Università degli Studi di Milano
Dipartimento di Informatica “Giovanni Degli Antoni”
Corso di Laurea Triennale in Informatica

Architettura degli Elaboratori II Laboratorio

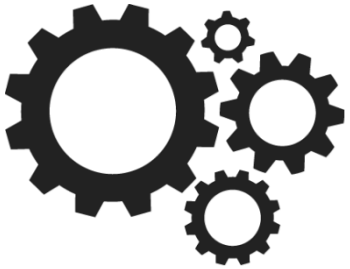
Progettare e assemblare software in MIPS

Introduzione

Linguaggio di alto livello

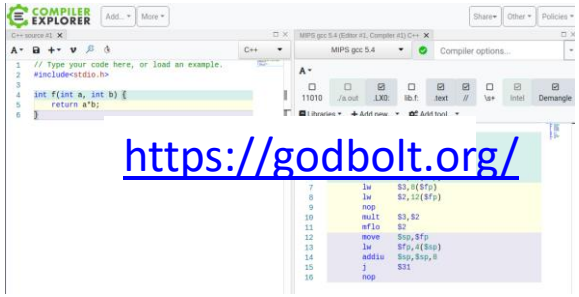
```
int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

compilatore



Assembly

```
multi $2, $5,4
add   $2, $4,$2
lw    $15, 0($2)
lw    $16, 4($2)
sw    $16, 0($2)
sw    $15, 4($2)
jr    $31
```



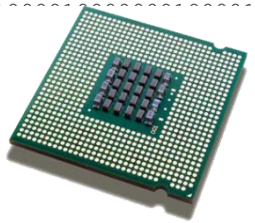
https://godbolt.org/

Livello più basso (vicino all'hardware) dove poter programmare le istruzioni di un elaboratore

Assembler + linker

Linguaggio macchina

hardware



Assembly

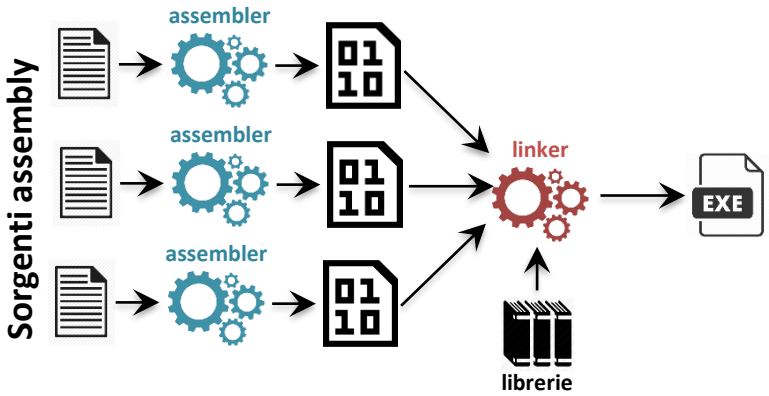
È la rappresentazione simbolica del linguaggio macchina di un elaboratore.

```
add $t0 $s2 $s3
Binary: 00000010010100110100000000100000
Hex: 0x02534020
```

MIPS instruction converter

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000 6	$s2 10010 5	$s3 10011 5	$t0 01000 5	0 00000 5	ADD 100000 6

- Dà alle istruzioni una forma *human-readable* e permette di usare **label** per referenziare con un nome parole di memoria che contengono istruzioni o dati.



- Programmi coinvolti:
  - **assembler**: «traduce» le istruzioni assembly (da un **file sorgente**) nelle corrispondenti istruzioni macchina in formato binario (in un **file oggetto**);
  - **linker**: combina i files oggetto e le librerie in un **file eseguibile** dove la «destinazione» di ogni label è determinata.

Assembly

- Il codice Assembly può essere il risultato di due processi:
  - *target language* del compilatore che traduce un programma in linguaggio di alto livello (C, Pascal, …) nell'equivalente assembly;
  - *linguaggio di programmazione* usato da un programmatore.

- Assembly è stato l'approccio principale con cui scrivere i programmi per i primi computer.

- Assembly come linguaggio di programmazione è adatto in certi casi particolari:
  - ottimizzare le performance (anche in termini di prevedibilità) e spazio occupato da un programma (ad es., sistemi embedded);
  - eredità di certi sistemi vecchi, ma ancora in uso, dove Assembly rappresenta l'unico modo conveniente per scrivere programmi;
  - rendere più efficienti certe istruzioni che hanno una semantica di basso livello.

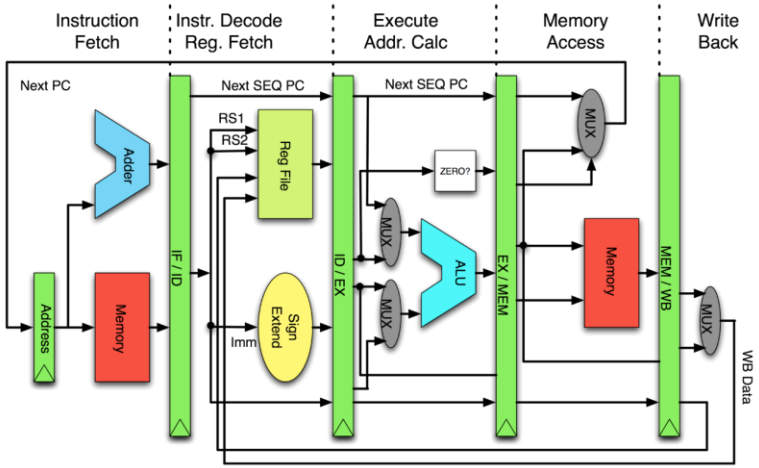
MIPS

- In questo laboratorio lavoreremo con **MIPS**
- MIPS: Multiprocessor without Interlocked Pipeline Stages → un'Instruction Set Architecture (ISA) di tipo RISC
- Nasce a metà anni '80 come architettura *general purpose;*

- Inizialmente è un progetto accademico (Stanford), poco dopo diventa commerciale
- Oggi è impiegata prevalentemente nell'ambito dei *sistemi embedded*

MIPS

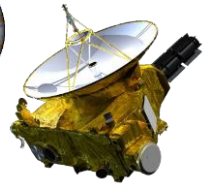
- La maggior parte dei corsi accademici di architetture adotta MIPS, perché?



- È una prima e lineare implementazione del concetto di pipeline
- È costruita su una semplice assunzione: ogni stadio della pipeline deve terminare in un ciclo di clock, ogni stadio non necessita di attendere il completamento degli altri (interlock)
- *(Oggi l'assunzione è rilassata per avere istruzioni come moltiplicazione e divisione, ma il nome è rimasto lo stesso)*

MIPS: passato e presente

NASA New Horizons,
$\sim 6180 \times 10^6$ Km
dalla Terra



Sony Playstation I

Nintendo 64

MIPS I
R2000 ,R3000

1985

MIPS II
R6000

1990

MIPS III
R4000, … R4700

1991

Noi siamo qui!

MIPS32, MIPS64
4K, …, 20K

1999

Home routers

Tesla S

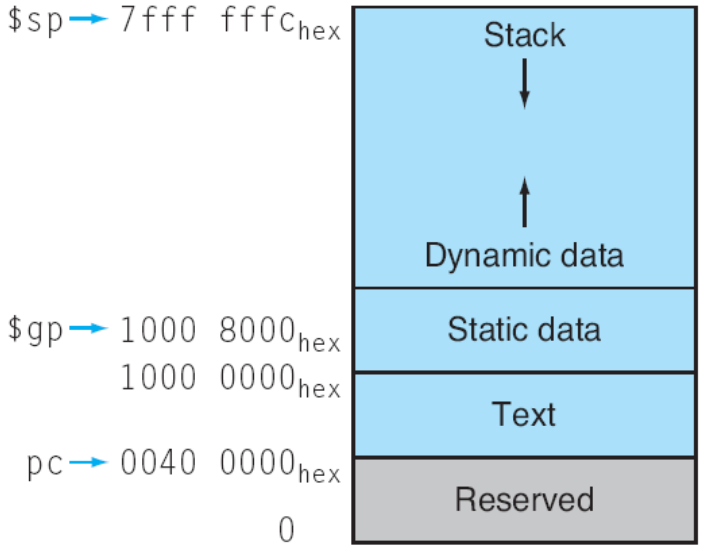
SGI server
Power Challenge

MIPS IV
R5000, … R18000

1996

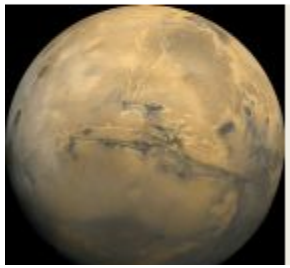
Il programma in memoria (in MIPS)

- **Segmento testo**: contiene le **istruzioni** del programma.
- **Segmento dati**:
  - ***dati statici*:** contiene dati la cui dimensione è conosciuta a *compile time* e la cui durata coincide con quella del programma *(e.g., variabili statiche, costanti, etc.);*
  - ***dati dinamici*:** contiene dati per i quali lo spazio è allocato dinamicamente a *runtime* su richiesta del programma stesso *(e.g., liste dinamiche, etc.).*
- **Stack**: contiene dati dinamici organizzati secondo una coda LIFO (Last In, First Out) (e.g., parametri di una procedura, valori di ritorno, etc.).



MARS

Missouri State UNIVERSITY



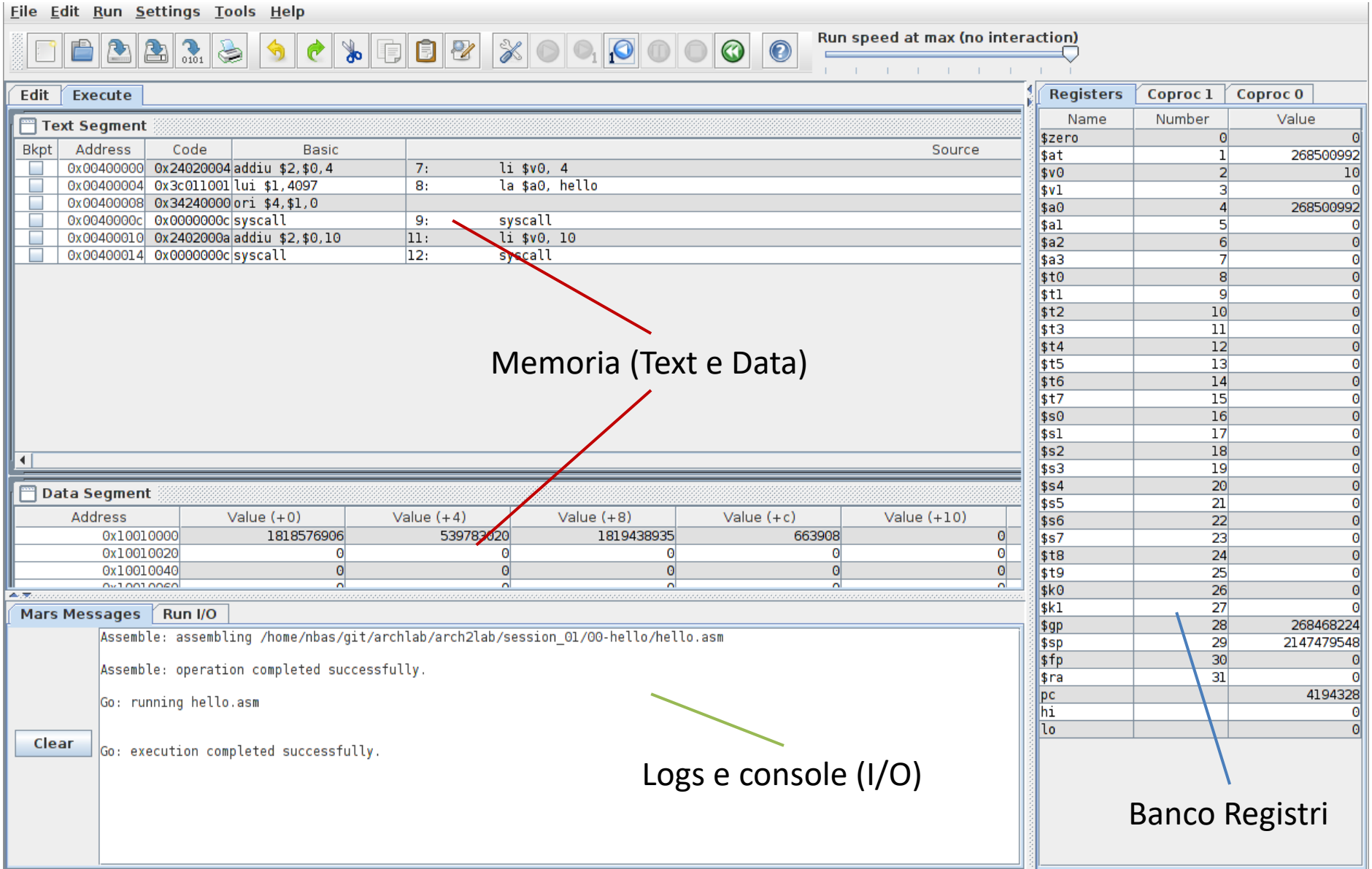
***MARS** (MIPS Assembler and Runtime Simulator)*

An IDE for MIPS Assembly Language Programming

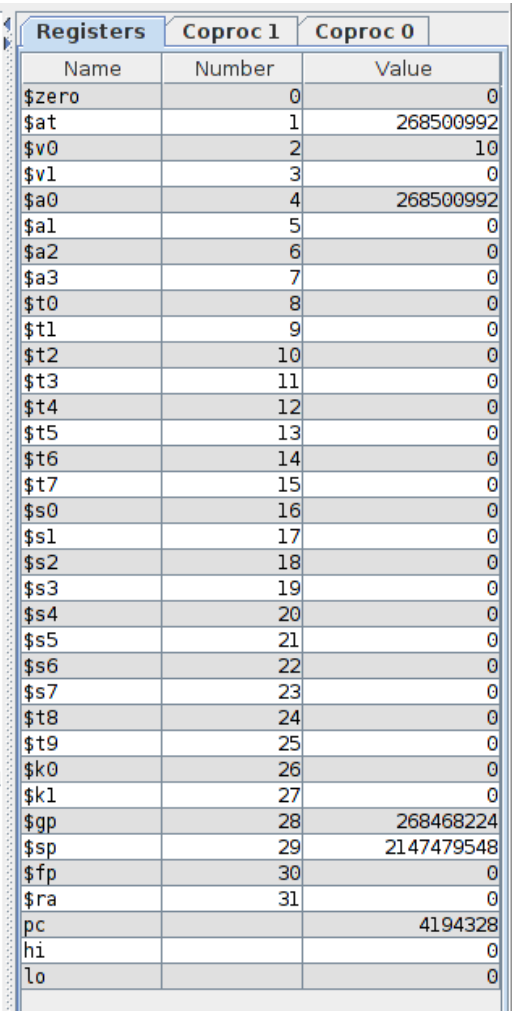
MARS is a lightweight interactive development environment (IDE) for programming in MIPS assembly language, intended for educational-level use with Patterson and Hennessy's *Computer Organization and Design*.

- È un emulatore di una CPU che obbedisce alle convenzioni MIPS32
- Perché usare un emulatore e non la macchina vera?
  - Usiamo tutti la stessa ISA indipendentemente dal calcolatore reale.
  - Ci offre una serie di strumenti che rendono la programmazione più comoda.
  - Maschera certi aspetti reali a cui non saremmo interessati (es., delays).
- Disponibile a questo URL http://courses.missouristate.edu/KenVollmar/MARS/index.htm

MARS (interfaccia)



MARS (Registri)



Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0
$at	1	268500992
$v0	2	10
$v1	3	0
$a0	4	268500992
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	0
$t1	9	0
$t2	10	0
$t3	11	0
$t4	12	0
$t5	13	0
$t6	14	0
$t7	15	0
$s0	16	0
$s1	17	0
$s2	18	0
$s3	19	0
$s4	20	0
$s5	21	0
$s6	22	0
$s7	23	0
$t8	24	0
$t9	25	0
$k0	26	0
$k1	27	0
$gp	28	268468224
$sp	29	2147479548
$fp	30	0
$ra	31	0
pc		4194328
hi		0
lo		0

- 32 registri a 32bit per operazioni su interi (**$0..$31**).
- 32 registri a 32 bit per operazioni in virgola mobile sul coprocessore 1 (**$FP0**..**$FP31**).
- registri speciali a 32bit:
  - il **Program Counter** (**PC**) l'indirizzo della prossima istruzione da eseguire;
  - **hi** e **lo** usati nella moltiplicazione e nella divisione;
  - **EPC, Cause, BadVAddr, Status** (coprocessore 0) vengono usati nella gestione delle eccezioni.
- I registri general-purpose sono chiamati col nome dato dalla convenzione MIPS e numerati da 0 a 31
- Il loro valore è ispezionabile nel formato esadecimale o decimale

Richiamo di istruzioni aritmetiche (somma, sottrazione)

- Convenzioni di notazione:
  - Identificativo con iniziale minuscola: deve essere un registro o un valore immediato (intero con segno su 16 bit);
  - Identificativo con iniziale «$»: deve essere un registro.

```
add $s1, $s2, $s3# $s1 = $s2 + $s3, rileva overflow

sub $s1, $s2, $s3# $s1 = $s2 - $s3, rileva overflow

addi $s1, $s2, 13# $s1 = $s2 + 13, rileva overflow

addu $s1, $s2, $s3        # $s1 = $s2 + $s3, unsigned, non rileva overflow

subu $s1, $s2, $s3        # $s1 = $s2 - $s3, unsigned, non rileva overflow

addui $s1, $s2, 27        # $s1 = $s2 + 17, unsigned, non rileva overflow
```

Istruzioni: moltiplicazione

- Due istruzioni:
  - `mult $rs $rt`
  - `multu $rs $rt` `# unsigned`

- Il registro destinazione è **implicito.**

- Il risultato della moltiplicazione viene posto sempre in due registri dedicati di una parola (special purpose) denominati **hi (High order word)** e **lo (Low order word)**.

- La moltiplicazione di due numeri rappresentabili con 32 bit può dare come risultato un numero non rappresentabile in 32 bit.

Istruzioni: moltiplicazione

- Il risultato della moltiplicazione si preleva dal registro **hi** e dal registro **lo** utilizzando le due istruzioni:
  - `mfhi $rd  # move from hi`
    - sposta il contenuto del registro **hi** nel registro **rd**;
  - `mflo $rd  # move from lo`
    - sposta il contenuto del registro **lo** nel registro **rd**.

Operazioni aritmetiche: divisione

