
**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
- **Programmazione di I/O**
- **Uso del simulatore SPIM**
- **Esercitazioni**

Linguaggio macchina / assembler

- È noioso **Ehm, si.**
- 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 meglio nel linguaggio di più alto livello (ad esempio, C), perché si capisce come poi viene eseguito
 - 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

Valore Pedagogico e Intrinseco

- **Imparare il linguaggio macchina / assembly vi aiuterà a capire**
 - le ripercussioni delle scelte di architettura del sistema.
 - il funzionamento e l'implementazione dei servizi offerti dai linguaggi ad alto livello.
 - quali sono le ottimizzazioni che può fare il compilatore
 - e quelle che potete fare voi anche in linguaggi ad alto livello, e quindi migliorare l'efficienza del codice che scriverete anche con linguaggi ad alto livello.
- **Anche se non scriverete mai un intero programma in linguaggio assembly, vi capiterà di dover accedere a caratteristiche di livello basso della macchina.**
 - **Approccio ibrido: potete usare il linguaggio macchina esattamente nei punti critici dove serve, e usare il vostro linguaggio di alto livello preferito nel resto del progetto**
 - **Questo avviene spesso nella programmazione di sistemi embedded, per il controllo dei processi con vincoli real-time**

Linguaggi di programmazione

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

MA. . .

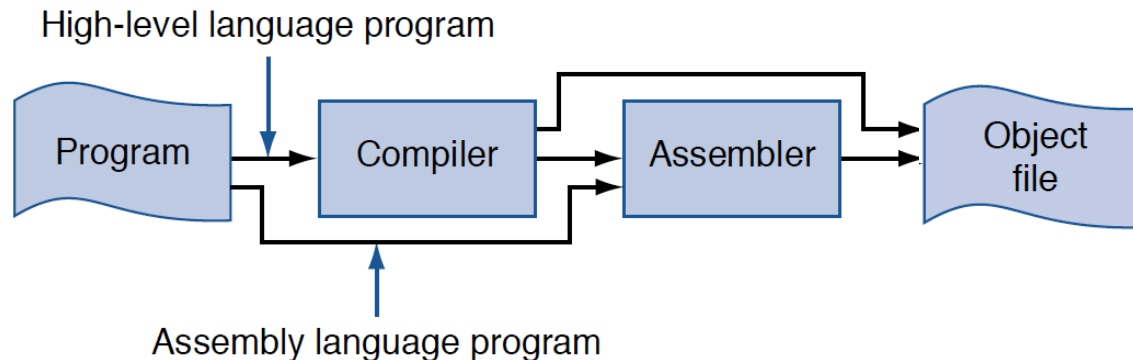
- Alla fine, l'unico linguaggio che il computer capisce è il linguaggio macchina
- Per usare altri linguaggi, occorre usare qualcosa in grado di “far capire” al computer gli altri linguaggi

Interprete

- Un **INTERPRETE** per un certo **linguaggio L** è un programma che “interpreta” appunto il linguaggio L e fa sì che questo venga correttamente eseguito dal computer
- Ogni istruzione del programma “viene interpretate” ogni volta che viene eseguita
- **ECCESSIVA LENTEZZA DI ESECUZIONE**
- Per eseguire un programma, ho sempre bisogno dell'interprete

Compilatore

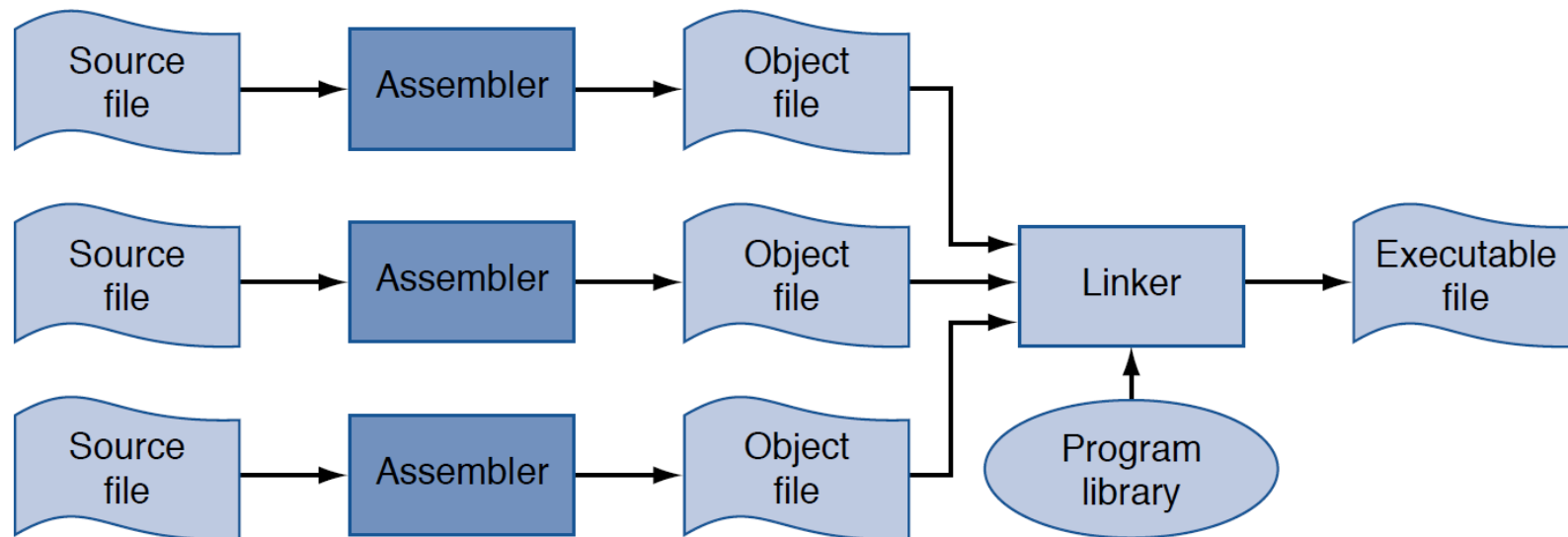
- Il **COMPILATORE**, invece, è un programma che prende in input un programma in un **linguaggio L** e lo traduce in un programma in un **altro linguaggio**
 - Il SOURCE LANGUAGE (“sorgente”) viene tradotto nel TARGET LANGUAGE (“obiettivo”)
- Tipicamente il TARGET LANGUAGE è il LINGUAGGIO MACCHINA, anche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



- **Java: Linguaggi a BYTE-CODE**
 - il linguaggio viene compilato in un LINGUAGGIO INTERMEDIO che poi viene INTERPRETATO

