Linguaggio Assembler MIPS

Corso di Calcolatori Elettronici

Corso di Calcolatori Elettronici A

A.A. 2000/2001

Dr. Daniela Fogli

Notazione

- Operazioni aritmetiche: **Somma**

  ```
  add   a, b, c
  ```

  fa la somma di b e c e il risultato è posto in a
- Ciascuna istruzione aritmetica MIPS esegue una sola operazione e contiene 3 variabili

  Es. sommare b, c, d, e e porre il risultato in a

  ```
  add   a, b, c       # a = b+c
  add   a, a, d       # a = a + d (=b+c+d)
  add   a, a, e       # a = a + e (= b+c+d+e)
  ```
- Ogni linea contiene un'istruzione; i commenti sono preceduti da #

Notazione

- Sottrazione

```
sub     a, b, c          # a = b - c
```

- Esempio: frammento di programma

```
a = b + c
d = a – e
```

viene tradotto in assembler MIPS come:

```
add     a, b, c
sub     d, a, e
```

Uso dei registri

- In realtà gli operandi delle istruzioni aritmetiche non possono essere variabili qualsiasi: devono essere ***registri***
- I registri del MIPS sono 32 e sono ampi 32 bit ciascuno
- La convenzione MIPS utilizza come nomi dei registri due caratteri preceduti dal segno $
- Si useranno `$s0`, `$s1`, … per i registri che corrispondono a variabili nei programmi C da compilare
- Si useranno `$t0`, `$t1`, … per i registri temporanei

Esempio

- Compilare l'istruzione C seguente

  f = (g + h) – (i + j)

  dove le variabili f, g, h, i, j vengono fatte corrispondere ai registri $s0, $s1, $s2, $s3, $s4

- Il programma compilato sarà:

```
add $t0, $s1, $s2     # $t0 contiene g+h
add $t1, $s3, $s4     # $t1 contiene i+j
sub $s0, $t0, $t1     # $s0 assume $t0-$t1 cioè
                        f =(g+h) – (i+j)
```

Strutture dati complesse e trasferimento dati

- Oltre a variabili semplici, dai programmi vengono di solito usate anche strutture dati complesse come i vettori
- Queste strutture possono contenere un numero di elementi maggiore del numero dei registri
- Le strutture dati complesse sono allocate in memoria
- Occorrono istruzioni assembler che permettano di trasferire dati fra memoria e registri
- Istruzione load (lw: load word)

Istruzione load (lw)

- Tradurre g = h + A[8]
- Assumendo che il compilatore associ i registri $s1 e $2 a g e h e l'indirizzo di partenza del vettore sia memorizzato in $s3 (***registro base***)
- Occorre ***caricare dalla memoria in un registro*** ($t0) il valore in A[8]
- L'indirizzo di questo elemento è dato dalla *somma dell'indirizzo di base* del vettore A (che è in $s3) *e del numero* (***offset***) che identifica l'elemento
- La compilazione dell'istruzione produrrà:

```
lw      $t0, 8($s3)
add     $s1, $s2, $t0
```

Una precisazione

- Il MIPS indirizza il singolo byte (8 bit)
- Gli indirizzi delle parole sono multipli di 4 (in una parola ci sono 4 byte)
- Gli indirizzi di due parole consecutive differiscono di 4 unità
- Nel MIPS le parole iniziano sempre ad indirizzi multipli di 4, cioè 0, 4, 8, 12, … (vincolo di allineamento)
- Quindi, nell'istruzione precedente, occorreva fare

```
lw $t0, 32($s3)
```

(l'ottava parola inizia al 32-esimo byte)

Istruzione store (sw)

- Istruzione complementare a quella di load
- Trasferisce un dato ***da un registro in memoria***
- Esempio: A[12] = h + A[8]

  dove h sia associata a $s2 e il registro base del vettore A sia $s3

```
lw      $t0, 32($s3)  # carica in $t0 il valore A[8]
add     $t0, $s2, $t0 # somma h + A[8]
sw      $t0, 48($s3)  # memorizza la somma in
                        A[12]
```

Indice variabile

- Si supponga che l'accesso a un vettore avvenga con indice variabile
- Es. : g = h + A[i]
- L'indirizzo base di A sia $s3 e alle variabili g, h, i siano associati i registri $s1, $s2, $s4

```
mul   $t1, $s4, 4    # i = i*4
add   $t1, $t1, $s3  # indirizzo di A[i] in $t1
lw    $t0, 0($t1)    # carica A[i] in $t0
add   $s1, $s2, $t0  # g = h + A[i]
```

Moltiplicazione nel MIPS

- L'istruzione da usare sarebbe mult che fa il prodotto del contenuto di due registri di 32 bit e pone il risultato a 64 bit nei registri di 32 bit hi e lo
- Poi, per caricare il prodotto intero su 32 bit in un registro general purpose, il programmatore può usare mflo (*move from lo*)
- Esempio:

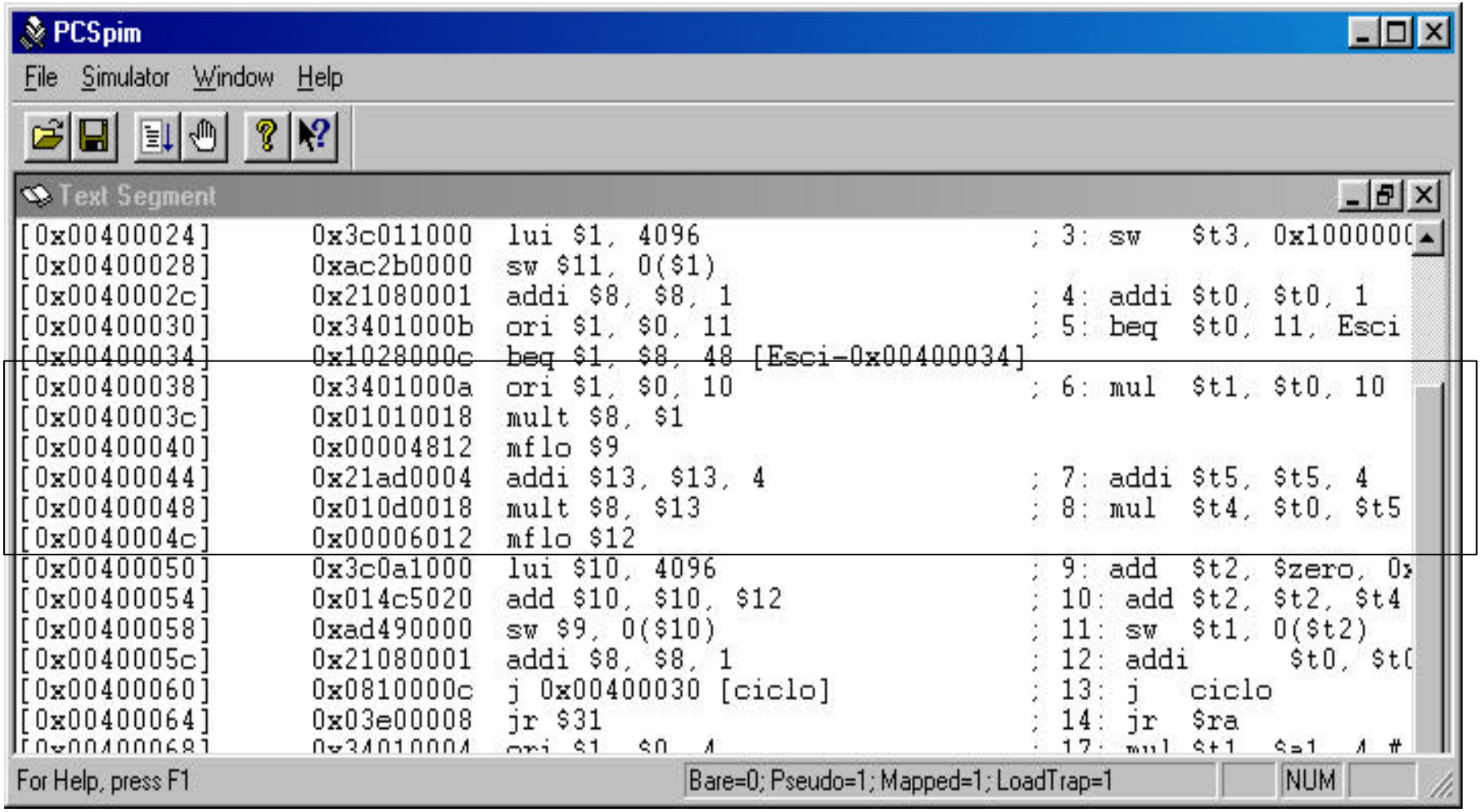
```
mult $s2, $s3     # hi, lo = $s2*$s3
mflo $s1          # s1 = lo
```

- In pratica useremo la *pseudoistruzione* mul

```
mul rdest, rsrc1, src2
```

che mette il prodotto fra rsrc1 e src2 nel registro rdest

La traduzione di “mul”



Salti condizionati

- Istruzioni beq e bne per tradurre gli if dei linguaggi di alto livello
- beq registro1, registro2, L1

  Vai all'istruzione etichettata con L1 se il valore contenuto in registro1 è uguale al valore contenuto in registro2 (beq = branch if equal)
- bne registro1, registro2, L1

  Vai all'istruzione etichettata con L1 se il valore contenuto in registro1 è diverso dal valore contenuto in registro2 (bne = branch if not equal)

Esempio

- Frammento di codice C:

```
        if (i = = j) go to L1;
        f = g + h;
L1:     f = f – i;
```

Supponendo che f, g, h, i, j corrispondano ai registri $s0, $s1, $s2, $s3, $s4, la traduzione è la seguente

```
        beq     $s3, $s4, L1    # va a L1 se i è uguale a j
        add     $s0, $s1, $s2   # f = g + h
L1:     sub     $s0, $s0, $s3   # f = f - i
```

Esempio con salto incondizionato

- Frammento di codice C:

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

  Supponendo che f, g, h, i, j corrispondano ai registri $s0, $s1, $s2, $s3, $s4, la traduzione è la seguente

```
        bne   $s3, $s4, Else
        add   $s0, $s1, $s2
        j     Esci   # salto incondizionato (jump)
Else:   sub   $s0, $s1, $s2
Esci:
```

Cicli

- Frammento di codice C:

```
Ciclo:  g = g + A[i];
        i = i + j;
        if (i != h) go to Ciclo;
```

Supponiamo che le variabili g, h, i, j siano associate ai registri $s1, $s2, $s3, $s4 e l'indirizzo di partenza del vettore A sia in $s5.