```
div $s2, $s3  # $s2 / $s3, divisione intera
```

- Il risultato della divisione intera va in:
  - Lo: $s2 / $s3 [quoziente];
  - Hi: $s2 mod $s3 [resto].

- Il risultato va quindi prelevato dai registri Hi e Lo utilizzando ancora la `mfhi` e la `mflo`.

Istruzioni: pseudo-istruzioni

- Le pseudoistruzioni sono un modo compatto ed intuitivo di specificare un insieme di istruzioni.
- La traduzione della pseudoistruzione nelle istruzioni equivalenti è attuata automaticamente dall'assemblatore.

Esempi:

- `move $t0, $t1` `# pseudo istruzione`
  - `add $t0, $zero, $t1` `# (in alternativa) addi $t0, $t1, 0`

- `mul $s0, $t1, $t2` `# pseudo istruzione`
  - `mult $t1, $t2`
  - `mflo $s0`

- `div  $s0, $t1, $t2` `# pseudo istruzione`
  - `div $t1, $t2`
  - `mflo $s0`

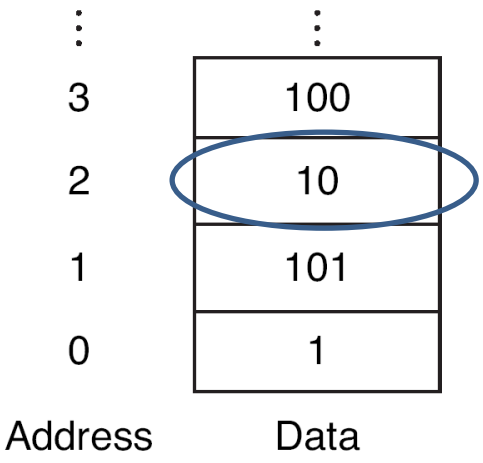
Primo programma in Assembly

Indirizzamento, lettura e scrittura della memoria

Organizzazione della memoria

- Cosa contiene la memoria?
  - Le istruzioni da eseguire
  - Le strutture dati su cui operare

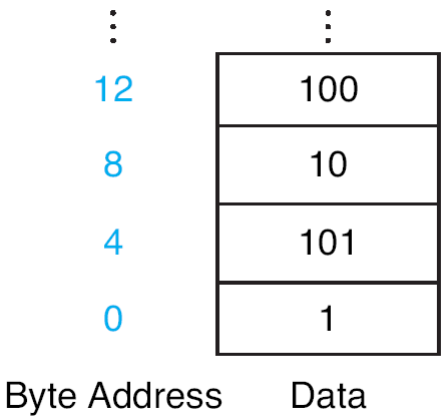
- Come è organizzata?
  - Array uni-dimensionale di elementi dette *parole*
  - Ogni parola è univocamente associata ad un *indirizzo* (come l'indice di un array)



Parola associata all'indirizzo 2, Mem[2]=10

Organizzazione della memoria

- In generale, la dimensione della parola di memoria non coincide con la dimensione dei registri nella CPU (ma nel MIPS sì).
- La parola è l'unità base dei trasferimenti tra memoria e registri (*load word* e *store word* operano per parole di memoria)
- In MIPS (e quindi anche nel simulatore MARS) una parola è composta da 32 bit e cioè 4 byte

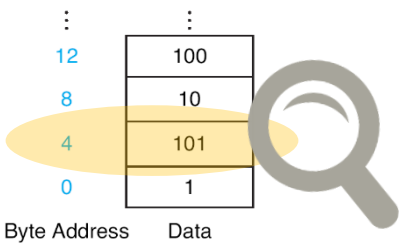


Il singolo byte è un elemento di memoria spesso ricorrente

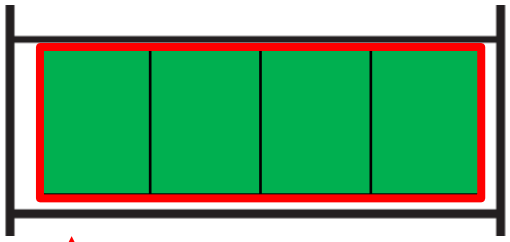
Costruiamo lo spazio degli indirizzi in modo che ci permetta di indirizzare ognuno dei 4 bytes che compongono una parola: gli indirizzi di due parole consecutive differiscono di 4

Endianness

- L'indirizzo di una parola di memoria è in realtà l'indirizzo di uno dei 4 byte che compongono quella parola



4



Questo indirizzo

che identifica questa parola

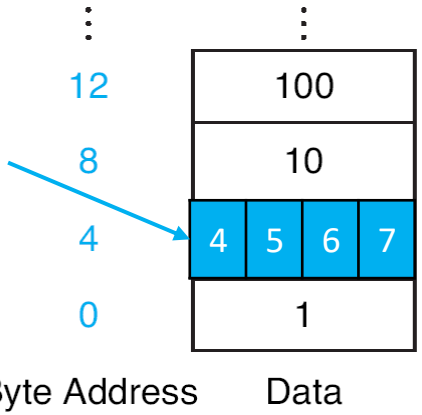
è in realtà l'indirizzo del primo dei 4 byte della parola

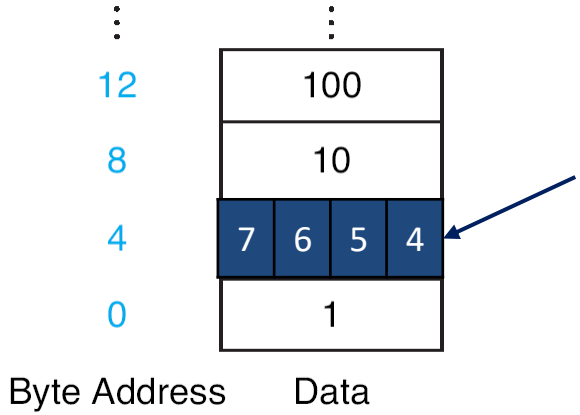
- Ma, tra i 4, quale è il **primo byte?** La risposta sta nell'ordine dei byte: la **endianness**

Endianness

- La **endianness** stabilisce l'ordine dei byte (quindi chi è il **primo** e chi l'ultimo)

Big endian: il primo byte è quello **più** significativo (quello più a **sinistra**, **big** end)





Little endian: il primo byte è quello **meno** significativo (quello più a **destra**, **little** end)

- MIPS è una architettura Big Endian, ma …
- … il nostro emulatore MARS eredita la endianness della macchina su cui è eseguito

Accesso alla memoria in Assembly

- Lettura dalla memoria: **Load Word**

```
lw $s1, 100($s2) # $s1 <- M[[$s2]+100]
```

- Scrittura verso la memoria: **Store Word**:

```
sw $s1, 100($s2) # M[[$s2]+100] <- $s1
```

- La memoria viene indirizzata come un vettore: indirizzo base + offset identificano la locazione della parola da scrivere o leggere

- L'offset è in byte

Inizializzazione esplicita degli indirizzi

- Come fare a caricare degli indirizzi nei registri? Obiettivo:
  - caricare in `$s1` l'indirizzo `0x10000000` (per es. indirizzo di `h`)
  - caricare in `$s2` l'indirizzo `0x10000004` (per es. base address di `A`)

- Soluzione?

```
addi $s1, $zero, 0x10000000     # $t0 = &h
addi $s2, $zero, 0x10000004     # $t1 = A
```

32 bit

- No! Il valore «immediato» in `addi` deve essere un intero (con segno, in C2) su 16 bit! (Un'istruzione richiede 32 bit nel suo complesso)
- Cosa succede se assembliamo queste due istruzioni in MARS?

0x00400000	0x3c011000	lui $1,0x00001000
0x00400004	0x34210000	ori $1,$1,0x00000000
0x00400008	0x00018820	add $17,$0,$1
0x0040000c	0x3c011000	lui $1,0x00001000
0x00400010	0x34210004	ori $1,$1,0x00000004
0x00400014	0x00019020	add $18,$0,$1

```
addi $s1, $zero, 0x10000000
```

```
addi $s2, $zero, 0x10000004
```

Inizializzazione esplicita degli indirizzi

- Metodo più comodo: usare la pseudo-istruzione «**load address**»:

```
la $s1, 0x10000000      # $t0 = &h
```

32 bit

Vettori

- Si consideri un vettore `v` dove ogni elemento `v[i]` è una parola di memoria (32 bit).
- Obiettivo: leggere/scrivere `v[i]` (elemento alla posizione `i` nell'array).
- Gli array sono memorizzati in modo sequenziale:
  - `b`: registro base di `v`, è anche l'indirizzo di `v[0];`
  - l'elemento i-esimo ha indirizzo `b + 4*i`.

```
la $s1, 0x10000000      # $t0 <- &h
la $s2, 0x10000004      # $t1 <- A
lw $t0, 0($s1)          # $t0 <- h
lw $t1, 32($s2)         # $t1 <- A[8]
add $t0, $t1, $t0       # $t0 <- $t1 + $t0
sw $t0, 48($s2)         # A[12] <- $t0
```

Direttive Assembler

- `.data` specifica che ciò che segue nel file sorgente è il segmento dati: vengono specificati gli elementi presenti in tale segmento (stringe, array, etc ...).

- `.text` specifica che ciò che segue nel file sorgente è il segmento testo

- `STRINGA: .asciiz “stringa_di_esempio”` memorizza la stringa “stringa_di_esempio” in memoria (aggiungendo terminatore di fine stringa), il suo indirizzo è referenziato con la label ‘’STRINGA’’ (significa che potremo scrivere ‘’STRINGA’’ anzichè l’indirizzo in formato numerico).

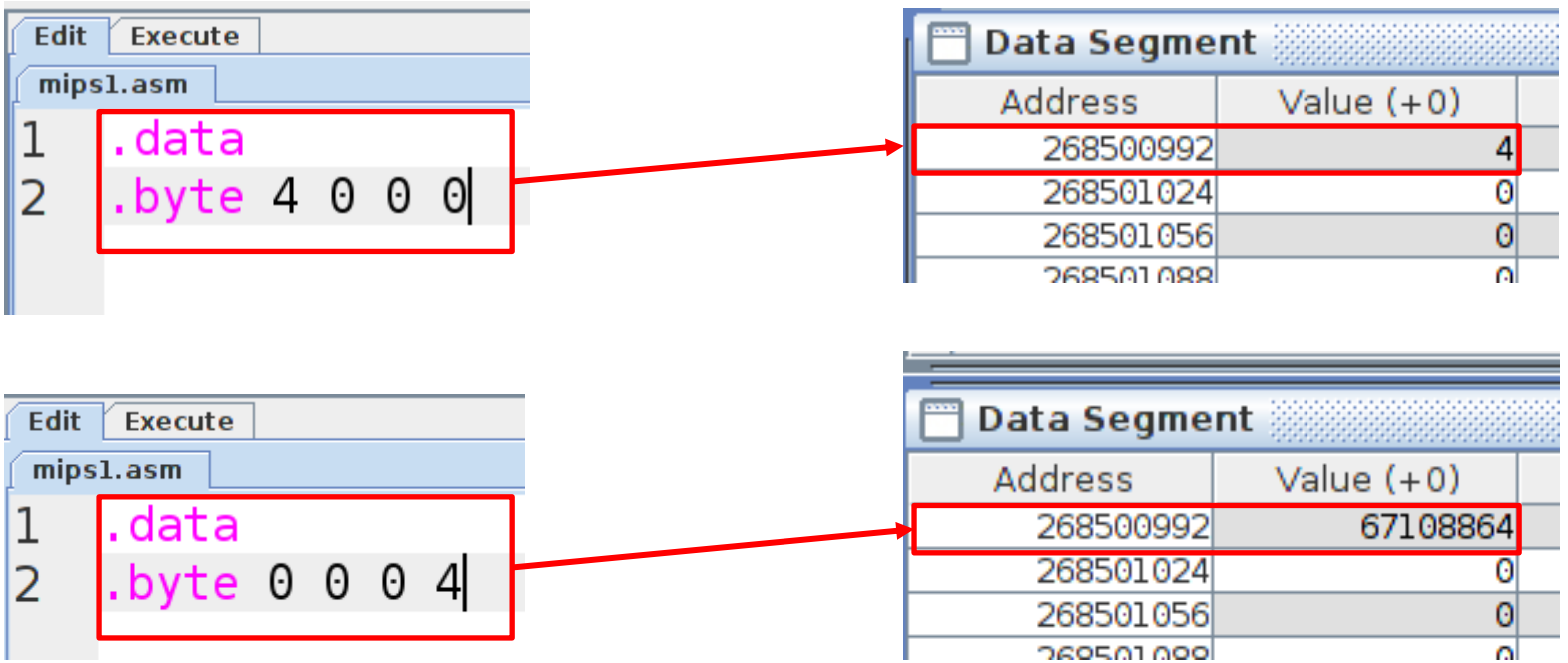
- `A: .byte b1, ..., bn` memorizza gli `n` valori in `n` bytes successivi di memoria, la label `A` rappresenta il base address della sequenza (indirizzo della parola con i primi quattro bytes).

- `A: .space n` alloca `n` byte di spazio nel segmento corrente (deve essere data), la label `A` rappresenta il base address (indirizzo della parola con i primi quattro degli `n` bytes).

La direttiva `.byte` e la endianness

- Testiamo la endianness della macchina su cui stiamo lavorando *(in Linux: comando «lscpu», proprietà «Byte order»)*:

- Cerchiamo di allocare la costante 4 in una parola di memoria usando la direttiva byte che permette di inserire parole di memoria specificando il valore di ogni singolo byte che la compone:

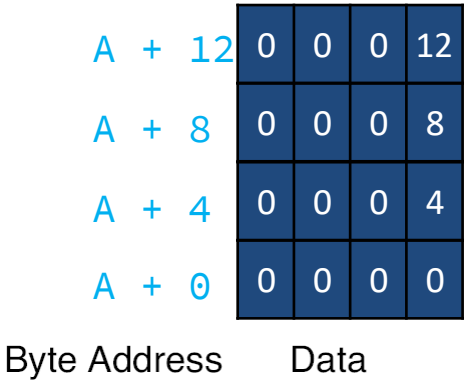


La direttiva `.byte` e la endianness

```
.data
A: .byte 0 0 0 0 4 0 0 0 8 0 0 0 12 0 0 0
```

Least Significant Byte

Most Significant Byte



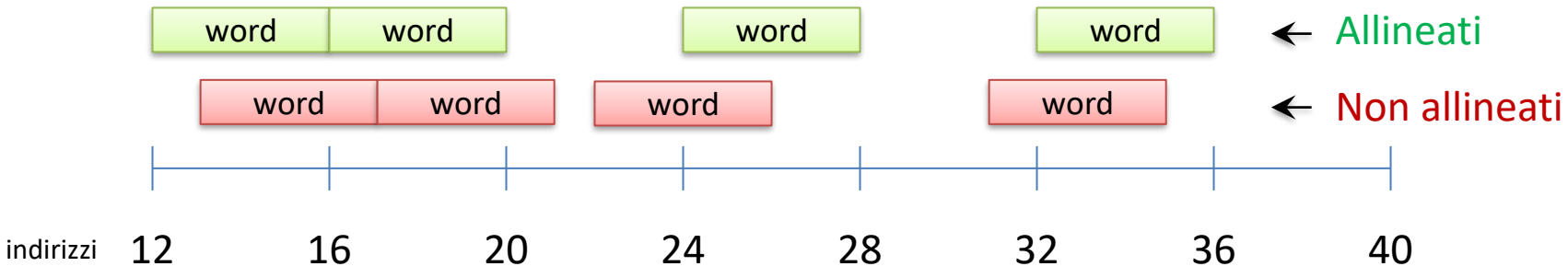
Attenzione!

I valori vengono scritti secondo il *byte order* della macchina: la famiglia di architetture x86 è **Little Endian** (Intel Core i7, AMD Phenom II, FX, …).

Allineamento dati

- L'accesso a memoria si dice allineato su n byte se:
  - ogni dato ha dimensione n byte
  - n è una potenza di 2
  - l'indirizzo di ogni dato è multiplo di n

- Nel nostro caso:
  - un dato è una word che ha dimensione 4 byte, quindi n=4
  - $(4 = 2^2)$
  - l'indirizzo di ogni word deve essere multiplo di 4



Esempio

```
        .data
        string: .asciiz "Ciao"
        A: .space 8

        .text
        .globl main

main:

        la $t0, A
        li $t1, 5
        sw $t1 0($t0)
```

Esempio

Il segmento dati inizia qui (indirizzo `0x10010000`), i dati che seguono sono allocati in modo sequenziale

```
.data
string: .asciiz "Ciao"
A: .space 8

.text
.globl main

main:

la $t0, A
li $t1, 5
sw $t1 0($t0)
```

La stringa «Ciao» verrà quindi allocata a partire dall'inizio del segmento:

Indirizzo	Valore
0x10010000	c
0x10010001	i
0x10010002	a
0x10010003	o
0x10010004	\0
0x10010005	prima word di A
0x10010009	seconda word di A

Proseguendo nel segmento dati incontriamo **A**, la prima posizione disponibile per allocarlo è nel byte all'indirizzo `0x10010005`

Convertendo in base 10 si osserva che non è multiplo di 4

$(0x10010005)_{16}=(268500997)_{10}$

Cosa succede se tento di accedere ad un indirizzo non allineato con `sw` o `lw` ?

Esempio

```
        .data
string: .asciiz "Ciao"
A: .space 8

        .text
        .globl main

main:

        la $t0, A
        li $t1, 5
        sw $t1 0($t0)
```

```
Go: running mips1.asm

Error in D:\Jacopo Essenziale\MEGA\MIPS_Stuff\mars\mips1.asm line 8: Runtime exception at 0x0040000c: store address not aligned on word boundary 0x10010005

Go: execution terminated with errors.
```

Le istruzioni di `sw` e `lw` richiedono di operare con accesso allineato con parole da 32 bit, quindi se specifichiamo un indirizzo **non** multiplo di 4 in MARS otteniamo un errore.

Esempio

```
        .data
string: .asciiz "Ciao"
        .align 2
        A: .space 8

        .text
        .globl main
main:

        la $t0, A
        li $t1, 5
        sw $t1 0($t0)
```

Aggiungendo la direttiva di allineamento viene lasciato spazio libero per mantenere l'allineamento

Ora A è viene allocato all'indirizzo $(0x10010008)_{16}=(67125250)_{10}$ che è multiplo di 4

Indirizzo	Valore
0x10010000	c
0x10010001	i
0x10010002	a
0x10010003	o
0x10010004	\0
0x10010005	
0x10010006	
0x10010007	
0x10010008	prima word di A
0x1001000C	seconda word di A

Università degli Studi di Milano
Dipartimento di Informatica “Giovanni Degli Antoni”
Corso di Laurea Triennale in Informatica

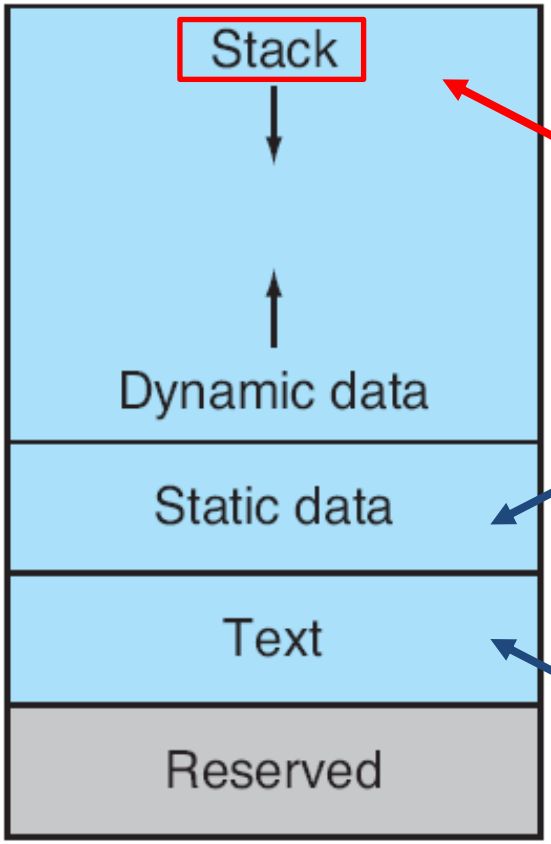
Architettura degli Elaboratori II Laboratorio

Università degli Studi di Milano
Dipartimento di Informatica “Giovanni Degli Antoni”
Corso di Laurea Triennale in Informatica

Architettura degli Elaboratori II Laboratorio

Uso dello Stack

Lo Stack



Regione nell'area dei dati dinamici che utilizziamo rispettando una regola: allocazioni e deallocazioni seguono un **coda LIFO** (last in, first out)

I dati statici usati dal nostro programma, ciò che sta dopo la direttiva `.data`

Le istruzioni del nostro programma, ciò che sta dopo la direttiva `.text`

- Il nostro programma può allocare/deallocare dati sullo stack durante l'esecuzione. **Come si fa?**

Stack Pointer

- Il registro `$sp` (stack pointer) contiene sempre l'indirizzo della parola che sta in cima allo stack. Questa parola è l'ultima ad essere stata inserita nello stack e sarà la prima ad essere rimossa.

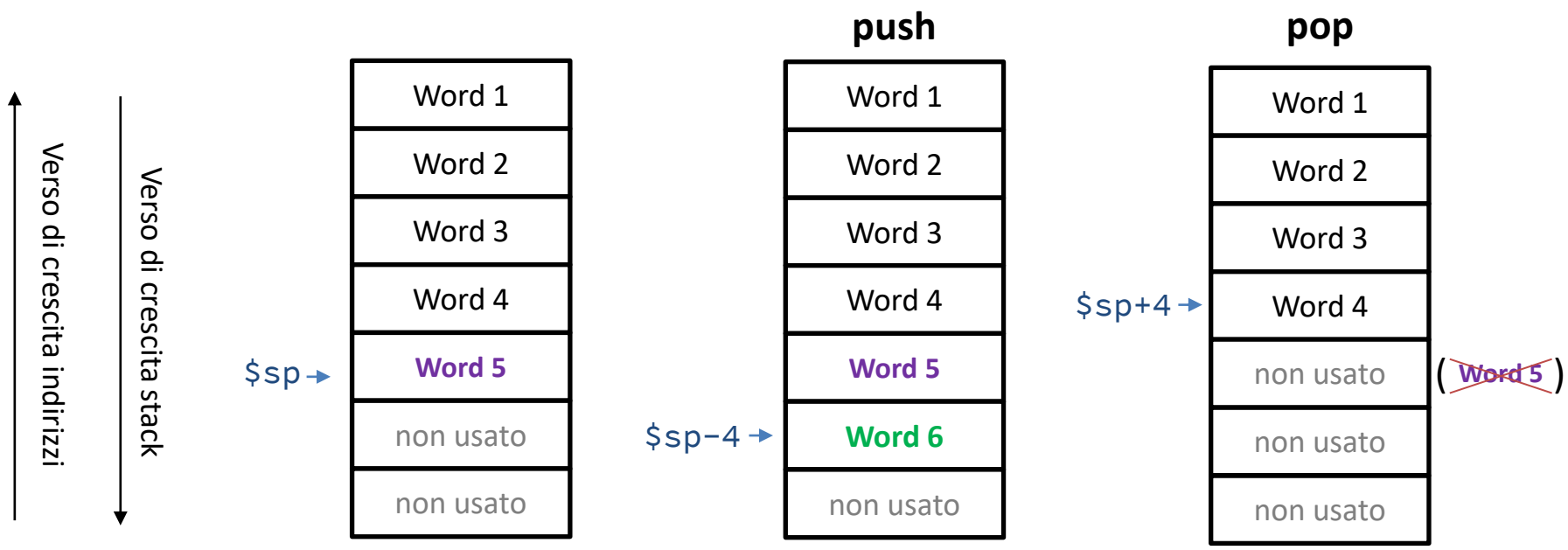
Regole per l'utilizzo dello stack

Per aggiungere un **dato** (push):

```
addi $sp $sp -4
sw $reg 0($sp)
```

Per consumare un **dato** (pop):

