Laboratorio di Architettura degli Elaboratori I

Barbara Carminati

carminati@dsi.unimi.it

http:/homes.dsi.unimi.it/~carminat/lab.htm

Programma del corso

- **Gerarchia dei linguaggi di programmazione**
- **Architetture Cisc/Risc**
- **Linguaggio assembly del processore MIPS R2000:**
  - **Formato delle istruzioni: R-I-J**
  - **Gestione degli operandi**
  - **Modalità di indirizzamento**
  - **Gestione delle subroutine on linguaggio assembly**
  - **Gestione dello stack e la ricorsione in liguaggio assembly**

Programma del corso

- **Gestione del software e delle operazione I/O**
  - **Dispositivi di I/O**
  - **Modalità di gestione dell'I/O**
  - **Gestione a controllo da programma**
  - **Polling**
  - **Gestione via Interrupt**
  - **Gestione software delle eccezioni**

Laboratorio

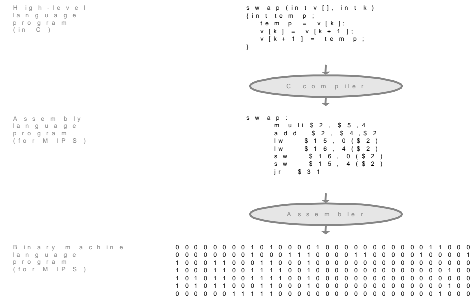
- 8 ore di laboratorio:
  - Si utilizzerà SPIM, un simulatore per processore MIPS scaricabile da:

Unix, Linux	**Spim, Xspim**	**http://www.cs.wisc.edu/~larus/SPIM/spim.tar.gz**
Windows 95, 98, NT, 2000	**Spim, PCSpim**	**http://www.cs.wisc.edu/~larus/SPIM/pcspim.exe**
Dos	**Spim**	**http://www.cs.wisc.edu/~larus/SPIM/spimdos.exe**

Bibliografia

- *Struttura, organizzazione e progetto – L'interfaccia Hardware.* 2a Ed, D.A. Patterson e J.L. Hennessy, Morgan Kaufmann ed., 1997
- *La Gestione software dei dispositivi di I/O.* D. Bruschi e E.Rosti, 1996

Livelli di programmazione



Architettura del set di istruzioni (ISA)

E' l'insieme di nozioni necessarie per creare un programma in linguaggio macchina. (istruzioni, dispositivi I/O, etc.)

Software

Instruction set Architecture

Hardware

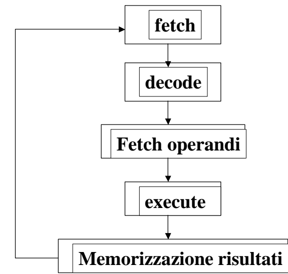
Il linguaggio Assembly

- E' una rappresentazione simbolica del linguaggio macchina
- E' più comprensibile del linguaggio macchina in quanto utilizza simboli invece che sequenze di bit
- Rispetto ai linguaggi ad alto livello, l'assembly fornisce limitate forme di controllo del flusso e di strutture dati

Linguaggio macchina

- Consente di gestire l'hardware di un calcolatore
- Le "parole" del linguaggio di un calcolatore sono le *istruzioni,* mentre il vocabolario è il *set di istruzioni (instruction set)*

Instruction set



Instruction set

- Come sono codificate le istruzioni?
- Dove risiedono gli operandi delle istruzioni?
- Quali tipi di dati? Che dimensione?
- Quali operazioni sono consentite?

Linguaggio C: esempio

```
Int
main(int argc, char *argv[ ])
{
        int i;
        int sum = 0;
        for (i = 0; i <= 100; i = i + 1)
          sum = sum + i;
        printf("The sum from 0 .. 100 is
  %d\n",sum);
}
```

Linguaggio Assembly: esempio

```
.text                          lw $t8, 24($sp)
.align 2                               addu $t9, $t8, $t7
.globl main                    sw  $t9, 24($sp)
main:                          addu $t0, $t6, 1
        subu $sp, $sp, 32      sw $t9, $t8, $t7
        sw $ra, 20($sp)        sw $t9, 24($sp)
        sw $a0, 32 ($sp)       addu $t0, $t6, 1
        sw $0, 24($sp)          ………………
        sw $0, 28($sp)
loop:   lw  $t6, 28($sp)
        mul $t7, $t6, $t6
```

Il linguaggio Assembly

I ruoli principali dell'Assembly sono **due**:

- **E' il linguaggio ottenuto dalla fase di compilazione di un programma scritto in un linguaggio ad alto livello (es: C, Pascal,ecc.)**
- **E' un vero e proprio linguaggio di programmazione**

L'Assembly come linguaggio di programmazione

E utile programmare in Assembly quando:

- **è fondamentale massimizzare la velocità di esecuzione**
- **si vogliono sfruttare al massimo le potenzialità dell'hardware sottostante**

L'Assembly come linguaggio di programmazione

I principali svantaggi della programmazione in Assembly sono:

- **i programmi non sono portabili su macchine diverse**
- **i programmi assembly sono molto più lunghi e di difficile comprensione rispetto ai programmi scritti in un linguaggio ad alto livello**