La traduzione è la seguente:

```
Ciclo:  mul $t1, $s3, 4          # $t1 = i * 4
        add $t1, $t1, $s5        # $t1 = indirizzo di A[i]
        lw $t0, 0($t1)           # $t0 = A[i]
        add $s1, $s1, $t0        # g = g + A[i]
        add $s3, $s3, $s4        # i = i + j
        bne $s3, $s2, Ciclo      # salta a Ciclo se i ≠ j
```

Ciclo *while*

- Frammento di codice C:

  ```
  while (salva[i] = = k)      i = i + j;
  ```

  Supponendo che i, j, k corrispondano ai registri $s3, $s4, $s5 e l'indirizzo base di salva sia in $s6.

  La traduzione è la seguente:

  ```
  Ciclo:  mul   $t1, $s3, 4      # $t1 = i * 4
          add   $t1, $t1, $s6    # $t1 = indirizzo di salva[i]
          lw    $t0, 0($t1)      # $t0 = salva[i]
          bne   $t0, $s5, Esci   # vai a Esci se salva[i] ≠ k
          add   $s3, $s3, $s4    # i = i + j
          j     Ciclo            # vai a Ciclo
  Esci:
  ```

  Si usano 1 salto condizionato e 1 salto incondizionato

Ottimizzazione

- Codice ottimizzato (con un solo salto condizionato)

```
Ciclo:  mul   $t1, $s3, 4      # $t1 = i * 4
        add   $t1, $t1, $s6    # $t1 = indirizzo di salva[i]
        lw    $t0, 0($t1)      # $t0 = salva[i]
        add   $s3, $s3, $s4    # i = i + j
        beq   $t0, $s5, Ciclo  # vai a Ciclo se salva[i]==k
        sub   $s3, $s3, $s4    # i = i – j
```

Istruzione slt e registro $zero

- Si può usare un'altra istruzione per confrontare due variabili: slt

  slt registro1, registro2, registro3

- Istruzione slt (*set on less than*): poni a 1 il valore di registro1 se il valore di registro2 è minore del valore di registro3, in caso contrario poni a 0 il valore di registro1
- Inoltre, il registro $zero che contiene sempre il valore 0, è un registro di sola lettura che può essere usato nelle condizioni relazionali

- Esempio:

```
slt     $t0, $s0, $s1          # $t0 diventa 1 se $s0 < $s1
bne     $t0, $zero, Minore     # vai a Minore se $t0 ≠ 0
```

Ciclo *for*

- Frammento di programma C:

  for (i=0; i<k; i++) f = f + A[i];

  siano i, k, f corrispondenti a $s1, $s2, $s3, supponendo che $s1 valga 0 all'inizio, e indirizzo base di A in $s4. La traduzione è:

```
Ciclo:  slt     $t2, $s1, $s2     # poni $t2 a 1 se i < k
        beq     $t2, $zero, Esci  # se $t2  = 0 vai a Esci
        mul     $t1, $s1, 4       # $t1 = i * 4
        add     $t1, $t1, $s4     # $t1 = indirizzo di A[i]
        lw      $t0, 0($t1)       # $t0 = A[i]
        add     $s3, $s3, $t0     # f = f + A[i]
        add     $s1, $s1, 1       # i = i + 1
        j       Ciclo
Esci:
```

Case/switch

- Frammento di programma 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;
}
```

- Si potrebbe trasformare in una catena di *if-then-else* e quindi tradurre di conseguenza con salti condizionati
- A volte però le diverse alternative possono essere specificate memorizzandole in una tabella di indirizzi in cui si trovano le diverse sequenze di istruzioni (*tabella degli indirizzi di salto o jump address table*)
- Il programma deve riconoscere un **indice** per saltare alla sequenza di codice opportuna
- La tabella è un **vettore di parole** che contiene gli **indirizzi corrispondenti alle etichette** presenti nel codice

Istruzione jr

- Istruzione jr (*jump register*): istruzione di salto tramite registro: effettua un salto incondizionato **all'indirizzo specificato in un registro**
- Premessa per la traduzione del case/switch
  - Si assuma che f, g, h, i, j, k corrispondano ai registri da $s0 a $s5 e che $t2 contenga il valore 4
  - La variabile k è usata per indicizzare la tabella degli indirizzi di salto
  - Sia in $t4 l'indirizzo di partenza della tabella dei salti
  - Si supponga che la tabella contenga in sequenza gli **indirizzi** corrispondenti alle **etichette** L0, L1, L2, L3

Traduzione del case/switch

```
        slt     $t3, $s5, $zero     # controlla se k < 0
        bne     $t3, $zero, Esci    # se k < 0 (cioè $t3 = 1) salta a Esci
        slt     $t3, $s5, $t2       # controlla se k < 4
        beq     $t3, $zero, Esci    # se k >= 4 (cioè $t3 = 0) salta a Esci
        mul     $t1, $s5, 4         # $t1 = k * 4 (k è l'indice di una tabella)
        add     $t1, $t1, $t4       # $t1 = indirizzo di TabelladeiSalti[k]
        lw      $t0, 0($t1)         # $t0 = TabelladeiSalti[k], cioè $t0
                                    # conterrà un indirizzo corrispondente
                                    # a un'etichetta
        jr      $t0                 # salto all'indirizzo contenuto in $t0
L0:     add     $s0, $s3, $s4       # k = 0, quindi f = i + j
        j       Esci                # fine del case, salta a Esci
L1:     add     $s0, $s1, $s2       # k = 1, quindi f = g + h
        j       Esci                # fine del case, salta a Esci
L2:     sub     $s0, $s1, $s2       # k = 2, quindi f = g -  h
        j       Esci                # fine del case, salta a Esci
L3:     sub     $s0, $s3, $s4       # k = 3, quindi f = i – j
Esci:
```

Le procedure

Durante l’esecuzione di un procedura il programma deve seguire i seguenti passi:

1. Mettere i parametri della procedura in un luogo accessibile alla procedura
2. Trasferire il controllo alla procedura
3. Acquisire le risorse necessarie alla memorizzazione dei dati
4. Eseguire il compito richiesto
5. Mettere il risultato in un luogo accessibile al programma chiamante
6. Restituire il controllo al punto di origine

L’istruzione jal

- Il MIPS alloca i seguenti registri per le chiamate di procedura:
  - $a0 - $a3: 4 registri argomento per il passaggio dei parametri
  - $v0-$v1: 2 registri valore per la restituzione dei valori
  - $ra: registro di ritorno per tornare al punto di origine
- Istruzione jal (*jump-and-link*): è l’istruzione apposita per le procedure, che salta a un indirizzo e contemporaneamente salva l’indirizzo dell successiva nel registro $ra

  jal IndirizzoProcedura
- In pratica jal salva il valore di PC+4 nel registro $ra (cioè il valore del Program Counter + 4, che contiene l’indirizzo della prossima istruzione)
- Questo crea un “collegamento” all’indirizzo di ritorno dalla procedura
- L’indirizzo di ritorno è necessario perché la stessa procedura p richiamata in più parti del programma

Chiamata di una procedura

- L’istruzione per effettuare il salto di ritorno è

  jr $ra

- In pratica il ***programma chiamante*** mette i valori dei **parametri** da passare alla procedura nei **registri $a0-$a3** e utilizza l’istruzione jal X per saltare alla procedura X (programma chiamato)
- Il ***programma chiamato*** esegue le operazioni richieste, memorizza i **risultati nei registri $v0-$v1** e restituisce il controllo al chiamante con l’istruzione jr $ra

L’uso dello stack

- Spesso, sono necessari più registri per i parametri di una procedura e i valori da restituire
- Lo **stack** (pila) è una struttura del tipo *last-in first-out* dove una procedura può memorizzare i registri e dove può recuperare i vecchi valori dei registri
- Si usa un puntatore allo stack (**stack pointer**) che contiene l’indirizzo del dato introdotto più recentemente nello stack
- Lo stack *cresce* a partire da indirizzi di memoria alti verso indirizzi di memoria bassi: quando vengono inseriti dei dati nello stack il valore dello stack pointer diminuisce, viceversa, quando sono estratti dati dallo stack, il valore dello stack pointer aumenta
- Il **registro** allocato dal MIPS come stack pointer è **`$sp`**

Esempio: procedura con uso dello stack

Procedura C

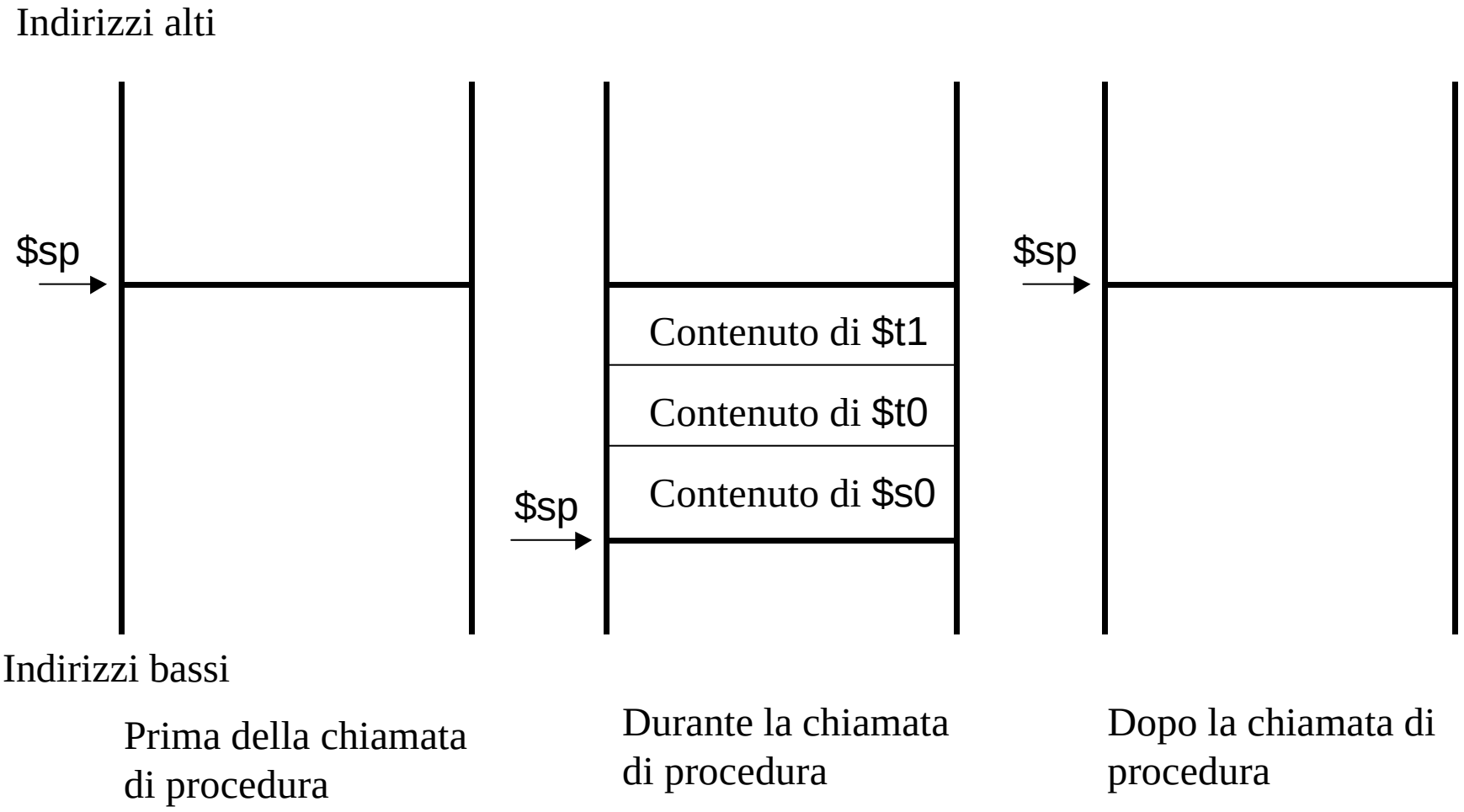
```
int proc(int g, int h, int i, int j)
{
        int f;
        f = (g + h) – (i + j)
        return f;
}
```

dove i parametri g, h, i, j corrispondono a $a0, $a1, $a2, $a3, e f corrisponde a $s0

Esempio: procedura con uso dello stack

