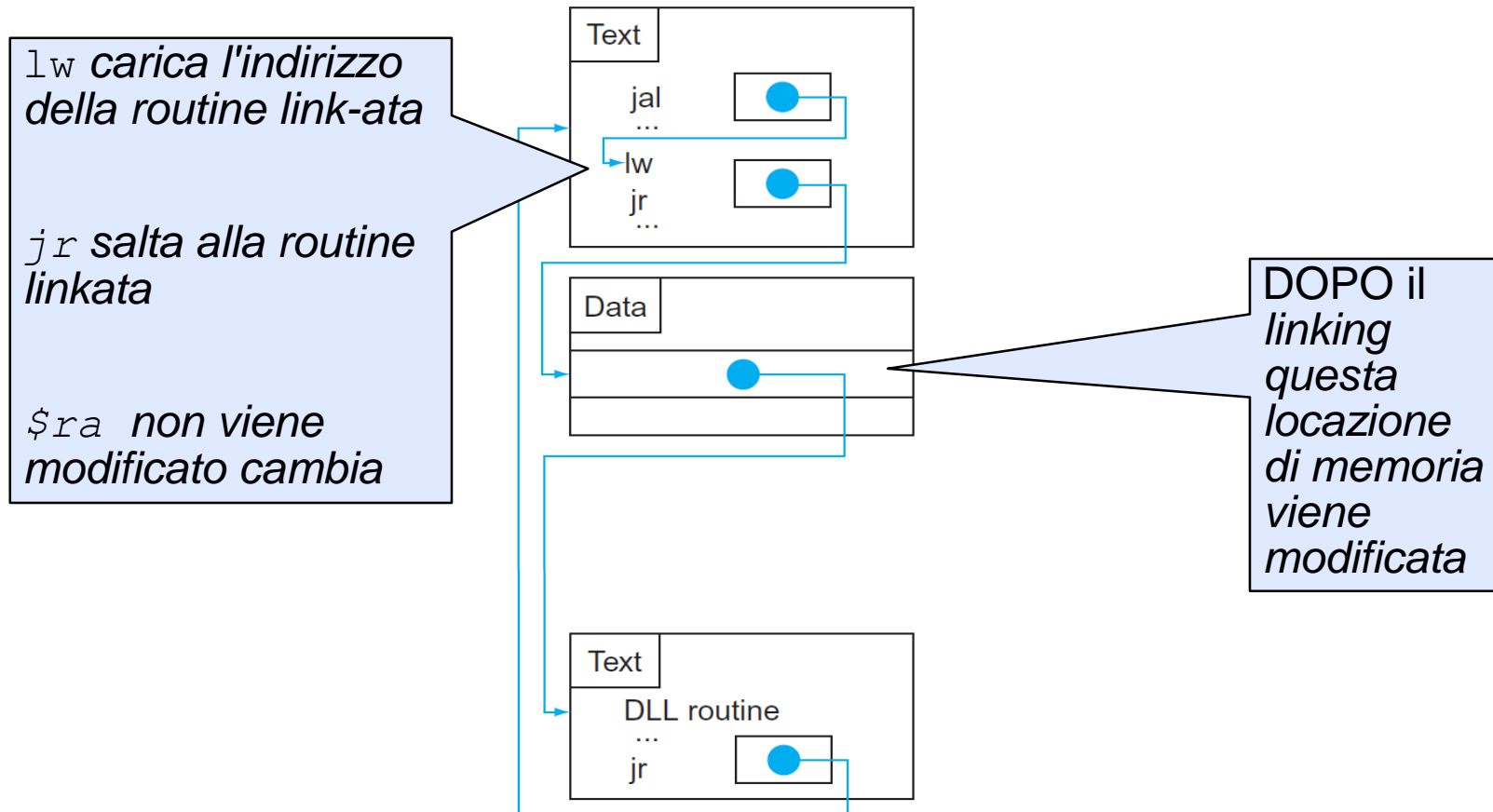
---

- **Svantaggi delle SLL:**
  - Codice della libreria è parte dell'eseguibile, assieme al codice dell'utente
  - Aggiornamenti della libreria necessitano la ripetizione del linking
  - Tutte le routine della libreria sono caricate (la standard C library è 2.5 MB), anche se solo alcune call sono realmente eseguite a run-time
  - Se più programmi usano la stessa libreria, questa non è condivisa ma replicata nei vari eseguibili
- Questi svantaggi hanno portato alle **DLL (dynamically linked libraries)**
  - Libreria link-ate e caricate in memoria a **run-time** e **on demand**.
  - Gli **shared objects** di Linux sono analoghe alle **DLL di Windows**