L'Assembly come linguaggio di programmazione

- L'assembly fornisce limitate forme di strutture di controllo
- La gestione delle strutture dati e delle chiamate a procedura deve essere fatta in modo esplicito

L'Assembly come linguaggio di programmazione

alcune volte si usa un approccio ibrido in cui le parti più critiche sono scritte in assembly e le altre in un linguaggio ad alto livello

Evoluzione dei microprocessori

- **Anni 50 – primi calcolatori: poche istruzioni e solo 1 o 2 modi d'indirizzamento.**
- **Anni 60 – introduzione della microprogrammazione: elevato numero di istruzioni e diversi metodi di indirizzamento.**
- **Anni 60-70 – diffusione dei linguaggi ad alto livello: aumento del gap semantico.**

  CISC Complex Instruction Set Computer

Evoluzione dei microprocessori

- **Anni 80 –nuova tendenza che ha fatto un passo "indietro", introducendo:**

  RISC Reduced Instruction Set Computer
- **Non si utilizzano i microprogrammi.**
- **Il set di istruzioni è ridotto alle sole istruzioni usate più frequentemente ed ha un formato uniforme.**
- **Un solo metodo d'indirizzamento (register-to-register).**

CISC

- **Porta la complessità da software ad hardware**
- **Vantaggi:**
  - **Migliora il compilatore**
  - **Diminuisce la lunghezza del codice (con conseguente risparmio di RAM)**
  - **Facilita l'operazione di debugging**

RISC

- **Porta la complessità da hardware a software**
- **Vantaggi:**
  - **Le operazioni di base sono eseguite in un solo ciclo.**
  - **Permette un pipeline efficace.**

Esempi di CISC e RISC

	CISC			**RISC**	
	IBM 370/168	**VAX 11/780**	**Intel 80486**	**SPARC**	**MIPS R4000**
Anno	**1973**	**1978**	**1989**	**1987**	**1991**
Num. Istru	**208**	**303**	**235**	**69**	**94**
Lung. Istru	**2-6**	**2-57**	**4**	**4**	**4**
Modi Ind	**4**	**22**	**11**	**1**	**1**
Numero registri general pu	**16**	**16**	**8**	**40-250**	**32**

- Attualmente:
  - CISC: Pentium (Intel)
  - RISC: PowerPc (Mac), Risc System/6000 (IBM)
  - RISC: MIPS (Sun Spar, HP PA-RISC, IBM, Power PC)

Architettura MIPS

- Il linguaggio assembly che noi vedremo è quello dell'**architettura MIPS**
- MIPS ha un architettura RISC
- Esempi: Sun Sparc, HP PA-RISC, IBM Power PC
- Gli operandi di una istruzione devono sempre risiedere nei **registri** (macchina load/store)
- Nel MIPS i registri sono **32** di dimensione 32 bit (**una word**)

I registri

- I registri sono associati alle variabili di un programma dal compilatore
- Per convenzione si usano $s0, $s1, ... $t0, $t1, … per denotare i registri
- I registri possono essere anche direttamente denotati mediante il loro numero (0, …, 31) preceduto da $

Formato istruzioni

- **Tutte le istruzioni MIPS hanno la stessa dimensione (32 bit)**
- **I 32 bit hanno un significato diverso a seconda del tipo di istruzione**
- **In MIPS le istruzioni sono di 3 tipi:**
  - **tipo R**
  - **tipo I**
  - **tipo J**

tipo R	op	rs	rt	rd	shamt	funct
tipo I	op	rs	rt	16 bit address		
tipo J	op	26 bit address				

Istruzioni aritmetiche

- Ogni istruzione aritmetica ha esattamente **tre** operandi
- L'ordine degli operandi **è fisso** (prima il registro contenente il risultato dell' operazione e poi i due operandi)

L'istruzione Add

Add serve per sommare il contenuto di due registri:

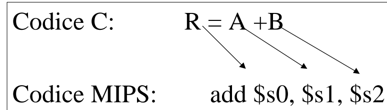
add rd rs rt

mette la somma del contenuto di rs e rt in rd

Tipo R	op	rs	rt	rd	shamt	funct

Add: esempio

Codice C: R = A +B



Codice MIPS: add $s0, $s1, $s2

i registri sono associati alle variabili dal compilatore

Istruzioni aritmetiche

Il fatto che ogni istruzione aritmetica abbia tre operandi consente di semplificare l'hw, ma complica alcune cose …

Codice C:	**A = B + C + D**
	E = F - A
Codice MIPS:	**add $t0, $s1, $s2**
	add $s0, $t0, $s3
	sub $s4, $s5, $s0

L'istruzione Sub

- **Sub** serve per sottrarre il contenuto di due registri:

sub rd rs rt

mette la sottrazione del contenuto di rs e rt in rd

Tipo R	op	rs	rt	rd	shamt	funct

- Analogamente alla somma => subu

Rappresentazione Binaria

- Ogni parola di memoria sono 32 bits.
- Cosa si può rappresentare in binario con 32 bit?
  - Numeri positivi (unsigned):
    - $[0, 2^{32}-1]$ => [0,4292967295]
  - Numeri con segno (complemento a due):
    - $[-2^{31},+ 2^{31}-1]$ => [-2147483648, 2147483647]

Add: varianti

- In generale, per ogni istruzione aritmetica esiste la corrispondente istruzione:
  - *Unsigned*
    addu $s0, $s1, $s2
  - *Immediate*
    addi $s1, $s2, 100
  - *Immediate & Unsigned*
    addiu $s0, $s1, 100

Overflow

- La dimensione finita delle parole di memoria implica che le operazioni aritmetiche possano generare risultati troppo grandi per essere rappresentati nella dimensione fissa delle parole.

