

Progetto di Architetture degli Elaboratori

A cura di:

- Bucciero Samuele, matricola: 5618430
E-mail: samuele.bucciero@gmail.com
- Mercatanti Elia - matricola: 5619856
E-mail: eliamercatanti@yahoo.it
- Pinto Andrea, matricola: 5641136
E-mail: andrea.pinto1@stud.unifi.it

Data consegna: 31/08/2014

Esercizio di progetto n. 1: Unione e Intersezione di Vettori

Testo:

Utilizzando [QtSpim](#), scrivere e provare un programma che permetta all'utente di inserire fino ad un massimo di 5 vettori di caratteri (Vet-1, ..., Vet-5), ognuno contenente al massimo 50 caratteri inseriti leggendo da tastiera un carattere alla volta (0 per terminare), e che generi e visualizzi su console i seguenti due vettori:

- Vettore Unione: vettore di caratteri che è l'unione (in senso insiemistico) di tutti i vettori inseriti dall'utente. Il vettore unione dovrà quindi contenere (in un qualsiasi ordine) tutti i caratteri dei vettori inseriti precedentemente evitandone le duplicazioni. Il nuovo vettore creato non potrà comunque contenere più di 50 caratteri, quindi gli eventuali caratteri eccedenti andranno persi ed il vettore troncato alla dimensione massima.

Per esempio, se ho inserito i seguenti tre vettori:

Vet-1: [3 3 9 3 3]

Vet-2: [g g 6 3 a a 9 h h 6 7 8]

Vet-3: [f 3 d 3 p p 9 à q]

si dovrà visualizzare su console :

Vet-1: [3 3 9 3 3]

Vet-2: [g g 6 3 a a 9 h h 6 7 8]

Vet-3: [f 3 d 3 p p 9 à q]

Vettore-Unione: [3 9 g 6 a h 7 8 f d p à q]

- Vettore Intersezione: vettore di caratteri che è l'intersezione (in senso insiemistico) di tutti i vettori inseriti dall'utente. Il vettore intersezione dovrà quindi contenere (in un qualsiasi ordine) solo i caratteri che appaiono almeno una volta in tutti i vettori precedentemente creati.

Per esempio, se ho inserito i seguenti tre vettori:

Vet-1: [3 3 9 3 3]

Vet-2: [g g 6 3 a a 9 h h 6 7 8]

Vet-3: [f 3 d 3 p p 9 à q]

si dovrà visualizzare su console :

Vet-1: [3 3 9 3 3]

Vet-2: [g g 6 3 a a 9 h h 6 7 8]

Vet-3: [f 3 d 3 p p 9 à q]

Vettore-Intersezione: [3 9]

Implementazione:

.data

```
sceltaAzione: .asciiz "\n Decidere se fare un'unione oppure un' intersezione (0=unione;1=intersezione)"
howManyArrays: .asciiz "\nQuanti array vuoi usare? (max 5)--> "
howManyElements: .asciiz "\nQuanti elementi vuoi inserire? (max 50)--> "
emptyArray: .asciiz "\n Un vettore e' vuoto , dunque l' intersezione dei vettori è un vettore vuoto"
vettoreUnione: .asciiz "\n Vettore-Unione: "
vettoreIntersezione: .asciiz "\n Vettore-Intersezione: "
vett0: .asciiz "\n Vett0: "
vett1: .asciiz "\n Vett1: "
vett2: .asciiz "\n Vett2: "
vett3: .asciiz "\n Vett3: "
vett4: .asciiz "\n Vett4: "
input: .asciiz "\nInserire un carattere --> "
fine: .asciiz "\nFine del programma!"
titolo: .asciiz "\n Esercizio 1: unione/intersezione di array"
array0: .space 51 #alloca 50 byte (ogni carattere vale 1 byte)
array1: .space 51 #alloca 50 byte
array2: .space 51 #alloca 50 byte
array3: .space 51 #alloca 50 byte
array4: .space 51 #alloca 50 byte
array5: .byte 0:53 #array unione
arrayDim: .space 28 #array per 7 interi, che esprimono la grandezza di ogni vettore, il numero di array usati, la grandezza e l' indice del piu piccolo
```

La prima parte del programma introduce i dati da caricare nel segmento dati:

- **sceltaAzione**, è una stringa che verrà utilizzata per stampare all'utente la richiesta su che operazione intraprendere.
- **howManyArrays**, è una stringa utilizzata per chiedere all'utente quanti array ha intenzione di usare.
- **howManyElements**, è una stringa utilizzata per chiedere all'utente quanti elementi vuole inserire nell'array che deve essere caricato.
- **emptyArray**, è una stringa che segnala la presenza di un vettore vuoto.
- **vettoreUnione**, **vettoreIntersezione**, **vett0**, **vett1**, **vett2**, **vett3**, **vett4** sono delle stringhe che verranno utilizzate per stampare correttamente i vari array che l'utente ha inserito.
- **input**, verrà utilizzata per richiedere all'utente l'inserimento di un carattere.
- **fine**, è una stringa che segnala l'uscita dal programma.
- **titolo**, è una stringa che verrà utilizzata per stampare all'utente un titolo per il programma.
- **array0**, **array1**, **array2**, **array3**, **array4**, allocano lo spazio per 51 byte destinati a contenere i 50 caratteri ricevuti dall'esterno ed il carattere di fine stringa.
- **array5**, alloca lo spazio per i 50 caratteri massimi che può contenere l'array unione, più il carattere di fine stringa e le due parentesi quadre. Notare che viene inizializzato a 0 in ogni sua posizione.
- **arrayDim**, alloca lo spazio per contenere le grandezze di ognuno dei cinque vettori, il numero di array effettivamente usati, la grandezza dell'array più piccolo ed il suo indice.

La seconda parte del programma contiene la parte relativa al codice, ovvero contiene tutte le varie istruzioni che il programma dovrà seguire per generare correttamente l'output richiesto.

Come prima cosa viene stampata la stringa "**titolo**" e vengono inizializzati i seguenti registri: **\$t7**, che conterrà il numero totale di elementi inseriti; **\$t1**, **\$t0** che vengono subito usati per inserire la parentesi aperta quadra nella prima posizione del vettore finale.

```
main:
    li $t7,0          #num elementi totali

    li $t1,0
    li $t0,91
    sb $t0,array5($t1) #carico la parentesi quadra aperta nella prima posizione del vettore risultato

    la $a0,titolo      #stampo il titolo
    li $v0,4
    syscall
```

Successivamente, all'interno delle etichette `quantiArray` e `quantiElementi`, viene richiesto all'utente di fornire il numero di array che vuole usare e, per ognuno di essi, quanti elementi dovrà contenere. Se vengono forniti dei dati errati il programma continuerà a richiedere all'utente il numero degli array da utilizzare finché non gli verrà fornito un dato corretto. Notare che vengono salvate di volta in volta le dimensioni di ogni vettore inserito e la quantità di vettori usati in un vettore di appoggio chiamato `arrayDim`, inoltre il registro `$t7` conterrà la **quantità totale di elementi** inseriti.

L'etichetta `richiestaCarattere` oltre a permettere all'utente l'inserimento di un carattere alla volta, ha anche la funzione di "crocevia": cioè dirige l'esecuzione del programma alla corretta etichetta per il salvataggio del carattere appena inserito nel vettore appropriato attraverso l'uso del registro `$t1` e di varie istruzioni di `branch on equal`.

Le etichette `caricaArray` permettono di inserire un carattere alla volta all'interno degli array.

Una volta terminato il caricamento di tutti gli array desiderati bisogna scegliere che azione intraprendere; cioè se effettuare un'unione o una intersezione fra tutti gli array caricati. Questa azione viene decisa dall'utente tramite l'inserimento del carattere "0" per l'unione oppure del carattere "1" per l'intersezione.

Unione:

Se viene scelta l'operazione di unione, allora subito i registri `$t1` e `$t2` vengono inizializzati rispettivamente a 0 ed 1. Il primo registro serve per tenere conto di quale array ho già effettuato l'unione, mentre `$t2` è l'indice dell'array risultato "array5" ed infatti verrà inizializzato ad 1 perché ho già caricato la prima posizione dell'array con il carattere "[". Il registro `$t0` sarà l'indice per i 5 array caricati dall'utente: viene caricato con la dimensione di un array alla volta e quindi scorre gli elementi dall'ultimo al primo.