Il processo di compilazione/esecuzione

- Il passaggio dal codice sorgente alla esecuzione del programma passa per 3 fasi.
 1. Compilazione / Assembling
 2. Linking
 3. Caricamento ed Esecuzione



Il processo di compilazione/esecuzione

- Il passaggio dal codice sorgente alla esecuzione del programma passa per 3 fasi.
 1. **Compilazione / Assembling**
 2. Linking
 3. Caricamento ed Esecuzione
- **Compilazione / Assembling**
 - Il programma viene analizzato e per ogni istruzione viene generata una porzione di linguaggio macchina che la implementa
 - Anche le istruzioni che riguardano dichiarazioni/allocazioni dei dati sono tradotte in modo opportuno
 - L'output è un *file oggetto*
 - un file tradotto *quasi completamente*
 - vengono mantenuti i simboli usati nel codice
 - Es.: etichette mnemoniche associate ai dati, etichette mnemoniche associate alla prima istruzione di una funzione/procedure, ecc.
 - formato dipendente dal sistema

Il processo di compilazione/esecuzione

- Il passaggio dal codice sorgente alla esecuzione del programma passa per 3 fasi.
 1. Compilazione / Assembling
 2. Linking
 3. Caricamento ed Esecuzione
- Linking
 - I file oggetto (tradotti in linguaggio macchina) sono collegati assieme, risolvendo i riferimenti ai simboli esterni usati nei vari file oggetto.
 - Tra i simboli esterni impiegati, abbiamo le invocazioni delle funzioni di libreria (es. printf())
 - Il risultato è un *file eseguibile*:
 - un file in cui oltre al codice c'è informazione riguardo alla posizione in memoria in cui va caricato il programma, nonché eventuali simboli ancora non “risolti”

Il processo di compilazione/esecuzione

- Il passaggio dal codice sorgente alla esecuzione del programma passa per 3 fasi.
 1. Compilazione / Assembling
 2. Linking
 3. Caricamento ed Esecuzione
- **Loader**
 - *è un componente del sistema operativo*
 - carica il programma in memoria (gerarchia) e poi passa il controllo della CPU alla prima istruzione del programma
 - in sistemi con con librerie dinamiche (DDL), invoca il linker dinamico per risolvere i simboli mancanti o altri meccanismi più complessi
- **Esecuzione**
 - C'è una locazione di memoria speciale, memorizzato nel registro PC della CPU, che contiene un indirizzo di memoria "speciale":
 - Prossima istruzione in linguaggio macchina da eseguire
 - Ciclo Fetch-Decode-Execution della CPU

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

```
00100111101111011111111111100000
10101111101111111000000000010100
1010111110100100000000000100000
1010111110100101000000000100100
101011111010000000000000011000
101011111010000000000000011100
100011111010111000000000011100
100011111011100000000000011000
000000011100111000000000011001
001001011100100000000000000001
0010100100000001000000001100101
101011111010100000000000011100
00000000000000000111100000010010
00000011000011111100100000100001
0001010000100000111111111110111
1010111110111001000000000011000
00111100000001000001000000000000
1000111110100101000000000011000
0000110000010000000000011101100
00100100100001000000010000110000
1000111110111111000000000010100
0010011110111101000000000100000
000000111110000000000000001000
00000000000000000001000000100001
```

- La CPU può *interpretare* certi numeri come corrispondenti a 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 però 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 per le varie locazioni di memoria
- I comandi che iniziano con i punti sono direttive assembler
 - **.text** **o** **.data**
 - Indica che le linee successive contengono istruzioni o dati
 - **.align n**
 - Indica l'allineamento su 2ⁿ
 - **.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 ad alto livello (C)?**

```
#include <stdio.h>

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

    for (i = 0; i <= 100; i = i + 1) 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)
 - Ogni registro è grande 32 bits, quindi il risultato è di 64 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 destinatario!!
 - 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

- In linguaggio macchina si possono effettuare anche manipolazioni dei dati a livello di bit
 - Spostamenti (shifts), AND, OR, XOR, NOR
- Interpretazione funzionale delle operazioni logiche
 - Tramite AND, OR e XOR possiamo modificare un bit a seconda se il secondo operando è 0 o 1
- AND
 - $\text{Dest_bit} = \text{Source_bit} \text{ AND } 0 \rightarrow$ **Spegni** il Source_bit
 - $\text{Dest_bit} = \text{Source_bit} \text{ AND } 1 \rightarrow$ **Copia** il Source_bit
 - In ASSEMBLY l'istruzione **and** agisce su words, il che significa che possiamo agire contemporaneamente sui 32 bits 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 siano 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
 - $\text{Dest_bit} = \text{Source_bit} \text{ OR } 0 \rightarrow \text{Copia il Source_bit}$
 - $\text{Dest_bit} = \text{Source_bit} \text{ OR } 1 \rightarrow \text{Accendi il Source_bit}$
 - 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
 - $A = \text{INVERT}(\text{INVERT}(A))$

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 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 ($s_b, 1_b$), 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 condizionale (branch)
 - salto assoluto (jump)

Istruzioni MIPS per modificare il flusso di controllo

- I salti condizionali 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 condizionali

`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 `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 `la`
 - 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`

- **2. Movimento di dati:**

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

- **3. 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 e memoria interna/esterna**
 - **Interna alla CPU: registri**
 - **Esterna alla CPU: RAM**

⇒ dovremo tradurre i costrutti attraverso particolare 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**
 - **In linguaggi ad alto livello come il C, ci sono diversi tipi di dato:**
 - 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 abbastanza trasparente all’utente in linguaggi di alto livello in linguaggio macchina ogni cosa deve essere esplicitata**
 - **bisogna conoscere la taglia di ogni tipo di dato, e avere abbastanza celle di memoria per ogni variabile di quel tipo**

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:

<code>lw \$t0, 104</code>	<code># carica ''b'' in t0</code>
<code>lw \$t1, 108</code>	<code># carica ''c'' in t1</code>
<code>add \$t2,\$t0,\$t1</code>	<code># mette la somma in t2</code>
<code>sw \$t2, 100</code>	<code># mette t2 in ''a''</code>

Semplice compilazione di un'espressione C

- In C: `a = 24;`
- Usiamo un 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 come 20 variabili `int` consecutive:
 `A[0] ⇒ 200, . . . , 203` (`A[i] ⇒ indirizzi al byte in cui è mappato`)
 `A[1] ⇒ 204, . . . , 207`
 . . .
- L'indirizzo in memoria di `A[x]` è allora
 - `200 + 4*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 con una LOAD e ogni scrittura con una STORE
- 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:

$a \Rightarrow \$s0$

$b \Rightarrow 200, \dots, 203$

$c \Rightarrow 204, \dots, 207$

- 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 valori mantenuti nei registri (veloci)
 - minimizzare gli accessi alla memoria esterna (lenta)

Etichette

- **Nelle istruzioni in linguaggio macchina si usano gli indirizzi**
 - Per un programmatore assembly non è conveniente dovere calcolare/ricordarsi esplicitamente l'indirizzo
 - si usano etichette mnemoniche (labels), che si associano a specifici punti del programma (istruzioni o dati)
 - l'assembly 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.)

- **E allora, potete tradurre 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

- **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
```