```
proc:                                   # etichetta della procedura
        sub     $sp, $sp, 12            # aggiornamento dello stack per fare
                                        # spazio a 3 elementi
        sw      $t1, 8($sp)             # salvataggio del registro $t1
        sw      $t0, 4($sp)             # salvataggio del registro $t0
        sw      $s0, 0($sp)             # salvataggio del registro $s0
        add     $t0, $a0, $a1           # $t0 = g + h
        add     $t1, $a2, $a3           # $t1 = i + j
        sub     $s0, $t0, $t1           # f = $t0 - $t1
        add     $v0, $s0, $zero         # restituzione di f
        lw      $s0, 0($sp)             # ripristino del registro $s0
        lw      $t0, 4($sp)             # ripristino del registro $t0
        lw      $t1, 8($sp)             # ripristino del registro $t1
        add     $sp, $sp, 12            # aggiornamento dello stack per
                                        # eliminare 3 elementi
        jr      $ra                     # ritorno al programma chiamante
```

Valori dello stack pointer



Registri preservati e non preservati

- Nell'esempio precedente abbiamo supposto che dei registri temporanei dovessero essere preservati (cioè il valore originale di $t0 e $t1 dovesse essere salvato e ripristinato)
- In realtà il MIPS offre 2 classi di registri:

  $t0-$t9: registri temporanei che non sono preservati in caso di chiamata di procedura

  $s0-$s7: registri che devono essere preservati (se utilizzati devono essere salvati e ripristinati dal programma chiamato)
- In questo modo si riduce la necessità di salvare registri in memoria
- Nell'esempio precedente si possono eliminare 2 istruzioni di store e 2 di load (quelle che si riferiscono a $t0 e $t1)

Procedure annidate

- Il programma principale chiama una certa procedura A passandole dei parametri, la procedura A chiama a sua volta una procedura B passandole dei parametri
- Occorre salvare nello stack tutti i registri che devono essere preservati, compresi i registri di ritorno:
  - Il programma *chiamante* memorizza nello stack i registri argomento ($a0 - $a3) o i registri temporanei ($t0 - $t9) di cui ha ancora bisogno dopo la chiamata
  - Il programma *chiamato* salva nello stack il registro di ritorno $ra e gli altri registri ($s0 - $s7) che utilizza
- Esempio, procedura ricorsiva (per il calcolo del fattoriale)

```
int  fatt(int n) {
        if (n < 1) return (1); else return (n *  fatt (n-1));
}
```

dove il parametro n corrisponde al registro argomento $a0

Procedura in assembler per il calcolo del fattoriale

```
fatt:
        sub     $sp, $sp, 8     # aggiornamento dello stack per fare
                                # spazio a 2 elementi
        sw      $ra, 4($sp)     # salvataggio del registro di ritorno (come
                                # programma chiamato)
        sw      $a0, 0($sp)     # salvataggio del parametro n (come
                                # programma chiamante)
        slt     $t0, $a0, 1     # test per n < 1
        beq     $t0, $zero, L1  # se n >= 1 salta a L1
        add     $v0, $zero, 1   # restituisce 1 (per n<1)
        add     $sp, $sp, 8     # aggiornamento dello stack
        jr      $ra             # ritorno all'istruzione successiva a jal
L1:     sub     $a0, $a0, 1     # n > 1: l'argomento diventa n-1
        jal     fatt            # chiamata a fatt con (n-1)
        lw      $a0, 0($sp)     # ritorno da jal: ripristino di n
        lw      $ra, 4($sp)     # ripristino del registro di ritorno
        sub     $sp, $sp, 8     # aggiornamento dello stack
        mul     $v0, $a0, $v0   # restituzione di n * fatt (n-1)
        jr      $ra             # ritorno al programma chiamante
```

Ripristino di $a0 e $r0 ma non sono cambiati

chiamata ricorsiva

Prodotto di fatt(n-1) ritornato in $v0 per il vecchio parametro $a0

Modifica di $a0

Modi di indirizzamento del MIPS

1. ***Indirizzamento tramite registro***: l'operando è un registro (già visto)
2. ***Indirizzamento tramite base o tramite spiazzamento***: l'operando è in una locazione di memoria individuata dalla somma del contenuto di un registro e di una costante specificata nell'istruzione (già visto)
3. ***Indirizzamento immediato***: l'operando è una costante specificata nell'istruzione
4. ***Indirizzamento relativo al program counter*** (PC-relative addressing): l'indirizzo è la somma del contenuto del PC e della costante contenuta nell'istruzione
5. ***Indirizzamento pseudo-diretto***: l'indirizzo è ottenuto concatenando i 26 bit dell'istruzione con i 4 bit più significativi del PC

Indirizzamento immediato

- E’ stato pensato per rendere veloce l’accesso a costanti di piccole dimensioni
- I programmi usano spesso costanti, ad esempio per: incrementare un indice di un vettore, contare il numero di iterazioni in un ciclo, aggiornare il puntatore allo stack
- Con le istruzioni viste, sarebbe necessario caricare la costante in memoria prima di poterla utilizzare

  Ad esempio, l’istruzione che abbiamo usato

  ```
  add     $sp, $sp, 4
  ```

  in realtà non è esatta.

  Assumendo che IndCostante4 sia l’indirizzo di memoria in cui si trova la costante 4, occorrerebbe fare

  ```
  lw      $t0, IndCostante4($zero)  # $t0 = 4
  add     $sp, $sp, $t0
  ```

Indirizzamento immediato

- In pratica, si usano versioni particolari delle istruzioni aritmetiche in cui uno degli operandi è una costante
- La costante è contenuta dentro l'istruzione
- Il formato di queste istruzioni è lo stesso di quelle di trasferimento dati e dei salti condizionati: cioè ***formato-I*** (dove I sta per immediato)
- L'istruzione precedente diventa quindi

