

# Assembler MIPS R2000/3000

## Assembly III

Riccardo Solmi

### Indice degli argomenti

---

- **Eccezioni**
- **Organizzazione della memoria**
  - Allineamento dati
  - Allocazione dinamica
- **Programmazione assembly**
  - Definizione e gestione di strutture dati
    - Puntatori
    - Array
    - Aggregati

## Eccezioni

---

- **Eccezione**

- Un evento che interrompe l'esecuzione del programma corrente
- Se l'evento che provoca l'eccezione è esterno al processore si dice che è un **interrupt**; se interno si chiama **program trap**.
- Sono abbastanza rare

- **Ogni volta che si verifica una eccezione:**

- Il processore svuota la pipeline, scartando le istruzioni incomplete
- Il controllo passa al gestore delle eccezioni registrato per il tipo che si è verificato
- L'eccezione viene gestita
- Se il programma può essere ripreso, viene riavviato dal punto in cui era stato interrotto, altrimenti termina.

## Gestione delle eccezioni

---

- **Obiettivo:**

- Trasparenza
- Il programma in esecuzione deve poter essere interrotto e riavviato senza che si accorga di nulla

- **Per raggiungere questo obiettivo:**

- Lo stato del processore (i registri) deve essere lo stesso dopo l'esecuzione del gestore dell'eccezione
- NB. Non valgono le convenzioni tra chiamante/chiamata di funzione. Per il semplice motivo che non esiste un chiamante.

## Principali tipi di eccezioni (trap)

---

- **Reset Exception**
  - All'accensione e ad ogni reset della macchina.
- **Address Error Exception**
  - Tentativo di leggere o scrivere dati non allineati
- **Bus Error Exception**
  - Tentativo di accedere ad un indirizzo inesistente
- **Integer Overflow Exception**
  - Quando una operazione aritmetica va in overflow
- **Reserved Instruction Exception**
  - Quando una istruzione non esiste o è usata in modo illegale
- **System Call and Break Exceptions**
  - Quando viene eseguita una SYSCALL o BREAK

## Allineamento dati

---

- **Nota:**
  - Le operazioni load/store viste finora operano su dati allineati
- **Definizione:**
  - Un dato si dice *allineato* in memoria se il suo indirizzo è un multiplo della sua dimensione in byte
  - Esempio:
    - le word devono essere collocate ad indirizzi multipli di 4
    - le halfword devono essere collocate ad indirizzi pari
- **Istruzioni MIPS:**
  - Esistono istruzione in grado di gestire dati non allineati
  - Esempi: ulw, ulh, lwl, lwr, usw, ush, swl, swr

## Direttiva per allineare i dati

---

- **Come gestire l'allineamento dei dati?**
- **Direttiva `.align n`**
  - Allinea il prossimo dato ad un indirizzo multiplo di  $2^n$
  - Esempio:
    - `.align 2` allinea il prossimo dato ad un indirizzo multiplo di 4
    - `.align 0` rimuove l'allineamento ( $2^0 = 1$ )

## Strutture dati

---

- **Per *struttura dati* si intende una sequenza di dati che viene usata dalle funzioni come se fosse una unità**
  - I dati sono allocati in una zona contigua di memoria.
  - Una struttura dati è identificata dall'indirizzo di memoria dove inizia e eventualmente da una etichetta.
  - I dati che compongono una struttura dati sono quelli gestiti direttamente dalle istruzioni: byte, halfword, word.
  - Una word può essere usata anche per contenere l'indirizzo di un'altra struttura dati; in questo caso si dice che **punta** o che è un **puntatore** a quest'ultima.
  - Una funzione che esegue un calcolo su una struttura dati si fa passare il suo indirizzo come argomento.

## Strutture dati omogenee: array

### ▪ Definizione

- Un **array** è un insieme di dati omogenei (dello stesso tipo).
- I dati o *elementi* di un array hanno tutti la stessa dimensione (*esize*).
- La posizione (*indice*) identifica i singoli elementi.
- L'indirizzo dell'elemento *iesimo* è dato da:  $base + indice * esize$  dove *base* è l'indirizzo di memoria dove inizia l'array.
- Se devo accedere a più elementi posso semplificare il calcolo dell'indirizzo e risparmiare una o entrambe le operazioni (+, \*) cambiando opportunamente il modo di indirizzamento e l'incremento:
  - $base + indice * esize$  e incremento *indice* di 1
  - $base + offset$  e incremento *offset* di *esize*
  - *pointer* e incremento *pointer* di *esize*

### ▪ Esempio:

```
array: .byte 12,34,53,13,54
```

© 2002-2004 Riccardo Solmi

9

## Esempio su array

### ▪ Accesso agli elementi calcolando: $base + indice * esize$

```
lw $t1,size          # $t1 array size
li $t2,0             # $t2 indice
loop: mul $t3,$t2,4   # indice * esize
      lw $t4,array($t3) # carico l'iesimo elemento
      #... Faccio quello che voglio con l'elemento
      addu $t2,$t2,1    # incremento indice
      blt $t2,$t1,loop
      .data
array: .word 12,43,23,54,23
size: .word 5
```