# Dynamic Linked Libraries

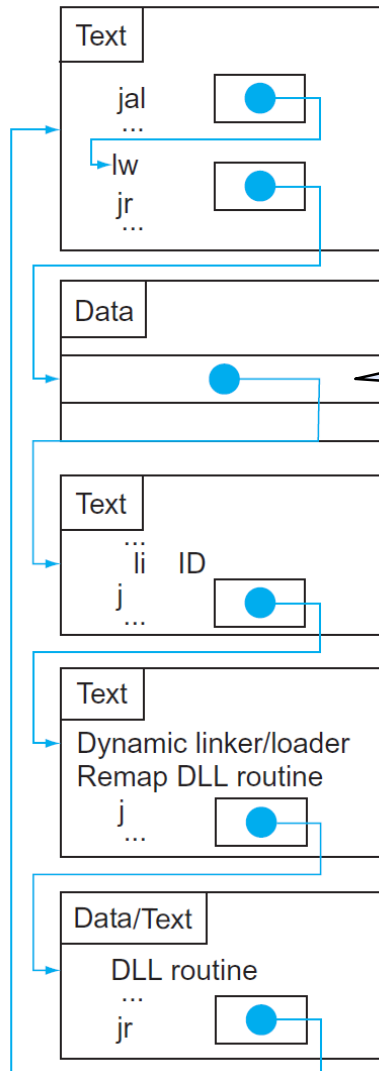
- Ogni routine è link-ata solo *dopo* che è stata chiamata la prima volta
- La chiamata alla funzione avviene con un *livello di indirezione*

## SCENARIO dopo il linking:



# Dynamic Linked Libraries

## SCENARIO alla prima chiamata della routine della DLL:



PRIMA del *linking* questa locazione di memoria punta al blocco dummy in basso

DOPO il *linking* questa locazione di memoria viene modificata (vedi slide precedente)

Il codice inserisce un codice (ID) in un registro prima di saltare al Dynamic linker/loader

L'ID serve ad identificare la routine della libreria

Il linker/loader usa strutture dati interne per trovare la funzione, e prima di saltare, cambia l'indirizzo usato dal salto indiretto per puntare direttamente alla routine della DLL



---

# SPIM e OS

# ... il Sistema Operativo

---

- I processori forniscono le istruzioni basilari (operazioni aritmetiche, salti, load, store...)
- Il computer, oltre al processore e alla memoria, include però altri accessori hardware (I/O) con cui è possibile interagire
  - Ad esempio, il video, la tastiera, la scheda video, il disco, etc...
- Per effettuare operazioni di I/O
  - per esempio, per scrivere caratteri sul video, per leggerli dalla tastiera, ecc.

servono particolari procedure (routine di gestione) che vanno a interagire con i controller dei dispositivi video e tastiera
- Tipicamente, tali operazioni sono molto complicate, soggette ad errori, con possibili conseguenze per gli altri utenti del sistema
- Il Sistema Operativo contiene il codice per queste operazioni !!!
  - Sotto forma di specifiche procedure, pre-caricate in memoria al momento dell'accensione

# ... il Sistema Operativo

---

- Possiamo invocare queste funzioni del Sistema Operativo
- Per invocare queste speciali funzioni, si usa si usa l'istruzione MIPS:
  - `syscall` (“system call”, chiamata di sistema)
  - L'invocazione provoca anche il passaggio dalla modalità di esecuzione *user* a quella *kernel*
- Tipicamente, siccome la `syscall` invoca una routine (anche se del Sistema Operativo), abbiamo bisogno di specifiche convenzioni di chiamata.
- **INPUT:**
  - `$v0` : codice della chiamata (cosa far fare al sistema operativo)
  - `$a0` : dato in input (se c'è)
  - `$a1` : dato in input (se c'è)
- **OUTPUT:**
  - `$v0`: dato in output (se c'è)

# SPIM

---

- Il simulatore SPIM del processore MIPS, oltre a simulare l'esecuzione di un programma utente, simula anche un I/O device e le routine di gestione
  - Un terminale memory-mapped sul quale i programmi possono leggere e scrivere caratteri
- Un programma MIPS in esecuzione su SPIM
  - può leggere i caratteri digitati sulla tastiera
  - stampare caratteri sul terminale
- Ma come facciamo, scrivendo un programma assembly in SPIM, a scrivere su video, o a leggere dalla tastiera?
  - ➔ SPIM ci fornisce un **mini sistema operativo** con delle funzioni di base
- “Mini” perché ci sono solo 10 possibili chiamate al sistema operativo
  - Di solito un sistema operativo ne ha molte di più...
  - Noi oggi ne vedremo 5 (codice 1, 4, 5, 8, 10)

