Linguaggio Assembler MIPS

Corso di Calcolatori Elettronici A

2007/2008

Sito Web:http://prometeo.ing.unibs.it/quarella

Prof. G. Quarella

prof@quarella.net

Architettura MIPS

- Architettura RISC sviluppata da John Hennessy nel 1981
- Usata da NEC, Nintendo, Silicon Graphics, Sony (Playstation), ecc.
- Architettura load-store
- Simile ad altre architetture RISC

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 operandi

  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
```

I registri del MIPS

- Gli operandi delle istruzioni aritmetiche sono dei ***registri***
- I registri del MIPS sono 32 di tipo **general-purpose** e sono ampi 32 bit ciascuno
- La convenzione MIPS utilizza come nomi dei registri due caratteri preceduti dal segno **$**
- In generale si usano $s0, $s1, …, $s7 per i registri che corrispondono a variabili nei programmi C da compilare, mentre $t0, $t1, …, $t9 per i registri temporanei
- Esistono particolari convenzioni nell'uso dei registri s e t all'interno di procedure.

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, nei 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 load word

- Tradurre g = h + A[8]
- Assumendo che il compilatore associ i registri $s1 e $s2 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
```

Indirizzi virtuali

Gli indirizzi di memoria che incontriamo non sono indirizzi fisici, ma indirizzi virtuali, in quanto l'architettura MIPS, come molte altre, implementa la memoria virtuale.

La memoria virtuale è una tecnica che permette di simulare una grande quantità di memoria avendo a disposizione una piccola quantità di memoria fisica (RAM).

Con indirizzi virtuali di 32 bit si possono indirizzare fino a 4GB di memoria virtuale (da 0x00000000 a 0xFFFFFFFF). Un meccanismo di traduzione implementato via hardware e software (S.O.) traduce gli indirizzi virtuali in indirizzi fisici.

La memoria virtuale

Uno spazio d'indirizzamento virtuale permette di raggiungere i seguenti obiettivi:

- Condivisione in maniera efficiente sicura della memoria e della CPU tra molti programmi.
- Eliminazione delle difficoltà di programmazione legate ad una quantità piccola e limitata di memoria principale.

Un programma, compilato in uno spazio di indirizzamento virtuale, ha quindi un proprio spazio di indirizzamento separato da quello degli altri programmi consentendo la condivisione e protezione di cui sopra. Può superare le dimensioni della memoria principale e non è necessario che occupi un blocco contiguo di memoria.

La gestione della memoria da assegnare ai programmi è quindi semplificata (es.: problemi di rilocazione, overlay).

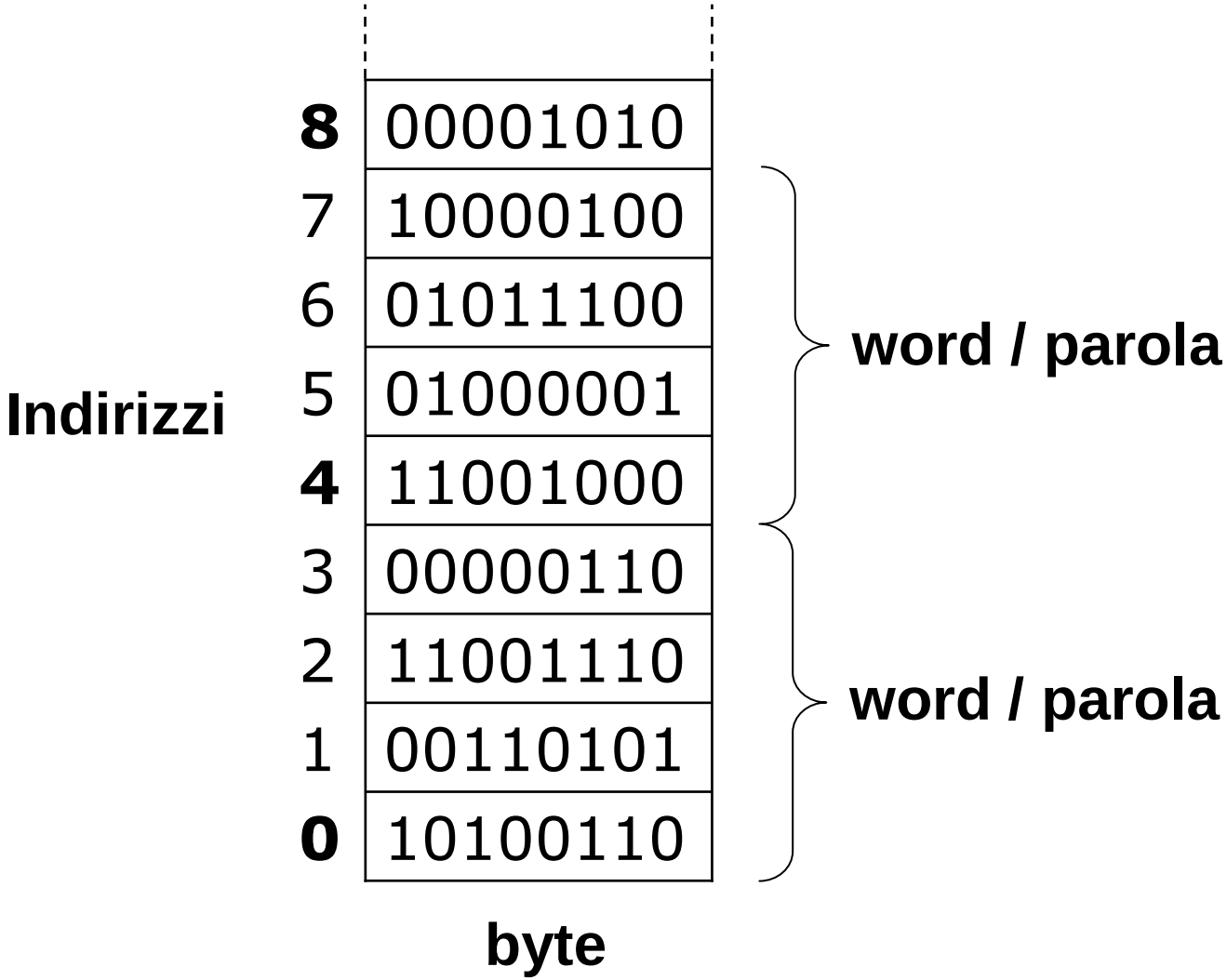
Indirizzamento a byte

- Il MIPS indirizza il singolo byte (8 bit)
- L'indirizzo punta al primo byte della parola (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 (vincolo di allineamento)
- Quindi, nell'istruzione precedente, occorreva fare

