

# Indice

<b>Appunti ACSO</b>	<b>1</b>
I principi fondamentali . . . . .	1
Data path . . . . .	2
Codifica delle istruzioni . . . . .	2
Architettura LOAD/STORE . . . . .	2
Interruzioni (interrupt) . . . . .	2
La gerarchia di memoria . . . . .	2
Indirizzi di memoria principale . . . . .	2
Bus di sistema . . . . .	3
Esempio: operazione di lettura da memoria . . . . .	3
MIPS assembler . . . . .	3
Sintassi . . . . .	3
Direttive . . . . .	3
Registri . . . . .	3
Istruzioni . . . . .	4
Modalità di indirizzamento . . . . .	7
Etichette . . . . .	7
Traduzione da C a MIPS . . . . .	7
Modello di memoria . . . . .	7
Syscall . . . . .	8
Dimensioni delle variabili . . . . .	8
Classi di variabili . . . . .	8
Variabili globali . . . . .	8
Variabili locali . . . . .	8
Parametri . . . . .	8
Variabili dinamiche . . . . .	8
Funzioni . . . . .	8
Salvataggio dei registri . . . . .	9
Gestione dello stack . . . . .	9
Funzioni foglia . . . . .	10
Valutazioni espressioni algebriche . . . . .	10

## Appunti ACSO

### I principi fondamentali

I processori eseguono le istruzioni in maniera sequenziale (non vedremo la Out of Order Execution). Il processore contiene una sezione di controllo che controlla l'esecuzione leggendo le istruzioni dalla memoria. I vari programmi da eseguire sono salvati in una memoria.

L'architettura di riferimento è l'architettura di Von Neumann. Esso contiene:

- Processore
  - Sezione di controllo
  - ALU
  - Registri
- Memoria
- Interfacce I/O
- Bus che collega i vari componenti sopra elencati

I registri sono delle piccole celle di memoria che il processore usa per salvare dati utili. Ogni registro ha una dimensione massima specificata in bit. I registri che studieremo saranno quelli a 32 bit.

Viene detta parola del processore la lunghezza dei suoi registri, ossia la lunghezza massima della sequenza di bit che può gestire.

Esistono dei registri particolare:

- PC (Program Counter): contiene l'indirizzo della prossima istruzione in memoria
- IR (Instruction Register): contiene l'intera istruzione da eseguire

Il ciclo di esecuzione di un'istruzione è divisa nelle seguenti fasi:

- fetch: viene presa dalla memoria attraverso il bus l'istruzione puntata dal program counter e incrementalo nel frattempo (memoria più lenta del processore)
- decode: decodifica l'istruzione e prepara l'esecuzione prelevando gli operandi
- execute: esegui e memorizza i risultati

I vari processori si differenziano in base alla lunghezza delle istruzioni che usano (può essere una singola parola (RISC) oppure più (CISC)).

## Data path

Il data path è formato da registri e da ALU. I registri sono divisi in:

- utilizzabili dall'assembler
- di supporto per altri componenti (es.: 3 registri di supporto dell'ALU (sorgente, sorgente, destinazione))

Abbiamo due classi di istruzioni:

- Prendono registri e memorizzano in altri registri (registro-registro)
- Gestiscono lo spostamento da registri a memoria (registro-memoria)

L'esecuzione di una istruzione registro-registro viene detta ciclo di data path. Il ciclo di data path è collegato al ciclo di clock.

## Codifica delle istruzioni

Ogni istruzione è divisa in codice operativo e campi (field). Il codice operativo indica il tipo di operazione. I campi vengono usati per gli operandi dell'istruzione.

Le modalità di indirizzamento descrivono le diverse modalità attraverso le cui far riferimento agli operandi nelle istruzioni.

## Architettura LOAD/STORE

Il numero di registri ad uso generale non è abbastanza grande per mantenere tutte le variabili di un programma. Ad ogni variabile viene quindi assegnata una locazione in memoria nella quale salvare il contenuto del registro che la rappresenta quando deve essere usato per altro.

Poiché gli operandi delle istruzioni possono provenire solamente dai registri ad uso generale, servono delle istruzioni di caricamento e salvataggio a memoria. Da qui il nome dell'architettura.

## Interruzioni (interrupt)

Il normale flusso dei programmi può essere interrotto attraverso le interruzioni. Quando un dispositivo esterno vuole richiedere l'attenzione del processore, esso attiva un segnale hardware (segnale di interrupt) per notificarlo.