```
addi    $sp, $sp, 4
```

- Il *formato decimale*, dove il codice operativo di addi è 8 e il numero corrispondente al registro $sp è 29, è il seguente

op	rs	rt	immediato
8	29	29	4

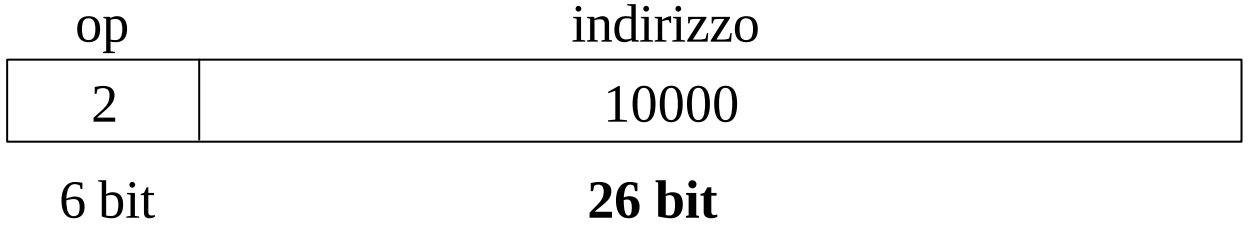
- Per controllare se il valore nel registro $s2 è minore di 10 si può scrivere

```
slti    $t0, $s2, 10        # $t0 = 1 se $s2 < 10
```

Indirizzamento nei salti

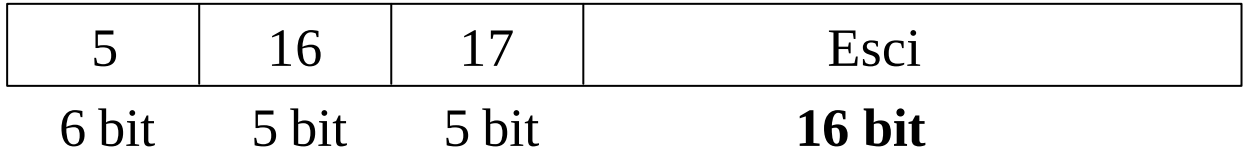
- Istruzione jump

```
j          10000   # vai alla locazione 10000
```



- Istruzione branch not equal

```
bne      $s0, $s1, Esci   # vai Esci se $s0 è diverso da $s1
```



(dove 16 e 17 sono i numeri dei registri $s0 e $s1)

Indirizzamento relativo al program counter

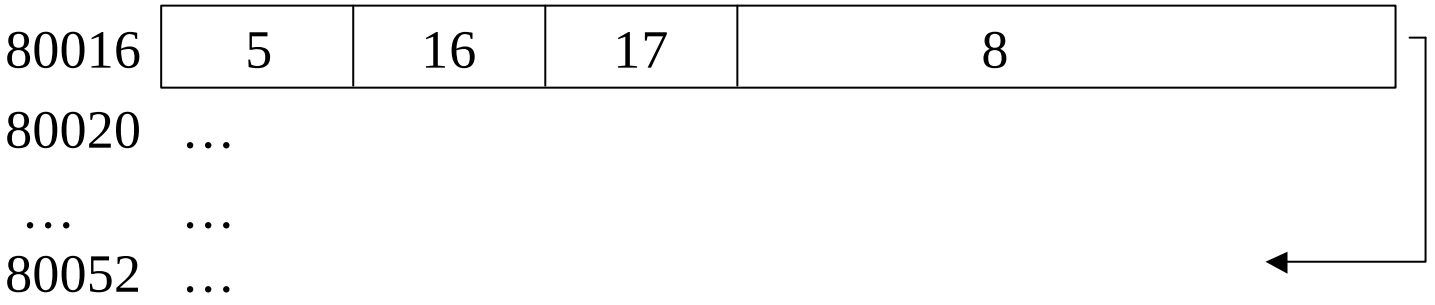
- Con 16 bit di indirizzo nessun programma può avere dimensioni maggiori di 2^{16}
- L'alternativa: specificare un registro il cui valore deve essere sommato al registro di salto, cioè:

  Program counter = *Registro* + Indirizzo di salto
- Ma quale *Registro* di partenza usare?
- Considerando che normalmente i salti condizionati vengono usati nei cicli e nei costrutti *if*, i salti sono in genere ad istruzioni vicine
- Il PC contiene l'indirizzo dell'istruzione corrente
- Quindi a partire da PC si può saltare fino a una *distanza di* $\pm 2^{15}$ *istruzioni* rispetto all'istruzione corrente e questo è sufficiente
- L'indirizzamento PC-relative è generalmente usato per tutti i tipi di *salto condizionato*, mentre le istruzioni di jump-and-link (chiamata di procedure) utilizzano altri modi di indirizzamento

Esempio di indirizzamento PC-relative

Supponiamo che alla locazione 80016 vi sia l'istruzione

```
bne    $s0, $s1, Esci
```



- L'indirizzo PC-relative si riferisce al numero di parole che lo separano dalla destinazione
- Quindi significherebbe che l'istruzione bne somma 8x4=32 byte all'indirizzo dell'istruzione successiva (perché il PC contiene l'indirizzo dell'istruzione successiva: è stato infatti incrementato nella fase preliminare dell'esecuzione dell'istruzione corrente)

Indirizzamento pseudo-diretto

- Se l'indirizzo specificato in un salto condizionato è ancora troppo lontano, l'assembler risolve il problema inserendo un salto incondizionato al posto di quello condizionato invertendo la condizione originale

```
        beq     $s0, $s1, L1
```

diventa

```
        bne     $s0, $s1, L2
        j       L1
L2:
```

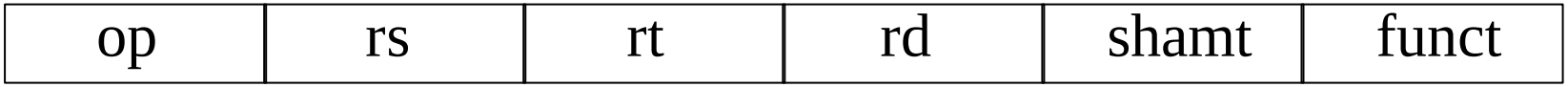
- I 26 bit all'interno dell'istruzione jump contengono un indirizzo in parole, questi 26 bit vengono concatenati con i 4 bit più significativi del PC

I 5 modi di indirizzamento

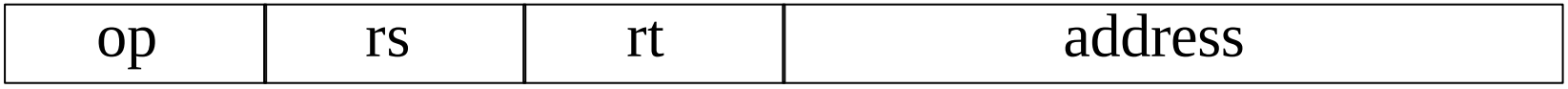
Fig. 3.17 (Patterson & Hennessy)

Formato delle istruzioni viste

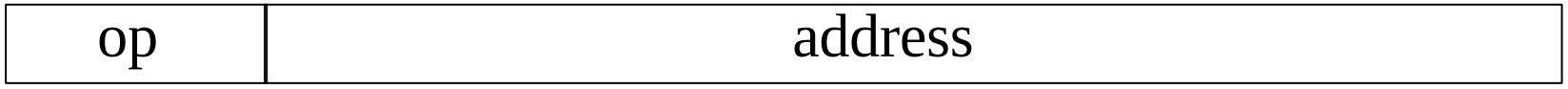
Formato-R o Tipo-R (istruzioni aritmetiche)



Formato-I o Tipo-I (istruzioni di trasferimento, salto condizionato, immediate)



Formato-J o Tipo-J (salto incondizionato)



I campi delle istruzioni MIPS

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

op: codice operativo
rs: primo operando sorgente (registro)
rt: secondo operando sorgente (registro)
rd: registro destinazione
shamt: shift amount (per operazioni di scalamento)
funct: seleziona una variante specifica dell'operazione base definita nel campo *op*

op	rs	rt	address

rs: registro base al cui contenuto va sommato address
rt: primo registro che compare nell'istruzione (registro destinazione per lw e registro sorgente per sw)

Esempio di traduzione in linguaggio macchina

- Istruzione C

  A[300] = h + A[300];

  assumendo che $t1 contenga l'indirizzo base di A e $s2 corrisponda alla variabile h

- Viene compilata in

```
lw      $t0, 1200($t1)   # $t0 = A[300]
add     $t0, $s2, $t0    # t0 = h + A[300]
sw      $t0, 1200($t1)   # A[300] = h + A[300]
```

- lw ha codice operativo 35, add ha codice operativo 0 e codice funzione 32, sw ha codice operativo 43
- I registri $s0-$s7 sono associati ai numeri da 16 a 23 e i registri $t0-$t7 sono associati ai numeri da 8 a 15

Esempio di traduzione in linguaggio macchina

Istruzioni macchina in formato decimale

op	rs	rt	rd	shamt/ address	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

Istruzioni macchina in formato binario

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

Pseudoistruzioni

- L’assemblatore può anche trattare versioni modificate delle istruzioni in linguaggio macchina come se fossero istruzioni vere e proprie
- Queste istruzioni sono dette ***pseudoistruzioni*** e permettono di semplificare le fasi di traduzione e di programmazione
- Consentono all’assembler MIPS di avere un insieme di istruzioni ricco
- Esempio: istruzione **move**