Il registro `$t7` conta quanti elementi vengono caricati nell'array risultato.

Di seguito viene spiegato come funziona nel dettaglio l'algoritmo dell'unione tra vettori:

Partendo dal primo array ne prelevo un elemento alla volta (all'interno delle varie etichette `unione0`, `unione1`, `unione2`, `unione3`, `unione4`) e lo confronto con ogni altro elemento dell'array risultato.

In base al risultato del confronto fra i due elementi (interno all'etichetta `checkExistence`) vengono intraprese diverse azioni:

- Se l'elemento attuale compare anche nell'array risultato allora non deve essere inserito: "azzerò" `$t2`, decremento `$t0` e decremento `$t7`, ora l'esecuzione salta all'etichetta `whereToGo` che "smista" l'esecuzione alla corretta etichetta di `unione`.
- Se l'elemento invece non compare allora viene inserito, il registro `$t2` "azzerato" e si procede a verificare l'elemento successivo

Una volta terminate queste operazioni, il vettore unione risultato (`array5`) è quasi pronto per essere stampato: bisogna inserire il carattere "]" in fondo al vettore. Questa operazione viene eseguita all'interno dell'etichetta `prepPrintUnione`.

```
prepPrintUnione:
    li $t5,0
    li $t0,93
    addi $t7,1
    sb $t0,array5($t7)
```

La codifica ASCII del carattere "]" è 93: lo inserisco nel registro `$t0` e poi nell'ultima posizione dell'array risultato indicata dal registro `$t7`.

L'etichetta `prepComune` viene utilizzata sia per l'intersezione che per l'unione: serve a prelevare il numero di array usati e la dimensione del primo.

```
prepComune:
    li $t4,7
    lb $t4,arrayDim($t4)    #prelevo il numero di array usati
    li $t3,0
    lb $t2,arrayDim($t3)    #prelevo la dimensione del primo array
    li $t0,0
    addi $t7,$t7,1
    j textVett0
```

Infine stampo il contenuto di ogni vettore usato.

Intersezione:

L' algoritmo per l'intersezione si basa sul principio che se un elemento dovrà appartenere al vettore-intersezione allora di sicuro apparterrà anche al più piccolo, in grandezza, dei vettori caricati (in caso più di un vettore hanno la stessa dimensione minima, viene usato il primo).

```
prepIntersezione:
    li $t0,9
    li $t1,0
    li $t2,8    #pos 8:grandezza del minore/pos 9:indice del minore
    move $t7,$t9
    li $t8,50
```

Inizialmente carico in `$t0` e `$t2` i valori 9 ed 8 che sono le posizioni di `arrayDim` che contengono l'indice dell'array più piccolo e la sua grandezza, mentre carico dentro `$t7` il numero di vettori che sono stati usati. Avrà la funzione di contatore e di indice di `arrayDim`.

`$t8` contiene la grandezza dell'attuale vettore più piccolo e viene inizializzato al massimo valore possibile.

Successivamente prelevo la grandezza dell'ultimo array caricato mettendola in `$t1` e la confronto con `$t8`:

- Se è minore, allora il nuovo minimo attuale diventa il valore di `$t1` (`move $t8, $t1`) e salvo i valori nelle posizioni 8 e 9 di `arrayDim`
- Altrimenti decremento `$t7`, prelevo la seguente grandezza degli array e rieseguo il confronto.

Al termine di queste operazioni prelevo la grandezza e l'indice del minore e le memorizzo in `$t1` e `$t2`; in base a quei due valori copio il contenuto dell'array più piccolo all' interno dell'array-intersezione risultato(`array5`).

Ora preparo i registri per la parte finale dell'operazione di intersezione.

```
prepCheck:    # RICORDA $t9 contiene il numero di vettori usati
    li $t0,1    #indice che scorre array5 fino a raggiungere $t1 (che contiene la grandezza del vettore5) RICORDA:posizione 0 occupata da [
    li $t2,0    #contiene il dato di cui si sta controllando l'esistenza negli altri array
    li $t3,0    #indice per i vari vettori
    li $t4,0    #contiene i dati dei vettori
    li $t5,-1    #tiene conto dei vettori controllati
```

L'algoritmo di intersezione consiste nel prendere un elemento alla volta dall' array risultato e controllare se è lecita la sua presenza nel vettore confrontandolo con ogni elemento di ogni altro vettore e, nel caso possa rimanerci, se sono presenti dei suoi duplicati (che vanno eliminati dal vettore).