Il check degli interrupt viene fatto alla fine dell'esecuzione di un'istruzione ma prima del fetch della successiva. Se viene trovato un segnale di interrupt, il processore interrompe la normale esecuzione del programma ed esegue la richiesta. Una volta gestita il processore torna ad eseguire normalmente il suo programma.

## La gerarchia di memoria

E' divisa in una gerarchia (dalla più veloce alla più lenta):

- Registri
- Cache (L1, L2, L3)
- Memoria fisica: RAM (es. DDR-SDRAM)
- Memoria a stato solido (SSD)
- Memoria virtuale: basata su file (HDD)

## Indirizzi di memoria principale

La memoria principale è suddivisa in celle. La dimensione di una cella è chiamata parola, come anche il numero massimo di bit che il processore può gestire. La parola di memoria non per forza avrà la stessa dimensione della parola del processore.

Nel MIPS useremo una parola di memoria di 8 bit (memoria indirizzabile al byte). Di conseguenza una parola MIPS è lunga 4 parole di memoria.

Il tempo di indirizzamento di ogni singola parola è uguale.

L'insieme di tutte le celle di memoria indirizzabili con 1 parola di processore è detto spazio di memoria: per una parola di 32 bit si ha uno spazio di memoria di 4 GB.

## Bus di sistema

Un bus è un insieme di fili. Ogni filo trasferisce un bit. I vari fili sono suddivisi in categorie:

- Bus dati: comprende le linee usate per trasferire dati da/verso memoria. La dimensione del bus dati deve essere abbastanza da garantire il trasferimento contemporaneo di una o più parole di memoria
- Bus indirizzi: comprende le linee sulle quali la CPU procede a trasmettere gli indirizzi di memoria delle risorse
- Bus di controllo: comprende le linee su cui transitano le informazioni ausiliarie per la corretta definizione delle operazioni e per sincronizzare CPU e memoria

Il bus può essere utilizzato per un solo trasferimento alla volta: in ogni istante solo due entità possono comunicare (master - slave). Il master è il processore, gli slave sono le periferiche. Il processore regola l'accesso delle varie periferiche. L'unico filo che può essere usato senza permesso del processore è quello di interrupt request.

L'architettura a bus singolo è molto flessibile e semplice. I dispositivi, però, hanno velocità diverse. Serve, quindi, un meccanismo di sincronizzazione tra le varie periferiche.

## Esempio: operazione di lettura da memoria

1. CPU fornisce l'indirizzo della parola desiderata sul bus indirizzi e richiede la lettura sul bus di controllo
2. Quando la memoria ha completato la lettura della parola richiesta, il dato viene trasmesso sul bus dati

## MIPS assembler

Il linguaggio assembler è simbolico. E' il primo livello di astrazione prima del linguaggio macchina.

Un programma è composto da più file (oggetti). Il compito di "collegare" i vari oggetti è delegato al linker.

L'assembler non istruisce che effettuano un controllo di flusso come inteso in C. L'unico strumento che ha è il jump/branch che permette di saltare da un'istruzione all'altra.

## Sintassi

Nel MIPS i registri sono numerati da 0 a 31 e hanno dei nomi simbolici. Vengono identificati dal simbolo \$. Mettere un registro tra parentesi toglie l'operazione di indirizzamento. Il primo registro nei field è quasi sempre la destinazione del risultato dell'istruzione.

**Direttive** I comandi che iniziano con . sono direttive del preprocessore. Esse sono:

- .text o .data: indica che le linee successive sono istruzioni o dati
- .align n: specifica l'allineamento a  $2^n$  bit
- .globl main: indica che l'etichetta main è visibile anche in altri file (visibilità globale)
- .ascii: specifica un'area di memoria che contiene una stringa ASCII terminata da \0
- .space n: riserva uno spazio di n byte
- .word n: equivalente a .space 4 e settare a n
- .half n: equivalente a .space 2 e settare a n
- .byte n: equivalente a .space 1 e settare a n
- .eqv A, n: equivalente alla #define del C

**Registri** Essi sono 32, numerati da 0 a 31 (occupa 5 bit). Ogni istruzione, quindi, può usare 3 registri alla volta. Gli operandi delle istruzioni devono essere prima salvati nei registri.