# SPIM e stampa

- Ci sono istruzioni per stampare su video:
  - un intero
  - una stringa

- **Stampa intero**

- **INPUT:**

- **\$v0 = 1 (codice)**
    - **\$a0 = intero da stampare**

- **OUTPUT:**

**Per stampare “7” (in decimale) sul video:**

```
addi $v0,$zero,1 #codice 1
addi $a0,$zero,7 #output 7
syscall
```

- **Stampa stringa**

- **INPUT:**

- **\$v0 = 4 (codice)**
    - **\$a0 = indirizzo della stringa**

- **OUTPUT:**

**Per stampare “hello” sul video:**

```
str: .asciiz "hello
```

```
...
```

```
addi $v0,$zero,4 # codice 4
la $a0, str # output str
syscall
```

# SPIM e lettura

- **Ci sono istruzioni per leggere dalla tastiera:**

- un intero
- una stringa

- **Leggi intero (digitato in decimale)**

- **INPUT:**
  - $\$v0 = 5$  (codice)
- **OUTPUT:**
  - $\$v0$  = intero letto da tastiera

## Per leggere un intero:

```
addi $v0,$zero,5 #codice 5
syscall
< usa $v0 >
```

- **Leggi stringa**

- **INPUT:**
  - $\$v0 = 8$  (codice)
  - $\$a0$  = dove mettere la stringa (buffer)
  - $\$a1$  = lunghezza della stringa (capacità del buffer, incluso '\0')
- **OUTPUT:**
  - il buffer conterrà la stringa letta

## Per leggere una stringa:

```
str: .space 10
...
addi $v0,$zero,8 # codice 8
la $a0, str # input str
addi $a1,$zero,10 # len=10
syscall
< usa str >
```

# SPIM e exit

---

- Vediamo l'ultima `syscall`
  - INPUT:
    - `$v0 = 10` (codice)
  - OUTPUT:
- Significato: EXIT
  - Esci da SPIM e termina l'esecuzione

# Un esempio di funzione ricorsiva (con stampa)

---

- Calcolo e stampa del fattoriale di  $n$ :

$$n! = n * (n-1) * (n-2) * \dots * 1$$

```
int fact(n) {  
    int ret;  
    if (n == 1)  
        ret = 1;  
    else {  
        ret = fact(n-1);  
        ret = n * ret;  
    }  
    return ret;  
}
```

```
main() {  
    int ret;  
    ret = fact(5);  
    <stampa ret>  
}
```



# Un esempio di funzione ricorsiva (con stampa)

- Calcolo e stampa del fattoriale di  $n$ :

$$n! = n * (n-1) * (n-2) * \dots * 1$$

```
int fact(n)
{
    int ret;    # use $v0 for this variable
    if (n != 1) goto else_lab;
    ret = 1;
    goto exit_lab;
else_lab:
    ret = n * fact(n-1);
exit_lab:
    return(ret);
}
```

```
main()
{
    int ret;    # allocato sullo stack
    ret = fact(5);
    <stampa ret>
}
```

# Un esempio di funzione ricorsiva

.text

fact:

```
subu    $sp, $sp, 8      # frame di 8 byte
sw      $ra, 0($sp)      # salva $ra (callee-save)
sw      $s0, 4($sp)      # salva $s0 (callee-save)
move    $s0, $a0         # usa $s0 per memorizzare n
bne     $s0, 1, else     # if (n != 1) goto else
li      $v0, 1           # caso finale della ricorsione
j       exit
```

else:

```
addu    $a0, $s0, -1
jal     fact
mult    $s0, $v0         # risultato in hi e lo
mflo    $v0              # copia in $v0 la parte bassa
                        # del risultato
```

# Un esempio di funzione ricorsiva

---

exit:

```
lw $ra, 0($sp)      # ripristina $ra (callee-save)
lw $s0, 4($sp)      # ripristina $s0 (callee-save)
addu $sp, $sp, 8
jr $ra
```

