Esercitazione del 14/04/2005 - Soluzioni

1. I registri del MIPS: convenzioni.

Il MIPS contiene 32 registri general-porpouse a 32bit per operazioni su interi (**$0..$31**), 32 registri general-porpouse per operazioni in virgola mobile a 32bit (**$FP0**..**$FP31**) più un certo numero di registri speciali, sempre a 32bit, usati per compiti particolari.
Tra questi ultimi, il **Program Counter** (**PC**) contiene l'indirizzo dell'istruzione da eseguire, **HI** e **LO** mantengono il risultato dell'ultima moltiplicazione effettuata e **Status** contiene i flag di stato della cpu.
Il MIPS è una cpu **RISC** ad architettura **load-store** che sfrutta massicciamente il **pipelining**. Ciò significa che il set di istruzioni disponibili nativamente sulla cpu è estremamente ridotto (Reduced Istruction Set Cpu) e limitato ad istruzioni che sfruttano appieno la struttura a fasi successive (il caricamento di una costante in un registro, ad esempio, è ottenuto con una sequenza equivalente di operazioni aritmetiche). Per analoghi motivi le operazioni da e verso la memoria sono limitate al solo caricamento e salvataggio dei registri. Non è possibile ad esempio sommare direttamente un dato contenuto in memoria con un registro: occorre prima spostare il dato in memoria in un registro della cpu.
Allo scopo di semplificare la programmazione in assembler del MIPS sono state adottate alcune convenzioni che permettono di creare delle pseudo-istruzioni facilmente traducibili nel set base del MIPS. Le più importanti riguardano:

- il registro **$0 ($zero)** settato costantemente al valore 0
- il registro **$1 ($at)** che viene usato come variabile temporanea nell'implementazione delle pseudo-istruzioni.

Ad esempio, la pseudo-istruzione load con indirizzamento immediato:

li $v0 , 4

che carica la costante 4 in $v0 è realizzata attraverso l'istruzione:

ori $vo, $zero, 4

che realizza un or logico con indirizzamento immediato tra il registro $0 (che vale sempre 0, elemento neutro per l'or) e la costante 4 e mette il risultato in $v0.

Altro esempio, la pseudo-istruzione di salto se minore tra registri:

ble $t6, $t1, Loop

che salta all'etichetta Loop se $t6 è minore (less) di $t1, è realizzata con la coppia di istruzioni:

slt $at, $t1, $t6
beq $at, $zero, Loop

Massimo Marchi – marchi@dsi.unimi.it

Le convenzioni adottate per i registri interi del MIPS sono riassunti di seguito:

Nome	Numero	Utilizzo
$zero	0	costante zero
$at	1	riservato per l'assemblatore
$v0-$v1	2-3	valori di ritorno di una procedura
$a0-$a3	4-7	argomenti di una procedura
$t0-$t7	8-15	registri temporanei (non salvati)
$s0-$s7	16-23	registri salvati
$t8-$t9	24-25	registri temporanei (non salvati)
$k0-$k1	26-27	gestione delle eccezioni
$gp	28	puntatore alla global area (dati)
$sp	29	stack pointer
$s8	30	registro salvato (fp)
$ra	31	indirizzo di ritorno

Il simulatore PCSpim mette a disposizione alcune procedure di sistema che possono essere usate per compiere delle azioni di I/O verso la console. Di seguito sono riportate le più comuni con un esempio di utilizzo:

Chiamate di sistema:

print_int:

```
                                # $a0 deve contenere l'intero da stampare,
                                # ex: move $a0 , $s0
            li $v0, 1           # $v0 codice della print_int
            syscall             # stampa della stringa
```

print_string:

```
            .data
str:        .asciiz "Inserire un numero intero:"
            [...]
            .text
            [...]
            li $v0, 4           # $v0 codice della print_string
            la $a0, str         # $a0 indirizzo della stringa con label str
            syscall             # stampa della stringa
```

read_int:

```
            li $v0, 5           # $v0 codice della print_string
            syscall             # stampa della stringa
```

exit:

```
            li $v0, 10          # $v0 codice della exit
            syscall             # exit
```