- ### ▪ Caricando array in un registro ed esplicitando la somma si guadagna in efficienza ... (vedi prossimo lucido)

© 2002-2004 Riccardo Solmi

10

## Esempio su array

- **Accesso agli elementi calcolando: base + indice \* esize**

```
la $t0,array      # $t0 array base address
lw $t1,size       # $t1 array size
li $t2,0          # $t2 indice
loop: mul $t3,$t2,4 # indice * esize
      addu $t3,$t0,$t3 # $t3 base + indice*esize
      lw $t4,($t3)    # carico l'iesimo elemento
      #... Faccio quello che voglio con l'elemento
      addu $t2,$t2,1  # incremento indice
      blt $t2,$t1,loop
      .data
array: .word 12,43,23,54,23
size: .word 5
```

## Esempio su array

- **Accesso agli elementi calcolando: base + offset**

```
la $t0,array      # $t0 array base address
lw $t1,msize      # $t1 array memory size
li $t2,0          # $t2 offset
loop: addu $t3,$t0,$t2 # $t3 base + offset
      lw $t4,($t3)    # carico l'iesimo elemento
      #... Faccio quello che voglio con l'elemento
      addu $t2,$t2,4  # offset + esize
      blt $t2,$t1,loop
      .data
array: .word 12,43,23,54,23
msize: .word 20 # = 5*4
```

## Esempio su array e puntatori

### ▪ Accesso agli elementi usando un puntatore

```
la $t0,array      # $t0 array pointer
la $t1,aend       # $t1 array end
loop: lw $t4,($t0) # carico l'iesimo elemento
      #... Faccio quello che voglio con l'elemento
      addu $t0,$t0,4 # puntatore + esize
      blt $t0,$t1,loop
      .data
array: .word 12,43,23,54,23
aend:
```

- Con i puntatori otteniamo la soluzione più efficiente
- I linguaggi ad alto livello in genere obbligano il programmatore ad usare gli indici e non permettono di fare "aritmetica dei puntatori"
- I compilatori sono abbastanza intelligenti da fare traduzioni efficienti che usano i puntatori al posto degli indici.

## Strutture dati eterogenee: aggregati

### ▪ Definizione

- Un **aggregato** (detto anche record o struttura dati) è un insieme di dati eterogenei (di tipo diverso)
- I dati o *campi* di un aggregato possono avere dimensione diversa.
- Lo scostamento dall'inizio dell'aggregato (*offset*) identifica i singoli campi e permette di accedervi.
- L'indirizzo di un campo è dato da:  $base + offset$  dove *base* è l'indirizzo di memoria dove inizia l'aggregato.

### ▪ Esempio:

```
studente: .word nome,cognome # puntatori a ...
          .word 3773         # matricola
          .byte 33          # età

# fine aggregato
nome:     .ascii "Riccardo"
cognome:  .ascii "Solmi"
```

## Esempio su aggregati

- **Esempio di accesso ai campi di un aggregato:**

```
la $t0,studente # $t0 indirizzo aggregato
lw $t1,0($t0)   # $t1 puntatore al nome
lw $t2,4($t0)   # $t2 puntatore al cognome
lw $t3,8($t0)   # $t3 numero di matricola
lbu $t4,12($t0) # $t4 età

studente:.word nome,cognome
        .word 3773      # matricola
        .byte 33       # età

# fine aggregato
nome:   .asciiz "Riccardo"
cognome: .asciiz "Solmi"
```

## Strutture dati e allocazione dinamica

- **Le strutture dati (array e aggregati) possono essere allocate dinamicamente in memoria**

- Un programma è costituito da un insieme di funzioni che operano su *diversi tipi* di strutture dati.
- Una struttura dati può essere allocata (*staticamente*) nel segmento dati come abbiamo visto fino ad ora.
- In alternativa esiste un servizio del sistema operativo che alloca (*dinamicamente*) lo spazio di memoria necessario ad una struttura dati tutte le volte che gli viene richiesto.
- Una struttura dati allocata dinamicamente non possiede una etichetta che la identifica.

## Allocazione dinamica della memoria

---

- **La system call sbrk**
  - è identificata (in \$v0) dal codice 9
  - prende in input (in \$a0) la dimensione del blocco di memoria da allocare (in byte)
  - ritorna (in \$v0) un puntatore al blocco di memoria allocato
  - La memoria non è inizializzata
- **Esempio: legge una stringa allocando il buffer**

```
li $a0,256
li $v0,9 # sbrk
syscall
move $a0,$v0
li $a1,256
li $v0,8 # read_string
syscall
```

## Uso dell'allocazione dinamica della memoria

---

- **Quando usiamo l'allocazione dinamica al posto dell'allocazione statica nel segmento dati?**
  - Tutte le volte che non sappiamo a priori (prima di eseguire il programma) quanti elementi deve avere un array o di quante copie di un aggregato abbiamo bisogno.
- **In genere la quasi totalità dei dati viene allocata dinamicamente in memoria.**