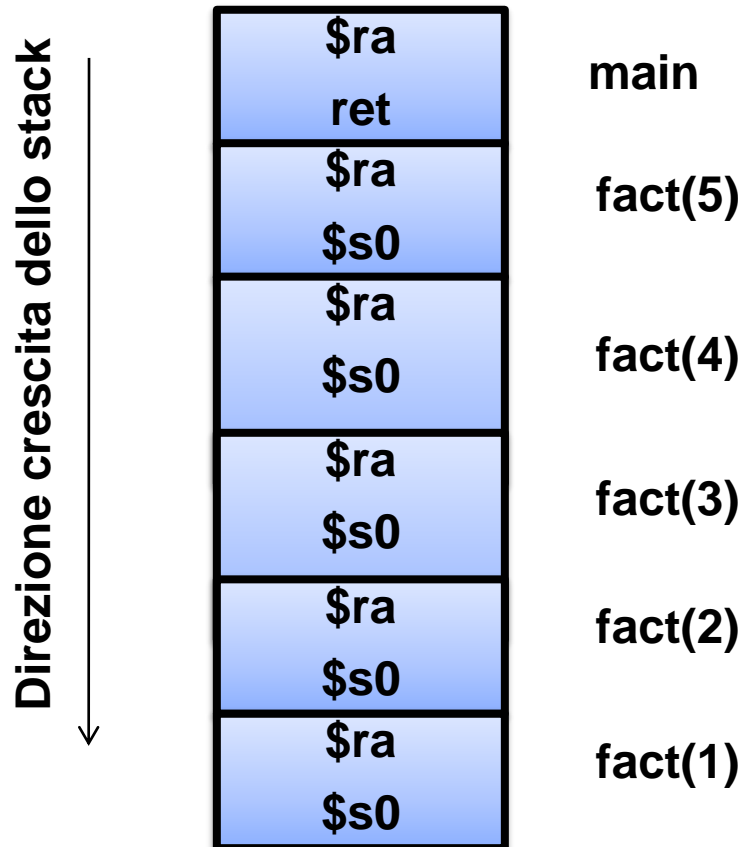
main:

```
subu $sp, $sp, 8
sw $ra, 0($sp)      # salva $ra (callee-save)
li $a0, 5           # invoca fact(5)
jal fact
sw $v0, 4($sp)      # memorizza il ritorno su ret (var.
                   # locale allocata sullo stack)
li $v0, 1           # stampa intero
lw $a0, 4($sp)
syscall

lw $ra, 0($sp)      # ripristina $ra
addu $sp, $sp, 8
jr $ra
```

# Un esempio di funzione ricorsiva

- E lo stack?



# QtSpim

Mappa dei registri

Stack segment  
Data segment

QtSpim

FP Regs [16] [16]

Data Text

PC = 4000  
EPC = 0  
Cause = 0  
BadVAddr = 0  
Status = 3000fff10

HI = 0  
LO = 0

R0 [r0] = 0  
R1 [at] = 0  
R2 [v0] = 8  
R3 [v1] = 0  
R4 [a0] = 5  
R5 [a1] = 7ffffd8c  
R6 [a2] = 7ffffd98  
R7 [a3] = 0  
R8 [t0] = 0  
R9 [t1] = 0  
R10 [t2] = 0  
R11 [t3] = 0  
R12 [t4] = 0  
R13 [t5] = 0  
R14 [t6] = 0  
R15 [t7] = 0  
R16 [s0] = 0  
R17 [s1] = 0  
R18 [s2] = 0  
R19 [s3] = 0  
R20 [s4] = 0  
R21 [s5] = 0  
R22 [s6] = 0  
R23 [s7] = 0  
R24 [s8] = 0  
R25 [s9] = 0  
R26 [k0] = 0  
R27 [k1] = 0  
R28 [gp] = 10008000  
R29 [sp] = 7ffffd78  
R30 [s8] = 0  
R31 [ra] = 400074