- **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 sequanza 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 condizionali: `beq`, `bne`
 - salti incondizionati: `j`
- . . . questi equivalgono in C a:
 - `if (condizione) goto label;`
 - `goto label;`
- Usando i *goto*, 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 rispetto a $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
    bne $t0, $zero, init-while
exit-while:
    ...
```

- Viene eseguito solo un salto condizionale per ciclo, invece di un salto condizionale 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 dei pezzi 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)

Procedures 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, B poi terminerà 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**

Passaggio di parametri e ritorno dei risultati (cont.)

- **Esempio:**

- A chiama B, e si mettono d'accordo per avere l'input in \$t3, e restituire l'output in \$t4
- B processa il dato fornitogli in \$t3 (lo moltiplica per 4)
- Infine mette il risultato in \$t4

B:

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

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

```
jr $ra
```

- **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

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 raddoppiato:

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**

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 di output
- Quando si chiama una procedura con side-effects, abbiamo due alternative
 1. Il chiamante si assicura che nei registri NON ci siano dati da utilizzare dopo la chiamata della procedura, la quale potrebbe infatti modificarli come *side effect*
 2. Qualcuno salva i registri importanti per il prosieguo della computazione in un'area di memoria memoria (store), si esegue la procedura, e poi si ripristinano dalla memoria i registri importanti (load)

Quindi?

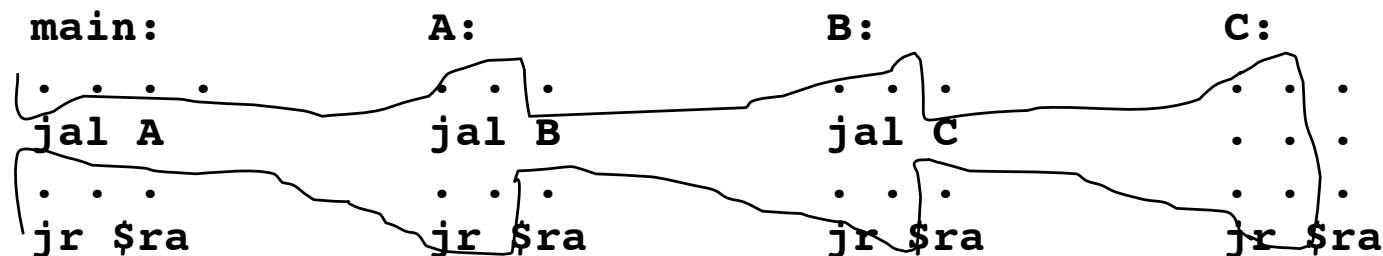
- Un possibili approccio potrebbe essere quello di documentare chiaramente:
 - Dov'è l'input e dov'è l'output
 - **Ma anche se ci sono altri registri modificati**

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

- Comunque 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

Registri MIPS e convenzioni d'uso

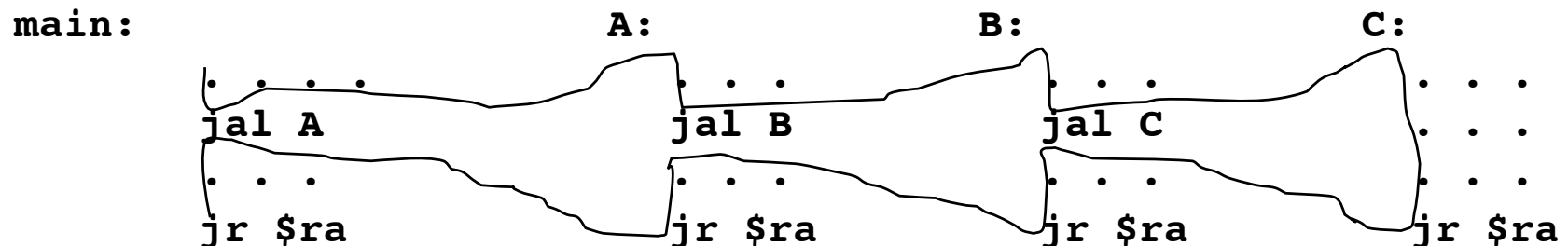
Register name	Number	Usage
\$zero	0	constant 0
\$at	1	reserved for assembler
\$v0	2	expression evaluation and results of a function
\$v1	3	expression evaluation and results of a function
\$a0	4	argument 1
\$a1	5	argument 2
\$a2	6	argument 3
\$a3	7	argument 4
\$t0	8	temporary (not preserved across call)
\$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 salvaguardati?
- **Soluzione che non funziona** in generale:
 - Ogni procedura ha uno spazio di memoria statico per salvare i registri
 - Ma se la procedura è ricorsiva, abbiamo bisogno di un numero arbitrario di questi spazi di memoria
- **Soluzione generale:**
 - 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**

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
```