  ```
  move $t0, $t1     # il registro $t0 assume il valore
                      del registro $t1
  ```

  questa istruzione viene accettata dall’assemblatore e convertita in:

  ```
  add $t0, $zero, $t1
  ```

- L’istruzione blt (branch on less than) viene convertita nelle due istruzioni slt e bne; e una cosa simile succede per bgt (branch on greater than), bge (branch on greater/equal than), ble (branch on less/equal than)

Input/Output con SPIM

- Un programma eseguito con SPIM può leggere i caratteri inseriti dall'utente e visualizzare eventuali uscite sulla finestra di console
- In pratica, si usano chiamate di sistema (syscall) per scrivere su (o leggere da) console
- Il programma deve caricare il codice della chiamata di sistema nel registro $v0 e gli argomenti nei registri $a0-$a3
- Le chiamate di sistema che forniscono un valore in uscita mettono il risultato in $v0 (o in $f0)
- Se si usano le chiamate di sistema occorre disabilitare l'opzione "*mapped I/O*"

Servizi di sistema e codici corrispondenti

Figura A.17
Patterson & Hennessy

Codice per una stampa semplice

Il seguente codice stampa “risposta = 5”

```
main:
        .data
str:    .asciiz "risposta = "
        .text
        li $v0, 4           # codice della chiamata di  sistema per
                            # print_string
        la $a0, str         # carica in $a0 l’indirizzo della stringa da
                            # stampare
        syscall             # stampa della stringa
        li $v0, 1           # codice della chiamata di sistema per
                            # print_int
        li $a0, 5           # carica in $a0 numero intero da stampare
        syscall             # stampa del numero risultato
```

Direttive all’assemblatore

- I nomi che iniziano con un punto specificano all’assemblatore come tradurre un programma ma non generano istruzioni macchina

  Esempi:

  .data (indica che le linee seguenti contengono dati)

  .text (indica che le linee seguenti contengono istruzioni)

  .asciiz (copia in memoria una stringa)

- I nomi seguiti dai due punti sono etichette che associano un nome alla locazione di memoria successiva

  Esempi:

  str

  main

Esercizio: somma dei quadrati da 0 a 100

- Un programma C che fa la somma dei quadrati dei numeri da 0 a 100 è il seguente:

```
somma = 0;
for (i=0; i<101; i++) somma = somma + i * i;
printf("La somma dei quadrati da 0 a 100 vale %d \n“, somma);
```

- Facciamo corrispondere i a $s0 e somma a $s1
- Inizializzazione dei registri:

```
move    $s0, $zero      # i = 0
move    $s1, $zero      # somma = 0
move    $t1, $zero      # $t1 = 0
addi    $t1, $t1, 101   # $t1 = 101
```

- Il ciclo for viene tradotto come segue:

```
ciclo:  mul     $t0, $s0, $s0       # j = i * i
        add     $s1, $s1, $t0       # somma = somma + j
        addi    $s0, $s0, 1         # i = i+1
        slt     $t2, $s0, $t1       # $t2 = 1 se i < 101
        bne     $t2, $zero, ciclo   # torna a ciclo se i < 101
```

Stampa del risultato

1. Definire in .data la stringa da visualizzare:

```
        .data
str:
        .asciiz "La somma dei quadrati da 0 a 100 vale \n"
```

2. Caricare in $v0 il codice della chiamata di sistema per la stampa di una stringa, che è 4, e caricare in $a0 l'indirizzo della stringa (str) da stampare, e infine chiamare una system call

```
li $v0, 4          # codice della chiamata di sistema per print_string
la $a0, str        # indirizzo della stringa da stampare
syscall            # stampa della stringa
```

3. Caricare in $v0 il codice della chiamata di sistema per la stampa di un intero, che è 1, e caricare in $a0 il numero intero da stampare che è stato accumulato in $s1 , e infine chiamare una system call

```
li $v0, 1          # codice della chiamata di sistema per print_int
move $a0, $s1      # numero intero da stampare
syscall
```

Programma completo per la somma dei quadrati da 0 a 100

```
        .text
main:
        move    $s0, $zero          # i = 0
        move    $s1, $zero          # somma = 0
        move    $t1, $zero
        addi    $t1, $t1, 101       # $t1 = 101
ciclo:  mul     $t0, $s0, $s0       # j = i * i
        add     $s1, $s1, $t0       # somma = somma + j
        addi    $s0, $s0, 1         # i = i+1
        slt     $t2, $s0, $t1       # $t2 = 1 se $s0 < 101
        bne     $t2, $zero, ciclo   # torna a ciclo se i < 101

        li $v0, 4                   # codice della chiamata di sistema per print_string
        la $a0, str                 # indirizzo della stringa da stampare
        syscall                     # stampa della stringa
        li $v0, 1                   # codice della chiamata di sistema per print_int
        move $a0, $s1               # numero intero da stampare
        syscall

        .data
str:
        .asciiz "La somma dei quadrati da 0 a 100 vale \n"
```

Programma con lettura da tastiera

- Programma che legge da console dei numeri interi e li somma finchè in ingresso non viene immesso 0
- Il corpo principale del programma è il ciclo seguente:

```
ciclo:   li $v0, 5                  # codice della chiamata di sistema per
                                    # read_int
         syscall                    # leggi numero da tastiera in $v0
         beq $v0, $zero, fine       # se $v0 vale 0 vai a fine
         add $t1, $t1, $v0          # somma nel registro $t1
         j ciclo                    # torna ad acquisire nuovo dato
```

- I numeri vengono inseriti uno alla volta dando “invio” dopo ciascuno

Programma completo per la somma dei numeri letti da tastiera

```
main:
            .text
            move $t1, $zero      # inizializza sommatore
ciclo:      li $v0, 5            # codice della chiamata di sistema per read_int
            syscall              # leggi numero da tastiera in $v0
            beq $v0, $zero, fine # se $v0 vale 0 vai a fine
            add $t1, $t1, $v0    # somma nel registro $t1
            j ciclo              # torna ad acquisire nuovo dato
fine:
            li $v0, 4            # codice della chiamata di sistema per print_string
            la $a0, str          # indirizzo della stringa da stampare
            syscall              # stampa della stringa
            li $v0, 1            # codice della chiamata di sistema per print_int
            move $a0, $t1        # numero intero da stampare
            syscall              # stampa del numero risultato
            .data
str:        .asciiz "risposta = "
```

Esercizio: procedura Scambia

- Codice C

```
scambia(int v[ ], int k)
{
        int temp;
        temp = v[k];
        v[k] = v[k+1];
        v[k+1] = temp;
}
```

- Per la traduzione in Assembler occorre:
  - allocare i registri per le variabili: i parametri v e k si troveranno nei registri $a0 e $a1 alla chiamata della procedura; l'altra variabile temp, la associamo al registro $t0
  - produrre il codice relativo al corpo della procedura

Codice relativo al corpo della procedura

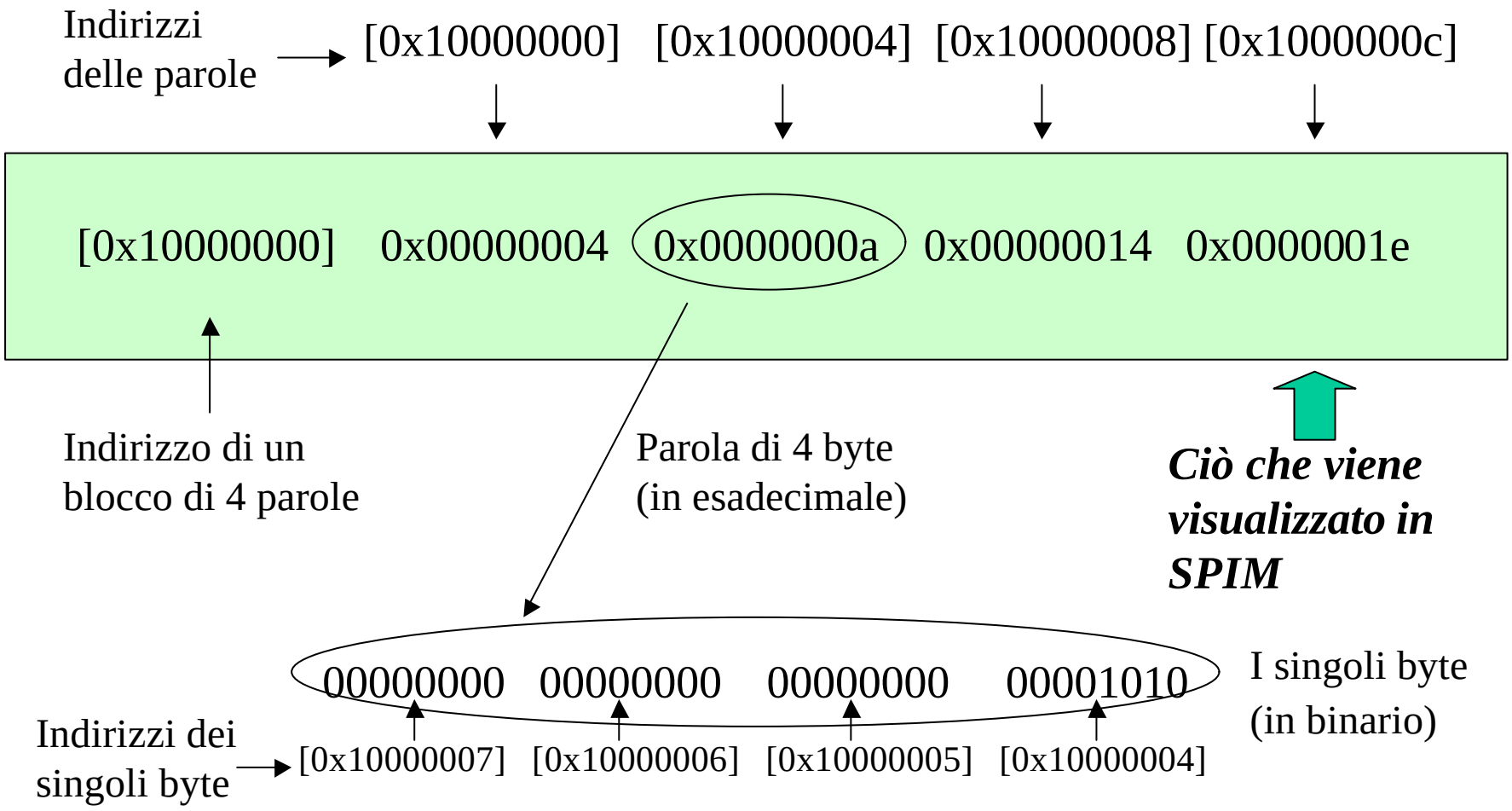
```
scambia:
        mul     $t1, $a1, 4         # $t1 = k * 4
        add     $t1, $a0, $t1       # $t1 = v + (k * 4) (in $t1 c'è
                                    #  l'indirizzo di v[k])
        lw      $t0, 0($t1)         # $t0 (temp) = v[k]
        lw      $t2, 4($t1)         # $t2 = v[k+1]
        sw      $t2, 0($t1)         # v[k] = $t2
        sw      $t0, 4($t1)         # v[k+1] = $t0 (temp)
        jr      $ra                 # ritorna al chiamante
```

Programma completo da provare con SPIM

- Per provare la procedura scambia con SPIM, prepariamo la memoria inserendo dei valori nella parte relativa ai dati
- In SPIM l'indirizzo di partenza della memoria dati è 0x10000000 (in esadecimale)
- Nella locazione 0x10000000 (v[0]) memorizziamo il valore di k, supponendo k = 4 (cioè vogliamo scambiare v[4] con v[5])
- Nelle 10 parole successive alla locazione 0x10000000 memorizziamo i valori v[1] = 10, v[2] = 20 … v[10] = 100
- Si ricordi che gli **indirizzi fanno riferimento al byte** ma che **ogni parola è di 4 byte**
- Quindi gli indirizzi relativi agli elementi del vettore v saranno:

(1)	0x10000004	(0x10000000+1*4)
(2)	0x10000008	(0x10000000+2*4)
(3)	0x1000000c	(0x10000000+3*4)
…	…	….

Memoria dati in SPIM



Codice per la preparazione della memoria

- Se k = 4, v[k] = 40 e v[k+1] = 50
- Usando la procedura scambia si avrà che v[k] = 50 e v[k+1] = 50
- Il codice C per preparare la memoria sarebbe:

```
prepara;
{
        v[0] = k
        for (i=1; i<11; i++)  v[i] = i * 10;
}
```

  dove k corrisponde a $t3, l'indirizzo base di v è 0x10000000, e i corrisponde a $t0
- Si può scrivere la procedura prepara che ha il compito di preparare la memoria come spiegato
- Questa procedura non ha argomenti, utilizza solamente delle variabili locali e quindi registri temporanei: $t0, $t1, $t2, $t3, $t4, $t5

Codice per la procedura “prepara”

```
Prepara: addi   $t3, $t3, 4          # $t3 (k) = 4
         sw     $t3, 0x10000000      # memorizza k=4 nella prima
                                     # parola della memoria dati (v[0])

         addi   $t0, $t0, 1          # i = 1
         addi   $t5, $t5, 11         # limite per beq
ciclo:   beq    $t0, $t5, Esci       # se i = 11 Esci
         mul    $t1, $t0, 10         # $t1 = i * 10
         mul    $t4, $t0, 4          # $t4 = i * 4
         la     $t2, 0x10000000      # $t2 = indirizzo di partenza
         add    $t2, $t2, $t4        # $t2 = $t2 + $t4 (indirizzo di v[i])
         sw     $t1, 0($t2)          # contenuto di v[i] = $t1 (= i * 10)
         addi   $t0, $t0, 1          # i = i + 1
         j      ciclo                # torna a ciclo
Esci:    jr     $ra
```

Programma da provare con SPIM

```
prepara:
        … mettere qui procedura “prepara”
scambia:
        … mettere qui procedura “scambia” …

main:
        jal prepara
        la $a0, 0x10000000      #  carica in $a0 l'indirizzo di partenza di v
        lw $a1, 0($a0)          #  carica in $a1 il valore di k (che è in v[0])
        jal scambia
```

Note:

(1) Le procedure non sono annidate, quindi non occorre salvare $ra sullo stack

(2) Sia “prepara” che “scambia” non usano registri da preservare, ma solo registri temporanei $t0, $t1, …, per cui nessun registro è da salvare sullo stack

Due semplici esercizi

1. Parametrizzare la procedura “prepara” in modo che anch’essa lavori utilizzando come argomenti di ingresso k e v (modificare di conseguenza anche il programma principale)

2. Es. A.7 del libro: Scrivere un programma che legga 3 numeri interi e stampi la somma dei due maggiori. In condizioni di parità si può agire in modo arbitrario

Una procedura di ordinamento

Procedura C:

```
ordina(int v[ ], int n)
{
        int i, j;
        for (i = 0; i < n; i = i + 1) {
                for (j = i-1; j >= 0 && v[j]>v[j+1]; j = j-1) scambia(v, j);
        }
}
```

I due parametri della procedura, v e n, sono nei registri argomento $a0 e $a1, mentre i registri $s0 e $s1 sono associati alle variabili i e j

Ciclo for esterno

Traduciamo prima il ciclo for più esterno:

```
                    for (i = 0; i < n; i = i +1)

            move    $s0, $zero          # i = 0
forext:     slt     $t0, $s0, $a1       # $t0 = 0 se $s0 >= $a1 (se i >= n)
            beq     $t0, $zero, esci1   # vai a esci1 se i >= n

            … “corpo del ciclo for esterno”

            addi    $s0, $s0, 1         # i = i + 1
            j       forext              # torna al test del ciclo esterno
esci1:
```

Ciclo for interno

Il ciclo for interno è:

```
for (j = i-1; j >= 0 && v[j]>v[j+1]; j = j-1) scambia(v, j);
```

```
        addi    $s1, $s0, -1        # j = i –1
forint: slti    $t0, $s1, 0         # $t0 = 1 se $s1 < 0 (j < 0)
        bne     $t0, $zero, esci2   # vai a esci2 se $s1 < 0 (j < 0)
        mul     $t1, $s1, 4         # $t1 = j * 4
        add     $t2, $a0, $t1       # $t2 = v + (j*4)
        lw      $t3, 0($t2)         # $t3 = v[j]
        lw      $t4, 4($t2)         # $t4 = v[j+1]
        slt     $t0, $t4, $t3       # $t0 = 0 se $t4 >= $t3
        beq     $t0, $zero, esci2   # vai a esci2 se $t4 >= $t3

        … “corpo del ciclo interno”

        addi    $s1, $s1, -1        # j = j –1
        j       forint
esci2:
```

Chiamata della procedura scambia

- La chiamata della procedura scambia è ovviamente

```
jal     scambia
```

- Problema dei parametri: la procedura ordina deve conservare i propri parametri nei registri $a0 e $a1, mentre la procedura scambia si aspetta i propri parametri sempre nei registri $a0 e $a1
- Quindi all'inizio della procedura ordina si copiano i parametri $a0 e $a1

```
move    $s2, $a0        # copia in $s2 l'indirizzo di v
move    $s3, $a1        # copia n in $s3
```

- E si passano i parametri alla procedura scambia facendo:

```
move    $a0, $s2        # copia in $a0 l'indirizzo di v (primo
                          parametro di scambia)
move    $a1, $s1        # copia j in $a1 (secondo parametro di
                          scambia)
```

Salvataggio dei registri sullo stack

- Occorre salvare il registro $ra perché ordina è una procedura ed è stata essa stessa chiamata (quando chiama scambia $ra viene modificato)
- Occorre salvare poi $s0, $s1, $s2, $s3 che sono usati da ordina (registri da preservare, supponendo che il chiamante di ordina li usi)

- Salvataggio sullo stack (all'inizio della procedura):

```
addi    $sp, $sp, -20       # crea spazio per 5 registri nello stack
sw      $ra, 16($sp)        # salva $ra nello stack
sw      $s3, 12($sp)        # salva $s3 nello stack
sw      $s2, 8($sp)         # salva $s2 nello stack
sw      $s1, 4($sp)         # salva $s1 nello stack
sw      $s0, 0($sp)         # salva $s0 nello stack
```

- Ripristino dei registri dallo stack (alla fine della procedura):

```
lw      $s0, 0($sp)         # ripristina $s0
lw      $s1, 4($sp)         # ripristina $s1
lw      $s2, 8($sp)         # ripristina $s2
lw      $s3, 12($sp)        # ripristina $s3
lw      $s4, 16($sp)        # ripristina $s4
addi    $sp, $sp, 20        # riaggiorna lo stack pointer
```

Procedura ordina completa

Figura 3.26
(Patterson&Hennessy)

Programma completo da provare con SPIM

```
prepara2:
        # questa procedura inserisce, a partire dalla posizione
        # 0x10000000 della memoria dati (v[0]), i numeri 100, 90, …, 10
        # (vettore di 10 elementi)
        ….