User Text Segment [00400000]..[004400000]

```
00400000: 8fa40000 lw $4, 0($29) ; 183: lw $a0 0($sp) # argc
00400004: 27a50004 addiu $5, $29, 4 ; 184: addiu $a1 $sp 4 # argv
00400008: 24a60004 addiu $6, $5, 4 ; 185: addiu $a2 $a1 4 # envp
0040000c: 00041080 sll $2, $4, 2 ; 186: sll $v0 $a0 2
00400010: 00c23021 addu $6, $6, $2 ; 187: addu $a2 $a2 $v0
00400014: 0c100019 jal 0x00400064 [main] ; 188: jal main
00400018: 00000000 nop ; 189: nop
0040001c: 3402000a ori $2, $0, 10 ; 191: li $v0 10
00400020: 0000000c syscall ; 192: syscall # syscall 10 (exit)
00400024: 27bdfff8 addiu $29, $29, -8 ; 5: subu $sp, $sp, 8 # frame di 8 byte
00400028: afbf0000 sw $31, 0($29) ; 6: sw $ra, 0($sp) # salva $ra (callee-save)
0040002c: afb60004 sw $16, 4($29) ; 7: sw $s0, 4($sp) # salva $s0 (callee-save)
00400030: 00048021 addu $16, $0, $4 ; 8: move $s0, $a0
00400034: 34010001 ori $1, $0, 1 ; 9: bne $s0, 1, else # if (n != 1) goto else
00400038: 14300003 bne $1, $16, 12 [else-0x00400038] ; 10: li $v0, 1 # caso finale della ricors.
0040003c: 34020001 ori $2, $0, 1 ; 11: j exit
00400040: 08100015 j 0x00400054 [exit] ; 13: addu $a0, $s0, -1
00400044: 2604ffff addiu $4, $16, -1 ; 14: jal fact
00400048: 0c100009 jal 0x00400024 [fact] ; 15: mult $s0, $v0 # risultato in hi e lo
0040004c: 02020018 mult $16, $2 ; 16: mflo $v0 # copia in $v0 la parte bassa
00400050: 00001012 mflo $2 ; 20: lw $ra, 0($sp) # ripristina $ra (callee-save)
00400054: 8fbf0000 lw $31, 0($29) ; 21: lw $s0, 4($sp) # ripristina $s0 (callee-save)
00400058: 8fb60004 lw $16, 4($29) ; 22: addu $sp, $sp, 8
0040005c: 27bd0008 addiu $29, $29, 8 ; 23: jr $ra
00400060: 03e00008 jr $31 ; 26: subu $sp, $sp, 8
00400064: 27bdfff8 addiu $29, $29, -8 ; 27: sw $ra, 0($sp) # salva $ra
00400068: afbf0000 sw $31, 0($29) ; 28: li $a0, 5 # invoca fact(5)
0040006c: 34040005 ori $4, $0, 5 ; 29: jal fact
00400070: 0c100009 jal 0x00400024 [fact] ; 30: sw $v0, 4($sp) # memorizza il ritorno su ret (var.
00400074: afa20004 sw $2, 4($29) ; 32: li $v0, 1 # stampa intero
00400078: 34020001 ori $2, $0, 1 ; 33: lw $a0, 4($sp)
0040007c: 8fa40004 lw $4, 4($29) ; 34: syscall
00400080: 0000000c syscall ; 36: lw $ra, 0($sp) # ripristina $ra
00400084: 8fbf0000 lw $31, 0($29) ; 37: addu $sp, $sp, 8
00400088: 27bd0008 addiu $29, $29, 8 ; 38: jr $ra
0040008c: 03e00008 jr $31
```

Kernel Text Segment [80000000]..[80010000]

```
80000180: 0001d821 addu $27, $0, $1 ; 90: move $k1 $at # Save $at
80000184: 3c019000 lui $1, -28672 ; 92: sw $v0 $1 # Not re-entrant and we can't trust $sp
80000188: ac220200 sw $2, 512($1) ; 93: sw $a0 $2 # But we need to use these registers
8000018c: 3c019000 lui $1, -28672 ; 95: mfc0 $k0 $13 # Cause register
80000190: ac240204 sw $4, 516($1) ; 96: srl $a0 $k0 2 # Extract ExCoDe Field
80000194: 401a6800 mfc0 $26, $13 ; 97: andi $a0 $a0 0x1f
80000198: 001a2082 srl $4, $26, 2 ; 101: li $v0 4 # syscall 4 (print_str)
8000019c: 3084001f andi $4, $4, 31 ; 102: la $a0 __ml_
800001a0: 34020004 ori $2, $0, 4 ; 103: syscall
800001a4: 3c049000 lui $4, -28672 [__ml_] ; 105: li $v0 1 # syscall 1 (print_int)
800001a8: 0000000c syscall ; 106: srl $a0 $k0 2 # Extract ExCoDe Field
800001ac: 34020001 ori $2, $0, 1 ; 107: andi $a0 $a0 0x1f
800001b0: 001a2082 srl $4, $26, 2 ; 108: syscall
800001b4: 3084001f andi $4, $4, 31 ; 110: li $v0 4 # syscall 4 (print_str)
800001b8: 0000000c syscall ; 111: andi $a0 $k0 0x3c
800001bc: 34020004 ori $2, $0, 4 ; 112: lw $a0 __excp($a0)
800001c0: 3344003c andi $4, $4, $26, 60
800001c4: 3c019000 lui $1, -28672
800001c8: 00240821 addu $1, $1, $4
800001cc: 8c240180 lw $4, 384($1)
```

SPIM version 9.1.7 of February 12, 2012  
Copyright 1990-2012, James R. Larus.  
All Rights Reserved.  
SPIM is distributed under a BSD license.  
See the file README for a full copyright notice.

Single Step

Text segment

1. [...] Ind. Memoria
2. Stringa esadecimale dell'istruzione macchina
3. Stringa mnemonica dell'istruzione macchina
4. Istruzioni assembly originali (nota bne \$s0,1,else interpretata come pseudo istruzione)

# QtSpim

1. Caricamento (e assembling) del programma (solitamente, file con estensione .asm)
2. Run
3. Degugging
  - Run step by step
  - Setting del breakpoint + Run

SPIM version 9.1.7 of February 12, 2012  
Copyright 1990-2012, James R. Larus.  
All Rights Reserved.  
SPIM is distributed under a BSD license.  
See the file README for a full copyright notice.