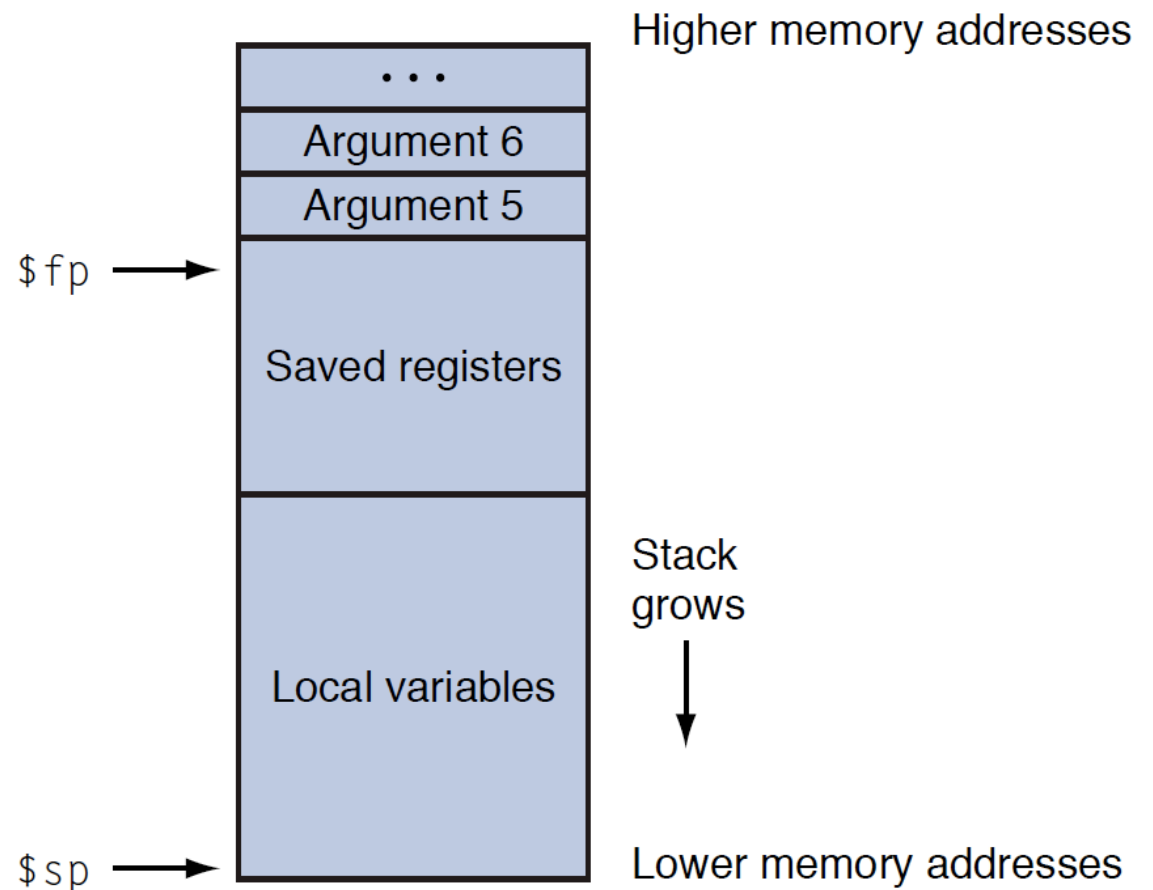
- **POP** toglie l'elemento in cima allo stack (e lo mette da qualche parte)

```
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)

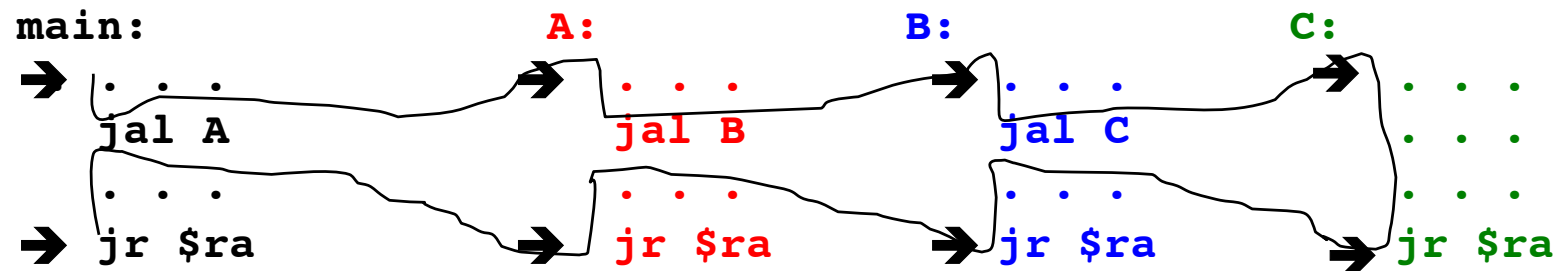
Stack frame

- 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, . . . , \$a3)

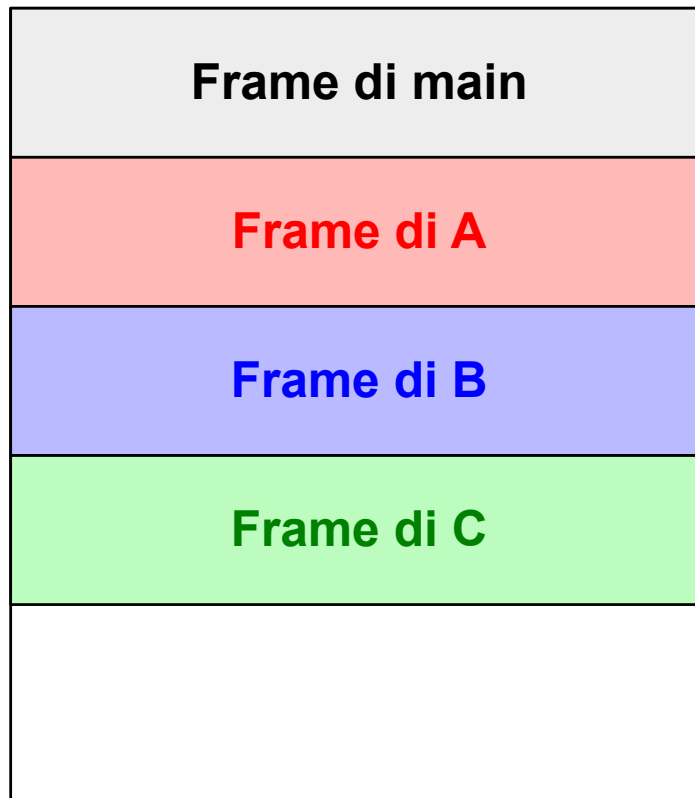


Allocazione/Deallocazione degli stack frame

- Esempio animato di allocazione dei frame sullo stack:



STACK



Indirizzi alti



Direzione di
crescita dello
stack

Chi salva i registri?

- Chi è responsabile di salvare i 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 modificherebbe
 - 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 prima di procedere a modificarli (**callee**)
- \$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 *ja1* (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 . . . \$t9Infatti, “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 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, ... $t9`)
 - nei registri temporanei (`$a0, ... $a3`)
 - in quelli usati convenzionalmente per i risultati (`$v0, $v1`)
 - in quelli non indirizzabili (`$Hi, $Lo`)
- La funzione *chiamante* non deve salvare nulla se non necessita che certi registri vengano preservati dopo la chiamata
 - (`$t0, ... $t9, $v0, $v1, $a0, ... $a3`)

Minimo salvataggio di registri (esempio)

- La funzione **main**, in qualità di funzione **chiamata**, invoca un'altra funzione **quadrato**, e deve quindi salvare
 - \$ra (**callee-saved**)
- La funzione **main**, in qualità di funzione **chiamante**, modifica solo \$v0