Registri referenziabili:

- \$0: contiene sempre 0
- \$1/\$at: riservato
- \$v0 e \$v1: usati per risultati di funzioni e calcolo espressioni
- \$a0 - \$a3: usati per il passaggio di argomenti
- \$t0 - \$t7: variabili temporanee
- \$s0 - \$s7: variabili da preservare
- \$k0 e \$k1: riservati al kernel
- \$gp: global pointer (punta ad area di dati globale/statica)

- `$sp`: stack pointer
- `$fp`: frame pointer (puntatore ai frame funzione; in aiuto a `$sp`)
- `$ra`: return address utilizzato nelle chiamate di funzione

Registri non referenziabili:

- `$pc`: program counter
- `$hi`: registro per moltiplicazione/divisione
- `$lo`: registro per moltiplicazione/divisione

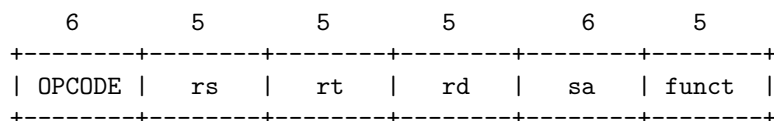
**Istruzioni** Le istruzioni si dividono in 4 categorie:

1. istruzioni aritmetico logiche
2. istruzioni di trasferimento da/verso memoria (ad es. `lw` e `sw`)
3. istruzioni di salto condizionato/incondizionato
4. istruzioni di ingresso/uscita (non fornite da tutti gli assembler)

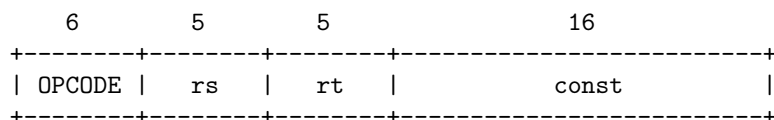
Le istruzioni sono tutte lunghe massimo 32 bit. I 6 bit più significativi si chiamano “codice operativo” (OPCODE) e indica il tipo di istruzione.

In MIPS ci sono 3 tipi di istruzioni:

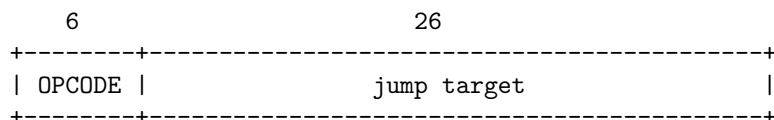
- tipo R: istruzioni aritmetico-logiche; strutturate in:



- tipo I: istruzioni di accesso alla memoria o contenenti costanti



- tipo J: salto



Sintassi:

`istruzione field1, field2, field3 # comment`

La lista completa di tutte le istruzioni non è riportata qui. Può essere trovata nella cartella con il materiale del corso.

### Istruzioni aritmetico-logiche

istruzione	campo	campo	campo
add	rd	rs	rt
addu	rd	rs	rt
addi	rd	rs	const
addui	rd	rs	const
sub	rd	rs	rt
subu	rd	rs	rt
subi	rd	rs	const
subui	rd	rs	const
mult	rs	rt	-
multu	rs	rt	-
div	rs	rt	-
divu	rs	rt	-

Le istruzioni immediate (\*i) hanno la peculiarità di prendere una costante al posto di un secondo registro. Per come è strutturato il formato delle istruzioni immediate, la costante sarà di massimo  $2^{16}$ .

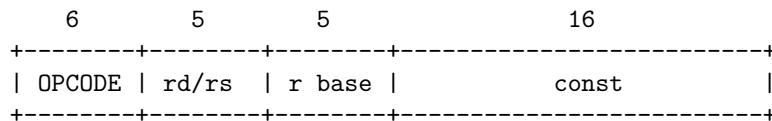
Il registro di destinazione della `mult` è implicito: il risultato viene salvato in `$hi` (32 cifre più significative) e `$lo` (32 cifre meno significative). Per spostare il risultato da questi due registri viene usato `mfhi rd` e `mflo rd`. Stessa cosa vale per la divisione solo che il quoziente è posto nel registro `$lo`, mentre il resto in `$hi`.

## Trasferimenti in memoria

istruzione	campo	campo
<code>lw</code>	<code>rd</code>	<code>const(registro base)</code>
<code>sw</code>	<code>rs</code>	<code>const(registro base)</code>
<code>la</code>	<code>rd</code>	<code>const</code>

La parte `const(registro base)` serve a calcolare l'indirizzo di memoria a cui si fa riferimento. L'indirizzo di memoria è pari a `const + indirizzo contenuto nel registro base`. Per come è strutturata il tipo di istruzione I, posso caricare/salvare nell'intervallo  $[-2^{15} ; +2^{15} - 1]$  rispetto al registro base.