Single Step

---

# Esercizi

# Esercizio 1

---

- **Si traduca in assembler MIPS (con le convenzioni di chiamata solite) la seguente funzione C, commentandone prima il funzionamento. Si suggerisce la modifica di registri temporanei per evitare di salvarli sullo stack.**

```
int find_substr(char str[], char first, char second)
{
    int i=0;
    while (str[i] != 0)
    {
        if ((first==str[i]) && (second==str[i+1]))
            return(i);
        else
            i++;
    }
    return(-1);
}
```



# Esercizio 1

---

- La funzione `find_substr()` ricerca all'interno della stringa `str` la sottostringa composta dai caratteri `first` e `second`. Se la ricerca ha successo, restituisce l'indice della sottostringa, altrimenti restituisce -1
- Vediamo come tradurre il corpo della procedura:

```
init_while:
    if (str[i] == 0) goto exit_while;
    if ((first != str[i]) || (second != str[i+1])) goto else;
    <set $v0=i>
    goto return;
else:
    i++;
    goto init_while
exit_while:
    <set $v0=-1>
return:
    <return with jr $ra>
```

# Esercizio 1

---

```
find_substr:
    ori  $t0, $0, 0          # i=0
init_while:
    add  $t1, $a0, $t0       # $t1 = str+i
    lbu  $t2, 0($t1)         # $t2 = str[i]
    beq  $t2, $0, exit_while # if (str[i] == 0) goto exit_while

    bne  $t2, $a1, else      # if ($t2 != first) goto else
    lbu  $t2, 1($t1)         # $t2 = str[i+1]
    bne  $t2, $a2, else      # if ($t2 != second) goto else
    ori  $v0, $t0, 0         # return i
    j    return
else:
    addi $t0, $t0, 1         # i++
    j    init_while
exit_while:
    li   $v0, -1             # return -1
return:
    jr   $ra
```

# Esercizio 1

- **Per invocare** find\_substr:

```
main:
    subu    $sp, $sp, 4
    sw      $ra, 0($sp)    # salva $ra
    la      $a0, str
    ori     $a1, $zero, 'a'
    ori     $a2, $zero, 'a'
    jal     find_substr    # cerca la sottostringa "aa"
    move     $t0, $v0      # salva il ritorno
    li      $v0, 4        # stampa stringa
    la      $a0, echo
    syscall
    li      $v0, 1        # stampa intero
    move     $a0, $t0
    syscall
    lw      $ra, 0($sp)    # ripristina $ra
    addu     $sp, $sp, 4
    jr      $ra

.data
str: .asciiz "bbaaccff"
echo: .asciiz "il ritorno e' "
```

## Esercizio 2

---

- **Si traduca in assembler MIPS (con le convenzioni di chiamata solite) la seguente funzione C, commentandone prima il funzionamento. Si suggerisce la modifica di registri temporanei per evitare di salvarli sullo stack.**

```
char s[5] = "roma";
```

```
void swap(char *c1, char *c2) {  
    char tmp;  
    tmp = *c1;  
    *c1 = *c2;  
    *c2 = tmp;  
}
```

```
int main() {  
    swap(&s[0], &s[3]);  
    swap(&s[1], &s[2]);  
    s[0]=s[0]-32;  
    <stampa stringa s>  
}
```

## Esercizio 2

- Il programma costruisce la stringa palindrome di `s[]` e trasforma in maiuscolo il primo carattere.

```
.data
```

```
s:  .ascii "roma"
```

```
.text
```

```
swap:
```

```
    lbu    $t0, 0($a0)    # tmp = *c1
    lbu    $t1, 0($a1)    # metti in $t1 il valore *c2
    sb     $t1, 0($a0)    # *c1 = *c2   (dove *c2 sta in $t1)
    sb     $t0, 0($a1)    # *c2 = tmp
    jr     $ra
```

```
main:
```

```
    subu   $sp, $sp, 4
    sw     $ra, 0($sp)    # salva $ra
    la     $t0, s
    addi   $a0, $t0, 0
    addi   $a1, $t0, 3
    jal    swap
```

## Esercizio 2

---

```
la    $t0, s
addi  $a0, $t0, 1
addi  $a1, $t0, 2
jal   swap

lbu   $t0, s($zero)
addi  $t0, $t0, -32
sb    $t0, s($zero)    # s[0] = s[0] - 32

li    $v0, 4
la    $a0, s
syscall                                # stampa s[]

lw    $ra, 0($sp)        # ripristina $ra
addu  $sp, $sp, 4
jr    $ra
```