<pre>.data x: .word 5 result: .space 4 .text main: # salvo ra # PUSH(\$ra) addi \$sp,\$sp,-4 sw \$ra,0(\$sp)</pre>	<pre># chiamo la procedura # quadrato lw \$a0,x(\$zero) jal quadrato sw \$v0,result # ripristino ra # POP(\$ra) lw \$ra, 0(\$sp) addi \$sp,\$sp,4 # return dal main jr \$ra</pre>	<pre># procedura che # calcola il quadrato # dell'argomento # intero passato # in \$a0 quadrato: mult \$a0,\$a0 mflo \$v0 jr \$ra # return</pre>
---	---	---

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)

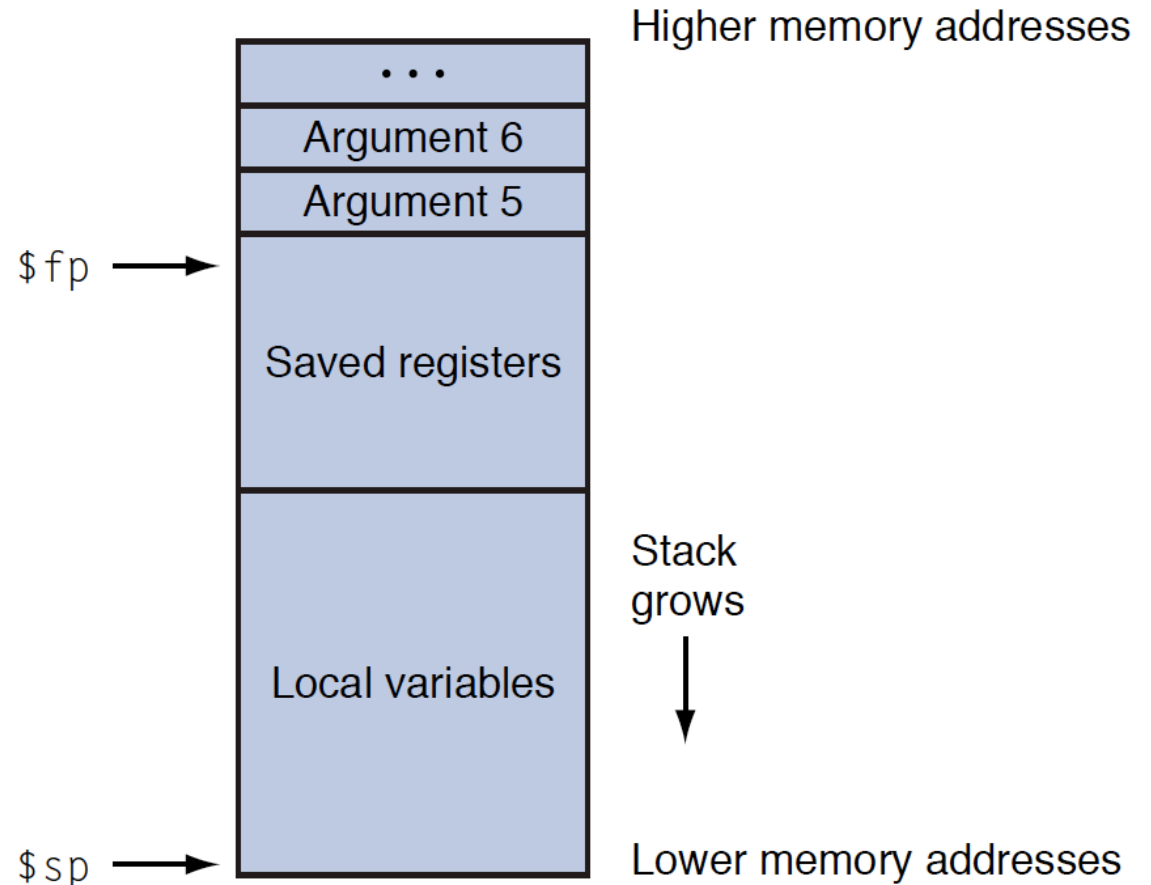
```
# procedura che calcola il quadrato  
# dell'argomento intero passato in $a0
```

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
- Dovete assegnare voi il valore a **\$fp** all’inizio della procedura (dopo averne salvato il valore nello 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**

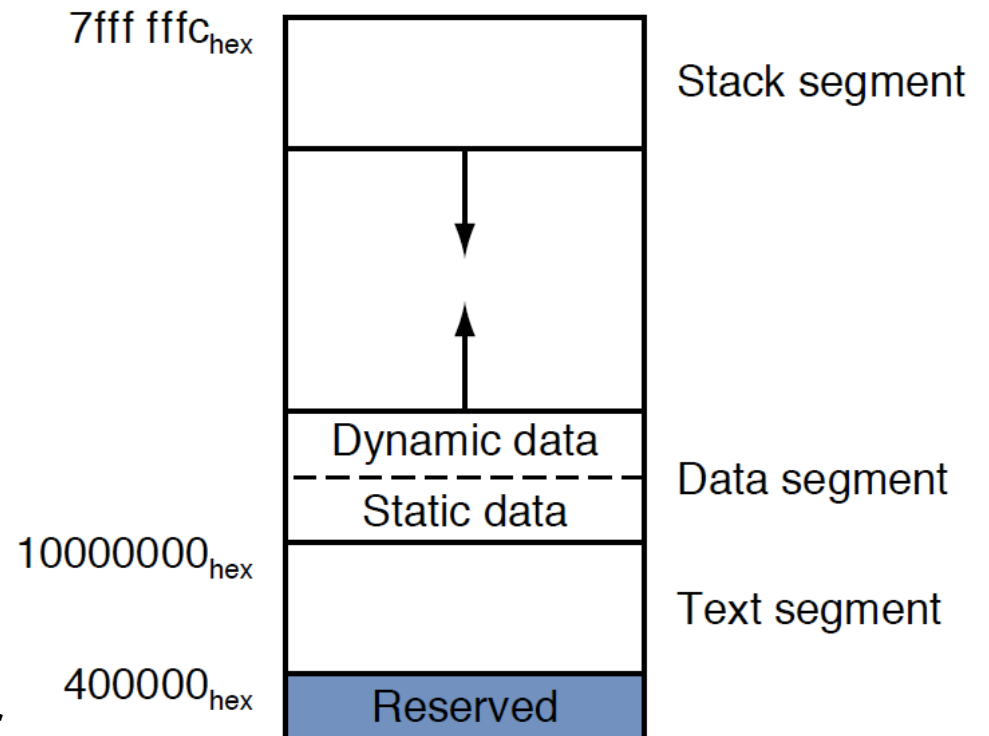
- **Static data** (a partire dall'indirizzo 0x10000000)
 - Contiene variabili globali, attive per tutta l'esecuzione del programma

- **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

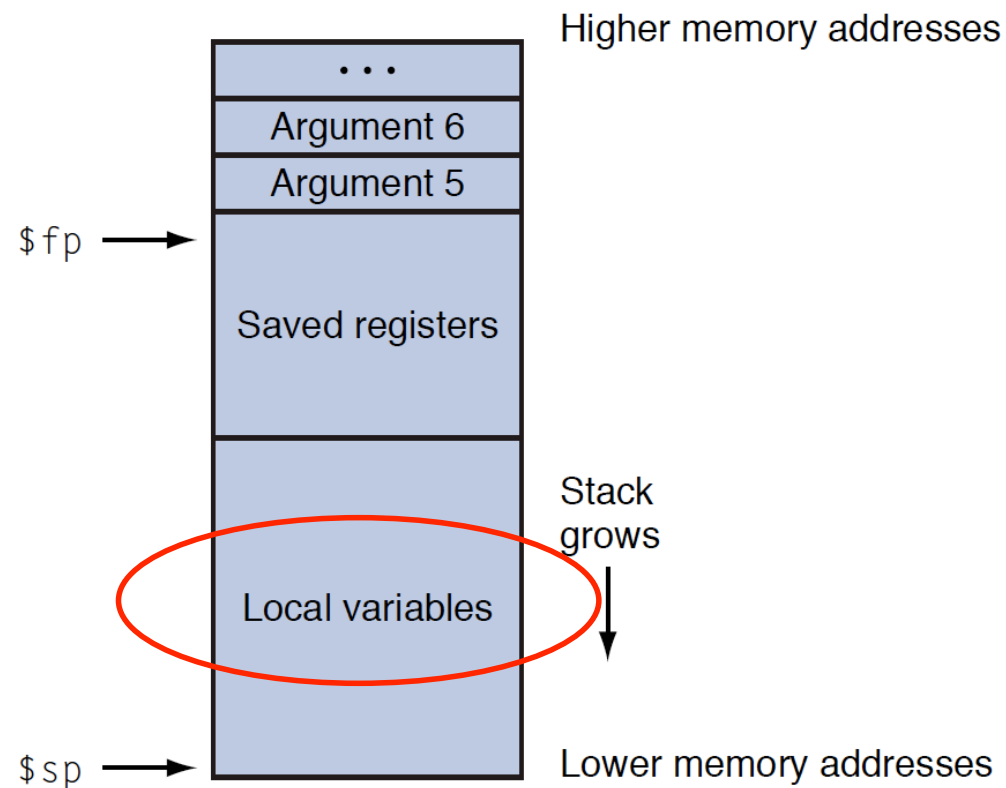


Variabili globali e locali

- 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 ottimiz.

foo:

addi \$sp,\$sp,-4 # fai spazio 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 # elimino spazio per temp

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 presente nella locazione di memoria `indirizzo+reg1`
- Motivo principale: manipolare le 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

- **Corrispondenza standard: ASCII**
 - **ASCII: American Standard Code for Information Interchange**
- **Codici diversi per rappresentare 128 caratteri:**
 - **da 0 a 127: sta in un byte**
- **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)**

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 il valore di '0' (48)
 - Esempio: il valore 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');  
}
```