OVERFLOW

Es. 2^{32}-1 + 1 = 4292967296=

100000000000000000000000000000000

…..…33 bits …..….

Overflow

- Gli interi senza segno (unsigned) sono generalmente usati per rappresentare gli indirizzi di memoria, dove l'overflow puo' essere ignorato.
- Per gestire gli questa distinzione:
  - Addi, sub causano eccezione in caso di overflow;
  - Addu, addiu, subu, non causano eccezione in caso di overflow.

Moltiplicazione

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

  n bit X m bit= nm bit
- Due istruzioni:
  - mult rs rt
  - multu rs rt (unsigned)

Moltiplicazione

- MIPS fornisce due registri *Hi* e *Lo* per contenere il risultato della moltiplicazione
- *nbit X mbit =nm bit :*
  - *in Lo sono memorizzati i primi 32 bit di nm (partendo da sx)*
  - *in Hi sono memorizzati i 32 successivi*
- Istruzioni per prendere i dati da *Hi* e *Lo*
  - mfhi rd (move from *Hi*)
  - mflo rd (move from *Lo*)

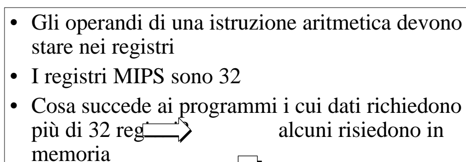
Moltiplicazione

- Se il prodotto e' contenuto in 32 bits, allora e' sufficiente prelevarlo da Lo attraverso *mflo*
- *Per controllare che prodotto contenuto in 32 bits:*
  - *Se Multu -> Hi deve contenere 0*
  - *Se Mult -> Hi deve contenere la replica del segno di Lo.*

Divisione

- Il MIPS utilizza *Hi* e *Lo* anche per l'operazione di divisione:
  - div rs rt (divide rs per rt)
  - divu rs rt
- Il quoziente della divisione è posto nel registro *Lo*, mentre il resto è posto in *Hi*

Istruzioni aritmetiche

- Gli operandi di una istruzione aritmetica devono stare nei registri
- I registri MIPS sono 32
- Cosa succede ai programmi i cui dati richiedono più di 32 reg  alcuni risiedono in memoria

servono istruzioni per trasferire dati da memoria a registri e viceversa

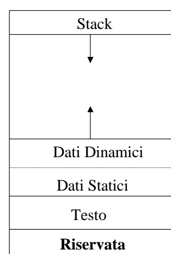
MIPS: utilizzo della memoria

Nei sistemi basati su processore MIPS la memoria è solitamente divisa in **tre** parti:
- **Segmento testo**: contiene le istruzioni del programma
- **Segmento dati**: ulteriormente suddiviso in:
  - *dati statici*: contiene dati la cui dimensione è conosciuta al momento della compilazione e il cui intervallo di vita coincide con l'esecuzione del programma

MIPS: utilizzo della memoria

Cont:
  - *dati dinamici*: contiene dati a cui lo spazio è allocato dinamicamente al momento dell'esecuzione del programma
- **Segmento stack**: contiene lo stack eventualmente allocato da un programma

Suddivisone della memoria



Indirizzamento della memoria

- La memoria è vista come un grosso array uni-dimensionale
- Un **indirizzo di memoria** è un indice all'interno dell'array
- MIPS utilizza un **indirizzamento al byte**, cioè l'indice punta ad un byte di memoria

Indirizzamento al byte

1 byte	1 byte	1 byte	1 byte	1 byte	1 byte
0	1	2	3	4	5

Indirizzamento della memoria

- Gli indirizzi di parole adiacenti differiscono per un fattore quattro
- In MIPS ogni parola deve iniziare ad un indirizzo multiplo di 4
- Per convenzione l'indirizzo di una parola coincide con l'indirizzo del suo byte *più a sinistra*

Indirizzamento della memoria

12	**32 bit**
8	**32 bit**
4	**32 bit**
0	**32 bit**

- 2^{32} byte con indirizzi $0,\ldots,2^{32}-1$
- 2^{30} parole con indirizzi $0,4,\ldots,2^{32}-4$

Istruzioni di trasferimento dati

- MIPS fornisce due operazioni base per attuare il trasferimento dati:
  - **lw** (load word) per trasferire una parola di memoria in un registro
  - **sw** (store word) per trasferire il contenuto di un registro in una parola di memoria

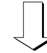
lw e sw richiedono come argomento l'indirizzo della locazione di memoria su cui devono operare

Istruzione lw

- L'istruzione lw ha tre argomenti:
  - il registro in cui va caricata la parola di memoria
  - una costante (*spiazzamento -- offset)*
  - un registro (*registro base -- base register*)
- L'indirizzo della parola di memoria da accedere è ottenuto dalla somma della costante e del contenuto del registro base

Istruzione lw

lw $s1, 100($s2)



Ad $s1 è assegnato il valore contenuto all'indirizzo contenuto nel registro $s2 + 100