## Esercizio 3

- Il programma C seguente accede ad un array bidimensionale, *allocato in memoria per righe* (la riga  $n$ -esima segue in memoria la riga  $n-1$ -esima, mentre gli elementi di ogni riga sono posti in locazioni contigue)

```
int a[5][2] = { {1, 2},
                {3, 4},
                {5, 6},
                {7, 8},
                {9, 10} };

int somma_righe[5] = {0, 0, 0, 0, 0};
int somma_colonne[2] = {0, 0};

char spazio[] = " ";
char newl[] = "\n";

void stampa_vett(int v[], int dim) {
    int i;
    for (i=0; i<dim; i++) {
        <stampa v[i]>
        <stampa la stringa spazio>
    }
    <stampa la stringa newl>
}
```

**Per accedere all'elemento  $(i, j)$  dell'array, ovvero per tradurre**

`var = a[i][j];`

**in assembly faccio qualcosa del genere:**

```
int displ, var;
int num_col = 2;

. . .
displ = i*num_col + j;
var = *(&a[0][0] + displ);
```

**In assembly bisogna fare però attenzione al `sizeof` del tipo di dato elementare, che in questo caso è `int` (4 Byte)**

- il `displ` deve essere moltiplicato per il `sizeof` (4 in questo caso)

# Esercizio 3

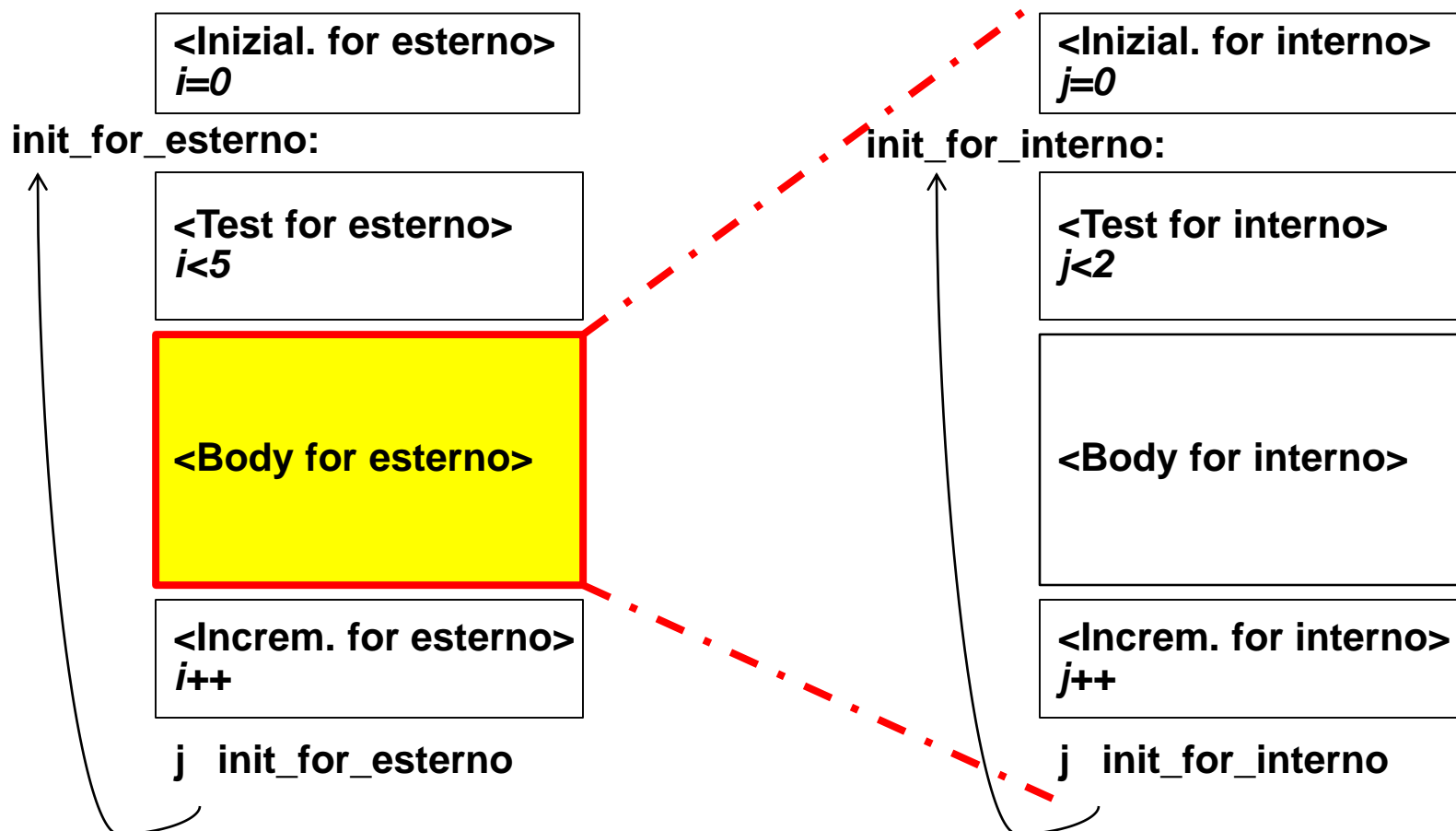
---

```
main() {  
    int i,j;  
    for (i=0; i<5; i++)  
        for (j=0; j<2; j++) {  
            somma_righe[i] = somma_righe[i] + a[i][j];  
            somma_colonne[j] = somma_colonne[j] + a[i][j];  
        }  
  
    stampa_vett(somma_righe, 5);  
    stampa_vett(somma_colonne, 2);  
}
```