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 (y[i] != 0) goto DOLOOP

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

Registri modificati:

\$s0 e \$s1: callee save

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

... 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 (programma di gestione) che vanno a interagire con i controller del dispositivo video
- 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
- L'invocazione provoca anche il passaggio dalla modalità di esecuzione *user* a quella *kernel*
- Per invocare queste speciali funzioni, si usa si usa l'istruzione MIPS:
 - `syscall` (“system call”, chiamata di sistema)
- Tipicamente, siccome la `syscall` invoca una procedura (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
 - 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)
{
    if (n == 1)
        return(1);
    else
        return(n * fact(n-1));
}
```

```
main()
{
    int ret;
    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
    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
```

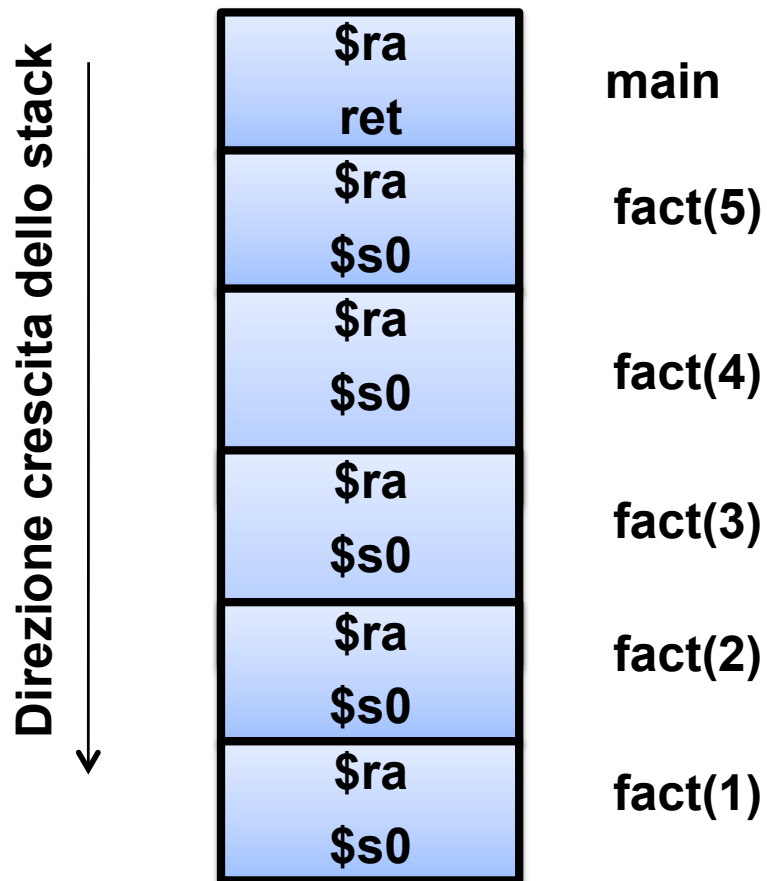
main:

```
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?



SPIM (PCSPIM)

The screenshot shows the PCSpim simulator window. The top menu bar includes File, Simulator, Window, and Help. Below the menu is a toolbar with icons for file operations and simulation control. The main window is divided into several sections:

- Registers:** A table showing the status of various registers.