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

Indirizzamento a byte
Vincolo di allineamento

(una quantità è allineata se il suo indirizzo di memoria
è un multiplo della sua dimensione espressa in byte)



Ordinamento dei byte

42044670_{10}=00000010 10000001 10001100 11111110_2

Architettura big-endian

3	11111110
2	10001100
1	10000001
0	00000010

big-end-first

Le locazioni di memoria sono occupate a partire dal byte più a sinistra, e quindi più significativo.

Architettura little-endian

3	00000010
2	10000001
1	10001100
0	11111110

little-end-first

Le locazioni di memoria sono occupate a partire dal byte più a destra, e quindi meno significativo.

L'architettura MIPS è di tipo big-endian

I vettori nel linguaggio C

Gli indici di un vettore in C partono sempre da zero.

La dichiarazione C

`long x[3]`

alloca spazio per i seguenti 3 elementi di tipo long:

`x[0]`, **`x[1]`** e **`x[2]`**

Compilando in assembler il registro base contiene l'indirizzo del primo elemento del vettore avente cioè indice zero.

In generale l'offset è dato quindi dalla seguente relazione:

offset=indice • [dimensione in byte dell'elemento]

Istruzione store (sw store word)

- 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 in $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:

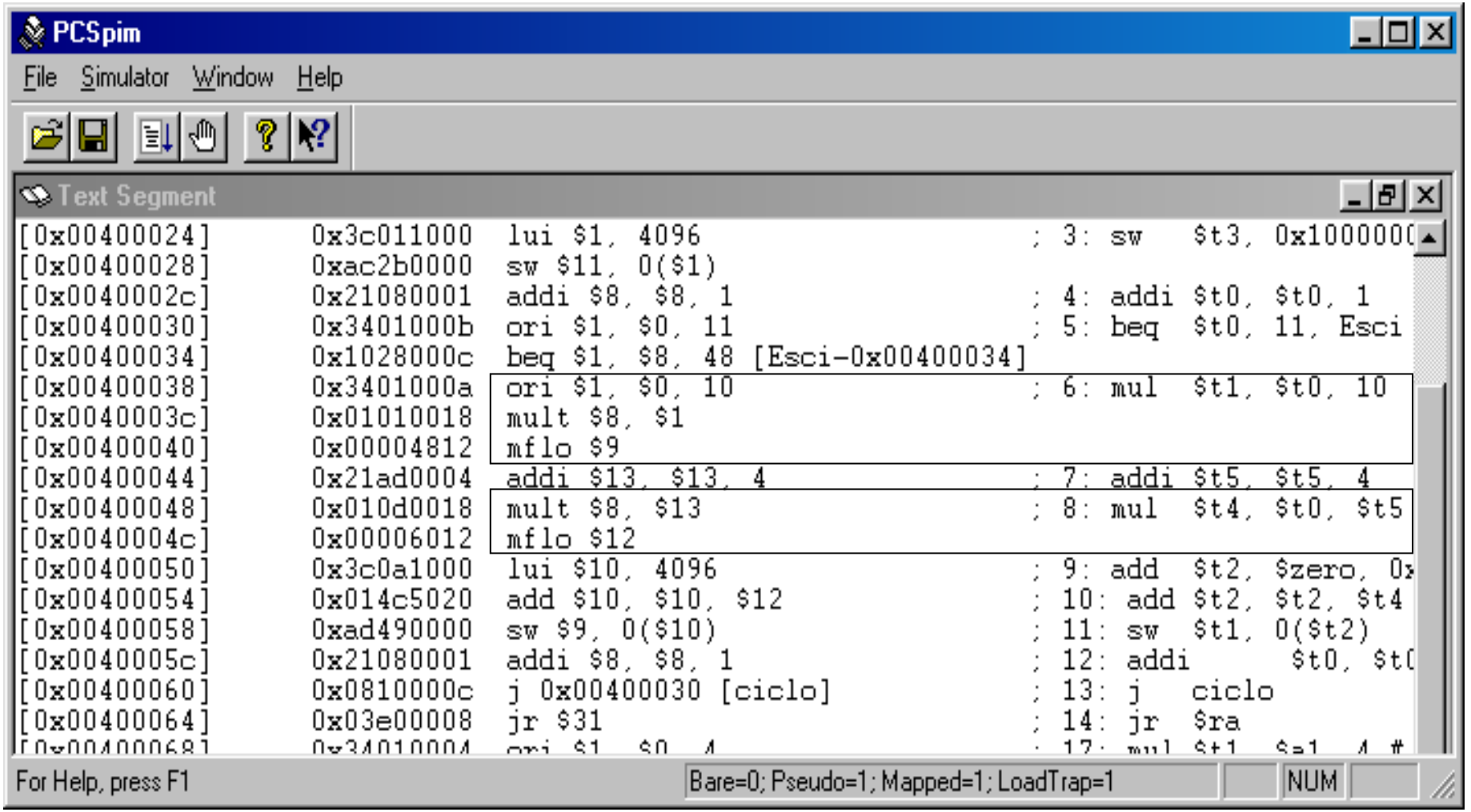
```
mult $s2, $s3     # hi, lo = $s2*$s3
mflo $s1          # s1 = lo
```

- In pratica useremo la **pseudoistruzione** mul

  mul rdest, rsrc1, rsrc2

  che mette il prodotto fra rsrc1 e rsrc2 nel registro rdest

La traduzione di “mul”



Salti condizionati

Istruzioni **beq** e **bne** sono utilizzate 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 on 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 on not equal**)

Esempio

- Frammento di codice C (goto è deprecato!):

```
        if (i = = j) goto 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
```

Salto incondizionato

Istruzione **j** (**jump**)

J LABEL

Vai all’istruzione etichettata con LABEL.