```
lw $reg 0($sp)
addi $sp $sp, 4
```



Register Spilling

- Quando serve salvare dati sullo stack? Una prima risposta è «Per fare **spilling** di registri»

- In generale, un programma potrebbe avere un numero di variabili maggiore rispetto al numero di registri disponibili. Non è possibile avere tutti i dati nel banco registri allo stesso momento.

- Una possibile soluzione è questa: tengo nei registri i dati di cui ho maggior bisogno (ad esempio quelli che devo usare più volte o più spesso) mentre i dati di cui non ho bisogno urgente li sposto temporaneamente in memoria.

- Trasferire variabili poco utilizzate da registri a memoria è **register spilling**.

- L'area di memoria di solito utilizzata per questa operazione è lo stack.

Register Spilling

Esempio

- Supponiamo di poter utilizzare solo i registri $t0 e $t1
- Dobbiamo calcolare il prodotto di due variabili che stanno nel segmento dati e i cui indirizzi sono identificati dalle label x e y

```
        .data
x:      .word 3
y:      .word 4

        .text
        .globl main
main:
        la $t0 x
        lw $t1 0($t0)

        addi $sp $sp -4
        sw $t1 0($sp)

        la $t0 y
        lw $t1 0($t0)

        lw $t0 0($sp)
        addi $sp $sp 4

        mult $t0 $t1
        mflo $t0
```

Spilling (push)

Spilling (pop)

Uso delle System Calls

System Calls

- **System call**: permette di utilizzare **servizi** la cui esecuzione è a carico del sistema operativo: tipicamente operazioni di input/output

- Ogni servizio è associato ad un codice numerico univoco (un intero **K**)

- Come si utilizza una system call che ha codice **K**?
  - Caricare **K** nel registro `$v0`;
  - caricare gli argomenti (se necessari) nei registri `$a0, $a1, $a2, $a3, $f12` (opzionale);
  - eseguire l'istruzione `syscall`;
  - leggere eventuali valori di ritorno nei registri `$v0, $f0` (opzionale).

System Calls "Canoniche"

Syscall	Codice	Argomenti	Valore di ritorno	Descrizione
print_int	1	intero da stampare in $a0	nessuno	Stampa l'intero passato in $a0
print_float	2	float da stampare in $f12	nessuno	Stampa il float passato in $f12
print_double	3	double da stampare in $f12	nessuno	Stampa il double passato in $f12
print_string	4	Indirizzo della stringa da stampare in $a0	nessuno	Stampa la stringa che sta all'indirizzo passato in $a0
read_int	5	nessuno	Intero letto in $v0	Legge un intero in input e lo scrive in $v0
read_float	6	nesuno	Float letto in $f0	Legge un float in input e lo scrive in $f0
read_double	7	nessuno	Double letto in $f0	Legge un double in input e lo scrive in $f0
read_string	8	Indirizzo nel segmento dati a cui salvare la stringa in $a0 e lunghezza in byte in $a1	nessuno	Legge una stringa di lunghezza specificata in $a1 e la scrive nel segment dati all'indirizzo specificato in $a0
sbrk	9	Numero di byte da allocare in $a0	Indirizzo del primo dei byte allocati in $v0	Accresce il segmento dati allocanto un numero di byte specificato in $a0, restituisce in $v0 l'indirizzo del primo di questi nuovi byte
exit	10	nessuno	nessuno	Termina l'esecuzione

System Calls "Apocrife"

Syscall	Codice	Argomenti	Valore di ritorno	Descrizione
Time	30	nessuno	32 bit meno significativi del system time in $a0, 32 bit più significativi del system time in $a1	Il system time è rappresentato nel formato Unix Epoch time, cioè il numero di millisecondi trascorsi dal 1 Gennaio 1970
random int	41	Id del generatore pseudo-random in $a0	Prossimo numero pseudo random in $a0	Ad ogni chiamata restituisce un numero intero in una sequenza pseudo-random
random in range	42	Id del generatore pseudo-random in $a0, massimo intero generabile in $a1	Prossimo numero pseudo random in $a0	Ad ogni chiamata restituisce un numero intero in una sequenza pseudo-random, ogni numero sarà compreso tra 0 e il massimo passato in $a1
MessageDialog	55	Indirizzo della stringa da stampare in $a0, intero corrispondente al tipo di messaggio in $a1		Mostra una finestra di dialogo con un messaggio dato dalla stringa passata in $a0. Viene anche mostrata una icona che dipende dal tipo di messaggio passato in $a1: errore (0), info, (1), warning (2), domanda(3)
InputDialogInt	51	Indirizzo della stringa da stampare in $a0	Intero letto in $a0, stato in $a1	

Esempio

```
        .data
msg1: .asciiz "Hello world!"
msg2: .asciiz "Inserisci un intero"

        .text
        .globl main
main:

        li $v0 4,
        la $a0, msg1
        syscall

        li $v0 55
        la $a0 msg1
        li $a1 1
        syscall

        li $v0, 51
        la $a0, msg2
        syscall

        li $v0 10
        syscall
```

Stampiamo una stringa nello standard output (la console)

Stampiamo una stringa in una finestra di diaologo

Leggiamo un intero in input con una finestra di dialogo

Exit

Università degli Studi di Milano
Dipartimento di Informatica “Giovanni Degli Antoni”
Corso di Laurea Triennale in Informatica

Architettura degli Elaboratori II Laboratorio

Controllo di flusso

Controllo di flusso nei linguaggi ad alto livello

Costrutti IF:

```
If(condizione){
 /*then*/
}
```

```
if (condizione) {
/*then*/
}
else {
}
```

Costrutto SWITCH:

```
switch (expr) {
        case 1: {…}
        case 2: {…}
        case 3: {…}
        default: {…}
}
```

Cicli (loops):

```
while (condizione) {
        fai qualcosa
}
```

```
do {
 fai qualcosa
}
while ( condizione )
```

```
for ( init ; condiz ; passo ) {
        fai qualcosa
}
```

Controllo di flusso nei linguaggi ad alto livello

Per esempio (in Go)

```
voti := [] int { 28, 21, 30, 18, 18 }
somma := 0
for i := 0; i < 5; i++ {
    somma += voti[ i ]
}
```

Controllo di flusso nei linguaggi a basso livello

Il controllo di flusso a basso livello si ottiene cambiando, a runtime, l'indirizzo della prossima istruzione da eseguire

- **PC** (program counter) = indirizzo della prossima istruzione da eseguire
- Di default PC viene automaticamente incrementato per andare all'istruzione successiva: **PC ← PC+4** (ricorda: la differenza tra indirizzi contigui è 4 byte)
- Modifica del flusso: in PC viene scritto il target address di un'istruzione diversa dalla successiva

Come possiamo farlo? Con due tipologie di istruzioni:

- Salti incondizionati, detti **«jump»**: cambiano sempre l'indirizzo della prossima istruzione
- Salti condizionati, detti **«branch»**: cambiano l'indirizzo della prossima istruzione se si verifica una data **condizione**

Salti Incondizionati (Jump)

- **Incondizionato** significa che il salto viene **sempre** eseguito
- Istruzioni: `j` (jump), `jr` (jump register)

```
j   INDIRIZZO  # salta a un dato indirizzo
```

Esempio:

```
J 0x00400084
```

```
jr  $rx       # salta all'indirizzo contenuto in $rx
```

Esempio:

```
la $s1 0x00400084
jr $s1
```

Salti Incondizionati (Jump)

- **Incondizionato** significa che il salto viene **sempre** eseguito
- Istruzioni: `j` (jump), `jr` (jump register)

```
j   INDIRIZZO  # salta a un dato indirizzo
```

Esempio:

```
J 0x00400084
```

?

```
jr  $rx       # salta all'indirizzo contenuto in $rx
```

Esempio:

```
la $s1 0x00400084
jr $s1
```

?

Ma come facciamo a conoscere l'indirizzo delle istruzioni a cui vogliamo saltare mentre scriviamo il nostro programma? Con le **label**

Label

- Se scriviamo "`Label1:` *element*" Assembler assocerà l'identificatore `Label1` **all'indirizzo** di *element*
- *element* può essere un **dato** o un'**istruzione**, quindi le label possono essere usate sia nel segmento dati che nel segmento testo

Le posso dichiarare così

Nel codice **array1** indicherà l'indirizzo di un array con 4 interi che sta nel segmento dati

Nel codice **blocco1** indicherà l'indirizzo della **add**

```
        .data
# dati ...
array1: .word 45 67 -3 7
# dati ...

        .text
        # istruzioni ...
blocco1:
        add $t0 $t0 $t1
        li $t2 4
        mul $t0 $t0
        # istruzioni ...

        la $s0 array1
        j blocco1
```

Le posso usare così

Carico un indirizzo nel registro

Salto alla **add**

Esempio

```
        .text
        .globl main
main:
        li $t0 4
        li $t1 5
        j qui
        li $t0 0
qua:
        li $t1 0
qui:
        add $t0 $t1 $t0
        j qua
```

Quanto vale t0 alla fine?

Esempio

```
        .text
        .globl main
main:
        li $t0 4
        li $t1 5
        j qui
        li $t0 0
qua:
        li $t1 0
qui:
        add $t0 $t1 $t0
        j qua
```

Quanto vale t0 alla fine?

Non termina mai!! 😐

Proviamo a correggere…

Esempio

```
        .text
        .globl main
main:
        li $t0 4
        li $t1 5
        j qui
        li $t0 0
qua:
        li $t1 0
        j end
qui:
        add $t0 $t1 $t0
        j qua
end:
```

Quanto vale t0 alla fine?

Esempio

```
        .text
        .globl main
main:
        li $t0 4
        li $t1 5
        j qui
        li $t0 0
qua:
        li $t1 0
        j end
qui:
        add $t0 $t1 $t0
        j qua
end:
```

Quanto vale t0 alla fine?

Risposta: 9

Jump in linguaggio macchina

- Il salto `j` (e anche `jal`, che vedremo poi) è un'istruzione J-type (J sta per Jump):

```
j   label
```

- Il **target address** è di 32 bit (come ogni indirizzo in MIPS32)

Problema: nell'istruzione ci sono solo 26 bit per specificare il target address
Soluzione: indirizzamento **pseudo-diretto**:

1. I bit in posizione 0 e 1 (i due meno significativi) sono **impliciti** ed uguali a 0 (allineamento)
2. I bit dalla posizione 2 alla 25 sono uguali ai 26 bit specificati nell'istruzione
3. I bit dalla posizione 26 alla 31 (i quattro più significativi) sono **impliciti** ed uguali ai quattro bit più significativi del PC

Effetto della jump:

Salti in linguaggio macchina MIPS

Il salto Jump:

- non può modificare i primi 4 bit del PC
  - per esempio, una jump all'indirizzo `0xC-------`
    può saltare solo ad un'altra istruzione di indirizzo `0xC-------`
  - Si dice che non può saltare «fuori dal blocco»

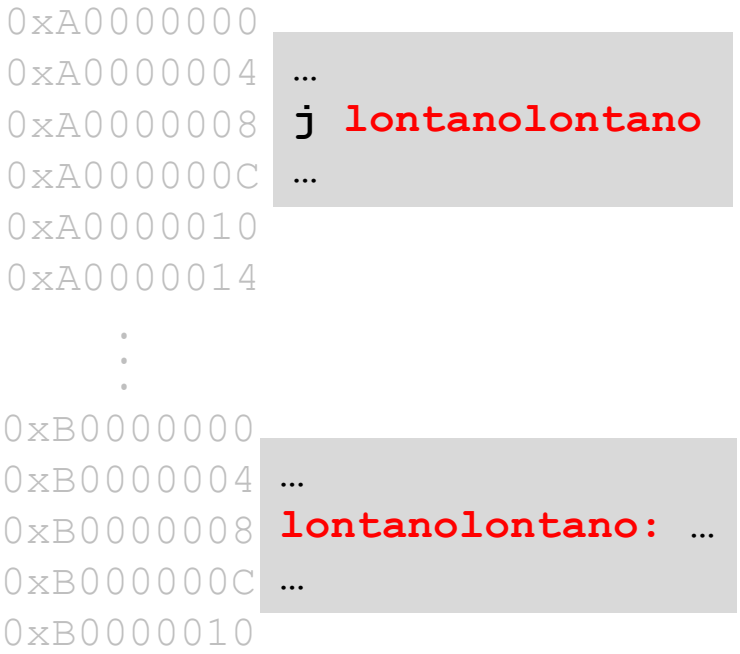
L'istruzione Jump Register non ha questa limitazione:

```
jr  $rx
```

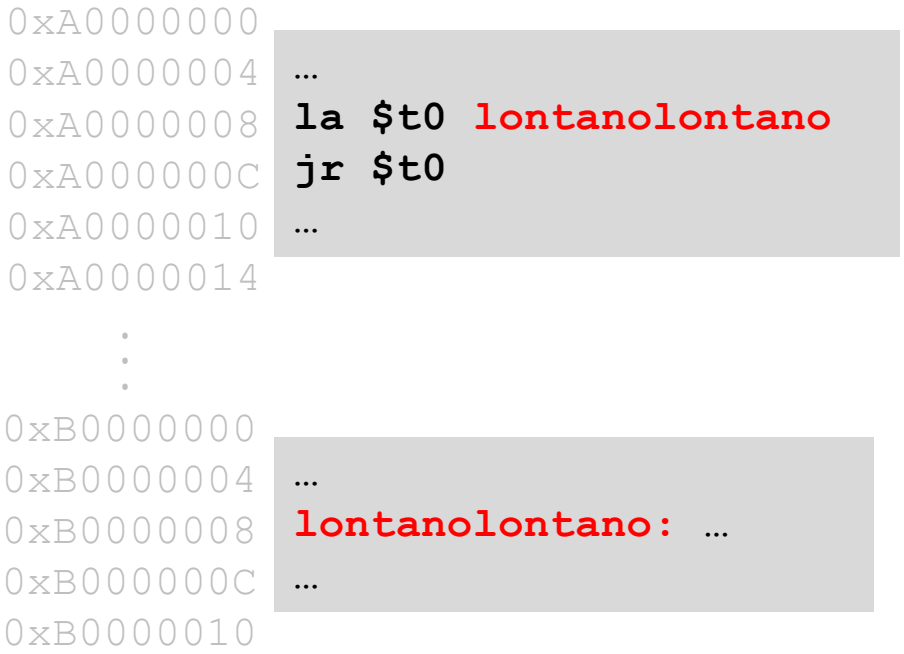
- Il target address sta dentro il registro $rx, non è un operando specificato dentro all'istruzione (nell'istruzione si specifica il numero di registro per cui bastano 5 bit)
- Non sarà Assembler a costruire il target address, dobbiamo farlo noi caricandone il valore nel registro $rx

Salti in linguaggio macchina MIPS

- Con jump register posso saltare «fuori dal blocco»



ERRORE: il target address è troppo distante



OK!

Branch – Bivio, Biforcazione

- Salto **condizionato**: viene eseguito solo se una certa **condizione** risulta verificata, altrimenti si continua normalmente con la prossima istruzione

- Esempio: **branch on equal**

```
beq $ra $rb label
```

- Se i registri `$ra` e `$rb` contengono lo stesso valore,
  allora salta all'istruzione memorizzata all'indirizzo rappresentato da *label*

Instruzioni di Branch

- Con confronto fra due registri

beq `$ra $rb` *addr*	branch on *equal*	$ra = $rb
bne `$ra $rb` *addr*	branch on *not equal*	$ra ≠ $rb
blt `$ra $rb` *addr*	branch on *less then*	$ra < $rb

- Con confronto fra registro e zero

bgez `$ra` *addr*	branch on *greater-or-equal zero*	$ra ≥ 0
bgtz `$ra` *addr*	branch on *greater-than zero*	$ra > 0
blez `$ra` *addr*	branch on *less-or-equal to zero*	$ra ≤ 0
bltz `$ra` *addr*	branch on *less-than zero*	$ra < 0

Branch in linguaggio macchina

- I Branch sono istruzioni I-type (I sta per Immediate)

```
beq $ra $rb label
```

Problema: nell'istruzione ci sono solo 16 bit per specificare il target address
Soluzione: indirizzamento **relativo al PC (PC-relative)**:

1. I bit in posizione 0 e 1 (i due meno significativi) sono **impliciti** ed uguali a 0 (allineamento)
2. I bit dalla posizione 2 alla 17 sono uguali ai 16 bit specificati nell'istruzione
3. I bit dalla posizione 17 alla 31 sono l'estensione del segno

Effetto della branch se il salto viene fatto:

Branch in linguaggio macchina

- L'offset sommato al PC è un numero in complemento a 2 ed è relativo all'**istruzione successiva** alla branch
- Massimo salto in avanti: $+4(2^{15}-1)$ bytes dall'istruzione successiva alla branch, quindi 2^{15} istruzioni **dopo** quella corrente
- Massimo salto all'indietro: $-4(2^{15})$ bytes dall'istruzione successiva alla branch quindi $2^{15}-1$ istruzioni **prima** di quella corrente
- Sono salti «corti», ma si può uscire dal blocco. Ad esempio posso saltare da 0x**A**FFFFFFE a di 0x**B**0000000

Nota: quando scrivo `beq $ra $rb label`
Assembler fa per noi il lavoro di ricostruire l'offset di 16 bit a partire dalla label che ho specificato:

- Sottrae all'indirizzo specificato dalla label l'indirizzo dell'istruzione successiva al branch
- se l'indirizzo target è troppo distante ($>2^{15}$) genera un errore

Posso saltare lontano condizionalmente?

- Sì, combinandolo con jump:

```
0xA0000000
0xA0000004   …
0xA0000008   bgez $t0 far
0xA000000C   …
0xA0000010
0xA0000014
   . . .
0xA5130000
0xA5130004   …
0xA5130008   far: …
0xA513000C   …
0xA5130010
```

ERRORE! too far!

```
0xA0000000
0xA0000004   …
0xA0000008   bltz $t0 near
0xA000000C   j far
0xA0000010   near: …
0xA0000014
   . . .
0xA5130000
0xA5130004   …
0xA5130008   far: …
0xA513000C   …
0xA5130010
```

OK

Condizioni di disuguaglianza

- Spesso è utile condizionare l'esecuzione di un'istruzione al fatto che una variabile sia minore di un'altra, istruzione **Set Less Than**.

```
slt $s1, $s2, $s3
```

- Assegna il valore 1 (set) a `$s1` se `$s2 < $s3` altrimenti assegna il valore 0.
- Con `slt, beq` e `bne` si possono implementare tutti i test sui valori di due variabili `(==, !=, <, <=, >,>=)`.

Condizioni di disuguaglianza

- Si completi la seguente tabella con il corrispettivo codice assembly

Pseudo codice	Assembly
if($s1==$s2) addi $s3, $s3, 1	bne $s1, $s2, L addi $s3, $s3, 1 L:
if($s1!=$s2) addi $s3, $s3, 1	beq $s1, $s2, L addi $s3, $s3, 1 L:
if($s1>$s2) addi $s3, $s3, 1	slt $t0, $s2, $s1 bne $t0, 1, L addi $s3, $s3, 1 L: ble $s1 $s2 L addi $s3 $s3 1 L:
if($s1>=$s2) addi $s3, $s3, 1	bne $s1, $s2, T j A T: slt $t0, $s2, $s1 bne $t0, 1, L A: addi $s3, $s3, 1 L: blt $s1 $s2 L addi $s3 $s3 1 L:
if($s1<$s2) addi $s3, $s3, 1	slt $t0, $s1, $s2 bne $t0, 1, L addi $s3, $s3, 1 L: bge $s1 $s2 L addi $s3 $s3 1 L:
if($s1<=$s2) addi $s3, $s3, 1	bne $s1, $s2, T j A T: slt $t0, $s1, $s2 bne $t0, 1, L A: addi $s3, $s3, 1 L: bgt $s1 $s2 L addi $s3 $s3 1 L:

Alcune strutture di controllo di alto livello in Assembly

If - Then

Codice C:

```
if (i==j)
   f=g+h;
…
```

Si supponga che le variabili `f, g, h, i` e `j` siano associate rispettivamente ai registri `$s0, $s1, $s2, $s3` e `$s4`

- Riscriviamo il codice C in una forma equivalente, ma più «vicina» alla sua traduzione Assembly

```
if (i!=j)
   goto L;
f=g+h;
L:
…
```

Codice Assembly:

```
bne $s3, $s4, L                # if i ≠ j
go to L
add $s0, $s1, $s2
L:
…
```

If - Then - Else

Codice C:

```
if (i==j)
   f=g+h;
else
   f=g-h
…
```

Si supponga che le variabili `f, g, h, i` e `j` siano associate rispettivamente ai registri `$s0, $s1, $s2, $s3` e `$s4`

Codice Assembly:

```
bne $s3, $s4, Else
add $s0, $s1, $s2
j End
Else:
sub $s0, $s1, $s2
End:
…
```

Do - While

Codice C:

```
i=0;
do{
   g = g + A[i];
   i = i + j;
}
while (i!=h);
```

Si supponga che:

`g` e `h` siano in `$s1, $s2`

`i` e `j` siano in `$s3, $s4`

`A` sia in `$s5`

- Riscriviamo il codice C:


```
i = 0;
Loop:
g = g + A[i];
i = i + j;
if (i != h)
  goto Loop
```

Codice Assembly:

```
li $s3, 0
Loop:
mul $t1, $s3, 4
add $t1, $t1, $s5
lw $t0, 0($t1)
add $s1, $s1, $t0
add $s3, $s3, $s4
bne $s3, $s2, Loop
```

While

Codice C:

```
while (A[i]==k){
  i=i+j;
}
```

Si supponga che:

`i` e `j` siano in `$s3, $s4`

`k` sia in `$s5`

`A` sia in `$s6`

- Riscriviamo il codice C:

```
Loop:
If (A[i]!=k)
  go to End;
i=i+j;
go to Loop;
```

Codice Assembly:

```
Loop:
mul $t1, $s3, 4
add  $t1, $t1, $s6
lw  $t0, 0($t1)
bne $t0, $s5, End
add $s3, $s3, $s4
j Loop
End:
```

Il costrutto switch

- Può essere implementato con una serie di `if-then-else`
- *Alternativa: uso di una jump address table*

Codice C:

```
switch(k){
case 0:
   f = i + j;
   break;
case 1:
   f = g + h;
   break;
case 2:
   f = g - h;
   break;
case 3:
   f = i - j;
   break;
default:
   break;
}
```

```
if (k < 0)
   t = 1;
else
   t = 0;
if (t == 1)                          // k < 0
   goto Exit;
t2 = k;
if (t2 == 0)                         // k = 0
   goto L0;
t2--; if (t2 == 0)                   // k = 1
   goto L1;
t2--; if (t2 == 0)                   // k = 2
   goto L2;
t2--; if (t2 == 0)                   // k = 3
   goto L3;
goto Exit;                           // k > 3

L0:  f = i + j; goto Exit;
L1:  f = g + h; goto Exit;
L2:  f = g - h; goto Exit;
L3:  f = i - j; goto Exit;

Exit:
```

Il costrutto switch

- Si supponga che `$s0, …, $s5` contengano `f,g,h,i,j,k`,

Codice Assembly:

```
slt $t3, $s5, $zero
bne $t3, $zero, Exit

beq $s5, $zero, L0

addi $s5, $s5, -1
beq $s5, $zero, L1

addi $s5, $s5, -1
beq $s5, $zero, L2

addi $s5, $s5, -1
beq $s5, $zero, L3
```

```
j Exit;

L0: add $s0, $s3, $s4
j Exit

L1: add $s0, $s1, $s2
j Exit

L2: sub $s0, $s1, $s2
j Exit

L3: sub $s0, $s3, $s4
Exit:
```

Università degli Studi di Milano
Dipartimento di Informatica “Giovanni Degli Antoni”
Corso di Laurea Triennale in Informatica

Architettura degli Elaboratori II Laboratorio

Università degli Studi di Milano
Dipartimento di Informatica “Giovanni Degli Antoni”
Corso di Laurea Triennale in Informatica

Architettura degli Elaboratori II Laboratorio

Procedure 1/2

Procedure

- Programmando ad alto livello, spesso organizziamo il programma in unità funzionali dette **procedure** (o anche, nei vari linguaggi, funzioni, routines, subroutines, sottoprogrammi …)

- Esempi:
  - procedura che volge in maiuscolo una data stringa
  - procedura che calcola l'interesse cumulato di una certa somma di denaro
  - procedura che legge il nome dell'utente da tastiera
  - procedura che verifica una password

- le procedure vengono **invocate** all'occorrenza, ogni volta che sia necessario
  - dal programma principale
  - da un'altra procedura

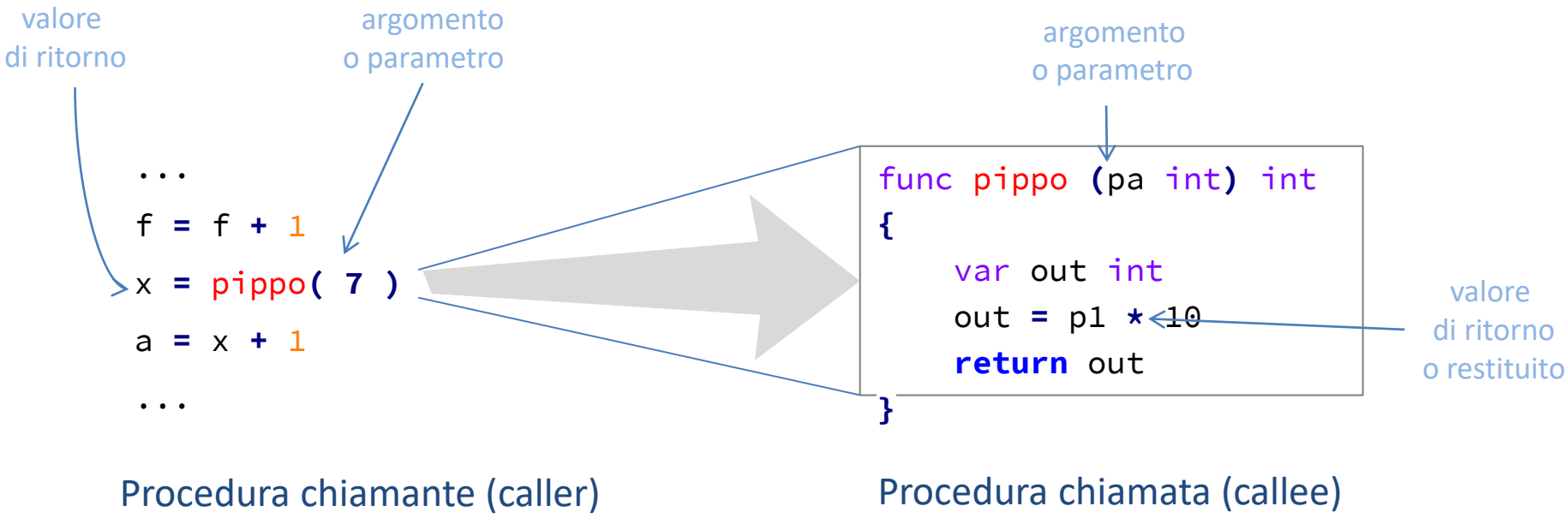
Procedure

- Chi implementa una procedura (ne scrive il codice) e chi la utilizza (scrive un codice che la invoca) sono spesso persone diverse, ad esempio:
  - l'autore di una libreria scrive una procedura
    gli utente della libreria la utilizza
  - membri diversi di un team di sviluppo si accordano sulle procedure da usare,
    e uno sviluppatore scrive codice «come se» le funzioni esistessero già,
    mentre un altro scrive le procedure

- I linguaggi ad alto livello impongono regole fisse con cui sviluppatore e utilizzatore possono coordinarsi. Per esempio, la sintassi con cui dichiarare e invocare una procedura. Se non rispettiamo queste regole il codice non compila o genera errori

- **A basso livello**, non esistono regole! Si adottano invece una serie di convenzioni **autoimposte**: sta al programmatore (noi), o al compilatore, rispettarle

Chiamata a procedura ad alto livello (es: in Go)

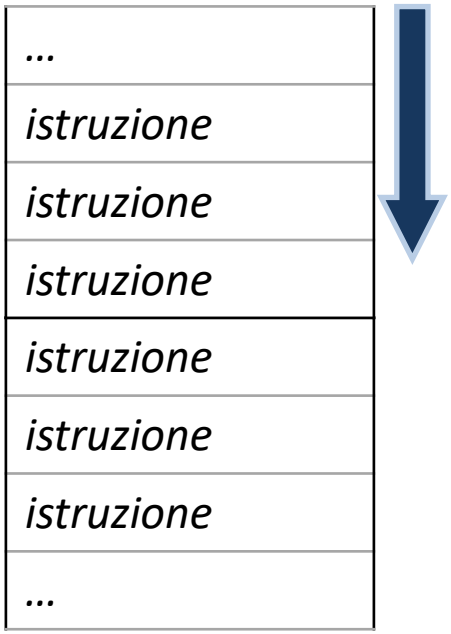


Caller e callee interagiscono attraverso:

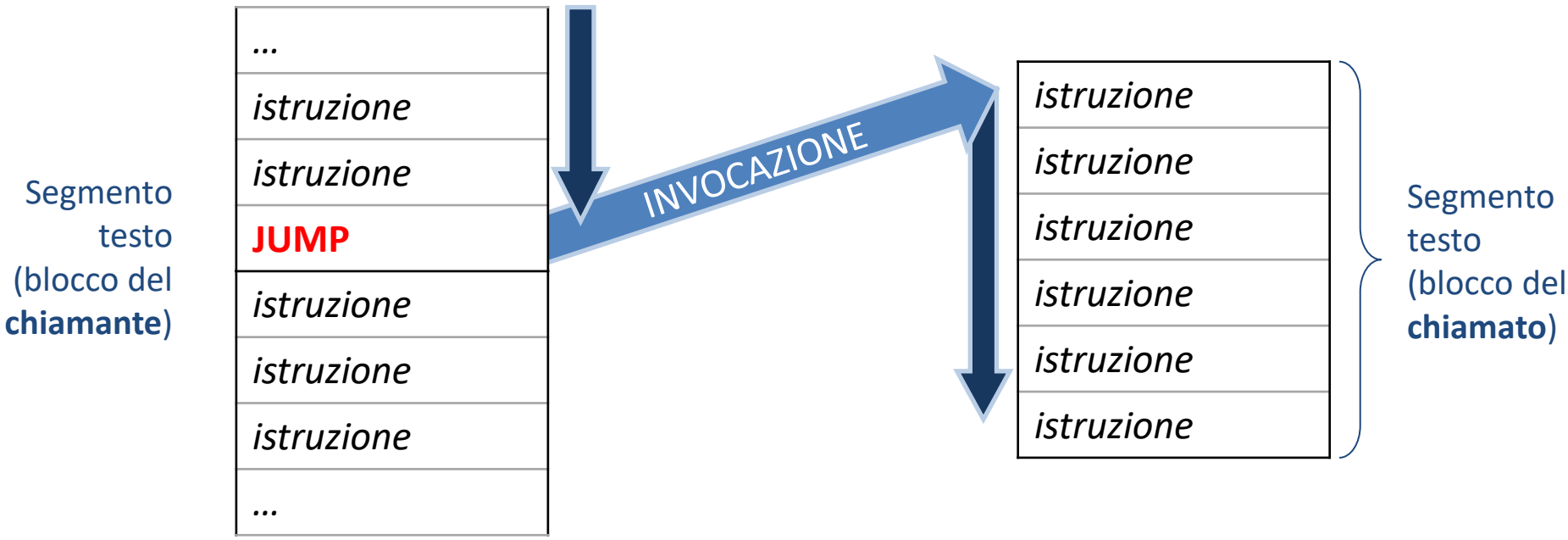
- passaggio di parametri di input (dal caller al callee)
- ritorno di valori di output (dal callee al caller)

Chiamata a procedura a basso livello

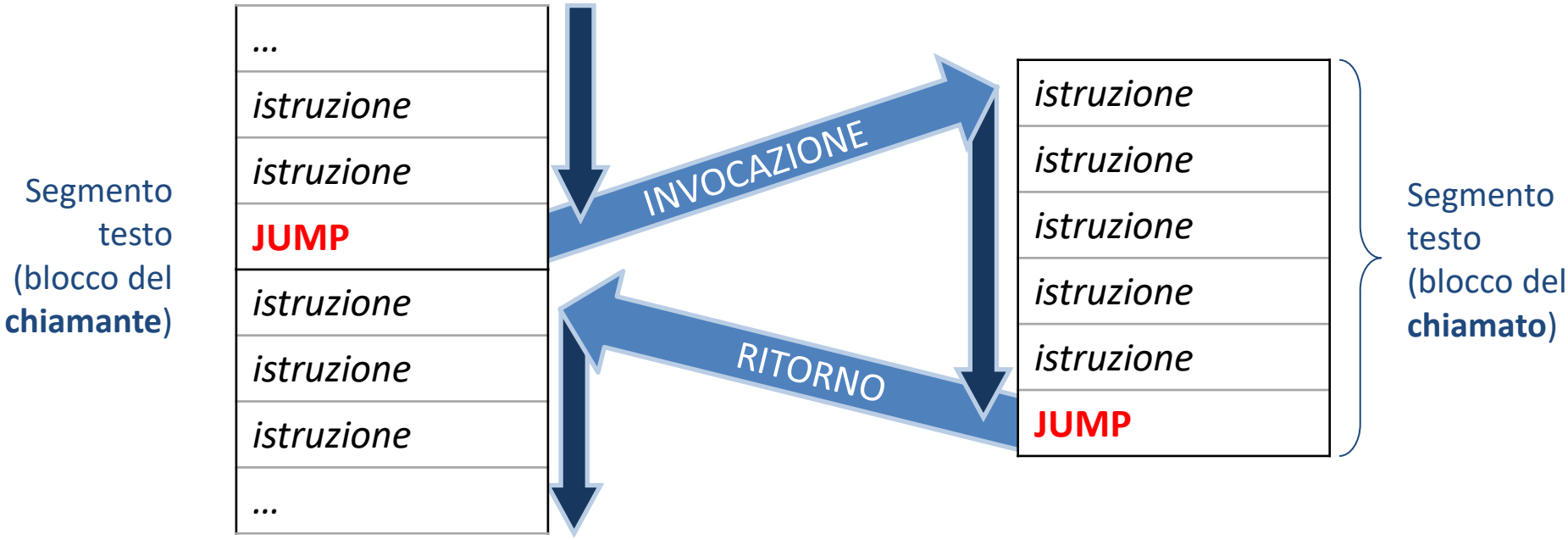
Segmento testo (blocco del **chiamante**)



Chiamata a procedura a basso livello



Chiamata a procedura a basso livello



JAL: Jump and Link

Registro:	$0	$1	$2	$3	$4	$5	$6	$7
Sinonimo:	**$r0**	**$at**	**$v0**	**$v1**	**$a0**	**$a1**	**$a2**	**$a3**
Registro:	$8	$9	$10	$11	$12	$13	$14	$15
Sinonimo:	**$t0**	**$t1**	**$t2**	**$t3**	**$t4**	**$t5**	**$t6**	**$t7**
Registro:	$16	$17	$18	$19	$20	$21	$22	$23
Sinonimo:	**$s0**	**$s1**	**$s2**	**$s3**	**$s4**	**$s5**	**$s6**	**$s7**
Registro:	$24	$25	$26	$27	$28	$29	$30	$31
Sinonimo:	**$t8**	**$t9**	**$k0**	**$k1**	**$gp**	**$sp**	**$s8**	**$ra**

RETURN ADDRESS

Salti di invocazione e ritorno

- In MIPS, un registro è dedicato a memorizzare l'indirizzo di ritorno:
  - **`$ra`** : «Return Address»

- Le istruzioni di jump hanno una variante «and link» che,
  prima di sovrascrivere il PC (per fare il salto),
  salvano in **`$ra`** il valore PC+4
  (cioè l'indirizzo a cui tornare al rientro dalla procedura):
  - **`jal`** `<label>` : Jump-and-link (j con link)
  - **`jalr`** `<registro>` : Jmp-and-link register (jr con link)

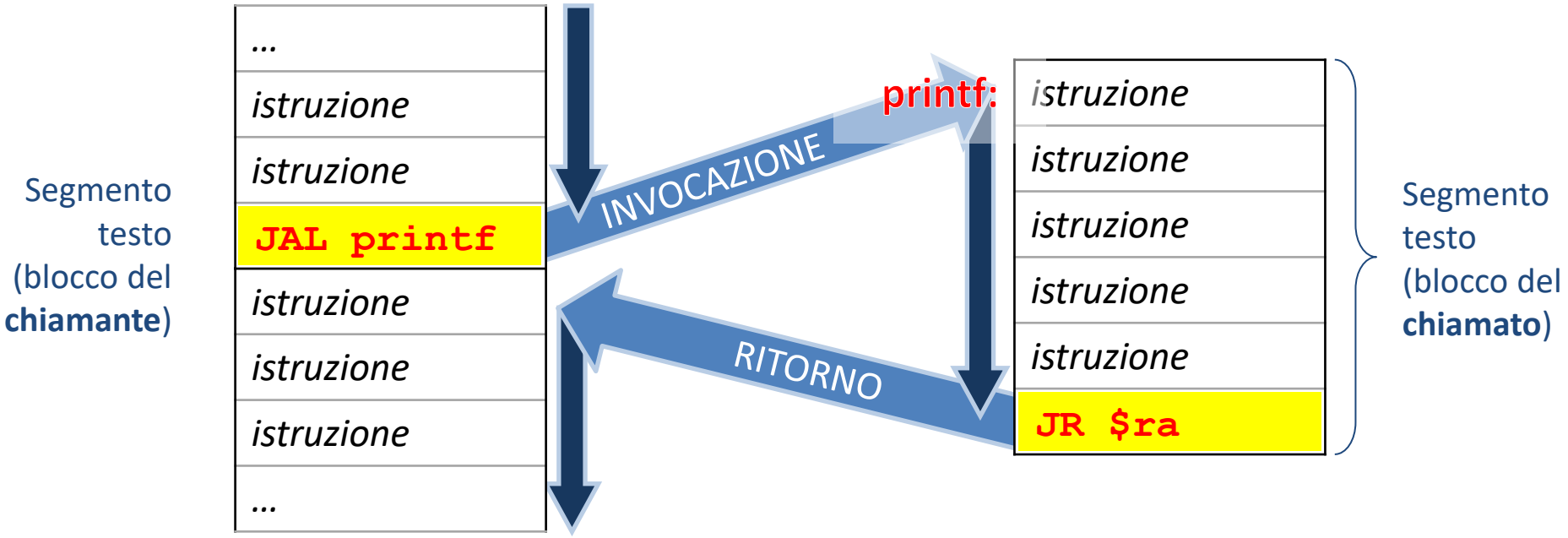
Salti di invocazione e ritorno

Salto di invocazione:

- Indicare l'inizio della procedura con una etichetta
- Saltare con una **`jal`** a quell'etichetta

Salto di ritorno

- Saltare con una **`jr`** al Return address



Comunicare alla procedura i suoi parametri

- Molte procedure si aspettano degli input
  - es: una procedura che volge in maiuscolo una stringa deve sapere l'indirizzo della stringa su cui lavorare

- Ad alto livello, questi sono gli argomenti (o i parametri) della procedura

- In MIPS, dedichiamo alcuni registri a memorizzare questi argomenti: `$a0`, `$a1`, `$a2`, `$a3` (a = argomento)

- Convenzione: (che sta ai programmatori / compilatori / studenti rispettare)
  - Il chiamante mette i valori dei parametri in `$a0`..`$a3` prima di invocare la procedura (quelli necessari)
  - La procedura assumerà di trovare gli input necessari in `$a0`..`$a3`

Comunicare al chiamante il valore di ritorno

- Molte procedure restituiscono degli output
  - es: una procedura che calcola l'interesse cumulato deve comunicare al chiamate il valore calcolato

- Ad alto livello, questo è il valore di ritorno della procedura (uno o più)

- In MIPS, dedichiamo alcuni registri a memorizzare i valori di ritorno: `$v0`, `$v1` (v = valore di ritorno)

- Convenzione: (che sta a noi rispettare)
  - Prima di restituire il controllo, la procedura mette in `$v0` (e/o `$v1`) il valore/i da restituire
  - Al ritorno il caller assume di avere in `$v0` (e/o `$v1`) il valore/i restituito/i dalla procedura

Come facciamo se abbiamo bisogno di più di 4 argomenti?

- Si usa lo stack
- Dal 5to argomento il poi:
  - Il chiamante: prima di invocare, fa una push nello stack dell'argomento
  - Il chiamato: per prima cosa, prende l'arogmento dallo stack, con una pop

```
addi $sp $sp -4
sw $** ($sp)
```

```
lw $** ($sp)
addi $sp $sp 4
```

- Nota: chiamante e chiamato devono sapere che questo è il caso per ogni data funzione
  - Quindi, quanti argomenti servono oltre i primi 4
  - Lo stack viene compromesso irrimediabilmente se non si mettono d'accordo! (viene fatta una push o, peggio, una pop di troppo)
- Info: in altri ISA, questo è il modo convenzionale di passare *tutti* gli argomenti

Come facciamo se abbiamo bisogno di più di 2 valori di ritorno?

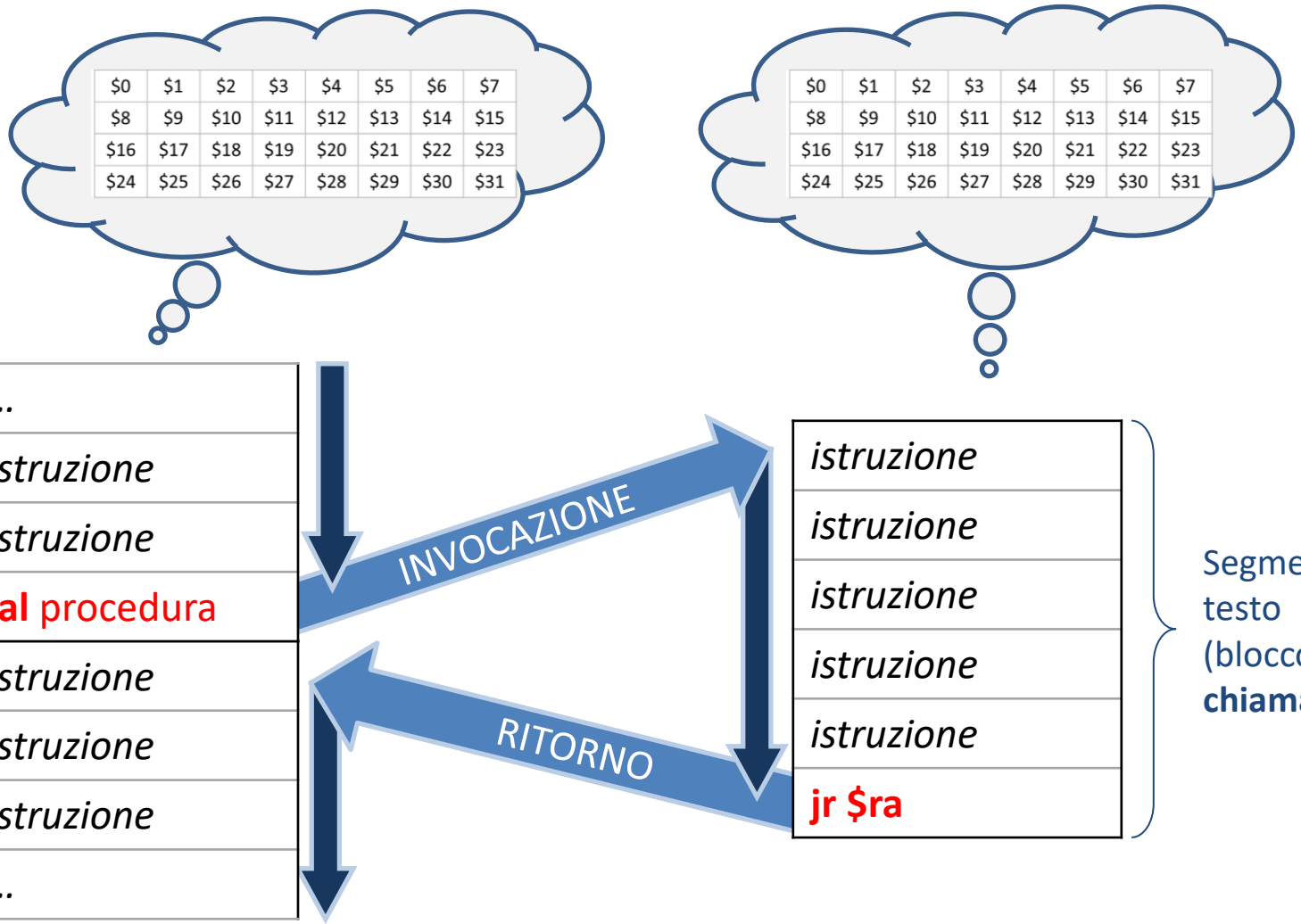
- Si usa lo stack
- Dal 3zo valore di ritorno il poi:
  - Il chiamato: prima di restituire il controllo, fa una push nello stack del valore prodotto
  - Il chiamante: dopo l'invocazione, prende il valore dallo stack, con una pop