PC = 00400000	EPC = 00000000	Cause = 00000000	BadVAddr = 00000000
Status = 3000ff10	HI = 00000000	LO = 00000000	
- General Registers:** A table showing the values of registers R0 through R29.

R0 (r0) = 00000000	R8 (t0) = 00000000	R16 (s0) = 00000000	R24 (t8) = 00000000
R1 (at) = 00000000	R9 (t1) = 00000000	R17 (s1) = 00000000	R25 (t9) = 00000000
R2 (v0) = 00000000	R10 (t2) = 00000000	R18 (s2) = 00000000	R26 (k0) = 00000000
R3 (v1) = 00000000	R11 (t3) = 00000000	R19 (s3) = 00000000	R27 (k1) = 00000000
R4 (a0) = 00000000	R12 (t4) = 00000000	R20 (s4) = 00000000	R28 (gp) = 10008000
R5 (a1) = 00000000	R13 (t5) = 00000000	R21 (s5) = 00000000	R29 (sp) = 7ffffeffc
- Assembly Code:** A list of instructions with their addresses and comments.


```

[0x00400024] 0x27bffff8 addiu $29, $29, -8      ; 5: subu    $sp, $sp, 8      # frame
[0x00400028] 0xafbf0000 sw $31, 0($29)          ; 6: sw      $ra, 0($sp)    # save
[0x0040002c] 0xafb00004 sw $16, 4($29)          ; 7: sw      $s0, 4($sp)    # save
[0x00400030] 0x00048021 addu $16, $0, $4        ; 8: move    $s0, $a0
[0x00400034] 0x34010001 ori $1, $0, 1           ; 9: bne     $s0, 1, else    # if (t
[0x00400038] 0x14300003 bne $1, $16, 12 [else-0x00400038]
[0x0040003c] 0x34020001 ori $2, $0, 1           ; 10: li     $v0, 1         # casc
[0x00400040] 0x08100015 j 0x00400054 [exit]      ; 11: j      exit
[0x00400044] 0x2604ffff addiu $4, $16, -1       ; 13: addu   $a0, $s0, -1
      
```
- Memory Segments:**
 - DATA:** [0x10000000]...[0x10040000] 0x00000000
 - STACK:** [0x7ffffeffc] 0x00000000
 - KERNEL DATA:**

[0x90000000]	0x78452020	0x74706563	0x206e6f69	0x636f2000
[0x90000010]	0x72727563	0x61206465	0x6920646e	0x726f6e67

At the bottom, there is a status bar showing the current PC, EPC, and Cause values, and a footer with copyright information and a note about the file assembly.

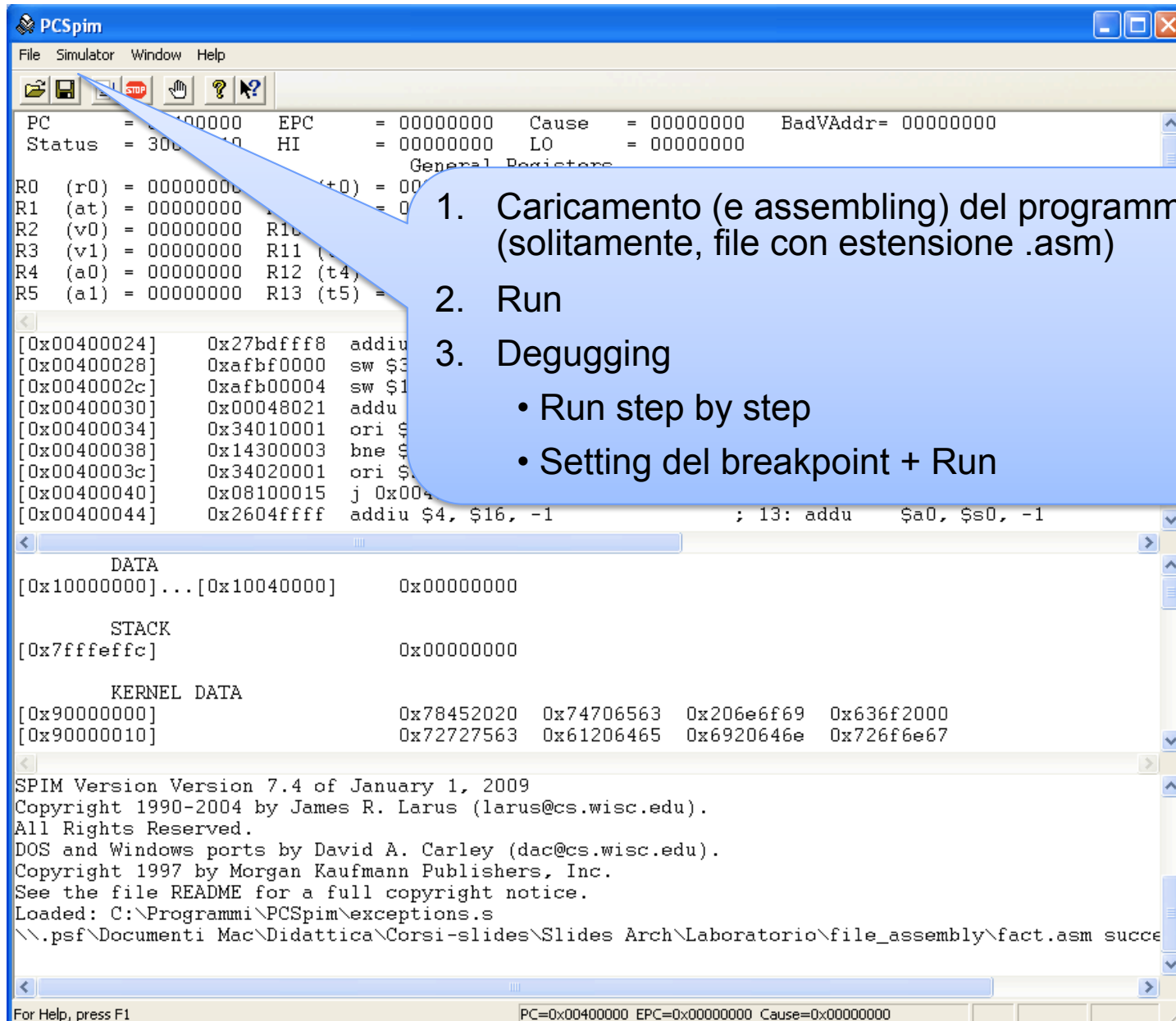
Mapa dei registri

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)

Text segment
Data segment

SPIM (PCSPIM)



PCSpim

File Simulator Window Help

PC = 00000000 EPC = 00000000 Cause = 00000000 BadVAddr= 00000000
Status = 30000000 HI = 00000000 LO = 00000000