In memoria una `lw` o una `sw` avranno questo formato:



L'istruzione `la` è una pseudo istruzione: poiché gli indirizzi sono di 32 bit, non è possibile specificare un intero indirizzo in un'istruzione. L'assemblatore, allora, espande la `la` in 2 istruzioni:

- si utilizza lui `$rt const` per caricare i 16 bit più significativi in `$at` (mettendo a 0 gli altri 16 bit meno significativi)
- si utilizza un'altra istruzione (ori un esempio) per caricare gli altri bit meno significativi e spostare tutto nel registro di destinazione

Un'altra pseudo istruzione simile è `li $rd, const`.

L'ordinamento dei byte di una parola non è da dare per scontato:

- big-endian: ordinamento da sinistra a destra
- little-endian: ordinamento da destra a sinistra

Il MIPS può operare con entrambe le modalità.

Load e store con indirizzi simbolici

Nelle chiamate a `lw` e `sw` si può omettere il registro base. In quel caso verrà usato `$gp` e l'offset verrà calcolato dal linker.

```
.data
...
A: .word
...

.text
...
    lw $t0, A
    # viene assemblato in
    #    lw $t0, %gp_rel(A)($gp)
    # dove %gp_rel(A) è una direttiva del linker che calcola l'offset di A da
    # $gp
```

Anche `la` possiede una forma simile: i bit più significativi e meno significativi non sono calcolati dal compilatore, ma il compito viene delegato al linker:

```
.data
...
A: .word
...
```

```
.text
...
    la $t0, A
    # viene assemblato in:
    #   lui $t0, %hi(A)
    #   addiu $t0, %lo(A) # oppure ori $t0, %lo(A)
    # dove come prima %hi(A) e %lo(A) sono direttive del linker
```

Queste forme vanno usate solo con valori dichiarati con `.word`, `.half` e `.byte`.

## Istruzioni logiche

istruzione	campo	campo	campo
and	rd	rs	rt
or	rd	rs	rt
nor	rd	rs	rt
sll	rd	rs	sa
srl	rd	rs	sa

`sll` di 1 bit equivale a moltiplicare per 2. `srl` di 1 bit equivale dividere per 2.

L'istruzione `not` viene eseguita con usando `nor $rd $rs $0`.

Esistono anche le varianti immediate per `and`, `or` e `nor`.

**Istruzioni di modifica del flusso** Le istruzioni di modifica del flusso servono a forzare la modifica del `$pc`, rompendo il flusso sequenziale standard. Il salto condizionato viene chiamato `branch`, quello incondizionato `jump`.

### Branch

Le istruzioni di `branch` hanno tutte la forma:

`branch_condizione rs, rt, offset`

Le istruzioni di `branch` sono di tipo I. Avrò quindi a disposizione un salto di  $[-2^{15} ; +2^{15} - 1]$  (salvato a complemento a 2) parole. Il salto però viene effettuato relativo al program counter. Il principio di località degli indirizzi, però, ci viene in aiuto: i programmi lavorano solo su segmenti vicini di indirizzi. La probabilità di saltare verso indirizzi più distanti di  $2^{15}-1$  è molto bassa.

istruzione	campo	campo	campo	condizione
beq	rs	rt	offset	<code>rs == rt</code>
bne	rs	rt	offset	<code>rs != rt</code>

L'assemblatore, a un'etichetta messa nel campo `offset`, sostituisce  $(L-(PC+4))/4$ .

Per verificare diseuguaglianze usiamo le seguenti istruzioni ausiliarie. Esse caricano 1 nel registro destinazione se la condizione è avverata e 0 altrimenti.

istruzione	campo	campo	campo	condizione
slt	rd	rs	rt	<code>rs &lt; rt</code>
sltu	rd	rs	rt	<code>rs &lt; rt</code>

Queste istruzioni sono di tipo R. Esistono anche le varianti immediate di queste istruzioni.

### Jump

Sono possibili 3 salti assoluti:

istruzione	campo
j	offset
jal	offset
jr	rs

Nel campo offset rizzo posso salvare fino a 26 bit. Poiché le istruzioni hanno allineamento 4, i due bit meno significativi saranno sempre 00. Posso, quindi, ignorare questi due bit salvando escludendo i due bit meno significativi ottenendo un salto totale di  $2^{28}$  indirizzi (ossia  $2^{26}$  parole). A runtime il processore farà uno shift di due a sinistra il campo indirizzo e lo concatenerà ai 4 bit più significativi del program counter, ottenendo l'indirizzo di salto.