scambia:
        # procedura per lo scambio di 2 elementi in un vettore
        ….
ordina:
        # procedura per l'ordinamento crescente di un vettore
        ….
main:
        addi    $a1, $a1, 10        # metti in $a1 il valore di n (10)
        la      $a0, 0x10000000 # carica in $a0 l'indirizzo di partenza
                                    # del vettore v
        jal     prepara2
        jal     ordina
```

Da scrivere per esercizio

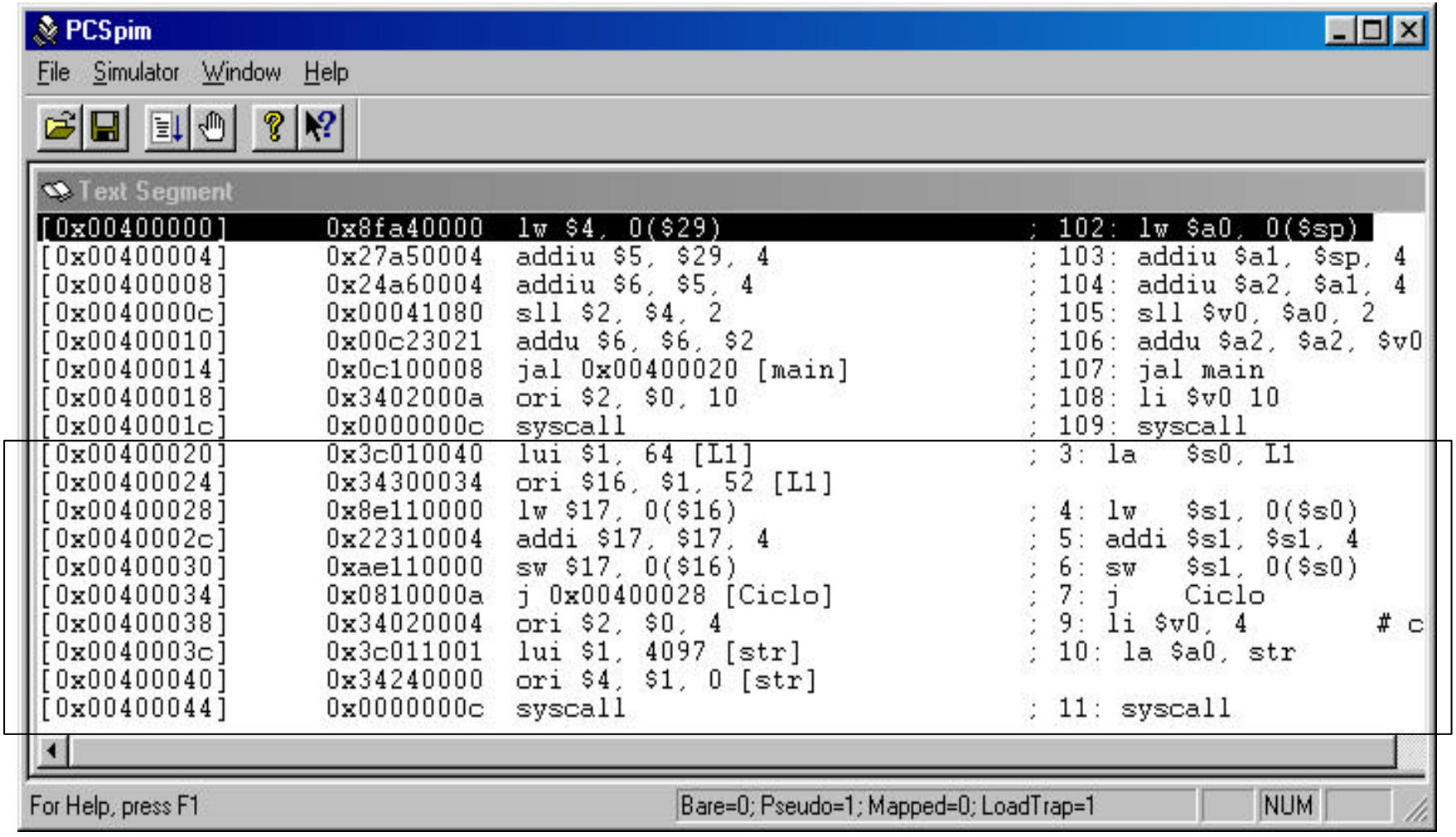
Programma assembler che si automodifica

```
main:
            .text
            la      $s0, L1             # metti in $s0 l'indirizzo di L1
Ciclo:      lw      $s1, 0($s0)         # metti in $s1 il contenuto dell'istruzione in L1
            addi    $s1, $s1, 4         # modifica il contenuto dell'istruzione in L1
            sw      $s1, 0($s0)         # memorizza l’istruzione modificata
L1:         j       Ciclo               # salta a Ciclo ??? (ma sarà vero???)