L’assemblatore sostituirà l’etichetta con l’indirizzo appropriato.

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) goto 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 l'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]
        addi  $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’istruzione 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 può essere richiamata in più parti del programma

Chiamata di una procedura

- L'istruzione per effettuare il salto di ritorno è

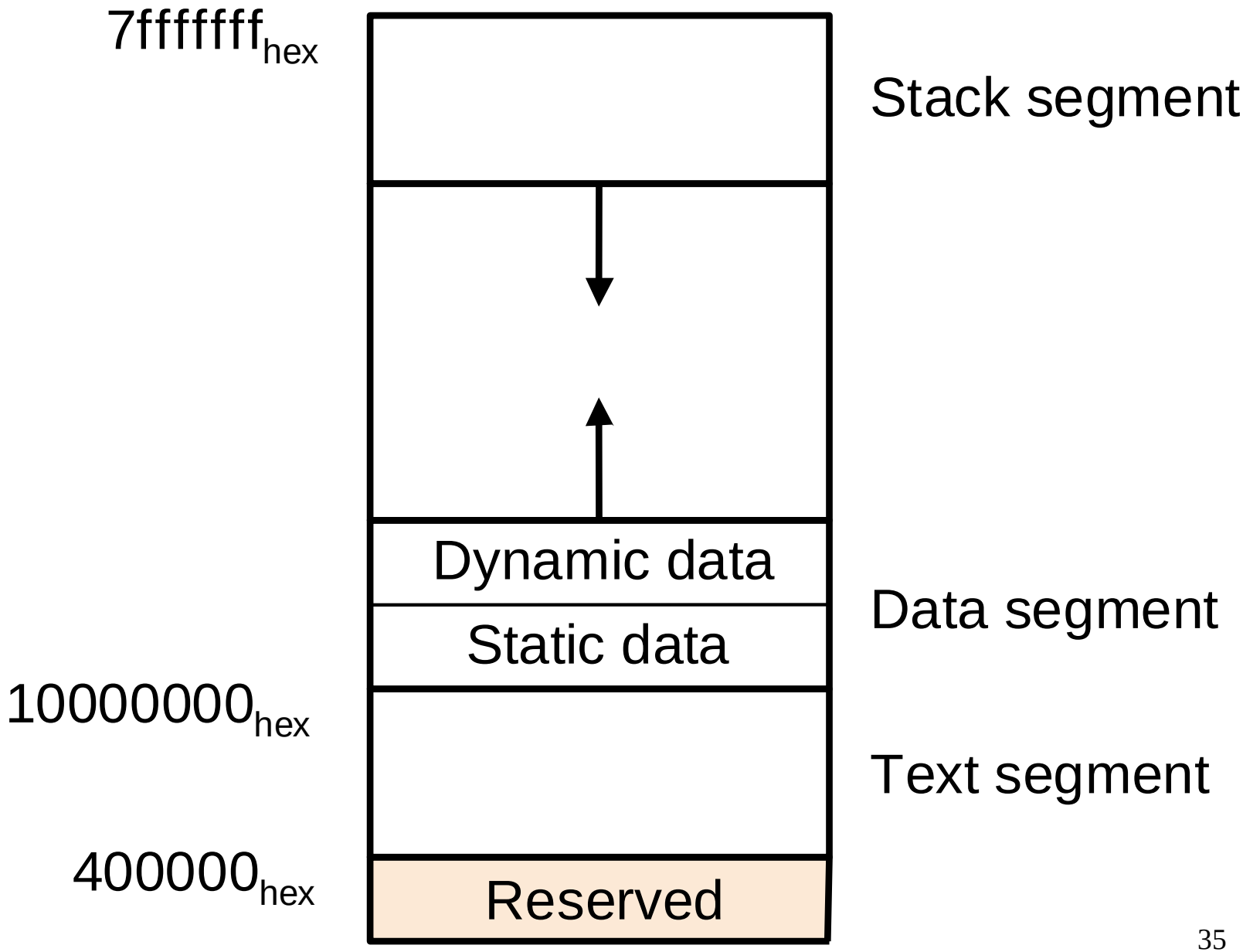
  jr $ra

- **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 eventualmente i risultati nei registri $v0-$v1 e restituisce il controllo al chiamante con l'istruzione jr $ra**

Lo 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 (LIFO)* dove una procedura può memorizzare i registri e dove può recuperare i vecchi valori dei registri
- In generale il processo di allocare in memoria le variabili usate meno frequentemente o usate in fasi successive del programma, si chiama **riversamento** (spilling) dei registri.
- Si usa un puntatore allo stack (**stack pointer**) che contiene l'indirizzo del dato introdotto più recentemente nello stack
- Per ragioni storiche 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**
- I termini **push** e **pop** sono comunemente usati per indicare rispettivamente la memorizzazione e l'estrazione di un dato dallo stack.

Organizzazione della memoria nell'architettura MIPS



Registri preservati e non preservati

- Il MIPS offre 2 classi di registri ed adotta la seguente convenzione:

  $t0-$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 tramite lo stack dal programma chiamato)

- In questo modo si riduce la necessità di salvare **tutti** i registri in memoria, ottimizzando implicitamente il codice.
- Visto che l'utilizzo dei registri s comporta delle dispendiose operazioni in memoria in una procedura **bisogna utilizzare prima i registri t e solo se questi non sono sufficienti utilizzare anche i registri s.**

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:
  - **Il programma *chiamante* *(caller)* 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* *(callee)* salva nello stack il registro di ritorno $ra, se chiama altre procedure, e gli altri registri $s0 - $s7 che utilizza, nel caso non siano sufficienti i registri temporanei $t0 - $t9.**

Una procedura che chiama un'altra procedura è sia 'chiamato' che 'chiamante'. L'utilizzo dei registri e la gestione dello stack è illustrata dall'esempio seguente.

Esercizio: scrivere una procedura in assembler MIPS, che riceva in ingresso l'indirizzo di una stringa ed un numero che, se pari a 0, determina la conversione dei caratteri in minuscolo altrimenti i caratteri vengono convertiti in maiuscolo. Per la conversione del singolo carattere si supponga di utilizzare le funzioni toupper e tolower.