General Registers

R0 (r0) = 00000000 R10 (t0) = 00000000
R1 (at) = 00000000 R11 (t1) = 00000000
R2 (v0) = 00000000 R12 (t2) = 00000000
R3 (v1) = 00000000 R13 (t3) = 00000000
R4 (a0) = 00000000 R14 (t4) = 00000000
R5 (a1) = 00000000 R15 (t5) = 00000000

[0x00400024] 0x27bdfff8 addiu \$4, \$16, -1 ; 13: addu \$a0, \$s0, -1
[0x00400028] 0xafbf0000 sw \$3, 0(\$a0)
[0x0040002c] 0xafb00004 sw \$1, 0(\$a0)
[0x00400030] 0x00048021 addu \$4, \$1, \$2
[0x00400034] 0x34010001 ori \$4, \$0, 1
[0x00400038] 0x14300003 bne \$4, \$0, 3
[0x0040003c] 0x34020001 ori \$4, \$0, 1
[0x00400040] 0x08100015 j 0x00400044
[0x00400044] 0x2604ffff addiu \$4, \$16, -1 ; 13: addu \$a0, \$s0, -1

DATA
[0x10000000]...[0x10040000] 0x00000000

STACK
[0x7ffffeffc] 0x00000000

KERNEL DATA
[0x90000000] 0x78452020 0x74706563 0x206e6f69 0x636f2000
[0x90000010] 0x72727563 0x61206465 0x6920646e 0x726f6e67

SPIM Version Version 7.4 of January 1, 2009
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
DOS and Windows ports by David A. Carley (dac@cs.wisc.edu).
Copyright 1997 by Morgan Kaufmann Publishers, Inc.
See the file README for a full copyright notice.
Loaded: C:\Programmi\PCSpim\exceptions.s
\\.\psf\Documenti Mac\Didattica\Corsi-slides\Slides Arch\Laboratorio\file_assembly\fact.asm succe

For Help, press F1 PC=0x00400000 EPC=0x00000000 Cause=0x00000000

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:
    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 (stri[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: .asciiz "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
lw    $ra, 0($sp)    # ripristina $ra

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[]
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)

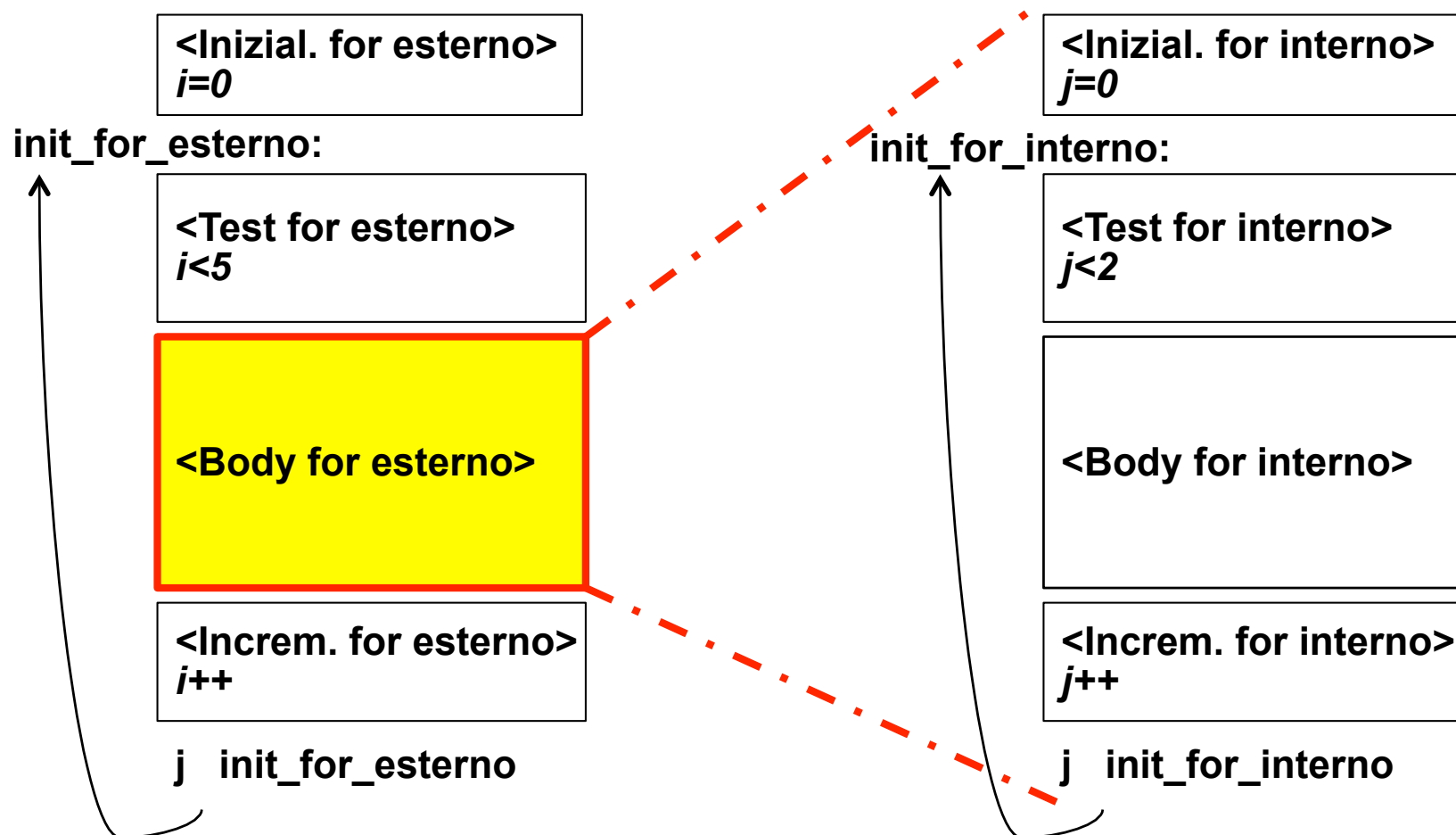
Esercizio 3

```
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, ed 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) è           # $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($4)` non può essere tradotta con una semplice macchina, e quindi diventa una pseudo-istruzione?
- Per invocare una funzione è sufficiente l'istruzione `j` (Jump)? Commentare.
- Qual è il razionale della convenzione di chiamata che impone di salvare i registri `$s0`, `$s1`, ecc. in accordo all'approccio callee-save?
- Qual è il razionale della convenzione di chiamata che impone di salvare i registri `$t0`, `$t1`, ecc. in accordo all'approccio caller-save?