Istruzione sw

- Ha argomenti analoghi alla lw

Esempio:

sw $s1, 100($s2)

Alla locazione di memoria di indirizzo $s2 + 100 è assegnato il valore contenuto in $s1

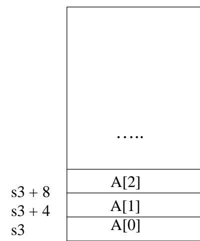
Lw & sw: esempio

Codice C: A[12] = h + A[8];

Codice MIPS:

```
lw $t0, 32($s3)
add $t0, $s2, $t0
sw $t0, 48($s3)
```

Array



Array

- L'elemento i-esimo di una array si troverà nella locazione br + 4 * i (dove br è il registro base)

Array

- A: array di 100 word
- g = h + A[i]
- si suppone che:
  - le variabili g, h, i siano associate a $s1, $s2, ed $s4
  - L'indirizzo del primo elemento è in $s3

Array

- L'elemento i-esimo si trova in $s3 + 4 * i:
  - add $t1, $s4, $s4
  - add $t1, $t1, $t1
  - add $t1, $t1, $s3
- Per trasferire A[i] in $t0:
  - lw $t0, 0($t1)
- Per sommare h e A[i]: add $s1, $s2, $t0

Riassumendo

- Le istruzioni aritmetiche leggono il contenuto di due registri, attuano su questo una computazione e scrivono il risultato in un terzo registro
- Le operazioni di trasferimento dati leggono e scrivono un solo operando senza effettuare nessuna computazione

Pseudoistruzioni

- Per semplificare la programmazione, MIPS fornisce un insieme di *pseudoistruzioni*
- 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

Esempio

- move $t0, $t1 (add $t0, $zero, $t1)
- mul $s0, $t1, $t2
- **div** $s0, $t1, $t2

…..

Linguaggio macchina

- Le istruzioni in linguaggio Assembly devono essere tradotte in linguaggio macchina (cioè in sequenze di 0 e 1) per poter essere eseguite
- Le istruzioni in linguaggio macchina sono lunghe **32 bit** (come i registri e le parole di memoria)

Linguaggio macchina

- E' necessaria una convenzione per rappresentare i registri tramite numeri
- In MIPS:
  – \$s0 = 16, \$s1 = 17, ….,\$s7 = 23
  – \$t0 = 8, \$t1 = 9, ….,\$t7 = 15

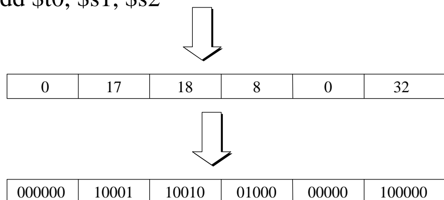
Formato istruzioni aritmetiche

op	rs	rt	rd	shamt	funct
6 bit	5 bit	5 bit	5 bit	5 bit	6 bit

- **op:** (opcode) identifica il tipo di istruzione
- **rs**: registro con il primo operando
- **rt**: registro con il secondo operando
- **rd**: registro contenente il risultato
- **shamt**: shift amount
- **funct**: indica la variante specifica dell'operazione

Istruzioni aritmetiche: esempio

add \$t0, \$s1, \$s2



Istruzioni di tipo R

- Istruzioni con il tipo di formato visto, vengono chiamate di **tipo R** (registro)

Linguaggio macchina

- Il formato delle istruzioni di tipo R non è adatto a rappresentare istruzioni di trasferimento dati
- Alla costante delle istruzioni lw e sw sarebbe riservato un campo di 5 bit ($2^5 = 32$)
- Si utilizza un formato diverso (**tipo I**) utilizzando sempre 32 bit complessivi

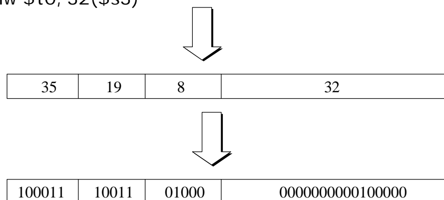
Istruzioni di tipo I

op	rs	rt	indirizzo
6 bit	5 bit	5 bit	16 bit

- Con questo formato una istruzione lw (sw) può indirizzare parole nel range -2^{15} $+2^{15}$ rispetto ad un indirizzo base

Istruzioni di tipo I

lw $t0, 32($s3)



35	19	8	32

100011	10011	01000	0000000000100000

Esempio

A[300] = h + A[300]

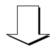
```
lw $t0, 1200($t1)
add $t0, $s2, $t0
sw $t0, 1200($t1)
```

$s2 ⟶ h

$t1 ⟶ Indirizzo base di A

Esempio (cont.)