Scrivere una bozza del corpo del programma:

```
        move $t0, $zero
loop:   addu $t1, $t0, $a0
        lb $t2, 0($t1)
        beq $t2,  $zero, esci
        move $a0, $t2          #err. si perde a0…
        beq $a1, $zero, tolow
        jal toupper
        j skip
tolow:  jal tolower
skip:   sb $v0, 0($t1)         #err. t1 è variato…
        addi $t0, $t0, 1       #err. t0 è variato…
        j loop
esci:
```

Procedere correggendo i registri: in particolare le chiamate alle funzioni possono modificare i registri $a0, $a1 e $t0, per cui bisogna utilizzare dei registri s. Apportare inoltre delle migliorie.

```
        move $s0, $a0
        move $s1, $a1
loop:   lb $t0, 0($s0)
        beq $t0,  $zero, esci
        move $a0, $t0
        beq $s1, $zero, tolow
        jal toupper
        j skip
tolow: jal tolower
skip:   sb $v0, 0($s0)
        addiu $s0, $s0, 1
        j loop
esci:
```

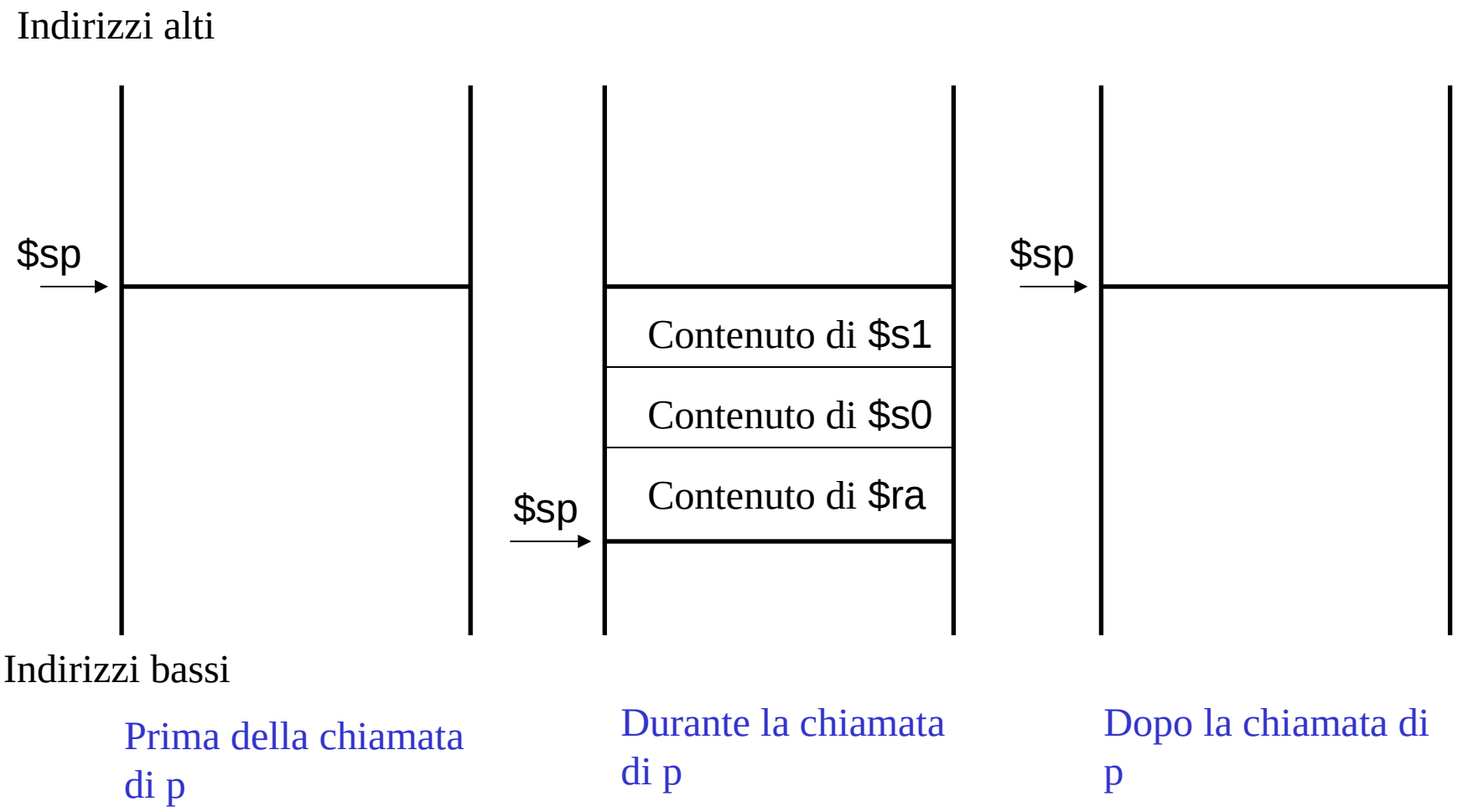
Completare il programma gestendo il salvataggio tramite lo stack dei registri s utilizzati e del registro **ra** che viene modicato dalle funzioni annidate. Gestire infine il ritorno al programma chiamante.

```
p:      subu $sp, $sp, 12
        sw $ra, 0($sp)
        sw $s0, 4($sp)
        sw $s1, 8($sp)

        [corpo del programma]

esci:   lw $ra, 0($sp)
        lw $s0, 4($sp)
        lw $s1, 8($sp)
        addiu $sp, $sp, 12
        jr $ra
```

Valori dello stack pointer



Procedura ricorsiva per il calcolo del fattoriale

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

Il parametro n corrisponde al registro argomento $a0.

In caso di procedure ricorsive bisogna considerare che programma chiamato e chiamante coincidono.