2. La struttura della memoria: area testo, area dati, area stack

Vedi slide L_15 Dal sorgente all'eseguibile. I programmi Assembly p.13

3. Uso delle costanti tramite indirizzamento immediato.

Attraverso l'indirizzamento immediato è possibile sostituire il secondo costante intera a 16bit. Nel caso sia necessario caricare un registro 32bit, es. caricare in **$a0** l'indirizzo di inizio di una stringa**,** si usa la pseudo-implementata dall'assemblatore in due passi: prima vengono caricati parola tramite l'istruzione **lui** (*load upper immediate*) quindi vengono bassi con un usuale istruzione **li.** (ogni istruzione MIPS è lunga sempre tolto lo spazio per il codice operativo dell'istruzione, non rimane spazio una costante a 32bit).

Ex: Programma dimostrativo dell'uso di costanti (file: somma.asm)

```
# Somma di due costanti.
# Dimostrativo dell uso delle costanti
# e del modo di indirizzamento immediato
        .data
pad:    .space 1            # spazio inserito per fare diventare un valore
                            # diverso da 0 la semi-word bassa di str
str:    .asciiz "10 + 15 = "
        .text
        .globl main
main:
        li $t1, 10          # carica il valore decimale 10 nel reg. $t1
        li $t2, 15          # carica il valore decimale 15 nel reg. $t2
        add $s0, $t1, $t2   # $s0 = $t1 + $t2

        li $v0, 4           # stampa la stringa che inizia all'indirizzo str
        la $a0, str         # pseudo-istruzione che carica una costante a 32 bit
        syscall

        li $v0, 1           # stampa risultato (10 + 15 = 25)
        move $a0 , $s0      #
        syscall

        li $v0, 10          # exit
        syscall
```

Ex2: Versione compatta (file: somma2.asm)

```
# Somma di due costanti (seconda versione compatta)
# Dimostrativo dell uso delle costanti
# e del modo di indirizzamento immediato
        .data
str:    .asciiz "10 + 15 = "
        .text
        .globl main
main:
        li $t1, 10          # carica il valore decimale 10 nel reg. $t1
        addi $s0, $t1, 15   # $s0 = $t1 + 15

        li $v0, 4           # stampa la stringa che inizia all'indirizzo str
        la $a0, str         # pseudo-istruzione che carica una costante a 32 bit
        syscall

        li $v0, 1           # stampa risultato (10 + 15 = 25)
        move $a0 , $s0      #
        syscall

        li $v0, 10          # exit
        syscall
```

4. Allocazione ed allineamento dei byte in memoria e nei registri.

Il MIPS trasferisce i dati da/verso la memoria in parole formate da **4 byte (word)** allineati a indirizzi multipli di 4. Nei trasferimenti che interessano word o half-word è necessario che i dati in memoria siano opportunamente allineati; in caso contrario si verifica una eccezione (*Exception 4 [Address error in inst/data fetch]*).