35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		



100011	01001	01000	0000010010110000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000010010110000		

Le strutture di controllo

- Queste istruzioni:
  - alterano l’ordine di esecuzione delle istruzioni
  - cambiano cioè la successiva istruzione da eseguire

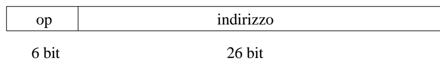
Istruzioni di scelta e salto

- Salti condizionati:
  - beq r1 r2 L (*branch if equal*)
  - bne r1 r2 L (*branch if not equal*)
- Salti incondizionati:
  - j label (*jump*)
  - jr r (*jump a registro*)
  - jal label (*jump and link*)

Istruzioni di scelta e salto

Nome	Formato
beq	I
bne	I
j	J
jr	R
jal	J

Formato J



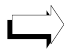
If … then

if (i != j) then
    h = i + j;

```
#$s4 e $s5 contengono i e j
#$s3 contiene h

beq $s4, $s5, Lab1
add $s3, $s4, $s5
Lab1:
```

If … then …else

if (i != j) then
    h = i + j;
else h = i - j;



```
beq $s4, $s5, Else
add $s3, $s4, $s5
j Exit
Else: sub $s3, $s4, $s5
Exit: ….
```

Do … while

do
    g = g + A[i];
    i = i + j;
while (i != h)

```
#$s1 e $s2 contengono g e h
#$s3 e $s4 contengono i e j
#$s5 contiene lo start address di A
```

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

While

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

```
#$s3 $s4, e $s5 contengono i, j e k
#$s6 contiene lo start address di A

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

Strutture di controllo

- Spesso è utile condizionare l'esecuzione di una istruzione al fatto che una variabile sia minore di una altra:
  - slt $s1, $s2, $s3
- Con slt, beq e bne si possono implementare tutti i test sui valori di due variabili (=, !=, <, <=, >,>=)

Esempio

if (i < j) then
    k = i + j;
else k = i - j;

```
#$s0 ed $s1 contengono i e j
#$s2 contiene k

slt $t0, $s0, $s1
beq $t0, $zero, Else
add $s2, $s0, $s1
j Exit
Else: sub $s2, $s0, $s1
Exit:
```

Struttura switch/case

- Può essere implementata mediante una serie di *if-then-else*
- Alternativa: uso di una *jump address table* cioè di una tabella che contiene una serie di indirizzi di istruzioni alternative

Struttura switch/case

```
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;
}
```

Jump address table

t4 + 12	L3
t4 + 8	L2
t4 + 4	L1
t4	Lo

Struttura switch/case

```
#$s0, .., $s5 contengono f,..,k
#$t2 contiene la costante 4
#$t4 contiene lo start address
#della jump address table

slt $t3, $s5, $zero
bne $t3, $zero, Exit
slt $t3, $s5, $t2
beq $t3, $zero, Exit
add $t1, $s5, $s5
add $t1, $t1, $t1
add $t1, $t1, $t4
lw $t0, 0($t1)
jr $t0
```

```
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:
```

Formato istruzioni

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	indirizzo		
J	op	indirizzo				

Direttive

- Le direttive (data layout directives) danno delle indicazioni all'assemblatore sul contenuto di un file (istruzioni, strutture dati, ecc.)
- Sintatticamente le direttive iniziano tutte con il carattere "."

Direttive

- .data <addr>: gli item successivi sono memorizzati nel segmento dati partendo da addr
- .asciiz str memorizza str terminandola con un valore nullo (.ascii str ha lo stesso effetto ad eccezione del valore nullo)
- .byte b1,…,bn memorizza i valori b1, .., bn in byte successivi di memoria

Direttive

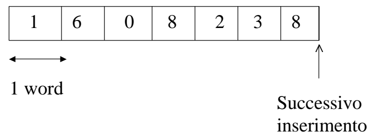
- .word w1, ..,wn memorizza i valori w1, .., wn in successive parole di memoria.
- .half h1, ..,hn memorizza i valori h1, .., hn in successivi 2 byte (mezze parole) di memoria
- .space n alloca n byte di spazio nel segmento dati
- .text <addr> memorizza gli item successivi nel segmento testo partendo da addr

Direttive

- .globl sym dichiara sym come etichetta globale (può essere riferita da altri file)
- .align n indica che i successivi dati devono essere allineati rispetto ad un limite di 2^n byte:
  - align 2 = .word
  - align 1 = .half

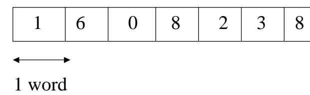
.word: esempio

.word 1, 6, 0, 8, 2, 3, 8



.word: esempio

array: .word 1, 6, 0, 8, 2, 3, 8



array contiene l'indirizzo del primo elemento

Direttive: esempio

```
# Somma valori in un array
         .data
array:  .word   1,2,3,4,5,6,7,8,9,10  #dichiarazione array
         .text
         .globl  main
main:
       li      $s0,10              #numero elementi
       la      $s1,array           #s1: registro base per array
       li      $s2,0               #contatore elementi per ciclo
       li      $t2,0               #azzero accumulatore
loop: lw    $t1,0($s1)          #accesso all'array
       add  $t2,$t1,$t2         #calcolo risultato
       addi $s1,$s1,4           #s1 contiene l'indirizzo del
                                #prossimo elemento
       addi $s2,$s2,1           #incremento contatore cicli
       bne  $s2,$s0,loop        #test terminazione
```

Direttive: esempio