Per saltare a indirizzi superiori a  $2^{28}$  Byte devo usare la **jr**.

La **jal** salva **\$pc + 4** nel registro **\$ra** prima di saltare. Essa viene usata per implementare la chiamata a funzione. La **jr** invece viene usata per implementare il ritorno al chiamante (**jr \$ra**).

Nota bene: anche i salti hanno delle restrizioni sull'indirizzo (26 bit dedicati)

**Gestione delle costanti** Se usiamo costanti di 32 bit, l'assemblatore deve fare 2 passi per caricarla: deve separare la costanti in due parti di 16 bit e trattarle con due istruzioni separate. Proprio come **la**, esiste la pseudo istruzione **li \$rs, const** che esegue gli stessi passaggi.

**Modalità di indirizzamento** In MIPS ci sono 5 modalità di indirizzamento:

- Immediato (quello usato dalle istruzioni di tipo I)
- A registro (quello usato dalle istruzioni di tipo R)
- Con base e offset (quello usato da **lw/sw**)
- Relativo al program counter (quello usato da **beq**)
- Pseudo diretto (quello usato da **jr**)

**Etichette** Le etichette vengono usate per dare nomi simbolici a delle celle di memoria. Sarà compito dell'assemblatore tradurre le etichette in indirizzi di memoria. Sintassi:

```
etichetta:
    add $1, $2, $3 # anche direttiva
```

## Traduzione da C a MIPS

Per tradurre da un linguaggio sorgente a un linguaggio macchina, bisogna definire un modello di architettura runtime. Alcune delle convenzioni di questo modello sono:

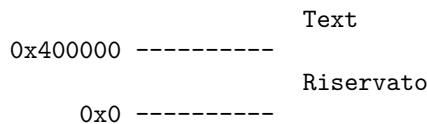
- collocazione e ingombro di tipi di variabili
- destinazione di uso dei registri

### Modello di memoria

La memoria di un programma è divisa in vari segmenti:

- text (**.text**): codice del programma (dichiarato dal programma)
- data (**.data**): dati statici e dinamici (dichiarato del programma)
- stack: la stack del processo (allocato dal sistema operativo)

```
0x7fffffff -----
                                Stack
                                -----
                                v
                                ^
                                -----
                                Dynamic Data
                                Static Data
0x10000000 -----
```



Gli indirizzi di impianto dei segmenti sono indirizzi virtuali, non fisici.

Programmi molto grandi e sofisticati possono avere due o più segmenti dati o testo, segmenti di dati condivisi, segmenti di libreria dinamica e altro.

## Syscall

La **syscall** è un'istruzione che passa il controllo al kernel. Il kernel offre vari servizi, ognuno indentificato da un codice.

Il codice, prima, viene salvato dal chiamante nel registro **\$v0**, seguito dagli argomenti in **\$a\*** o **\$f12**. Viene poi chiamata l'istruzione **syscall**.

Se una **syscall** ritorna un valore, esso viene salvato nel registro **\$v0**, anche **\$f0** se è più grande di 32 bit.

## Dimensioni delle variabili

tipo	dimensione (B)
<b>char</b>	1
<b>short</b>	2
<b>int</b>	4
<b>void*</b>	4

Per dichiarare un array allochiamo semplicemente **n\*dimensione** byte. Per accederci, usiamo l'aritmetica dei puntatori (come si fa anche in C): **base + (dimensione \* n)**.

Una sintassi alternativa di **.word** è: **.word n, m, ...**. Essa equivale all'inizializzazione diretta di un array (come in C)

Per dichiarare una **struct**, basta semplicemente allocare una dimensione di pari alla dimensione totale della **struct**. Per accedere ai vari elementi, semplicemente ci accede come ad un array, tenendo conto dell'ordine degli elementi.

## Classi di variabili

In C abbiamo diversi tipi di variabili: globali, locali, parametri e allocate dinamicamente.

**Variabili globali** Le variabili globali sono allocate in **.data** a partire da 0x10000000. Per indirizzarla usiamo un'etichetta che gli assegniamo.

Per puntare al segmento dati statici usiamo il registro **\$gp**, inizializzato all'indirizzo 0x10008000

**Variabili locali** Per le variabili locali posso usare i registri **\$s\***. Se, però, ho bisogno di usare l'indirizzo di una variabile locale, essa andrà salvata sulla stack.