            li $v0, 4               # codice della chiamata di sistema per print_string
            la $a0, str             # indirizzo della stringa da stampare
            syscall                 # stampa della stringa
            .data
str:        .asciiz "ho già finito!"
```

Ad una prima occhiata il programma sembra eseguire un ciclo infinito, invece modifica il proprio codice ed esce subito dal ciclo

Codice prima dell'esecuzione



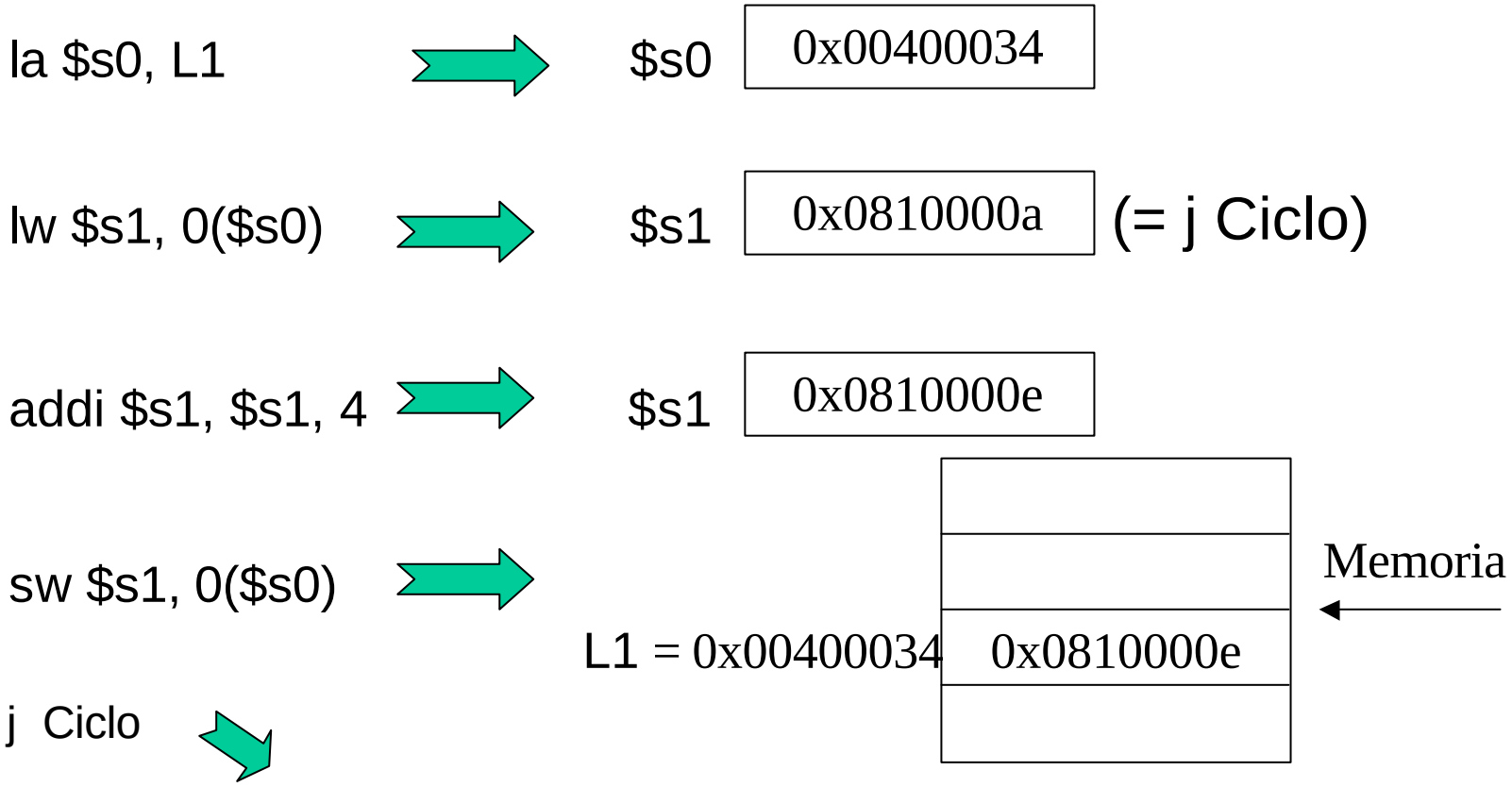
Significato delle linee nel segmento di testo di SPIM

- Ciascuna istruzione è visualizzata su una linea del tipo

  [0x0040002c] 0x22310004 addi $17, $17, 4 ; 5: addi $s1, $s1, 4

- Il primo numero, che compare fra parentesi quadre, è l'**indirizzo esadecimale** dell'istruzione
- Il secondo numero è la **codifica esadecimale** dell'istruzione
- Il terzo elemento è la **descrizione mnemonica** dell'istruzione
- Ciò che segue dopo il punto e virgola è l'effettiva **linea di codice del programma utente** che ha generato l'istruzione; il numero 5 è la linea del file in cui si trova l'istruzione
- Se non compare la parte dopo il punto e virgola significa che l'istruzione è stata generata da SPIM come parte della traduzione di una pseudoistruzione

Esecuzione



in L1 non c'è più j Ciclo ma un'altra istruzione, quale?
La sua codifica esadecimale è 0810000e… è un salto
all'istruzione successiva - li $v0, 4 - (con indirizzo 00400038)

Codifica esadecimale e binaria delle istruzioni

la $s0, L1 dove L1 = 0x00400034

è una pseudoistruzione che viene tradotta con 2 istruzioni (si noti che i 32 bit che codificano L1 non possono essere contenuti interamente in una istruzione)

lui $1, 64 (lui sta per *load upper immediato* 64 è la codifica decimale della parte più significativa dell'indirizzo 00400034)

carica la costante 64 nella mezza parola più significativa del registro $1 (che sarebbe il registro $at, un registro riservato all'assemblatore)

ori $16, $1, 52 (ori sta per *or immediato*)

fa l'or logico del contenuto di $1 con 52 (codifica decimale della parte meno significativa di 00400034) e pone il risultato in $16 (che sarebbe il registro $s0)

Istruzione	**Codifica esadecimale**	**Codifica binaria**
lui $1, 64	3c010040	0011 1100 0000 0001 0000 0000 0100 0000
ori $16, $1, 52	34300034	0011 0100 0011 0000 0000 0000 0011 0100

Codifica binaria nel formato delle istruzioni MIPS

Formato binario

op	rs	rt	Immediate
001111	00000	00001	0000 0000 0100 0000
001101	00001	10000	0000 0000 0011 0100

Formato decimale

op	rs	rt	Immediate
15 (lui)	0	1 ($at)	64
13 (ori)	1 ($at)	16 ($s0)	52

Con lui, $at conterrà: 0000 0000 0100 0000 0000 0000 0000 0000
Dopo ori, $s0 conterrà: 0000 0000 0100 0000 0000 0000 0011 0100
ovvero $s0 conterrà 00400034 - in esadecimale - (nota che è l'indirizzo dell'istruzione j Ciclo)

Codifica esadecimale, binaria e decimale delle istruzioni

lw $s1, 0($s0)

Istruzione	Codifica esadecimale	Codifica binaria
lw $s1, 0($s0)	8e110000	1000 1110 0001 0001 0000 0000 0000 0000

Formato	op	rs	rt	address
Binario	100011	10000	10001	0000 0000 0000 0000
Decimale	35 (lw)	16 ($s0)	17 ($s1)	0

Questo significa: vai a prendere la parola di memoria il cui indirizzo è dato dal contenuto di $s0 + 0 (cioè 00400034) e memorizzala in $s1

Da ciò, $s1 conterrà 0810 000a (che è la codifica di j Ciclo)

Codifica esadecimale, binaria e decimale delle istruzioni

addi $s1, $s1, 4

Istruzione	Codifica esadecimale	Codifica binaria
addi $s1, $s1, 4	22310004	0010 0010 0011 0001 0000 0000 0000 0100

Formato	op	rs	rt	immediate
Binario	001000	10001	10001	0000 0000 0000 0100
Decimale	8 (addi)	17 ($s1)	17 ($s1)	4

Questo significa: somma al contenuto di `$s1` (0810000a) il valore decimale/esadecimale 4

Si ottiene: $s1 = 0810000e (in esadecimale)

Codifica esadecimale, binaria e decimale delle istruzioni

sw $s1, 0($s0)

Istruzione	Codifica esadecimale	Codifica binaria
sw $s1, 0($s0)	ae110000	1010 1110 0001 0001 0000 0000 0000 0000

Formato	op	rs	rt	address
Binario	101011	10000	10001	0000 0000 0000 0000
Decimale	43 (sw)	16 ($s0)	17 ($s1)	0

Questo significa: memorizza nella parola di memoria il cui indirizzo è dato dal contenuto di $s0 $s1 (che è 0810000e)

In questo modo viene sovrascritta la locazione di memoria 00400034 che conteneva 0810000a (cioè j Ciclo) con la nuova istruzione 0810000e

Cosa significa la nuova istruzione?

Istruzione	Codifica esadecimale	Codifica binaria
???	0810000e	0000 1000 0001 0000 0000 0000 0000 1110

Formato	op	rs	rt	rd	shamt/ address	funct
Binario	000010	0000 0100 0000 0000 0000 0011 10				
Decimale	2 (j)	1048590				

L'indirizzo dell'istruzione jump è un indirizzo di 26 bit in **parole**

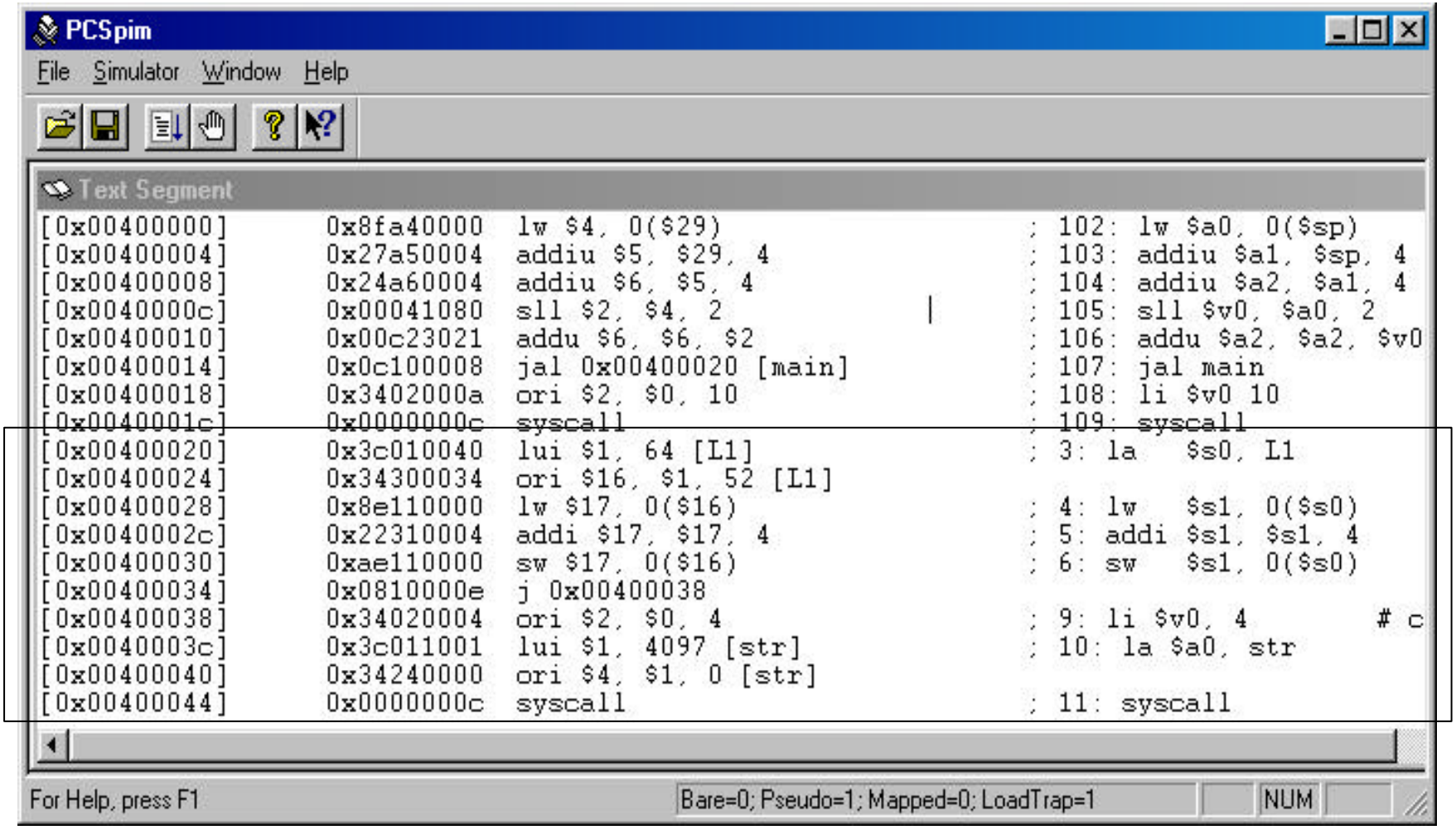
Occorre moltiplicarlo per 4, che in binario significa uno scorrimento a sinistra di 2 posizioni (o meglio si aggiungono due 0 nelle posizioni meno significative)

Si ottiene un indirizzo a 28 bit che va concatenato con i 4 bit più significativi di (PC+4) per ottenere l'effettivo indirizzo di salto

Si ottiene l'indirizzo esadecimale 00400038 e quindi la nuova istruzione risulta

j 0x00400038 (cioè un salto all'struzione successiva)

Codice dopo l'esecuzione



ATTENZIONE!

- La tecnica di programmazione appena mostrata è altamente **sconsigliata**!
- Tuttavia, lo stesso principio è applicabile nel caso di programmi che utilizzano istruzioni come dati, ad esempio programmi che elaborano altri programmi (es. i linker, soprattutto in passato)
- Ricordare: una parola di memoria può essere un'**istruzione** o un **dato** a seconda dell'uso che se ne fa

Esercizi

- Analizzare, con l’aiuto di SPIM, le **istruzioni macchina** corrispondenti ai programmi assembler visti (es. somma dei quadrati, scambia, etc…)