Procedura ricorsiva per il calcolo del fattoriale

```
fatt:
        subu    $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)
        slti    $t0, $a0, 1     # test per n < 1
        beq     $t0, $zero, L1  # se n >= 1 salta a L1
        addi    $v0, $zero, 1   # restituisce 1 (per n<1)
        addi    $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
        addiu   $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 $ra 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
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
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

- È stato pensato per rendere veloce l'accesso a costanti di piccole dimensioni (2^{16} o $\pm 2^{15}$)
- 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

  ```
  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 codice macchina in *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
```

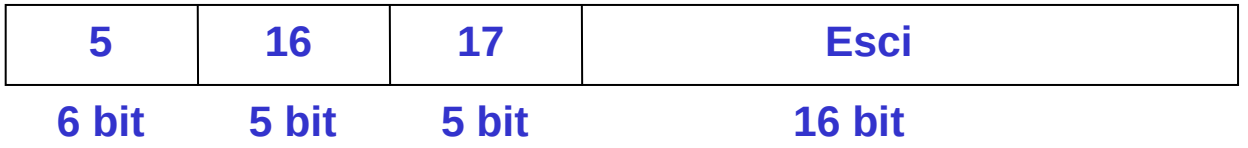
Indirizzamento nei salti

- Istruzione jump

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

- Istruzione branch on 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

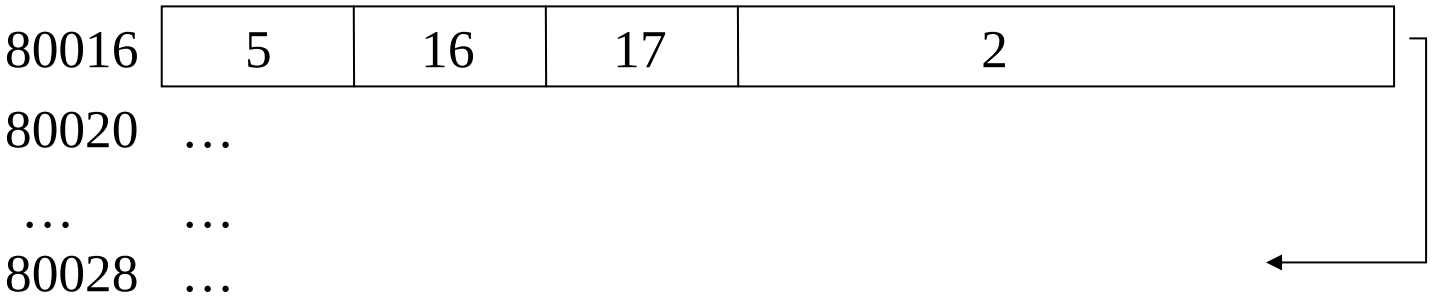
- Con 16 bit di indirizzo nessun programma potrebbe avere dimensioni maggiori di 2^{16}
- L'alternativa: specificare un registro il cui valore deve essere sommato al registro di salto, cioè:

  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 usato per tutti i tipi di *salto condizionato*, mentre le istruzioni di jump e 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**
- Significa quindi che nell'istruzione bne la destinazione del salto si ottiene sommando 2x4=8 byte all'indirizzo dell'istruzione successiva (poiché il PC contiene già 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:
```

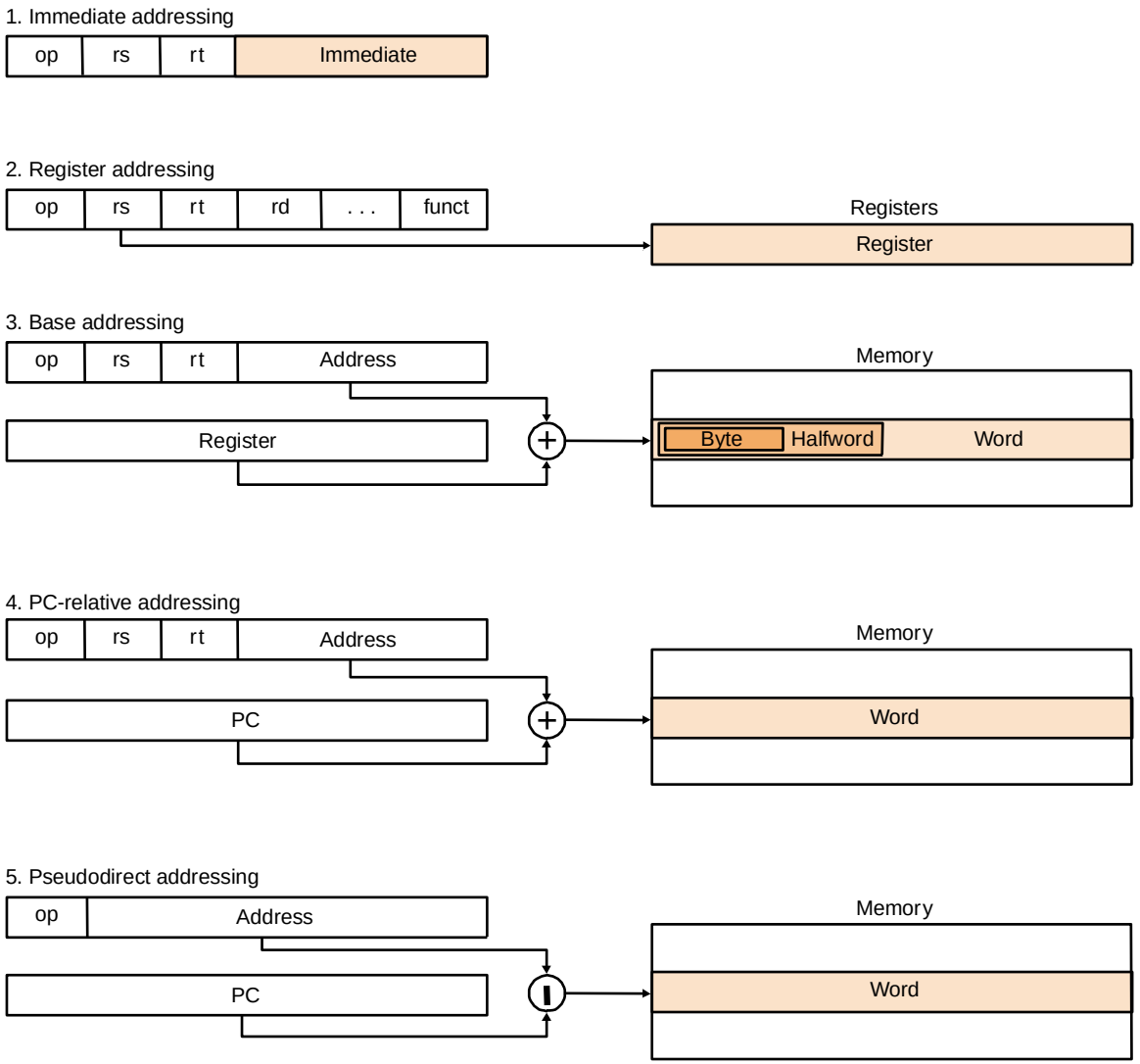
- **I 26 bit all'interno dell'istruzione jump contengono un indirizzo in parole, equivalgono quindi a 28 bit che vengono concatenati con i 4 bit più significativi del PC (indirizzamento pseudo-diretto).**