**Parametri** I primi quattro parametri vengono passati tramite i registri **\$a\***. Se la funzione ha più di 4 parametri essi vengono salvati sulla stack.

Nota bene: per array e struct viene passato l'indirizzo!

Il chiamante, prima della chiamata, salva in memoria i registri che vuole mantenere inalterati durante la chiamata.

Per i valori in uscita si usa il registro **\$v0**. Se la return è più lunga di 32 bit si usa anche il registro **\$v1**.

**Variabili dinamiche** Come anche in C, bisogna ricorrere a una funzione che ci allochi memoria sulla heap. Viene utilizzata la syscall **sbrk** (codice 9 su SPIM) passando in entrata il numero di byte da allocare. L'indirizzo dello spazio di memoria allocato è ritornato in **\$v0**.

## Funzioni

La chiamata a funzione, in linguaggi come il C, ha come effetto la creazione di un record di attivazione sulla stack. Il record di attivazione contiene:



- i parametri formali e i loro valori
- l'indirizzo di ritorno al chiamante
- le informazioni per gestire lo spazio allocato per il record
- le variabili locali
- il valore restituito

Il chiamante esegue le seguenti operazioni preliminari:

1. Predispone i parametri in ingresso negli opportuni registri
2. Salva sulla stack il valore di registri che vuole mantenere
3. Trasferisce il controllo alla procedura tramite `jal`

Il chiamato esegue le seguenti operazioni:

1. Alloca lo spazio di memoria necessario alla memorizzazione dei dati e alla sua esecuzione
2. Salva alcuni registri sulla stack
3. Eseguire le sue funzioni
4. Memorizzare il risultato nell'apposito registro
5. Ripristina i registri salvati sulla stack
6. Ritornare il controllo al chiamante tramite `jr`

**Salvataggio dei registri** Vengono salvati sullo stack solo un particolare gruppo di registri. I registri che vengono preservati dal chiamato sono: `$s*`, `$fp`, `$ra`.

I registri che il chiamante può salvare, invece, sono: `$t*`, `$a*` e `$v*`.

Entrambe le parti devono salvare i registri solo se verranno modificati.

### Minimo salvataggio dei registri

- Il chiamato non deve salvare nulla se:
  - Non chiama nessun'altra funzione
  - scrive solo in registri temporanei, per parametri, per risultati o non indirizzabili
- La funzione chiamante non deve salvare nulla se:
  - non vuole salvare il contenuto dei suoi registri temporanei, parametro e risultato

**Gestione dello stack** Lo stack cresce dall'altro verso il basso. Lo stack pointer contiene l'indirizzo della cima della stack. Quindi ogni volta che dobbiamo spingere qualcosa sulla stack dobbiamo decrementare lo stack pointer e ogni volta che dobbiamo rimuovere qualcosa dobbiamo incrementarlo.

#### PUSH:

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

#### POP:

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

Nello stack sono salvati i record di attivazione (frame). Lo spazio sulla stack viene allocato dal programmatore in una sola volta all'inizio della procedura. Lo stesso vale per la deallocazione.

**Stack frame** Lo stack frame è allocato dal chiamato e ha la seguente forma:

```
-----
Registri salvati
dal chiamante
-----
Parametri (5, 6...)
----- <- $fp

Registri salvati
-----
```

Variabili locali

```
----- <- $sp
```

Il registro **\$fp** punta alla prima parola del frame, mentre **\$sp** punta all'ultima parola del frame. L'utilizzo del frame pointer è opzionale.

Il layout dei registri salvati è il seguente:

```
...
-----
Vecchio $fp
-----
Vecchio $ra
-----

Vecchi $s0 - $s7

-----
....
```

La generalizzazione dello stack frame è l'area di attivazione e comprende anche i registri salvati dal chiamante.

**Funzioni foglia** Si dice funzione foglia una funzione che non ha annidate al suo interno altre chiamate a procedure. La peculiarità di questo tipo di funzioni è che non deve salvare il valore di **\$ra**.

### Valutazioni espressioni algebriche

Adotteremo la regola più semplice senza ottimizzare. Basta seguire un semplice procedimento:

1. Completa l'espressione associando a sinistra
2. Finché non hai finito valuta il primo operatore valutabile da sinistra

Un operatore viene considerato valutabile se da entrambi i lati sono presenti solo costanti o risultati di sotto-espressioni già calcolate.