```
# Somma numeri memorizzati in memoria centrale
        .data
val:    .word  10, 20, 30, 40, 50, 60
        .text
        .globl main
main:
        move $t0, $zero         # inizializza indice
        addi $t3, $zero, 20     # carica l'offeset dell'ultimo elemento
        lw $t1, val($t0)        # carica primo valore
loop: addi $t0, $t0, 4          # avanza indice
        lw $t2, val($t0)        # carica prossimo valore
        add $t1, $t1, $t2       # effettua la somma
        bne $t0, $t3, loop      # ripeti fino all'ultimo valore
```

System call

- Sono fornite delle funzioni di sistema pre-definite che implementano particolari servizi (stampa a video, ecc.)
- Ogni system call ha:
  - un codice
  - degli argomenti (opzionali)
  - dei valori di ritorno (opzionali)

System call

Nome	Codice	Argomenti	Risultato
print_int	1	$a0	
print_float	2	$f12	
print_double	3	$f12	
print_string	4	$a0	
read_int	5		$v0
read_float	6		$f0
read_double	7		$f0
read_string	8	$a0,$a1	
sbrk	9	$a0	$v0
exit	10		

System call

- Per chiamare una system call:
  - mettere il codice in $v0
  - mettere gli argomenti in $a0 - $a3 ($f12 - $f15)
  - eseguire syscall
  - l'eventuale valore di ritorno sarà in $v0 ($f0)

Esempio

```
#programma che stampa: la risposta è 5

str:  .asciiz "la risposta è"

  .text
      li $v0, 4      #codice della print_string
      la $a0, str    #indirizzo della stringa
      syscall        #stampa la stringa

      li $v0, 1      #codice della print_integer
      li $a0, 5      # intero da stampare
      syscall        #stampa l'intero
```

Modi di indirizzamento

- Definiscono come reperire gli operandi di una istruzione
- L'esempio più comune di indirizzamento è l'indirizzamento **a registri** (cioè gli operandi sono contenuti in registri (es: add $s0, $s1, $s2)

Utilizzo di costanti

- Spesso le operazioni richiedono l'uso di costanti (ad esempio: somma 4 ad un valore, ecc.)
- Tre opzioni:
  - le costanti risiedono in memoria e sono caricate con lw
  - utilizzo di registri speciali (es: $zero)
  - utilizzare versioni alternative delle operazioni aritmetiche in cui un operando è una costante

Utilizzo di costanti

- Tali istruzioni utilizzano un **indirizzamento immediato** (tipo I)
- La costante è memorizzata nel campo di 16 bit

Esempio

- addi $s0, $s0, 4
- slti $t0, $s2, 10
- andi $s0, $s0, 6
- ori $s0, $s0, 10
- li $s0, 20
- I valori immediati possono anche essere esadecimali o binari

Esempio di indirizzamento a registri & immediato

```
# Somma

.text          #Definizione segmento codice
.globl  main

main:
 li $t1,10  # viene caricato il valore decimale 10 nel registro
   $t1
 li $t2,15
 add $a0,$t2,$t1
# stampa risultato
print_result:
 li  $v0,1
 syscall
```

Esempio di indirizzamento a registri & immediato

```
# L'accesso immediato è usato anche dalle operazioni
# aritmetiche
        .text
        .globl  main
main:
        li  $t1,10
        addi $a0,$t1,15      # viene sommato  15 al registro $t1
# stampa risultato
print_result:
        li  $v0,1
        syscall
```

Esempio di indirizzamento a registri & immediato

```
# I valori immediati possono essere dichiarati anche come
    esadecimali
# abbiamo 10 e 15 espressi in esadecimale con 0xa e 0xf.
        .text
        .globl  main
main:
        li        $t1,0xa
        addi    $a0,$t1,0xf
# stampa risultato
print_result:
        li      $v0,1
        syscall
```

Costanti > di 16 bit?

- Le istruzioni di tipo I consentono di rappresentare costanti esprimibili in 16 bit
- Se 16 bit non sono sufficienti si devono fare due passi:
  - si utilizza l’istruzione **lui** per caricare i 16 bit più significativi della costante in un registro
  - una successiva istruzione specifica i rimanenti 16 bit

Istruzione lui: esempio

Si consideri la costante a 32 bit:

0000 0000 0011 1101 0000 1001 0000 0000

lui $s0, 61 (61 = 0000 0000 0011 1101)

valore di $s0:

0000 0000 0011 1101 0000 0000 0000 0000

addi $s0, $s0, 2304 (2304 = 0000 1001 0000 0000)

valore di $s0:

0000 0000 0011 1101 0000 1001 0000 0000

Indirizzamento relativo al PC

- Motivazioni:
  - bne $so,$s1,Exit ⟹ per Exit si hanno a disposizione solo **16 bit**
  - soluzione: specificare un registro il cui contenuto va **sempre** sommato all'indirizzo contenuto nell'istruzione bne per ottenere l'indirizzo vero e proprio a cui saltare

Indirizzamento relativo al PC

- Vale il principio della **local...** ⟹ un buon candidato è il PC
- Questo tipo di indirizzamento è chiamato indirizzamento **relativo al PC** ed è utilizzato nelle istruzioni di salto
- In realtà l'indirizzamento è relativo al PC + 4 byte

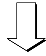
Esempio