Indirizzamento pseudo-diretto

Con 28 bit riesco ad indirizzare 2^{28} byte ovvero 256MB equivalenti a 64M istruzioni, all'interno però del blocco di memoria i cui indirizzi hanno i 4 bit più significativi pari ai 4 bit più significativi dell'indirizzo in cui è contenuta l'istruzione `jump`.

Il loader ed il linker devono fare in modo che il programma stia all'interno di un tale blocco di 256MB, diversamente si ricorre ad una **`jump register`**, che può indirizzare 4GB con la necessità di un istruzione aggiuntiva che carichi l'indirizzo desiderato in un registro.

I 5 modi di indirizzamento



Formati delle istruzioni viste

Formato-R o Tipo-R (istruzioni aritmetiche)

op	rs	rt	rd	shamt	funct

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

op	rs	rt	address

Formato-J o Tipo-J (salto incondizionato)

op	address

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 più 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)

SPIM

- SPIM (MIPS letto al contrario) è un simulatore software che permette l'esecuzione di programmi scritti per i processori MIPS R2000/R3000.
- Contiene un debugger e fornisce servizi analoghi a quelli di un sistema operativo.
- Esegue direttamente programmi scritti in assembler.
- Utilizza l'ordinamento dei byte proprio dell'hardware su cui gira, per cui PCSpim, la versione per l'architettura Intel 80x86, è little-endian.
- Utile per studiare i calcolatori ed i programmi eseguiti su di esso.

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

Servizio	Codice della chiamata di sistema	Argomenti	Risultato
print_int	1	$a0=intero	
print_float	2	$f12=virgola mobile	
print_double	3	$f12=doppia precisione	
print_string	4	$a0=stringa	
read_int	5		intero (in $v0)
read_float	6		virgola mobile (in $f0)
read_double	7		doppia precisione (in $f0)
read_string	8	$a0=buffer, $a1=lunghezza	
sbrk	9	$a0=quantità	indirizzo (in $v0)
exit	10		

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 terminandola con Nul)

- 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 finché 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 = "
```

Scambio registri ottimale

```
        .text
        .globl __start
__start:
        # swap values of $t0 and $t1 (xor-
        based)
        xor $t0, $t0, $t1
        xor $t1, $t0, $t1
        xor $t0, $t0, $t1
```

Da eseguire in SPIM senza caricare il trap file.

Scambio registri ottimale

(Dimostrazione)

t0′ = t0 XOR t1

t1′ = t0′ XOR t1= t0 XOR t1 XOR t1 = t0 XOR 0 = t0

t0″ = t0′ XOR t1′ = t0 XOR t1 XOR t0 = t1 XOR 0 = t1

Nota

A XOR A = 0

A XOR 0 = A

Funzione sn

sn(x, k) restituisce il numero delle somme di k addendi che danno x (x e k sono quindi interi positivi) ed è così definita:

sn(x, k) = sn(x-k, k) + sn(x-1, k-1)

sn(n, n) = 1

sn(x, k) = 0 per x < k e k=0

Funzione sn

x \ k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	1	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	1	2	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	1	3	3	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	1	3	4	3	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	1	4	5	5	3	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0
9	0	1	4	7	6	5	3	2	1	1	0	0	0	0	0	0	0	0	0	0	0
10	0	1	5	8	9	7	5	3	2	1	1	0	0	0	0	0	0	0	0	0	0
11	0	1	5	10	11	10	7	5	3	2	1	1	0	0	0	0	0	0	0	0	0
12	0	1	6	12	15	13	11	7	5	3	2	1	1	0	0	0	0	0	0	0	0
13	0	1	6	14	18	18	14	11	7	5	3	2	1	1	0	0	0	0	0	0	0
14	0	1	7	16	23	23	20	15	11	7	5	3	2	1	1	0	0	0	0	0	0
15	0	1	7	19	27	30	26	21	15	11	7	5	3	2	1	1	0	0	0	0	0
16	0	1	8	21	34	37	35	28	22	15	11	7	5	3	2	1	1	0	0	0	0
17	0	1	8	24	39	47	44	38	29	22	15	11	7	5	3	2	1	1	0	0	0
18	0	1	9	27	47	57	58	49	40	30	22	15	11	7	5	3	2	1	1	0	0
19	0	1	9	30	54	70	71	65	52	41	30	22	15	11	7	5	3	2	1	1	0
20	0	1	10	33	64	84	90	82	70	54	42	30	22	15	11	7	5	3	2	1	1

Funzione sn

Implementazione in C

```
int sn(int x, int k)
{
 if (x==k) return 1;
 else if (x<k || k<=0) return 0;
 else return sn(x-k,k)+sn(x-1,k-1);
}
```

Funzione sn

Implementazione in MIPS: x = $a0, k = $a1

```
li  $v0, 1                  # sn=1
beq $a0, $a1, ESCI          # Esci se x=k
move $v0, $zero             # sn=0
slt $t0, $zero, $a1
beq $t0, $zero, ESCI        # Esci se k<=0
slt $t0, $a0, $a1
bne $t0, $zero, ESCI        # Esci se x<k
```

Funzione sn

$s0, $s1 e $s2 registri di appoggio per memorizzare i parametri ed un risultato intermedio, evitando l'uso dello stack tra le jal.