```
addi $sp $sp -4
sw $** ($sp)
```

```
lw $** ($sp)
addi $sp $sp 4
```

- Nota: chiamante e chiamato devono sapere che questo è il caso per ogni data funzione
  - Quindi, quanti valori vengono restituiti oltre i primi due
  - Lo stack viene compromesso irrimediabilmente se non si mettono d'accordo! (viene fatta una push o, peggio, una pop di troppo)
- Info: in altri ISA, questo è il modo convenzionale di restituire *tutti* i valori

Problema:
i registri sono gli stessi!



Registri $s e $t

- Problema: i registri sono usati tanto dalla procedura quanto dal chiamante
  - Quindi, dopo una chiamata ad una procedura, il chiamante rischia di trovare i registri che stava utilizzando completamente cambiati («sporcati»)
- Ogni linguaggio assembly usa delle convenzioni per consentire a chiamante e chiamato usare i registri per non interferire
- In MIPS, adottiamo una convenzione:
  - gli otto registri **`$s0`** .. **`$s7`** (s = save) devono essere **preservati** dalla procedura: quando la procedura restituisce il controllo, il chiamante deve trovare in questi registri gli stessi valori che avevano al momento dell'invocazione
  - i dieci registri **`$t0`** .. **`$t9`** (t = temp) possono invece essere modificati da una procedura: il chiamante sa che invocare una procedura potrebbe modificare ("sporcare") questi registri

Convenzione sull'uso dei registri da parte delle procedure

Registro:	$0	$1	$2	$3	$4	$5	$6	$7
Sinonimo:	$r0	$at	$v0	$v1	$a0	$a1	$a2	$a3
Registro:	$8	$9	$10	$11	$12	$13	$14	$15
Sinonimo:	$t0	$t1	$t2	$t3	$t4	$t5	$t6	$t7
Registro:	$16	$17	$18	$19	$20	$21	$22	$23
Sinonimo:	$s0	$s1	$s2	$s3	$s4	$s5	$s6	$s7
Registro:	$24	$25	$26	$27	$28	$29	$30	$31
Sinonimo:	$t8	$t9	$k0	$k1	$fp	$sp	$s8	$ra

deve rimanere invariato

può essere modificato dalla procedura

Anche detti: registri caller-saved e callee-saved

Caller-saved

«salvati dal chiamante»
*sono i registri rispetto a cui **non** vige una convenzione di preservazione attraverso chiamate a procedura*

`$t0 … $t9, $a0 … $a3, $v0, $v1`

Un callee è libero di sovrascrivere questi registri: se un chiamante vuole essere sicuro di non perderne il valore deve salvarli sullo stack prima della chiamata a procedura

Esempio: `main` *ha un dato importante nel registro* `$t0`*, prima di invocare* `f` *salva* `$t0` *sullo stack, una volta riacquisito il controllo lo ripristina.*

Callee-saved

«salvati dal chiamato»
sono i registri rispetto cui la convenzione esige che vengano preservati attraverso chiamate a procedura

`$s0 … $s9 $ra $sp $fb`

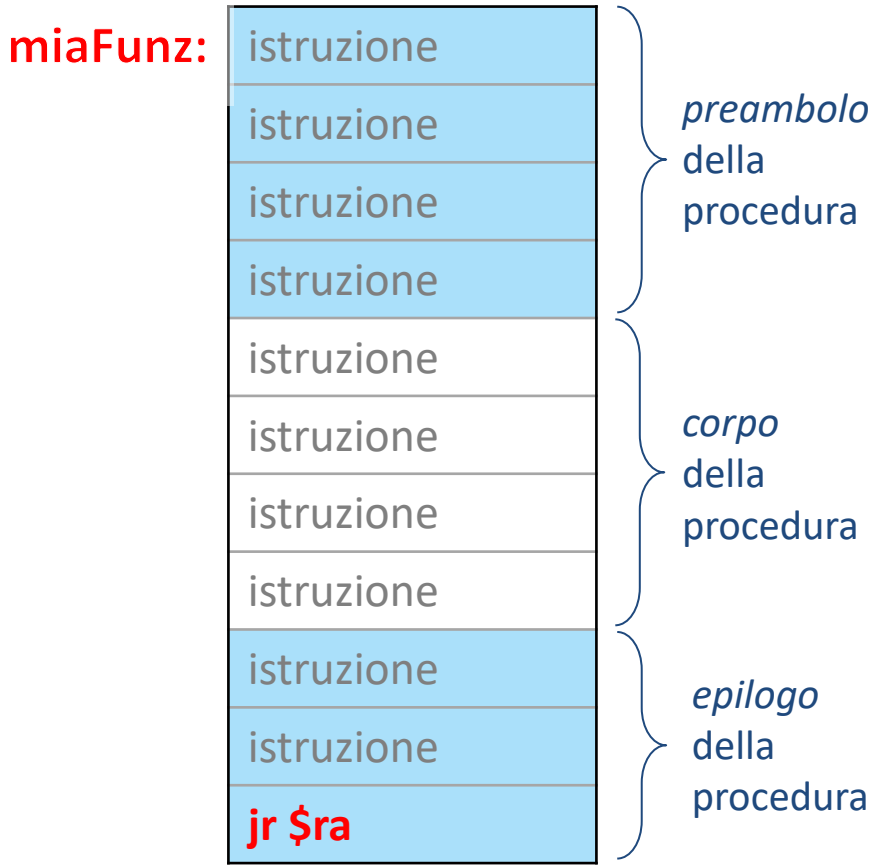
un callee non può sovrascrivere permanentemente questi registri: il chiamante si aspetta che restino invariati dopo la chiamata a procedura. Se il callee li vuole usare, deve prima salvarli sullo stack per poi ripristinarli una volta terminato

Esempio: `main` *ha un dato importante nel registro* `$s1` *e invoca* `f`*;* `f` *salva* `$s1` *sullo stack prima di utilizzarlo, una volta terminato lo ripristina.*

Registri $s e $t: per la procedura

La procedura può
rispettare la convenzione
attraverso diversi modi

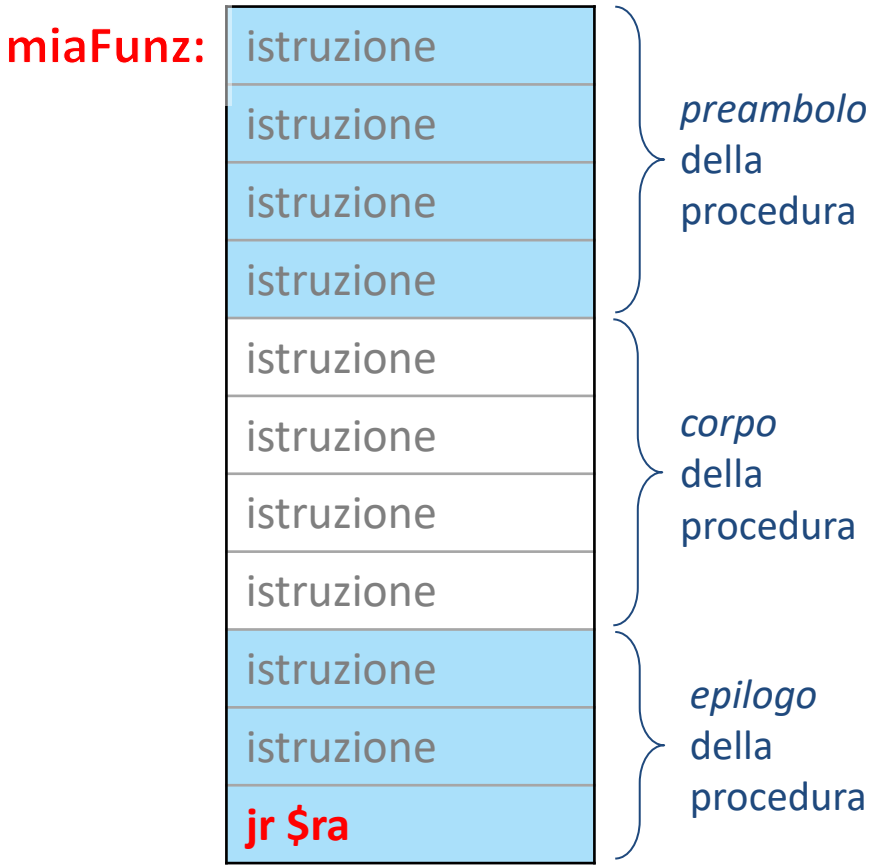
- Modo 1: non scrivere mai i registri $s
- Modo 2: salvare I registri $s nello stack



Registri $s e $t: per la procedura

La procedura può rispettare la convenzione attraverso diversi modi

- Modo 1: non scrivere mai i registri $s
  - usare quindi solo i registri $t
- Modo 2: salvare I registri $s nello stack
  - Prima di scrivere su un dato registro $s, (ad esempio, nel «*preambolo*» della proc.) o cmq salvarne una copia con una **push** nello stack
  - Poi, usare questi registri come normale
  - prima di restituire il controllo al chiamante, (ad esempio, nel «*epilogo*» della proc, subito prima della j $ra finale) ripristinare il valore originale di questi registri con una **pop**
  - Nota: dal punto di vista del chiamante, lo stack rimane inalterato



Conclusione: manuale per invocare una procedura

1. Caricare in $a0.. $a3 i parametri della procedura (se previsti)
2. Se necessario, salvare una copia dei registri $t nei registri $s oppure sullo stack frame (vale anche per $a0..$a3, $v0 e $v1)
3. Invocare la procedura: jal <label>
4. Trovare in $v0 (o $v1) l'eventuale valore restituito (se previsto)
5. Se necessario ripristinare il valore dei registri salvati nel passo 2

Manuale per scrivere una procedura

1. Preambolo:
   salvare una copia dei registri $s che si intende usare nello stack frame
   - con store word agli indirizzo $sp – 4, $sp – 8, $sp – 12, …

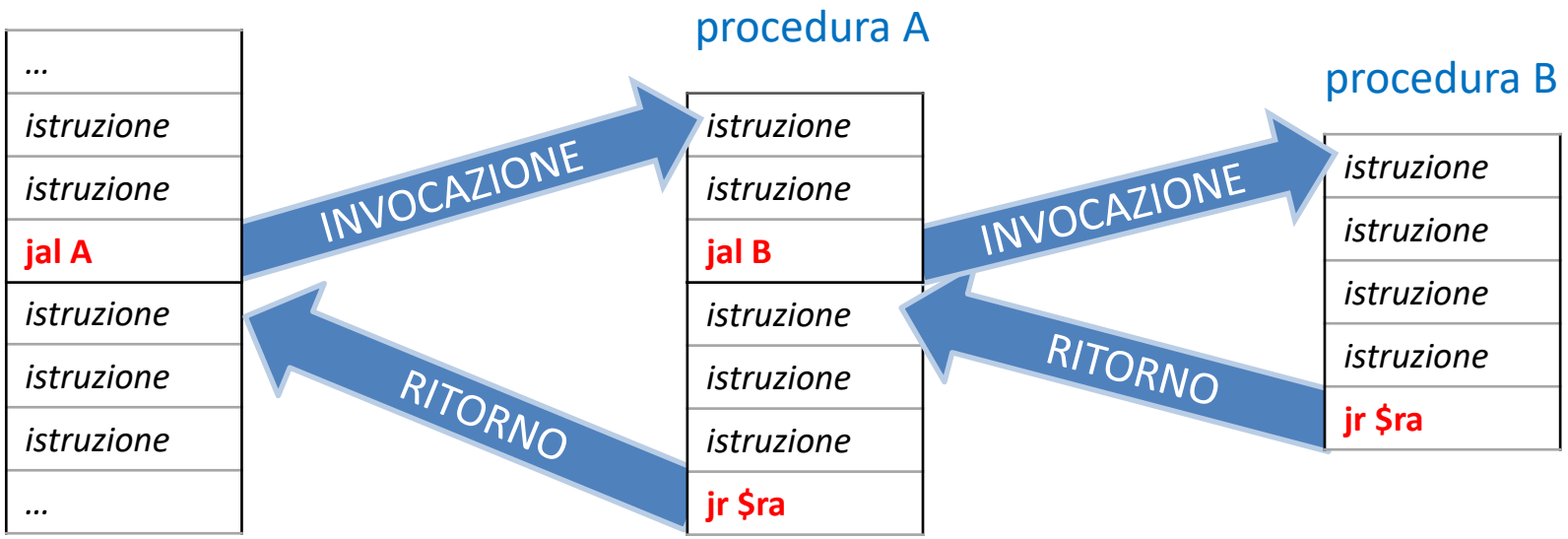
2. implementare la procedura (scrivere codice)
   - leggendo gli eventuali input da $a0 .. $a3
   - scrivendo liberamente su $t0..$t9
   - scrivendo su $s0..$s8 che siano stati salvati precedentemente
   - scrivere l'eventuale output in $v0 (e/o $v1)

3. Epilogo: ripristinare tutti i registri salvati nel passo 1
   - con altrettante load word agli stessi indirizzi

4. restituire il controllo al chiamante
   - jr $ra

nota: se la procedura ne invoca un'altra, la situazione si complica. Vedi prossima lezione

Prossima lezione: invocazione di procedura annidate



Università degli Studi di Milano
Dipartimento di Informatica “Giovanni Degli Antoni”
Corso di Laurea Triennale in Informatica

Architettura degli Elaboratori II Laboratorio

Università degli Studi di Milano
Dipartimento di Informatica “Giovanni Degli Antoni”
Corso di Laurea Triennale in Informatica

Architettura degli Elaboratori II Laboratorio

Procedure 2/2: Procedure annidate e ricorsive

Procedure «foglia»

- Scenario più semplice: `main` chiama la procedura `funct` che, senza chiamare a sua volta altre procedure, termina e restituisce il controllo al `main`

main

```
f = f + 1;

if (f == g)
        res = funct(f,g);

else
        f = f -1;

print(res)
```

funct

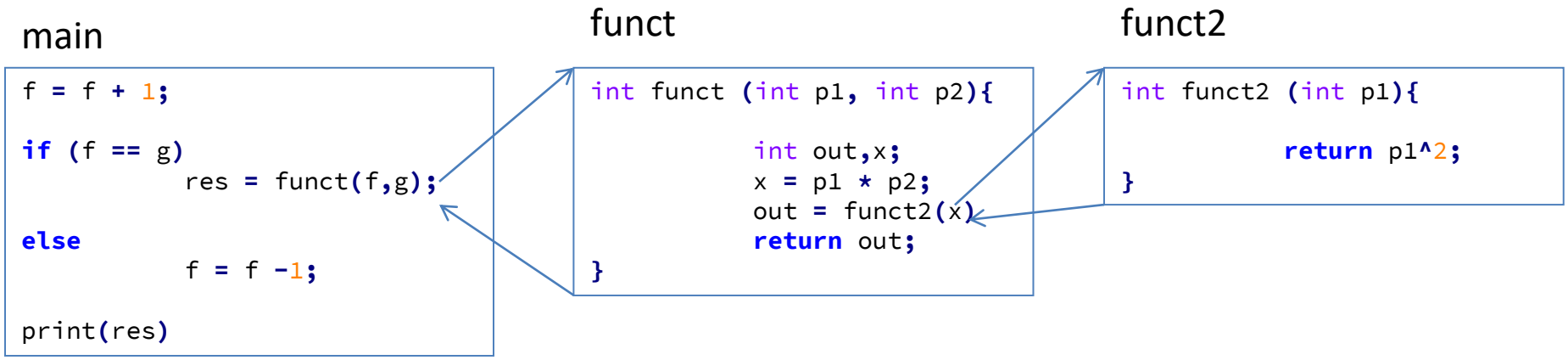
```
int funct (int p1, int p2){

        int out;
        out = p1 * p2;
        return out;
}
```

- Una procedura che non ne chiama un'altra al suo interno è detta procedura *foglia*

  Perché? Rappresentiamo le nostre procedure con un albero: le procedure diventano nodi e un arco tra due nodi x e y indica che x contiene almeno una chiamata a y

Procedure non «foglia»

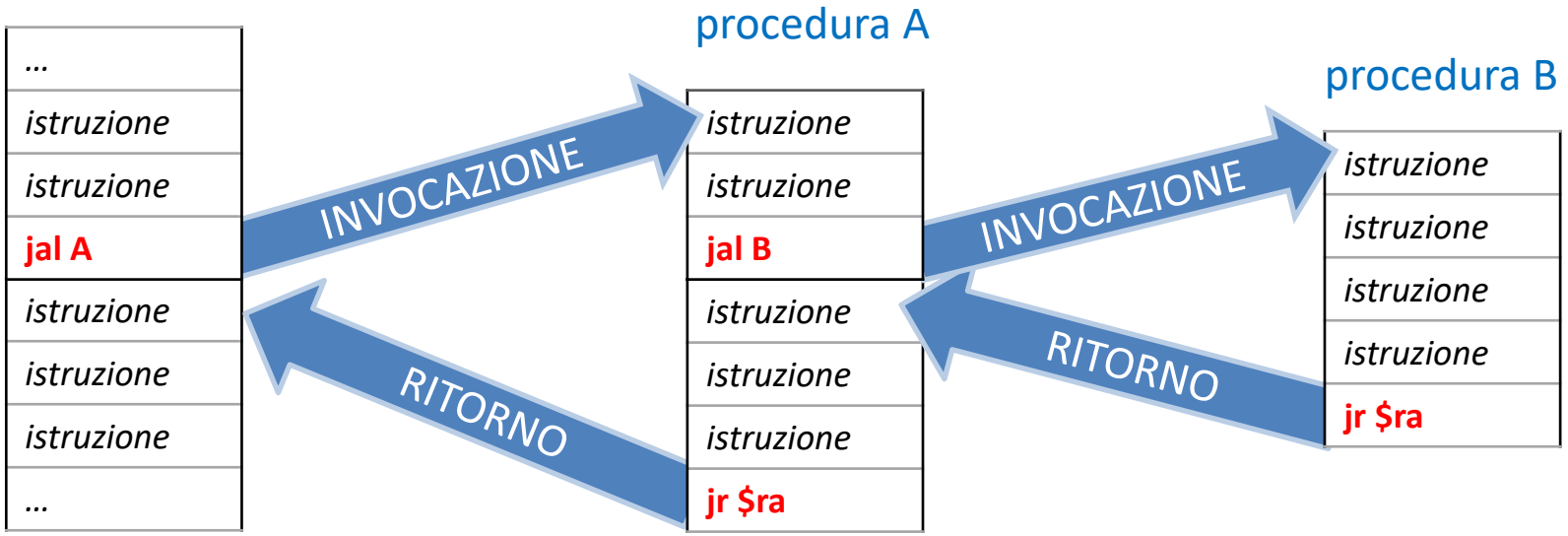
- Una procedura che può invocarne un'altra durante la sua esecuzione non è una procedura foglia, ha annidata al suo interno un'altra procedura:



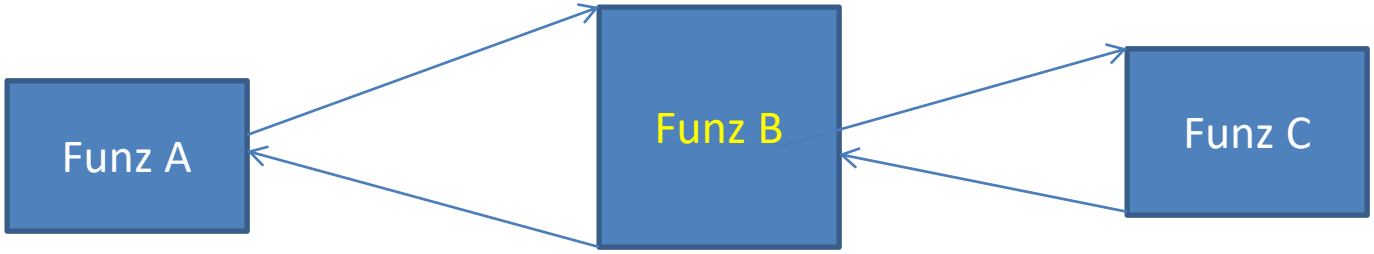
- Se una procedura contiene una chiamata ad un'altra procedura dovrà effettuare delle operazioni per (1) garantire la non-alterazione dei registri opportuni (2) consentire una restituzione del controllo consistente con l'annidamento delle chiamate.

- *Ricordiamo: in assembly la modularizzazione in procedure è un'assunzione concettuale sulla struttura e sul significato del codice. Nella pratica, ogni «blocco» di istruzioni condivide lo stesso register file e aree di memoria*

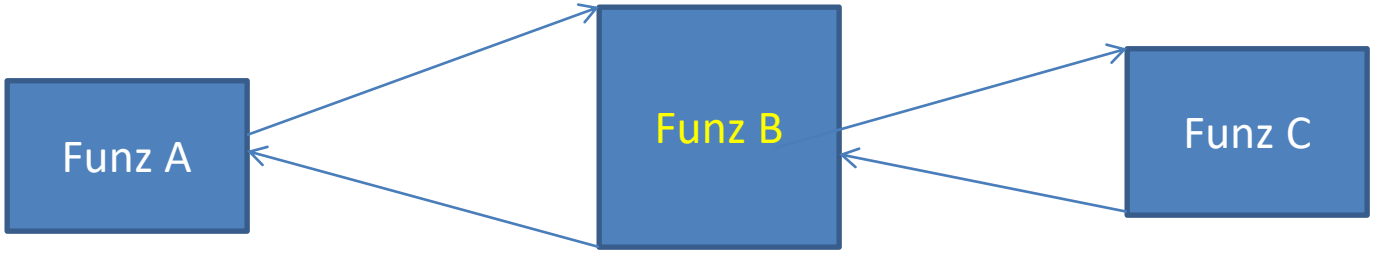
Invocazione di procedura annidate



Problemi per
le procedure non «foglia»



Problemi per le procedure non «foglia»



La funzione B ha alcuni problemi da risolvere…
Problema 1:

- Se B usa registri $t questi vengono (potenzialmente) distrutti da C, lecitamente ☹
- Se B usa registri $s (in scrittura), contravviene al «contratto» con A ☹
- Quali registri deve usare B?

Problema 2:

- Quando B usa la JAL per invocare C, sovrascrive il $ra
- Al momento di tornare ad A, non ha più l'indirizzo di ritorno!

Local variables (in Go)