Ex: Programma dimostrativo per gli allineamenti (file: wordmem.asm).

```
# Programma dimostrativo per gli allineamenti delle word
# in memoria e nei registri.

        .data
carat:  .byte  0x21
        .byte  0x22
        .align 2         # riallineo i dati alle word
testo:  .asciiz "0123456789"
space:  .asciiz " "
        .text
        .globl main
main:
        li $v0 , 4       # seleziono con $v0 la syscall print_str (4)
        la $a0 , testo   # indico in $a0 la stringa da stampare
        syscall          # invoco la syscall

        li $v0 , 4       # stampo uno spazio
        la $a0 , space
        syscall

        la $a0 , testo   # carico i primi 4 caratteri come fossero una word
        lw $t0 , 0($a0)
        #lw $t0 , 1($a0) # questa linea genera un eccezione perchè
                         # la word puntata non è allineata correttamente
        li $v0 , 1       # seleziono con $v0 la syscall print_int (1)
        move $a0 , $t0   # indico in $a0 l'intero da stampare
        syscall

        li $v0 ,....     # stampo uno spazio (vedi segmento di codice precedente)

        la $a0, carat    # carico l'indirizzo del byte puntato da carat
        lbu $t0, 0($a0)
        move $a0 , $t0
        li $v0 , 1
        syscall

        li $v0 ,....     # stampo uno spazio (vedi segmento di codice precedente)

        la $a0, carat    # carico l'indirizzo del byte puntato da carat
        lbu $t0, 1($a0)  # qui l'allineamento è irrilevante perchè stiamo
        li $v0 , 1       # trasferendo un byte.
        move $a0 , $t0
        syscall

        li $v0 ,....     # stampo uno spazio (vedi segmento di codice precedente)

        la $a0, carat    # carico l'indirizzo del byte puntato da carat
        lhu $t0, 2($a0)  # qui l'allineamento è importante perchè stiamo
        #lhu $t0, 1($a0)  # qui l'allineamento è importante perchè stiamo
        li $v0 , 1       # trasferendo una half-word.
        move $a0 , $t0
        syscall

        li $v0 , 1       # exit
        syscall
```

5. Implementazione dei cicli for e uso della moltiplicazione.

I registri speciali **HI** e **LO** del MIPS sono usati perconservare il risultato dell'ultima moltiplicazione eseguita. La pseudo istruzione

mul rd , rs1 , rs2

carica in **HI** e **LO** il risultato della moltiplicazione **rs1 * rs2** quindi sposta **LO** nel registro **rd.**

Ex. Programma dimostrativo dell'uso di cicli e mul (file sommaQuadrati.asm)

```
# Programma che calcola la somma dei quadrati da 0 a N
# Dimostrativo dell'implementazione di cicli for
# e dell'uso di istruzioni mul

        .data
msg:    .asciiz "Inserisci N "
str:    .asciiz "La somma dei quadrati da 0 a N vale "
        .text
        .globl main

main:
        li $v0, 4           # Stampo messaggio di richiesta
        la $a0, msg
        syscall

        li $v0 , 5          # Carico un intero da console
        syscall

        move $s0 , $v0      # $s0 contiene N
        li $s1, 0           # $s1 contiene la somma dei quadrati
        li $t0, 0           # $t0 e' l'indice di ciclo
Loop:
        mul $t1, $t0 $t0    # calcolo il quadrato: $t1 = $t0 x $t0
        addu $s1, $s1, $t1  # accumulo le somme:  $s0 += $t1
        addi $t0, $t0, 1    # incremento l'indice $t0++
        ble $t0, $s0, Loop  # se indice < N salta a Loop  if $t0<$s0 goto Loop

        li $v0, 4           # Stampo messaggio di output
        la $a0, str
        syscall

        li $v0, 1           # Stampo somma
        add $a0, $s1, $zero # eq. alla pseudo-instr. move $a0 , $s1
        syscall

        li $v0, 10          # $v0 codice della exit
        syscall             # esce dal programma
```

6. Uso del registro $ra.

Ogni qualvolta viene eseguita l'istruzione **jal** il MIPS memorizza l'indirizzo dell'istruzione seguente nel registro **$31** (**$ra**). Se il registro non viene sovrascritto, alla fine della routine invocata è possibile usare questo valore per tornare indietro al punto di chiamata con l'istruzione

jr $ra.