```
move $s0, $a0
move $s1, $a1
sub $a0, $s0, $s1
jal sn                  # sn(x-k,k)
move $s2, $v0
addi $a0, $s0, -1
addi $a1, $s1, -1
jal sn                  # sn(x-1,k-1)
add $v0, $v0, $s2
```

Funzione sn

```
sn:     sub $sp, $sp, 16
        sw  $ra, 12($sp)
        sw  $s0, 8($sp)
        sw  $s1, 4($sp)
        sw  $s2, 0($sp)
        … (corpo della funzione)
ESCI:   lw  $s0, 8($sp)
        lw  $s1, 4($sp)
        lw  $s2, 0($sp)
        lw  $ra, 12($sp)
        addi $sp, $sp, 16
        jr  $ra
```

Funzione sn

È stata effettuata un'ottimizzazione in modo da utilizzare lo stack solo quando la funzione effettua le chiamate ricorsive e non quando è già in grado di restituire il risultato.

È stata spostata…

```
sn:     li  $v0, 1
        beq $a0, $a1, ESCI
        move $v0, $zero
        slt $t0, $zero, $a1
        beq $t0, $zero, ESCI
        slt $t0, $a0, $a1
        bne $t0, $zero, ESCI
        sub $sp, $sp, 16
        sw  $ra, 12($sp)
        sw  $s0, 8($sp)
        sw  $s1, 4($sp)
        sw  $s2, 0($sp)
        move $s0, $a0
        move $s1, $a1
        sub $a0, $s0, $s1
        jal sn
        move $s2, $v0
        addi $a0, $s0, -1
        addi $a1, $s1, -1
        jal sn
        add $v0, $v0, $s2
        lw  $s0, 8($sp)
        lw  $s1, 4($sp)
        lw  $s2, 0($sp)
        lw  $ra, 12($sp)
        addi $sp, $sp, 16
ESCI:   jr  $ra
```

Non è necessario salvare alcun registro

PUSH

Corpo della funzione con le chiamate ricorsive

POP

Esercizio

Data l'implementazione multi-ciclo discussa nel 5°capitolo del libro di testo, descrivere che cosa avviene nel 19°ciclo di clock relativamente all'esecuzione delle seguenti istruzioni MIPS, supposto che i registri $t1 e $t0 contengano rispettivamente 40 e 4.