```
func pippo() {
    var a , b , c int
    a = 10
    b = 20
    c = a + b
    if a%2 == 0 {
        var d , e , f int
        ...
        for j := 7; j <= 9; j++ {
            k := j+3
            fmt.Println(k)
        }
    } else {
        pippo := 6
        …
    }
    …
}
```

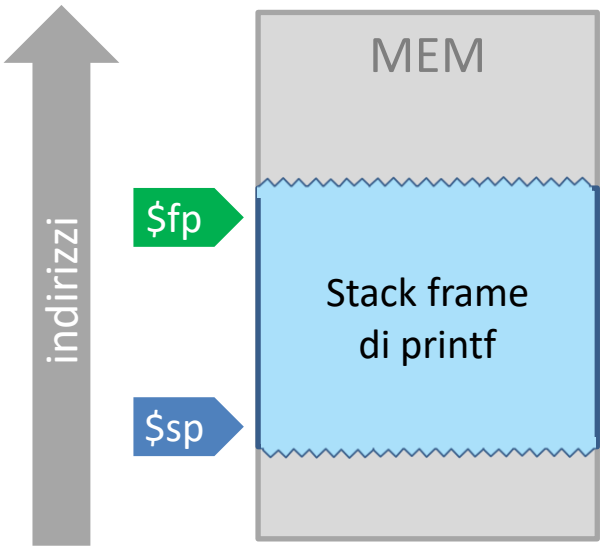
Nuove variabili locali sono aggiunte in vari punti dell'esecuzione di una funzione (compreso il main)

Record di attivazione – Stack frame

- Una procedura ha bisogno di usare la memoria
  - Per memorizzare le sue variabili locali
  - Per memorizzare la copia dei registri da preservare

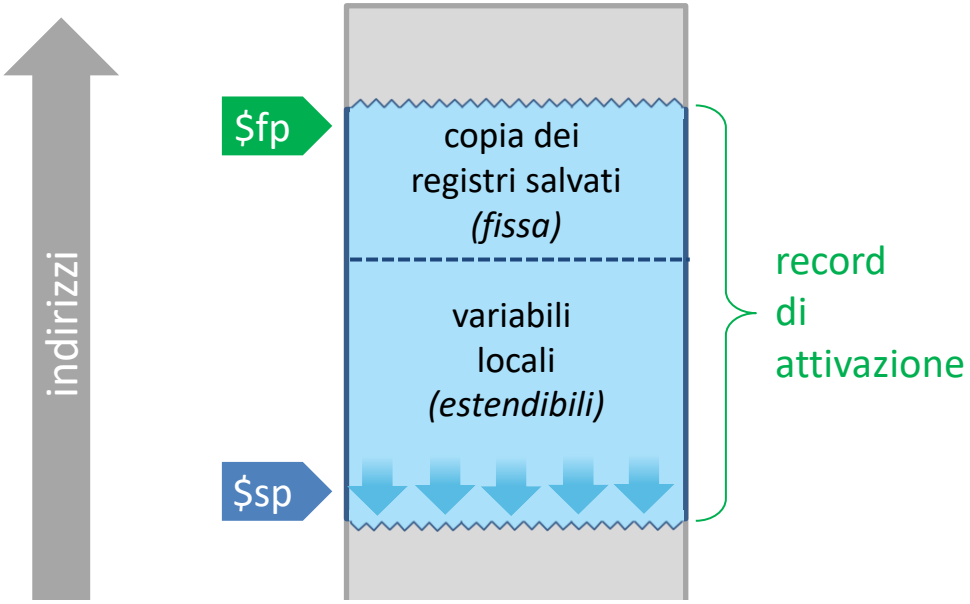
- Dedichiamo ad ogni procedura in esecuzione una sua area di memoria *sullo stack*, detta record di attivazione o stack frame

- MIPS riserva due registri per indirizzare lo stack frame della procedura attualmente in esecuzione:
  da `$sp` (stack pointer)
  a `$fp` (frame pointer)
  compresi!



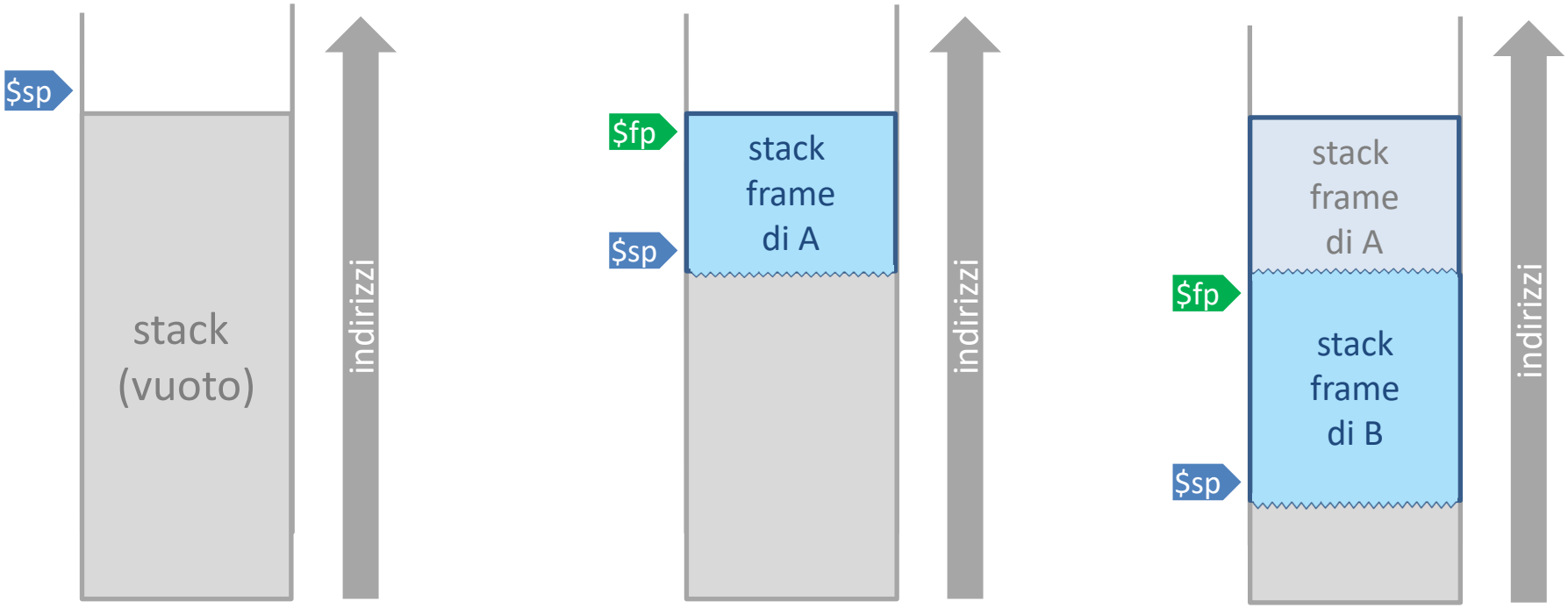
Il record di attivazione di una funzione

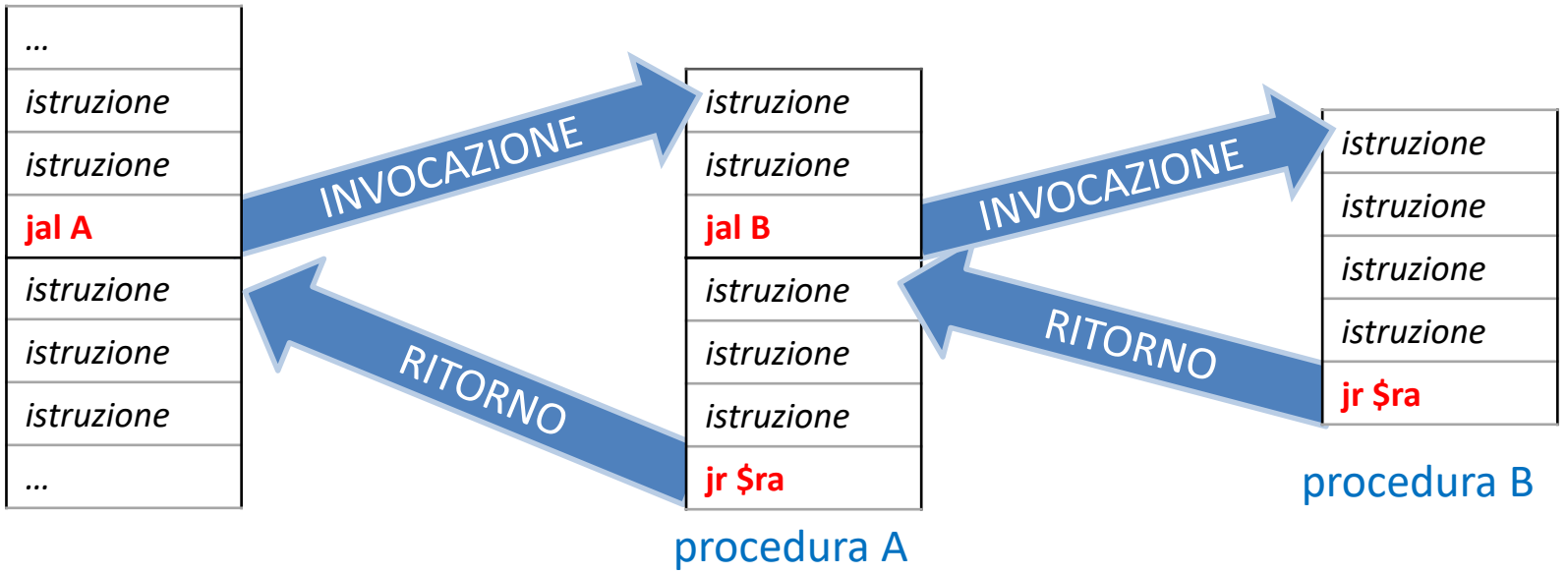
- Il record di attivazione di una procedura memorizza
  - La copia dei registri da preservare per il chiamante
  - Le variabili locali (attaverso push e pop, come normale)



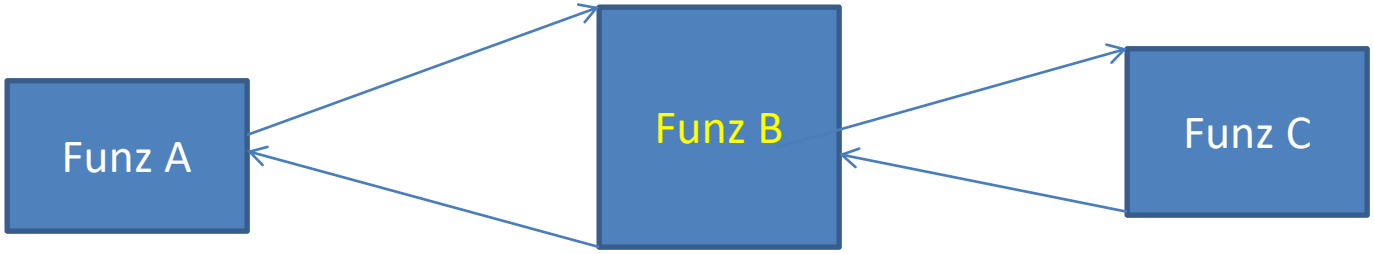
Allocazione e deallocazione degli stack frame

- I record di attivazione si impilano (LIFO) in memoria sullo stack
- quando una procedura viene invocata,
  un nuovo record di attivazione viene impilato nello stack
  - sotto al precedente
  - modificando i registri $sp e $fp
- quando una procedura termina, il suo record di attivazione (che è sempre quello inferiore) viene rimosso
  - modificando i registri $sp e $fp
  - nota: non è necessario «cancellare la memoria» semplicemente, l'area dello stack verrà riutilizzata dalle prossime procedure o variabili locali


$sp
stack
(vuoto)
indirizzi
$fp
stack
frame
di A
$sp
indirizzi
stack
frame
di A
$fp
stack
frame
di B
$sp
indirizzi


...
istruzione
istruzione
jal A
istruzione
istruzione
istruzione
...
INVOCAZIONE
RITORNO
istruzione
istruzione
jal B
istruzione
istruzione
istruzione
jr $ra
procedura A
INVOCAZIONE
RITORNO
istruzione
istruzione
istruzione
istruzione
jr $ra
procedura B

Problemi per le procedure non «foglia»



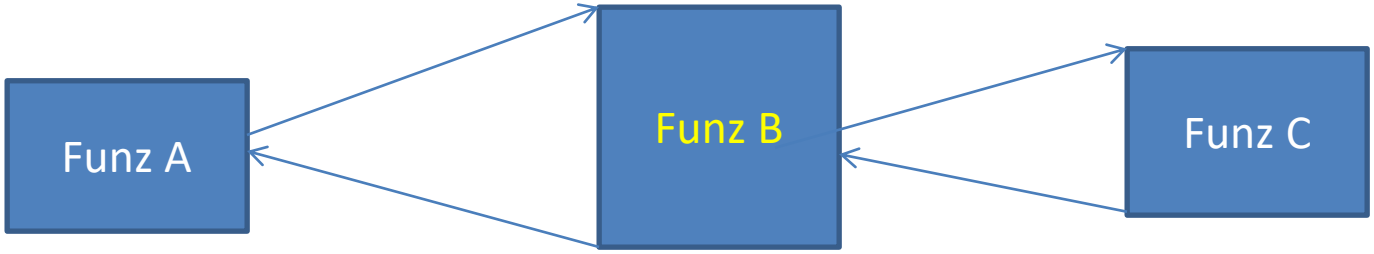
La funzione B ha alcuni problemi da risolvere…
Problema 1:

- Se B usa registri $t questi vengono (potenzialmente) distrutti da C, lecitamente ☹
- Se B usa registri $s (in scrittura), contravviene al «contratto» con A ☹
- Quali registri deve usare B?

Problema 2:

- Quando B usa la JAL per invocare C, sovrascrive il $ra
- Al momento di tornare ad A, non ha più l'indirizzo di ritorno!

Problemi per le procedure non «foglia»



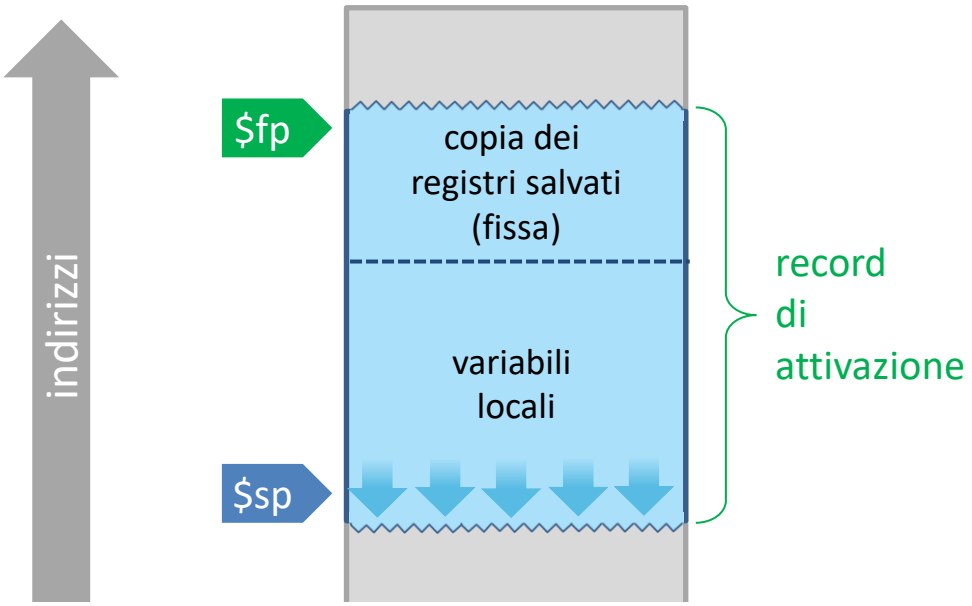
Soluzione ad entrambi i problemi:

Prima di usare i registri $s (ma anche $ra, e gli altri)
Funz B li memorizza nel proprio RECORD DI ATTIVAZIONE

La stessa strategia vale anche per $fp e $sp

$fp e $sp

- $sp può variare nel corso della procedura: viene decrementato quando una si allocano nuove variabili locali (compreso, da parte di sottofunzioni)
- $fp invece *non cambia* durante l'esecuzione della procedura
- $fp è utile a tener traccia di dove sono stati salvati i registri



Manuale per scrivere una procedura (versione completa)

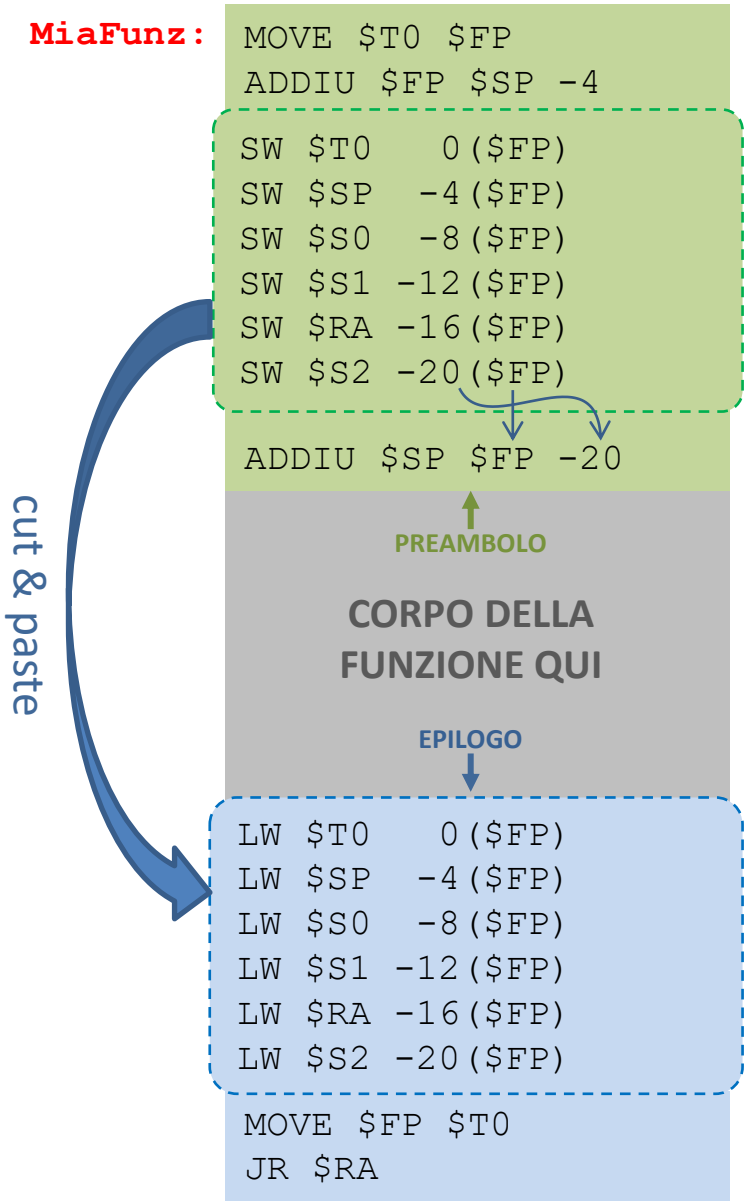
1. salvare una copia di $fp nel frame stack
   - con una store word ad indirizzo $sp – 4
2. aggiornare il valore di $fp a $sp -4
3. salvare una copia di $sp nello stack frame all'indirizzo $fp -4
4. salvare una copia dei registri $s nel frame stack
   - con un store word agli indirizzo, decre $fp – 12, $fp – 16, etc
   - solo quelli che si intende usare
5. salvare una copia di $ra nel frame stack
   - necessario solo se si invoca una funzione
6. aggiornare il valore di $sp
   - [dimensione record di attivazione] da $sp

Preambolo

7. implementare la procedura! (codice)
   - leggere gli eventuali input da $a0 .. $a3
   - scrivere l'eventuale output in $v0 (e/o $v1)
   - se servono nuove variabili locali, ingrandire il record di attivazione (decrementando $sp)
8. ripristinare tutti i registri salvati nel frame stack nei passi 1-5
   - con altrettante load word agli stessi indirizzi
   - $sp verrà ripristinato automaticamente
9. restituire il controllo al chiamante
   - jr $ra

Epilogo

Guida pratica per funzioni non-foglia



Copia temporanea del Frame Pointer *iniziale* (in T0).
Il *nuovo* record di attivazione comincia subito dopo il vecchio.

I valori dei registri *iniziali* sono i salvati (in qualsiasi ordine) nel (nuovo) record di attivazione.
Compreso lo stack pointer SP, il Return Address RA, e anche FP stesso (sotto forma di T0)

Aggiornamento dello SP
(punta sempre all'ultimo elemento occupato dello stack)

La funzione può

- usare i registri S solo se sono stati salvati (qui: s0, s1, s2).
- invocare altre funzioni (quindi usando RA),
- allocare variabili nello stack (quindi usano SP).
- usare i registri T, sapendo che non vengono mantenuti dopo l'invocazione di funzione

Recupero valore iniziale di tutti i registri salvati, compreso lo SP (flush dello stack)

… e compreso il FP
Ritorno al chiamante

Università degli Studi di Milano
Dipartimento di Informatica “Giovanni Degli Antoni”
Corso di Laurea Triennale in Informatica

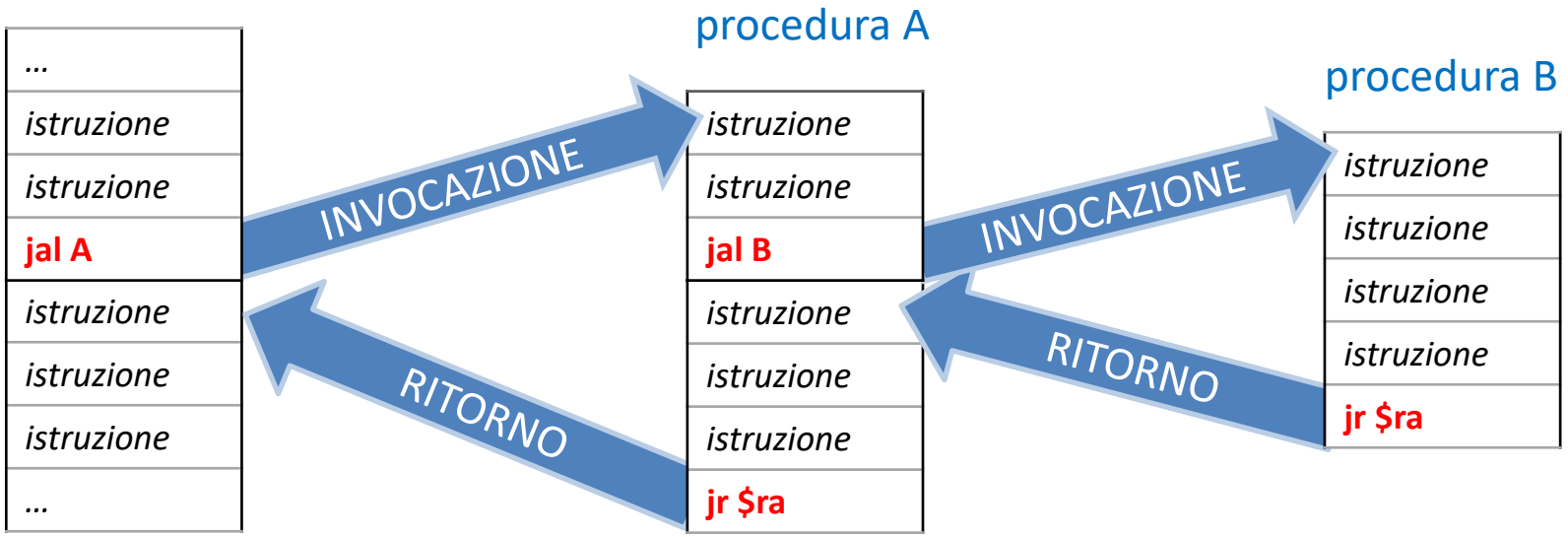
Architettura degli Elaboratori II Laboratorio

Università degli Studi di Milano
Dipartimento di Informatica “Giovanni Degli Antoni”
Corso di Laurea Triennale in Informatica

Architettura degli Elaboratori II Laboratorio

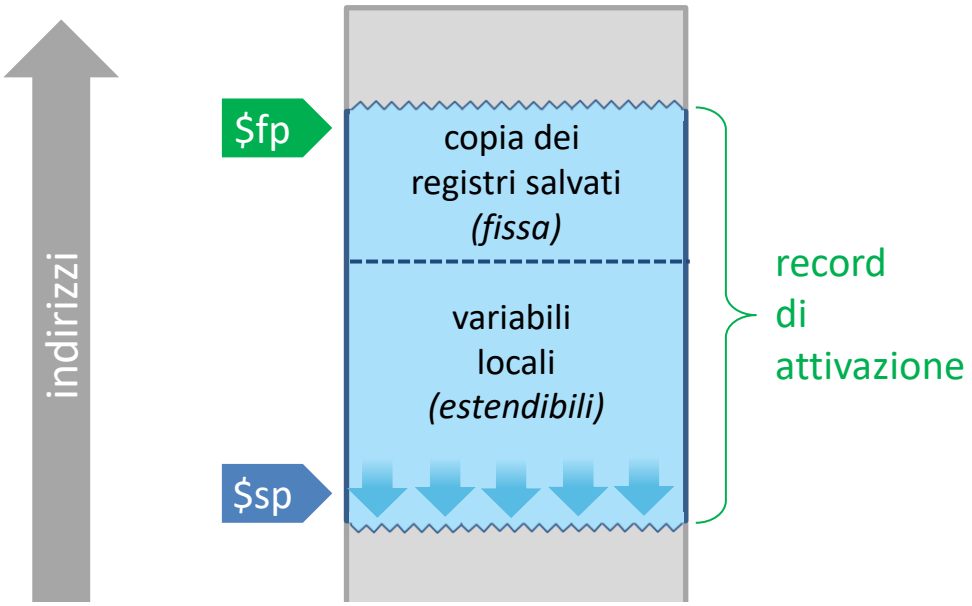
Procedure ricorsive

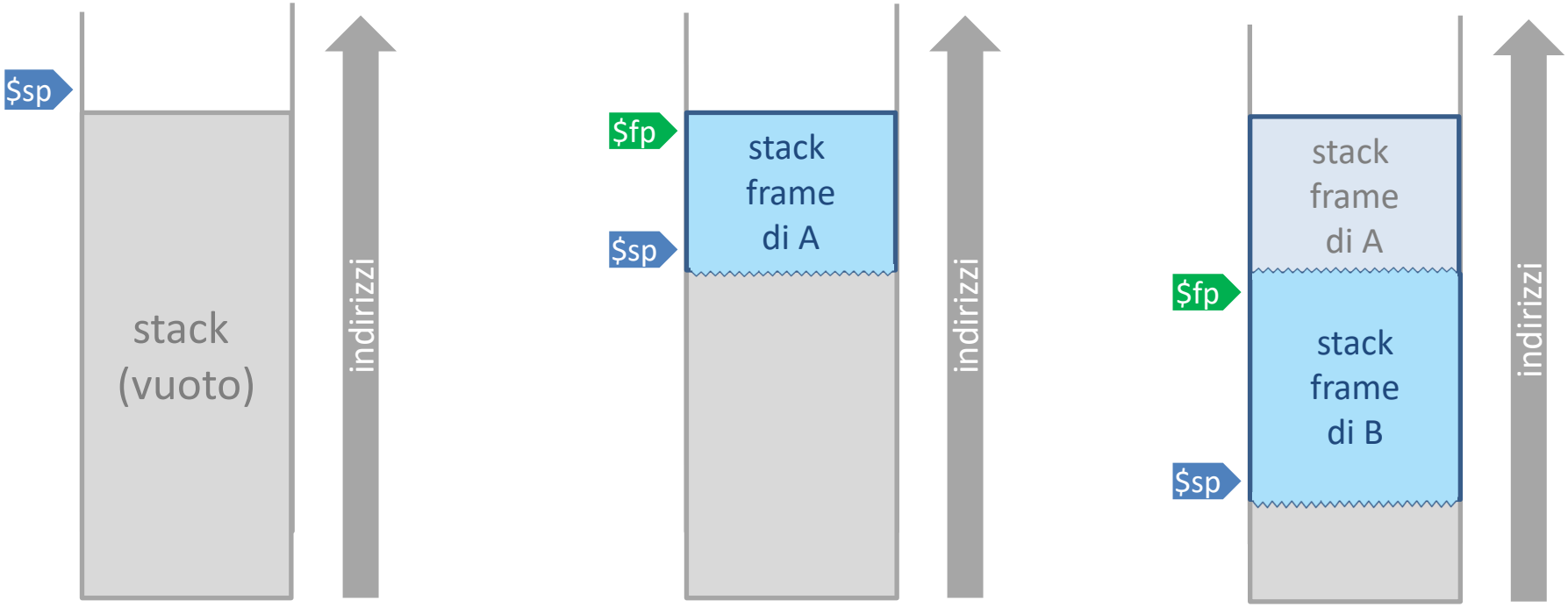
Invocazione di procedura annidate

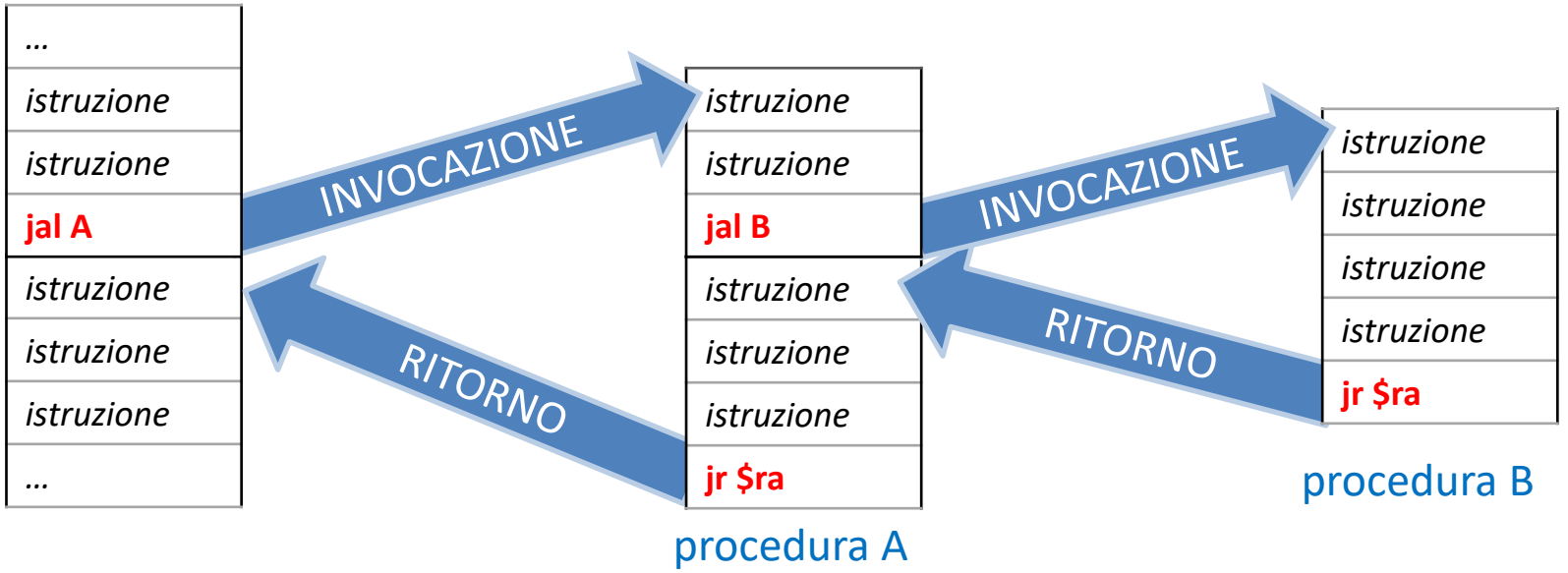


Il record di attivazione di una funzione

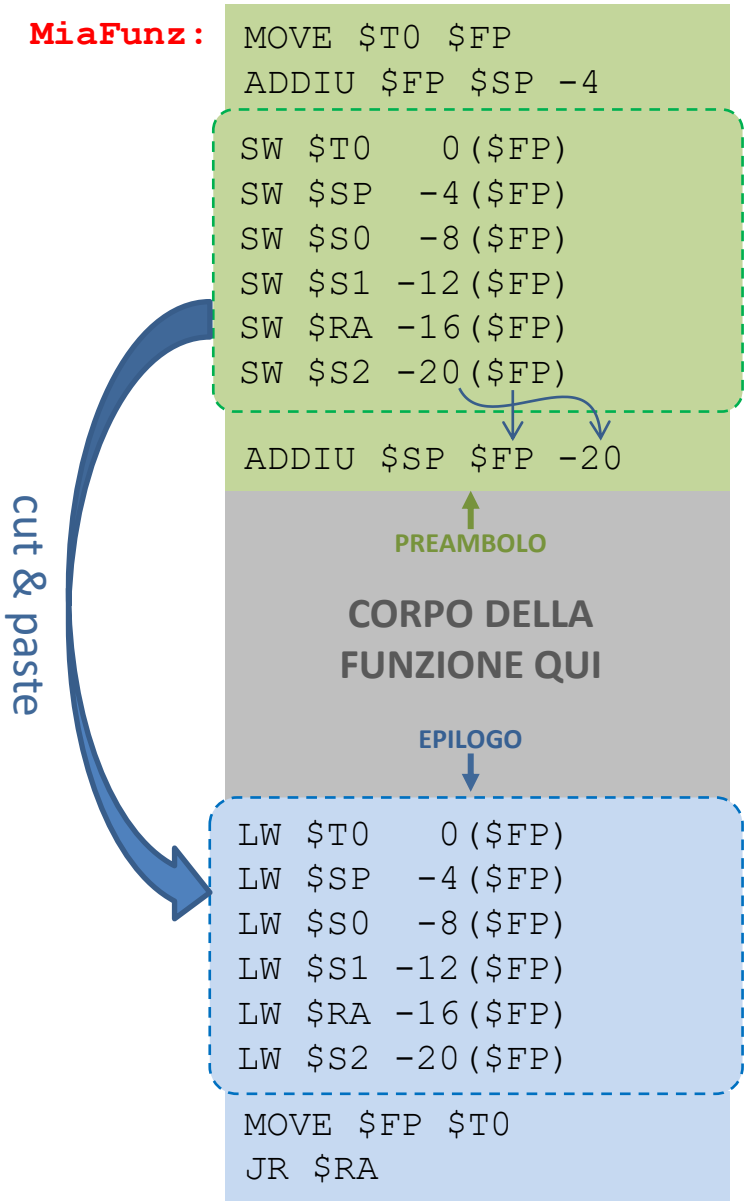
- Il record di attivazione di una procedura memorizza
  - La copia dei registri da preservare per il chiamante
  - Le variabili locali (attaverso push e pop, come normale)




$sp
stack
(vuoto)
indirizzi
$fp
stack
frame
di A
$sp
indirizzi
stack
frame
di A
$fp
stack
frame
di B
$sp
indirizzi


...
istruzione
istruzione
jal A
istruzione
istruzione
istruzione
...
INVOCAZIONE
RITORNO
istruzione
istruzione
jal B
istruzione
istruzione
istruzione
jr $ra
procedura A
INVOCAZIONE
RITORNO
istruzione
istruzione
istruzione
istruzione
jr $ra
procedura B

Guida pratica per funzioni non-foglia



Copia temporanea del Frame Pointer *iniziale* (in T0).