Il PCSpim prima di lanciare effettivamente un programma caricato in memoria effettua alcune operazioni di servizio. Al termine di queste operazioni, prosegue l'esecuzione all'indirizzo indicato nel programma tramite l'etichetta **main:** attraverso un'istruzione **jal** opportuna. Se il programma termina con un istruzione **jr $ra** e il registro **$ra** non è stato cambiato dal programma, allora l'esecuzione del programma ritornerà al termine alle istruzioni seguenti **jal main:** che altro non sono che un'invocazione alla syscall **exit.**

Ex: Programma dimostrativo dell'uso del registro $ra (file: do_nothing.asm)

```
# Programma che non fa niente.
# Dimostrativo dell'uso del registro $ra
        .data
        .text
        .globl main
main:
        move $s0 , $ra          # salvo $ra in $s0 poichè la subroutine
                                # do_nothing lo riscriverà quando invocata
        jal do_nothing          # richiamo una sub-routine che non fa niente
        move $ra , $s0          # ripristino $ra

        jr $ra                  # salto indietro all'exit di PCSpim

do_nothing:
        nop                     # non fare niente
        jr $ra                  # return
```

7. Uso delle tabelle di lookup.

Tramite una tabella di lookup è possibile selezionare una procedura in un set attraverso un indice numerico che rappresenta l'ordinale della procedura all'interno set. Nella tabella saranno memorizzati tutti gli indirizzi di inizio delle varie procedure. Dato l'ordinale **i**, l'indirizzo di partenza della procedura **i**-esima sarà memorizzzato nella word di indirizzo **base_lookup** + **i** * 4, dove base è l'indirizzo di inizio della tabella di lookup.

Ex. Programma dimostrativo delle tabelle di lookup (lookup.asm *).*

```
# Programma dimostrativo dell'uso di tabelle di lookup
# per richiamare un set di procedure attraverso un indice
# (ex. syscall o switch)
        .data
lookup:  .space 12
msga:    .asciiz "Inserisci l'indice della funzione[0..2]: "
msg1:    .asciiz "Prima procedura"
msg2:    .asciiz "Seconda procedura"
msg3:    .asciiz "terza procedura"
        .text
        .globl main
main:
# carico la lookup table con i valori opportuni
        la $s0 , lookup   # $s0 contiene l'indirizzo della tabella lookup
        la $t0 , proc1    # salvo l'indirizzo dalla prima procedura
        sw $t0 , 0($s0)
        la $t0 , proc2    # salvo l'indirizzo dalla seconda procedura
        sw $t0 , 4($s0)
        la $t0 , proc3    # salvo l'indirizzo dalla terza procedura
        sw $t0 , 8($s0)

        li $v0 , 4        # stampo messaggio di richiesta
        la $a0 , msga
        syscall
        li $v0 , 5        # leggo un intero da console
        syscall           # numero in $v0

# calcolo l'indirizzo della procedura richiesta e la richiamo
        li $t0 , 4
        mul $t0 , $t0 , $v0   # $t0 = $vo * 4
        add  $t1 , $t0 , $s0  # $t1 = $t0 + lookup
        lw $t2 , 0($t1)       # $t2 = ($t1)
        jal $t2               # salto alla procedura

        li $v0, 10            # exit
        syscall

# set di procedure
proc1:   li $v0 , 4
        la $a0 , msg1
        syscall
        jr $ra

proc2:   li $v0 , 4
        la $a0 , msg2
        syscall
        jr $ra

proc3:   li $v0 , 4
        la $a0 , msg3
        syscall
        jr $ra
```