```
CICLO: sub $t1, $t1, $t0
       add $t2, $s0, $t1
       lw $t3, 0($t2)
       add $s1, $s1, $t3
       bne $t1, $zero, CICLO
```

Esercizio - soluzione

Evidenziando i cicli di clock richiesti da ogni istruzione, si ricava che durante il 19°ciclo è in corso il 2°ciclo dell'istru zione bne; quindi siamo nella fase di decodifica e caricamento dei registri. In questo caso specifico:

A = $t1

B = $zero

ALUOut = PC + (sign-ext(CICLO) << 2)

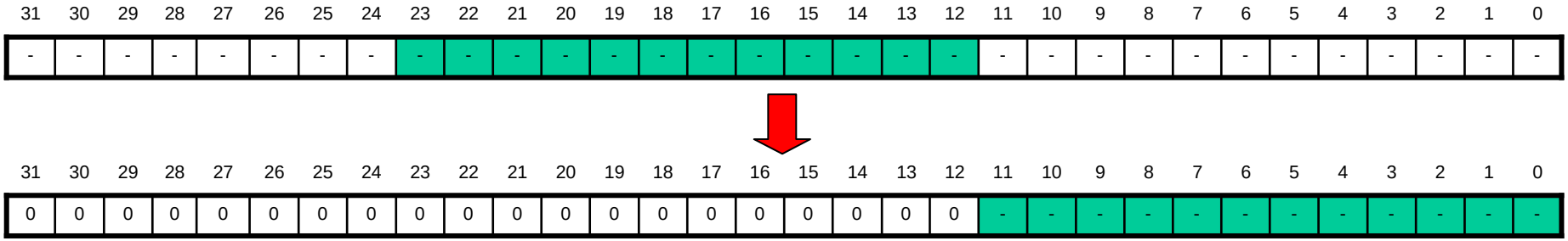
CICLO:	**sub $t1, $t1, $t0**	**4**
	add $t2, $s0, $t1	**4**
	lw $t3, 0($t2)	**5**
	add $s1, $s1, $t3	**4**
	bne $t1, $zero, CICLO	**3**

Istruzioni logico-aritmetiche

MIPS	C	Pseudo	Immediate	Unsigned	Note
add r, s, t	r = s + t		×	×	
sub r, s, t	r = s - t			×	
mult s, t	(Hi Lo) = s * t			×	
mul r, s, t	r = s * t	×		-	
div s, t	Lo = s / t ; Hi = s % t			×	
div r, s, t	r = s / t	×		×	
rem r, s, t	r = s % t	×		×	
and r, s, t	r = s & t		×		
or r, s, t	r = s | t		×		
not r, s	r = ~ s	×			
xor r, s, t	r = s ^ t		×		
nor r, s, t	r = ~ (s | t)				
sll r, s, c	r = s << c				(1)
sllv r, s, t	r = s << t				
srl r, s, c	r = s >> c				(1)
srlv r, s, t	r = s >> t				
sra r, s, c					(1)
srav r, s, t					

Note
1) c costante, formato-R (rs ignorato)

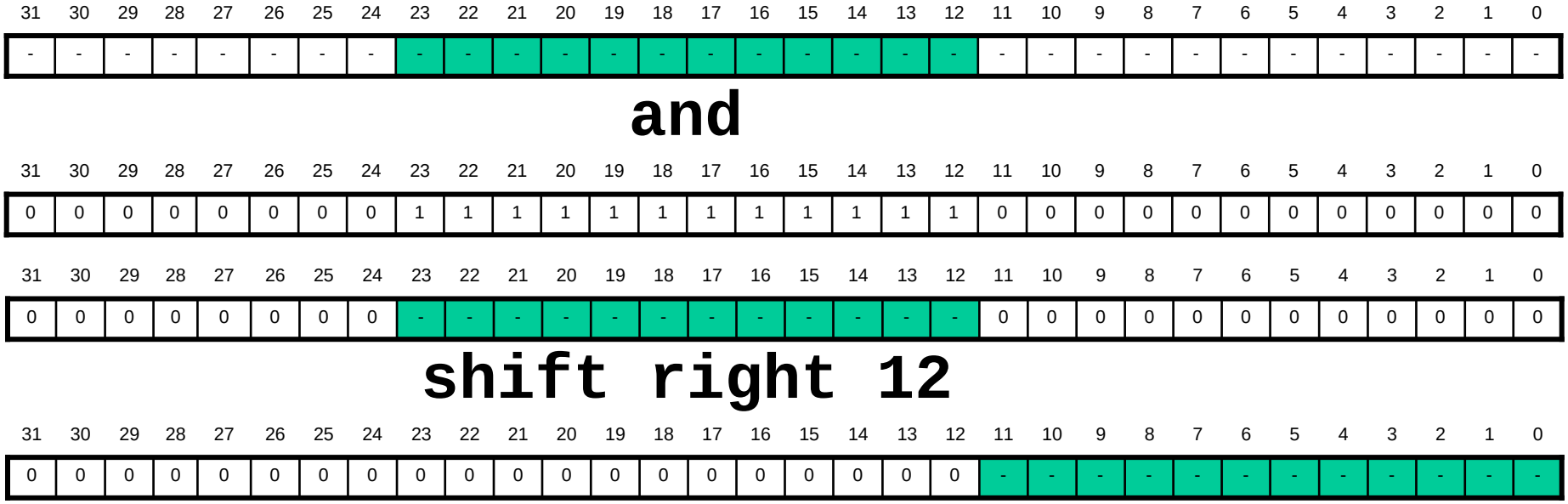
Scrivere un frammento di codice per estrarre dal registro s0 il numero positivo rappresentato dai bit dalla posizione 12 alla 23.



Si possono utilizzare due tecniche:

1) **Maschera di bit + operazione di scorrimento**
2) **2 operazioni di scorrimento**

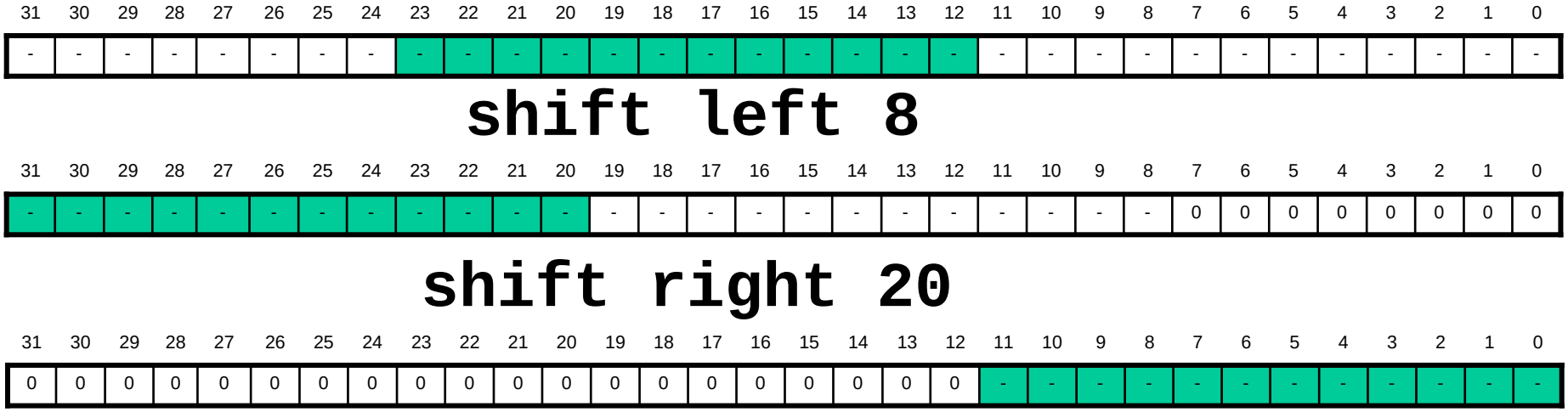
1) Maschera di bit + operazione di scorrimento




```
li $t0, 0xFFF000
and $s1, $s0, $t0   #andi se const. <= 16 bit
srl $s1, $s1, 12
```

In questo esempio sarebbe bastato uno srl seguito da andi.

2) 2 operazioni di scorrimento




```
sll $s1, $s0, 8
srl $s1, $s1, 20
```

Hexify

```
          .text
          .globl __start
__start:
          li $v0, 5                    # read integer n
          syscall
          move $t0, $v0                # save n
          li $t1, 7                    # 7 (+ 1) hex digits for 32 bits
hexify:   and $t2, $t0, 0x0F           # extract least significant 4 bits
          srl $t0, $t0, 4              # prepare for next digit
          lb $t3, hex_table($t2)       # convert 4 bit group to hex digit
          sb $t3, hex_digits($t1)      # store hex digit
          sub $t1, $t1, 1              # next digit
          bgez $t1, hexify             # more digits?
          li $v0, 4                    # print string
          la $a0, the_result_is
          syscall
          li $v0, 10                   # exit
          syscall

          .data
hex_table:          .ascii "0123456789ABCDEF"
the_result_is:      .ascii "Hexadecimal value: 0x"
hex_digits:         .asciiz "XXXXXXXX"
```

Hexify: note

Le istruzioni **`lb`** e **`sb`** non utilizzano la modalità di indirizzamento **`c(rx)`** consentita dall'hardware, ma un'altra concessa solo dall'assemblatore che deve quindi effettuare la seguente traduzione:

```
lui $at, 4097              ; lb $t3, hex_table($t2)
addu $at, $at, $t2
lb $t3, 0($at)
lui $at, 4097              ; sb $t3, hex_digits($t1)
addu $at, $at, $t1
sb $t3, 37($at)
```

Riferimenti

Computer Organization and Design

The Hardware/Software Interface 3rd Edition

David A. Patterson, John L. Hennessy

Capitolo 2

Appendice A

Versione italiana:

Struttura e Progetto dei Calcolatori

L'Interfaccia Hardware-Software

2ª edizione Zanichelli

http://en.wikipedia.org/ o http://it.wikipedia.org/