Il *nuovo* record di attivazione comincia subito dopo il vecchio.

I valori dei registri *iniziali* sono i salvati (in qualsiasi ordine) nel (nuovo) record di attivazione.
Compreso lo stack pointer SP, il Return Address RA, e anche FP stesso (sotto forma di T0)

Aggiornamento dello SP
(punta sempre all'ultimo elemento occupato dello stack)

La funzione può

- usare i registri S solo se sono stati salvati (qui: s0, s1, s2).
- invocare altre funzioni (quindi usando RA),
- allocare variabili nello stack (quindi usano SP).
- usare i registri T, sapendo che non vengono mantenuti dopo l'invocazione di funzione

Recupero valore iniziale di tutti i registri salvati, compreso lo SP (flush dello stack)

… e compreso il FP

Ritorno al chiamante

Ricorsione

- La risoluzione di un problema P è costruita sulla base della risoluzione di un sottoproblema di P

- Esempio classico: il fattoriale di n

$$n! = \prod_{k=1}^{n} k = n \prod_{k=1}^{n-1} k = n \times (n-1)!$$

- il fattoriale di n è uguale a n moltiplicato per il fattoriale di n-1, non vero se n=0! Serve una regola aggiuntiva

$$n! = \begin{cases} n \times (n-1)! & \text{if } n > 0 \\ 1 & \text{if } n = 0. \end{cases}$$

Ricorsione

- Applico la regola in cascata

$$n! = \begin{cases} n \times (n-1)! & \text{if } n > 0 \\ 1 & \text{if } n = 0. \end{cases}$$

$$4! = 4 \times (3)!$$

Ricorsione

- Applico la regola in cascata

$$n! = \begin{cases} n \times (n-1)! & \text{if } n > 0 \\ 1 & \text{if } n = 0. \end{cases}$$

$$4! = 4 \times (3)!$$

$$3! = 3 \times (2)!$$

Ricorsione

- Applico la regola in cascata

$$n! = \begin{cases} n \times (n-1)! & \text{if } n > 0 \\ 1 & \text{if } n = 0. \end{cases}$$

$$4! = 4 \times \underbrace{(3)!}$$

$$3! = 3 \times \underbrace{(2)!}$$

$$2! = 2 \times (1)!$$

Ricorsione

- Applico la regola in cascata

$$n! = \begin{cases} n \times (n-1)! & \text{if } n > 0 \\ 1 & \text{if } n = 0. \end{cases}$$

$$4! = 4 \times \underbrace{(3)!}$$
$$3! = 3 \times \underbrace{(2)!}$$
$$2! = 2 \times \underbrace{(1)!}$$
$$1! = 1 \times (0)!$$

Ricorsione

- Applico la regola in cascata

$$n! = \begin{cases} n \times (n-1)! & \text{if } n > 0 \\ 1 & \text{if } n = 0. \end{cases}$$

$$4! = 4 \times \underbrace{(3)!}$$
$$3! = 3 \times \underbrace{(2)!}$$
$$2! = 2 \times \underbrace{(1)!}$$
$$1! = 1 \times \underbrace{(0)!}$$
$$0! = 1$$

Ricorsione

- Applico la regola in cascata

$$n! = \begin{cases} n \times (n-1)! & \text{if } n > 0 \\ 1 & \text{if } n = 0. \end{cases}$$

$$4! = 4 \times \underbrace{(3)!}$$
$$3! = 3 \times \underbrace{(2)!}$$
$$2! = 2 \times \underbrace{(1)!}$$
$$1! = 1 \times \underbrace{(0)!}$$
$$0! = 1$$
$$= 4 \times 3 \times 2 \times 1 \times 1$$

Procedure ricorsive

- Procedura ricorsiva: è una procedura che per risolvere il problema P invoca se stessa per risolvere un sotto-problema di P

- In generale una procedura ricorsiva non è una procedura foglia: invoca se stessa per sua definizione

- Una procedura ricorsiva è:
  - Un callee: deve salvare i registri callee-saved ($s0, …, $ra, $fp)
  - Un caller: deve salvare i registri caller-saved ($t0, …, $a0, …, $v0, $v1)

- Al momento del ritorno dalla chiamata ricorsiva è necessario ripristinare il valore di `$ra`

Procedure ricorsive

Possono essere strutturate in diversi blocchi funzionali:

- Punto di ingresso
- Push sullo stack dei registri usati
- Check caso base / step ricorsivo
  - Caso base
  - Step ricorsivo
- Ripristino dei registri usati
- `jr $ra`

Università degli Studi di Milano
Dipartimento di Informatica “Giovanni Degli Antoni”
Corso di Laurea Triennale in Informatica

Architettura degli Elaboratori II Laboratorio

Università degli Studi di Milano
Dipartimento di Informatica “Giovanni Degli Antoni”
Corso di Laurea Triennale in Informatica

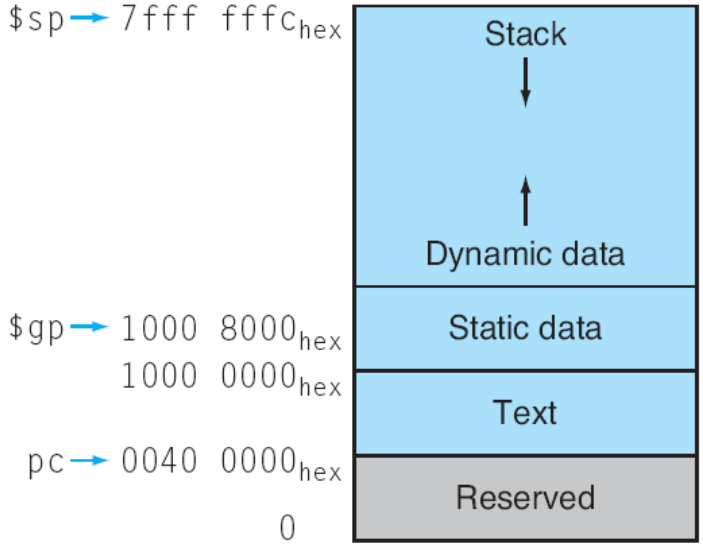
Architettura degli Elaboratori II Laboratorio

Allocazione dinamica della memoria

Utilizzo della memoria

Richiamo: in MIPS la memoria viene divisa in:

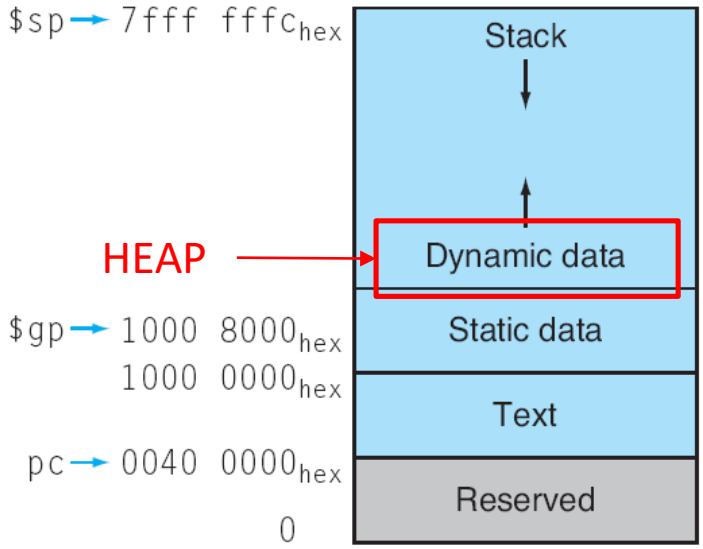
- **Segmento testo**: contiene le **istruzioni** del programma.
- **Segmento dati**:
  - ***dati statici*:** contiene dati la cui dimensione è conosciuta a *compile time* e la cui durata coincide con quella del programma *(e.g., variabili statiche, costanti, etc.);*
  - ***dati dinamici*:** contiene dati per i quali lo spazio è allocato dinamicamente a *runtime* su richiesta del programma stesso *(e.g., liste dinamiche, etc.).*
- **Stack**: contiene dati dinamici organizzati secondo una coda LIFO (Last In, First Out) *(e.g., parametri di una procedura, valori di ritorno, etc.)*.



Utilizzo della memoria

Richiamo: in MIPS la memoria viene divisa in:

- **Segmento testo**: contiene le **istruzioni** del programma.
- **Segmento dati**:
  - ***dati statici***: contiene dati la cui dimensione è conosciuta a *compile time* e la cui durata coincide con quella del programma *(e.g., variabili statiche, costanti, etc.);*
  - ***dati dinamici***: contiene dati per i quali lo spazio è allocato dinamicamente a *runtime* su richiesta del programma stesso *(e.g., liste dinamiche, etc.).*
- **Stack**: contiene dati dinamici organizzati secondo una coda LIFO (Last In, First Out) *(e.g., parametri di una procedura, valori di ritorno, etc.)*.



Allocazione statica vs. allocazione dinamica

STATICA

- Avviene a **compile time**
- Non può cambiare dimensione
- Usata per costanti, variabili, e array

DINAMICA

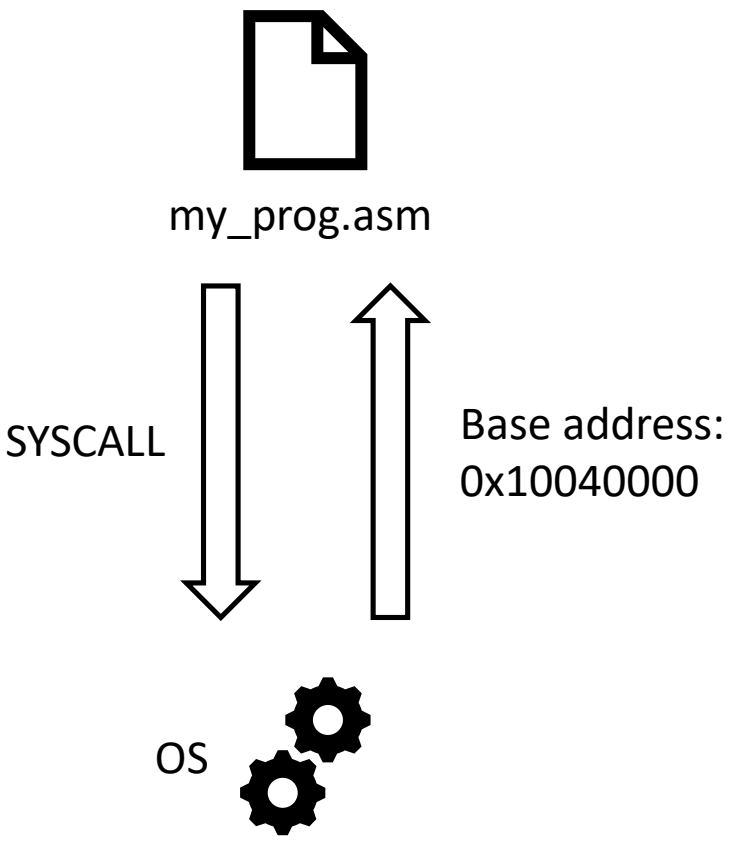
- Avviene a **run time**
- La dimensione può variare
- L'allocazione è demandata al programmatore
- Usata per stringhe di lunghezza non determinata, strutture dati dinamiche

Come avviene l'allocazione?

1. Il programma richiede al sistema operativo di allocare un certo numero di byte in memoria tramite una syscall

2. Il sistema operativo verifica l'effettiva disponibilità dello spazio in memoria e, se possibile, alloca lo spazio richiesto e restituisce al programma il base address dell'area di memoria allocata

3. In caso di memoria non allocabile, di norma, il sistema operativo restituirà un codice di errore (tipicamente un numero negativo) al posto del base address permettendoci di gestire il problema



In MARS: la richiesta di allocare più spazio di quanto disponibile, genera un'eccezione a run time

SBRK (segment break)

La syscall utilizzata per allocare spazio dinamicamente sullo heap è la **SBRK** (codice **9**).

- Input: $a0 <- numero di byte da allocare
- Output: $v0 <- base address dell'area di memoria allocata