8. Uso dello stack.

Nel MIPS non esistono istruzioni specifiche per gestire lo stack dei dati. L'implemetazione dello stack viene fatta in software usando il registro **$29** (**$sp**) e le usuali operazioni di load/store e add con indirizzamento immediato.
Le procedure assembler si possono suddividere in due classi distinte: procedure **rientranti** e procedure **non rientranti.** Nelle procedure rientranti è possibile durante l'esecuzione di una chiamata effettuare chiamate ricorsive alla procedura stessa. Al contrario nelle procedure non rientranti la procedura non può essere richiamata finché l'ultima chiamata non è terminata. Il principale problema da risolvere per ottenere procedure rientranti è evitare che modifiche sui registri effettuate da livelli di chiamata più profondi influenzino la computazione del livelli superiori. Questo in genere è ottenuto salvando lo stato dei registri utilizzati dalla procedura nello stack all'inizio della chiamata. Una volta terminata la procedura, appena prima di effettuare il ritorno al punto di chiamata, lo stato dei registri eventualmente alterati dalla procedura stessa viene ripristinato recuperandolo dallo stack. Questa operazione equivale a creare delle variabili locali alla procedura.
L'implementazione di procedure ricorsive deve essere fatta ovviamente sempre con procedure rientranti.

Ex. Segmento di codice C che calcola il valore di Fibonacci per un intero n.

```
main(){
  int n,f;

  printf("Inserire un numero intero: ");
  scanf("%d",&n);

  f = fib(n);

  printf("Numero di Fibonacci: ");
  printf("%d",f);

  exit();
}

int fib(int n) {
  int f ;

  if (n>2) f=fib(n-1)+fib(n-2) ;
  else f=1;

  return f;
}
```

Poiché la procedura fib(.) richiama se stessa se n è maggiore di due, allora occorre ad ogni chiamata creare due nuove aree di memoria dove memorizzare i valori di f ed n attuali per la chiamata, in modo da non alterarne i valori per i livelli di chiamata superiori.

Ex. Programma dimostrativo dell'uso dello stack (fibonacci.asm *).*

```
# Programma che calcola il numero di Fibonacci di un intero
# tramite un'algoritmo ricorsivo:
# fib(n) = fib(n-1) + fib (n-2) se n > 2
#        = 1                    se n = 1,2
# Ex: fib(1)=1, fib(2)=1, fib(3)=2, fib(4)=3, 5, 8, 13, ....
# Dimostrativo dell'uso dello stack.

        .data
prompt: .asciiz "Inserire un numero intero: "
output: .asciiz "Numero di Fibonacci: "
        .text
        .globl main
main:
        li $v0, 4           # stampa la stringa
        la $a0, prompt
        syscall

        li $v0, 5           # legge l'intero
        syscall
        move $s0 , $v0      # salva l'intero in $s0

# calcola fibonacci(n)
        move $a0, $s0       # Chiama la procedura con $a0 = n
        jal fib

# stampa il risultato ed esci
        move $s1, $v0       # salva il valore restituito da fib in $s1
        li $v0, 4           # stampa il messaggio di output
        la $a0, output
        syscall

        move $a0, $s1       # stampa il risultato
        li $v0, 1
        syscall

        li $v0, 10          # exit
        syscall

fib:
        addi $sp, $sp,-12   # crea uno spazio di 4 word nello stack
        sw $ra, 0($sp)      # salva l'indirizzo di ritorno della chiamata
        sw $a0, 4($sp)      # salva il valore di $a0 e $s0 che verranno usate
        sw $s0, 8($sp)      # dalla procedura ($t0 non viene salvato)

        li $t0, 2           # se n> 2 esegui ricorsione
        bgt $a0, $t0, ricorsione

        li $v0, 1           # altrimenti restituisci 1
        j return

ricorsione:
        addi $a0, $a0,-1    # n -> n-1
        jal  fib            # chiama fib(n-1)
        move $s0, $v0       # salva fib(n-1) in una variabile che non viene
                            # alterate dalla procedura
        addi $a0, $a0,-1    # $a0 diventa n-2
        jal  fib            # esegue fib(n-2)
        add  $v0, $v0, $s0  # somma  fib(n-1) e fib(n-2)

return:
        lw $ra, 0($sp)      # ripristina i vecchi valori di $ra
        lw $a0, 4($sp)      # $a0 e
        lw $s0, 8($sp)      # $s0
        addi $sp, $sp, 12   # disalloca lo spazio creato nello stack
        jr $ra              # ritorna al punto di chiamata
```