```
Loop: add $t1, $s3, $s3         8000: 0 19 19 9 0 32
      …………
      bne $t0, $s5, Exit        80016: 5 8 21 2
      add $s3, $s3, $s4
      j Loop
      Exit:                     80028: Exit ...
      …………
```

80028 = 80016 + 4 + 2 *4

Salti di dimensioni maggiori

beq $s0, $s1, L1



bne $s0, $s1, L2

j L1

L2: ……

Modi di indirizzamento

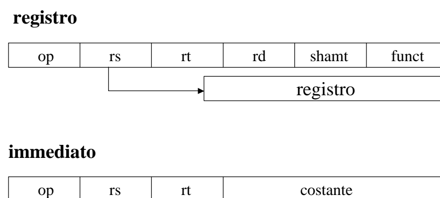
- Quattro opzioni principali:
  - **indirizzamento a registri**: gli operandi sono nei registri
  - **indirizzamento immediato**: l'operando è una costante contenuta nell'istruzione

Modi di indirizzamento

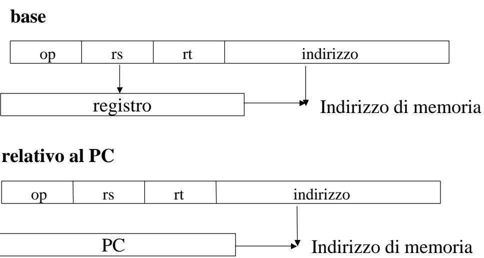
- Quattro opzioni (cont):
  - **indirizzamento con base**: l'operando è in una locazione di memoria il cui indirizzo è registro + cost. nell'istruzione
  - **indirizzamento relativo al PC**: l'indirizzo dell'operando è la somma del PC e di una cost. nell'istruzione

Modi di indirizzamento

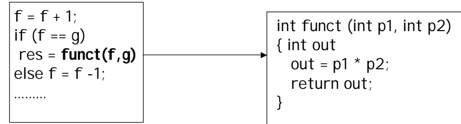
registro



Modi di indirizzamento



Chiamata a procedura



Chiamata a procedura

- Il chiamante deve eseguire le seguenti operazioni:
  - salvare i parametri di input della procedura in un posto accessibile alla procedura
  - trasferire il controllo alla procedura

Chiamata a procedura

- La procedura chiamata deve eseguire le seguenti operazioni:
  - acquisire lo spazio di memoria necessario alla sua esecuzione
  - svolgere la computazione
  - memorizzare il risultato della computazione in un posto accessibile al chiamante
  - restituire il controllo al chiamante

Chiamata a procedura

- Convenzioni:
  - $a0, …, $a3 ($f12, …, $f15) sono usati dal chiamante per il passaggio dei parametri
  - $v0,$v1 ($f0, …, $f3) sono usati dalla procedura per memorizzare i valori di ritorno
  - $ra (return address) memorizza l'indirizzo della prima istruzione del chiamante da eseguire al termine della procedura

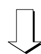
Chiamata a procedura

- Convenzioni (cont):
  - Nuova istruzione:
    jal Indirizzo_Procedura

  salta alll'indirizzo con etichetta Indirizzo_Procedura e memorizza contemporaneamente l'indirizzo dell'istruzione successiva in $ra
- Il chiamato esegue jr $ra come ultima istruzione

Chiamata a procedura

Può succedere che una procedura abbia bisogno di più registri di quelli a disposizione (es: riceve 5 parametri in input)



utilizzo dello stack

Lo stack

- Lo stack è una coda LIFO (last-in-first-out)
- E' necessario un puntatore al TOP dello stack ⇨
- Per inserire dati nello stack ⇨ *push*
- Per prelevare dati dallo stack *pop*
- Il registro $sp contiene l'indirizzo del top dello stack

Lo stack

- Lo stack cresce dall'alto verso il basso
- L'inserimento di un dato nello stack avviene decrementando $sp
- Il prelevamento di un dato dallo stack avviene incrementando $sp

Lo stack

- Tutto lo spazio in stack di cui ha bisogno una procedura (**record di attivazione**) viene *esplicitamente* allocato in una sola volta, all'inizio della procedura
- Lo spazio nello stack viene allocato sottraendo a $sp la dimensione voluta

Esempio

subu $sp,$sp,24 #alloca 24 byte nello stack

Al ritorno da una procedura il record di attivazione viene deallocato dalla procedura incrementando $sp della stessa quantità di cui lo si era decrementato al momento della chiamata tramite:
addu $sp,$sp,24

Lo stack

- Per inserire elementi nello stack:
  - sw $t0, 20($sp)
- Per recuperare elementi dallo stack:
  - lw $t0, 20($sp)

Lo stack

- Quando si chiama una procedura i registri utilizzati dal chiamato vanno:
  - salvati nello stack
  - il loro contenuto va ripristinato alla fine dell'esecuzione della procedura

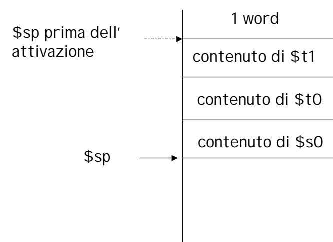
Esempio

```
int esempio (int g, int h, int i, int j)
{
    int f;

    f = (g + h) - (i + j);
    return f;
}
```

Esempio

```
#g,h,i e j sono associati a $a0, ..., $a3; f è associata ad $s0
#la computazione di f richiede 3 registi: $s0, $t0, $t1

example:
        subu $sp, $sp, 12 #alloca lo spazio per i 3 registri
        sw $t1, 8($sp)
        sw $t0, 4($sp)
        sw $s0, 0($sp)
```

Esempio (cont.)



Esempio (cont.)