```
    .text
    .globl main

main:
    li $v0 9 # codice per SBRK
    li $a0 100 # chiedo di allocare 100 bytes
    syscall

    # $vo ora contiene il base address dell'area di memoria allocata


    # carico il valore 5 nella prima parola di memoria allocata
    li $to 5
    sw $t0 0($v0)

    # carico 6 nella seconda
    li $to 6
    sw $t0 4($v0)
```

SBRK vs. malloc e free

- La syscall SBRK si occupa solo di «spostare» il puntatore alla fine dello heap più avanti aumentando lo spazio di memoria dinamica disponibile al nostro programma

- non ci permette di gestire come allochiamo la memoria all'interno dello heap, ma solo di richiedere al sistema operativo di darci più memoria

- Le funzioni «malloc» e «free» appartenenti alla libreria stdlib distribuita con libc (installata praticamente su tutti i sistemi operativi moderni) servono per gestire in maniera più fine lo heap e sono implementate «sopra» a SBRK, permettendo in maniera automatica di allocare e deallocare spazio per le nostre strutture dati dinamiche, gestendo automaticamente problemi di frammentazione della memoria e minimizzando le chiamate a SBRK utilizzando opportuni buffer di memoria

- In MARS e in SPIM non abbiamo accesso alle funzioni malloc e free, che sono disponibili solo su sistemi «reali»

Strutture Dati

Una struttura dati rappresenta un modo particolare di **organizzare i dati** in memoria **in modo da massimizzare** l'**efficienza** di esecuzione di **determinate operazioni** su di essi (accesso, ricerca di elementi, ecc.)

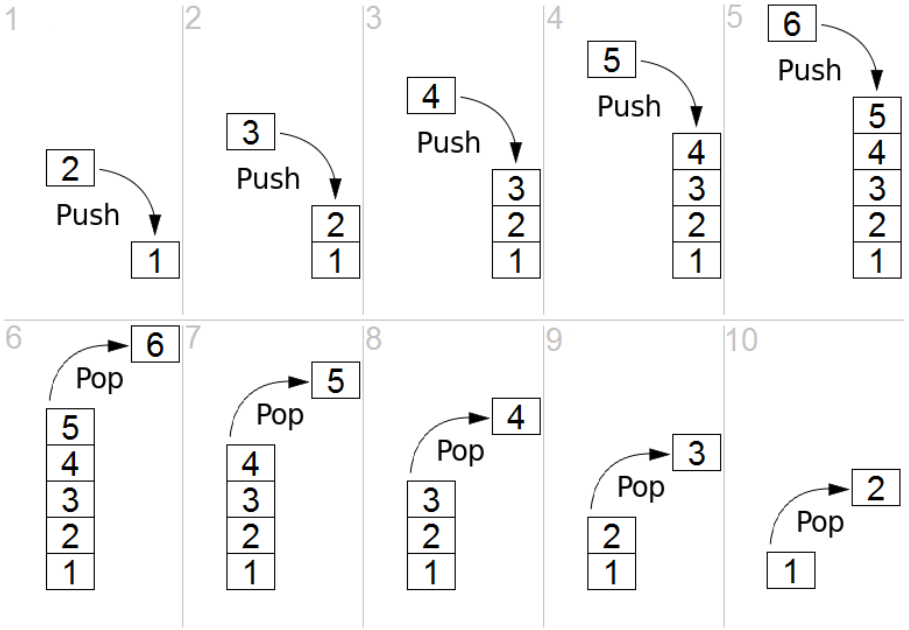
Una struttura dati contiene le seguenti informazioni:

- I **valori** dei dati in essa contenuti (interi, stringhe, strutture …)
- Le eventuali **relazioni** tra i dati in essa contenuti (ordine …)
- Le **funzioni** e le **operazioni** che possono essere applicate sui vari dati in essa contenuti (cancellazione, inserimento, ricerca …)

«stack» o «pila»

- Lo stack è un tipo particolare di struttura dati, appartenente alla categoria delle code **LIFO** (Last In First Out)

- Il suo scopo è quello di massimizzare l'efficienza nelle operazioni di salvataggio di un nuovo elemento e di accesso all'ultimo elemento in esso inserito



«stack» o «pila»
implementazione

Obiettivo: implementare uno stack nel segmento dati dinamico:

1. Identificare i dati con cui lavoreremo e come li rappresenteremo in memoria
2. Stabilire come collegare fra di loro questi dati
3. Implementare le funzioni di PUSH e POP dando così un'interfaccia all'utente per utilizzare la struttura dati

Attenzione: questo è uno stack implementato da noi all'interno dello heap, non è lo stack «di sistema» a cui per convenzione riserviamo la parte alta del segmento di memoria.

«stack» o «pila»
rappresentazione dei dati

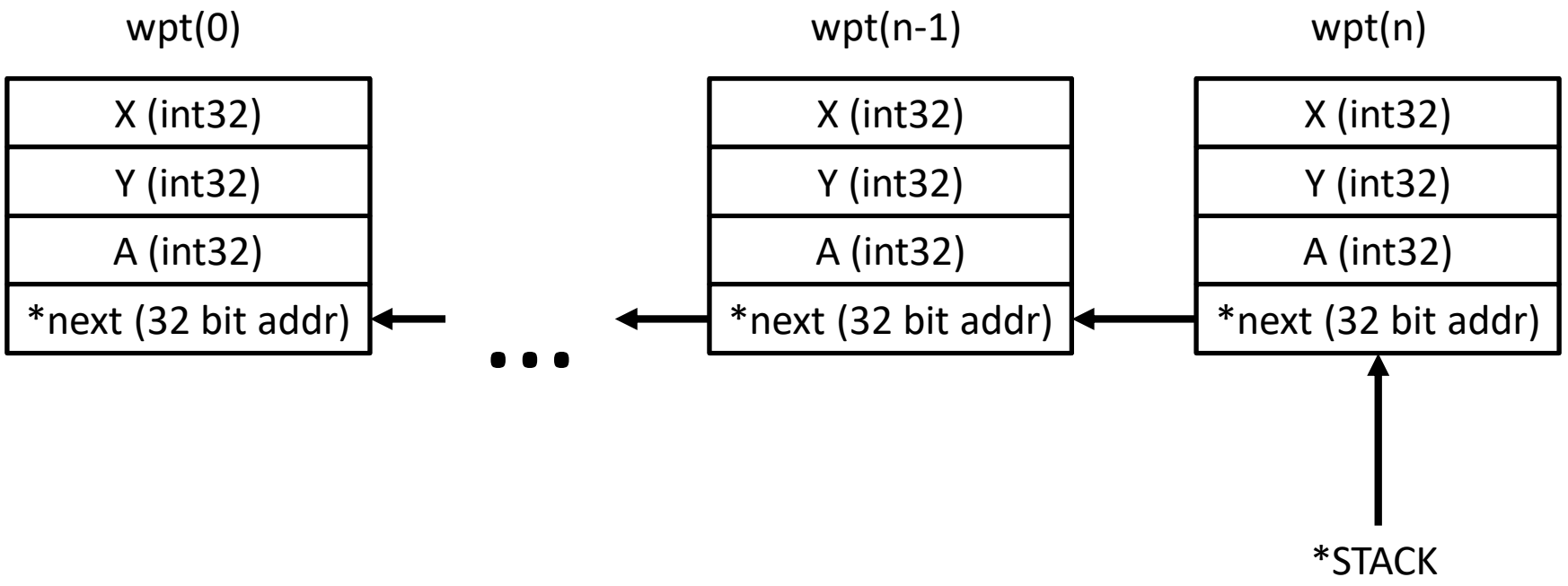
Supponiamo di volere memorizzare nello stack dei waypoint di un percorso che verranno «consumati» da un robot in ordine inverso a come sono stati inseriti (il primo waypoint visitato dal robot sarà l'ultimo inserito)

- Un waypoint è definito come una posizione su una mappa bidimensionale (x,y) e un angolo di rotazione (r) espresso in gradi
- Per semplicità trattiamo le coordinate come numeri interi a 32 bit

```
struct{
    int x; // coordinata x
    int y; // coordinata y
    int r; // rotazione
    *next; // puntatore al prossimo elmento nella lista
}
```

«stack» o «pila»
collegamento fra i dati

Possiamo rappresentare lo stack, come una «lista linkata», ovvero come una lista in cui ogni elemento punta all'elemento adiacente



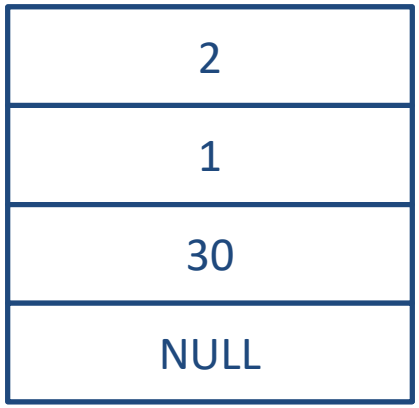
«stack» o «pila»
implementazione funzione push (1)

Push(x,y,a)

INPUT: x,y,a

1. Alloca spazio per un nuovo elemento e lo inizializza con i valori passati come argomento alla funzione.

Push(2,1,30)

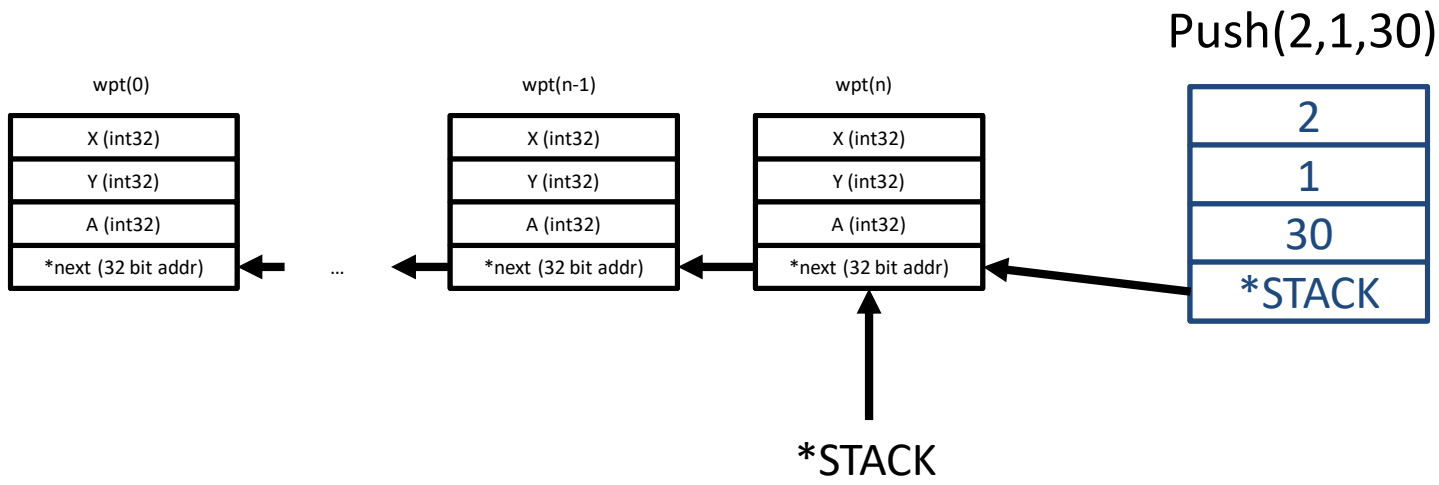


«stack» o «pila»
implementazione funzione push (2)

Push(x,y,a)

INPUT: x,y,a

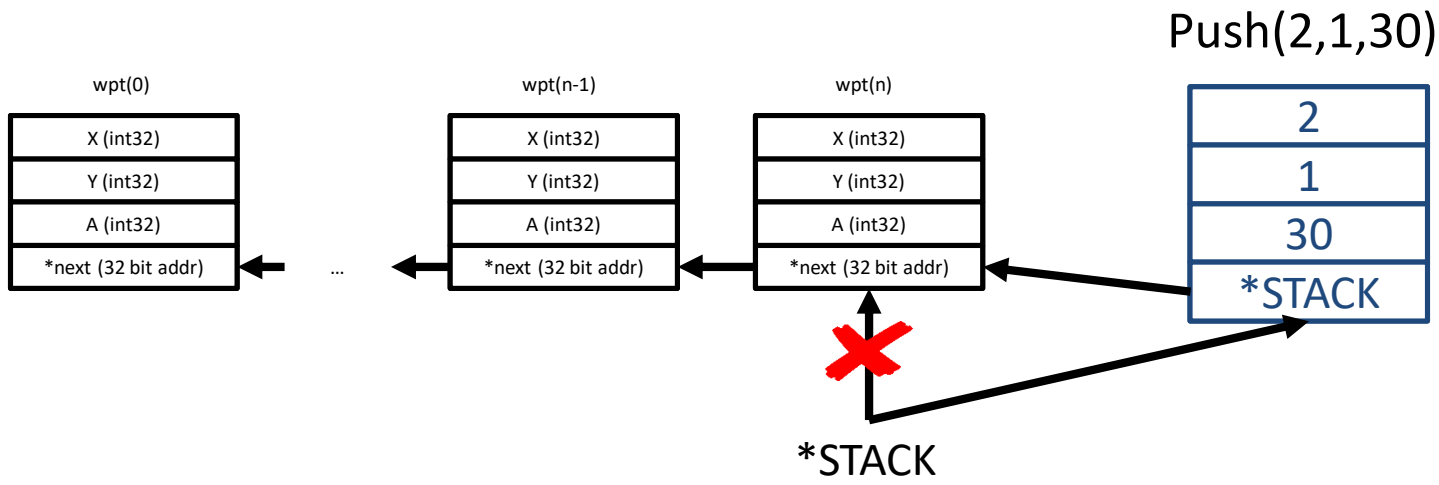
2. Inizializza il puntatore al prossimo elemento con il valore corrente del puntatore allo stack



«stack» o «pila»
implementazione funzione push (3)

Push(x,y,a)
INPUT: x,y,a

3. Aggiorna il puntatore allo stack facendolo puntare all'elemento appena creato



Università degli Studi di Milano
Dipartimento di Informatica “Giovanni Degli Antoni”
Corso di Laurea Triennale in Informatica

Architettura degli Elaboratori II Laboratorio