# Esercizio 3

- Il problema è come tradurre i due **for annidati** del main
  - basta iniziare a tradurre dal **for più esterno** con il noto schema di traduzione, e iterare l'applicazione dello stesso schema al **for più interno**



# Esercizio 3

```
.data
a:                .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
somma_righe:      .word 0, 0, 0, 0, 0
somma_colonne:    .word 0, 0
bianco:           .ascii " "
newline:          .ascii "\n"
```

```
.text
#
# stampa_vett$a0
# &v[0] in $a0, dim in $a1
#
stampa_vett:
    ori $t0, $a0, 0          # salva $a0 in $t0
    ori $t1, $zero, 0        # i=0
initfor:
    bge $t1, $a1, exitfor    # (i < dim) ?

    sll $t2, $t1, 2          # $t2 = i*4
    add $t3, $t0, $t2        # $t3 = &v[i]
    lw  $a0, 0($t3)          # $a0 = v[i]
    ori $v0, $zero, 1
    syscall                  # stampa v[i]
```

# Esercizio 3

```
ori $v0, $zero, 4
la  $a0, bianco
syscall                # stampa " "

addi $t1, $t1, 1
j  initfor
exitfor:
ori $v0, $zero, 4
la  $a0, newline
syscall                # stampa "\n"
jr $ra

#
# main
#
main:
    addiu $sp, $sp, -4
    sw $ra, 0($sp)

    ori $t0, $zero, 0                # i=0
init_for_esterno:
    bge $t0, 5, exit_for_esterno    # (i<5) ?
```

# Esercizio 3

```
ori $t1, $zero, 0                # j=0
init_for_interno:
bge $t1, 2, exit_for_interno     # (j<2) ?

sll $t2, $t0, 1                  # $t2 = i*2
add $t2, $t2, $t1                # $t2 = i*2 + j
sll $t2, $t2, 2                  # $t2 = (i*2 + j) * 4 (tiene conto
                                # della dim. dei dati)
lw $t3, a($t2)                  # $t3 = a[i][j]

sll $t4, $t0, 2                  # $t4 = i*4
sll $t5, $t1, 2                  # $t5 = j*4
lw $t6, somma_righe($t4)        # $t6 = somma_righe[i]
lw $t7, somma_colonne($t5)      # $t7 = somma_colonne[j]

add $t6, $t6, $t3
sw $t6, somma_righe($t4)

add $t7, $t7, $t3
sw $t7, somma_colonne($t5)

addi $t1, $t1, 1                # j++
j init_for_interno
```

# Esercizio 3

```
exit_for_interno:
    addi $t0, $t0, 1          # i++
    j init_for_esterno
exit_for_esterno:
    la $a0, somma_righe
    ori $a1, $zero, 5
    jal stampa_vett

    la $a0, somma_colonne
    ori $a1, $zero, 2
    jal stampa_vett

    lw $ra, 0($sp)
    addiu $sp, $sp, 4
    jr $ra
```

# Domande

---

- Perché `$ra` (`$31`) è un registro callee save?
- Dato una funzione, sarebbe una soluzione praticabile associare ad essa un'area di memoria statica in cui allocare le variabili locali, passare i parametri, salvare i registri?
- In quali casi una funzione non necessita di salvare alcun registro sullo stack?
- In quali casi l'istruzione `lw $5, costante($4)` non può essere tradotta con una semplice macchina, e quindi diventa una pseudo-istruzione?
- Per invocare una funzione è sufficiente l'istruzione `j` (Jump)? Commentare.
- Qual è il rationale della convenzione di chiamata che impone di salvare i registri `$s0`, `$s1`, ecc. in all'approccio callee-save?
- Qual è il rationale della convenzione di chiamata che impone di salvare i registri `$t0`, `$t1`, ecc. in accordo all'approccio caller-save?