```
add $t0, $a0, $a1 #t0 contiene g + h
add $t1, $a2, $a3 #t1 contiene i + j
sub $s0, $t0, $t1 #f = $t0 - $t1
add $v0, $s0, $zero #restituisce f

#ripristino il contenuto dei registri

lw $s0, 0($sp)
lw $t0, 4($sp)
lw $t1, 8($sp)
addu $sp, $sp, 12 #deallocazione dello stack

jr $ra          #ritorno al chiamante
```

Lo stack

- Per evitare di salvare inutilmente il contenuto dei registri, i registri sono divisi in due classi:
  - **registri temporanei**: \$t0, …, \$t9 (\$f4, .. \$f11, \$f16, .., \$f19) il cui contenuto non è salvato dal chiamato nello stack;
  - **registri non-temporanei**: \$s0, …, \$s8 (\$f20, …, \$f31) il cui contenuto è salvato nello stack se utilizzati dal chiamato.

Uso dei registri: convenzioni

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
$sp	29	stack pointer
$s8	30	registro salvato
$ra	31	indirizzo di ritorno

Uso dei registri: convenzioni

Nome	Utilizzo
$f0-$f3	valori di ritorno di una procedura
$f4-$f11	registri temporanei (non salvati)
$f12-$f15	argomenti di una procedura
$f16-$f19	registri temporanei (non salvati)
$f20-$f31	registri salvati

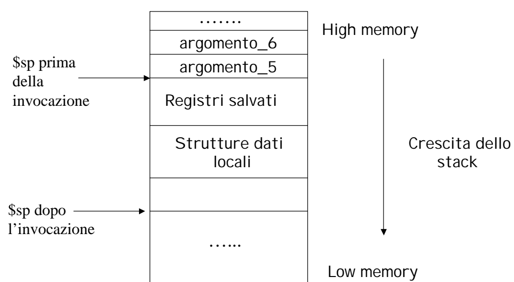
Lo stack

- Nel caso di procedure **foglia** il chiamante mette nello stack:
  - i registri temporanei di cui vuole salvare il contenuto ($a0, …, $a3, $t0, …, $t9,...)
  - eventuali argomenti aggiuntivi oltre a quelli che possono essere contenuti in $a0, …, $a3

Lo stack

- Nel caso di procedure foglia il chiamato mette nello stack:
  - i registri non temporanei che vuole utilizzare ($s0, …, $s8)
  - strutture dati (es: array, matrici) locali al chiamante

Lo stack: procedure foglia



Costruzione di una procedura

- Ogni procedura ha:
  - un prologo
  - un corpo
  - un epilogo

Prologo

- Definire un etichetta per la procedura (es: proc_name:)
- Determinare la dimensione del record di attivazione

Prologo

- Per determinare la dimensione del record di attivazione si deve stimare lo spazio per:
  - variabili locali
  - registri generali da salvare
  - registri floating-point da salvare
  - argomenti

Prologo

- Aggiornare il valore di $sp:

```
subu $sp , $sp, record_att_size

# lo stack pointer viene decrementato
# della dimensione del r.a.
# lo stack cresce verso indirizzi bassi di
# memoria
```

Prologo

- Salvare i registri per cui è stato allocato spazio nello stack:

```
sw    registro,record_att_size-N($sp)
```

N (N >= 4) viene incrementato di 4 ad ogni salvataggio

Esempio

- Record di attivazione: 16 byte

```
sw $s0, 12($sp)
sw $s1, 8($sp)
sw $s2, 4($sp)
sw $s3, 0($sp)
```

Corpo della procedura

- Vengono scritte le istruzioni per l'esecuzione delle funzionalità previste dalla procedura

Epilogo

- Recuperare i registri salvati:
  - lw registro, record_att_size - N($sp)
- Pulire lo stack:
  - addu $sp , $sp, recor_att_size
- Passare il controllo al chiamante:
  - jr $ra

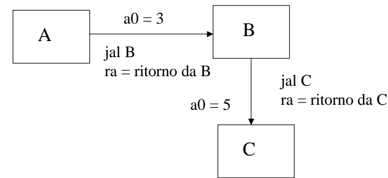
Chiamante

- Salva i registri temporanei di cui vuole preservare il contenuto
- Mette eventuali argomenti nello stack (oltre a quelli contenuti nei registri)
- Esegue jal proc_name

Procedure ricorsive & annidate

- Sono procedure che richiamano al loro interno altre procedure
- Le procedure ricorsive contengono una chiamata a se stesse
- Occorre salvare nello stack:
  - i parametri di input della procedura
  - l'indirizzo di ritorno

Procedure ricorsive & annidate



Gestione dei caratteri

- Ogni carattere è rappresentato univocamente mediante un codice numerico rappresentabile usando **un byte** (codice ASCII)
- MIPS fornisce le istruzioni **lb** e **sb** per trasferire un byte da memoria a registro e viceversa
- Il byte è quello "più a destra" nel registro (meno significativo)

Rappresentazione delle stringhe

- Tre opzioni:
  - la prima posizione della stringa contiene la sua lunghezza
  - la lunghezza è memorizzata in una variabile separata
  - l'ultima posizione della stringa è segnalata da un carattere speciale (NULL) la cui codifica ASCII è zero