L'eliminazione dei duplicati, dopo aver riscontrato che l'elemento proveniente dall' array-intersezione è uguale a quello proveniente dall' array "utente", consiste nel cercare nelle posizioni successive del primo array se è presente una copia del carattere in esame. In questo caso l'eliminazione della copia viene effettuata spostando a "sinistra" di una posizione tutti gli elementi successivi all' elemento duplicato e decrementando di 1 la grandezza del vettore finale ([array5](#)).

La stessa operazione viene effettuata anche se l'elemento in esame non può rimanere nel vettore finale perché non è presente in uno dei vettori "utente"; in questo caso mi servo del registro `$t8` come flag: nel caso contenga 0 alla fine del confronto tra il carattere in esame con tutti quelli di un vettore "utente" vuol dire che al suo interno non è presente e va quindi eliminato.

Simulazione:

Qui di seguito vengono riportati i risultati dell'esecuzione del programma fornendo in input i seguenti vettori: [1 2 -3 a s], [6 5 4 4 4 5 s e] e [p o i]

Unione:

```
Vett0: 123as
Vett1: 654445se
Vett2: poi
Vettore-Unione: [sa321e546iop]
Fine del programma!
```

Intersezione:

```
Vett0: 123as
Vett1: 654445se
Vett2: poi
Vettore-Intersezione: []
Fine del programma!
```

Fornendo invece i seguenti vettori: [o-a-i-c], [o-d-n-o-m] e [] (vettore vuoto)

Unione:

```
Vett0: oaic
Vett1: odnom
Vett2:
Vettore-Unione: [ciaomnd]
Fine del programma!
```

Intersezione:

```
Un vettore e' vuoto , dunque l' intersezione dei vettori è un vettore vuoto
Fine del programma!
```

Esercizio di progetto n. 2: Traduttore di Stringhe

Testo:

Utilizzando [QtSpim](#), scrivere e provare un programma che prenda in input una stringa qualsiasi di dimensione massima di 100 caratteri (es, "uno due ciao 33 tree tre uno Uno di eci"), e traduca ogni sua sottosequenza di caratteri separati da almeno uno spazio (nell'esempio, le sottosequenze "uno", "due", "ciao", "tree", "tre", "uno", "Uno", "di", "eci") applicando le seguenti regole:

- ogni sottosequenza "uno" si traduce nel carattere '1';
- ogni sottosequenza "due" si traduce nel carattere '2';
- ...
- ogni sottosequenza "nove" si traduce nel carattere '9';
- qualsiasi alta sottosequenza si traduce nel carattere '?'.

Nell'esempio considerato, se "uno due ciao 33 tree tre uno Uno di eci" è la stringa di input inserita da tastiera, a seguito della traduzione il programma dovrà stampare su console la seguente stringa di output:

Risultato della traduzione: 1 2 ? ? ? 3 1 ? ? ?

Implementazione:

```
##### Data segment #####
.data
titolo:      .asciiz "Esercizio 2 - Traduttore di stringhe\n"           # stringa per il titolo
input:       .asciiz "\nInserire un stringa qualsiasi (MAX 100 Caratteri) : " # stringa per l'input utente
erroreIns:   .asciiz "\nNon hai inserito nessun carattere!! \n"       # stringa per messaggio di errore
output:      .asciiz "\nRisultato della traduzione: "                  # stringa per messaggio dell'output del programma
uno:         .asciiz "uno"
due:         .asciiz "due"
tre:         .asciiz "tre"
quattro:     .asciiz "quattro"                                         # stringhe per i vari numeri
cinque:      .asciiz "cinque"
sei:         .asciiz "sei"
sette:       .asciiz "sette"
otto:        .asciiz "otto"
nove:        .asciiz "nove"
string:      .space 101        # spazio allocato per la stringa che viene inserita dall'utente
result:      .space 101        # spazio allocato per la stringa finale che viene resa in output a fine programma
array:       .space 101        # spazio allocato per l'array di appoggio, usato per i vari confronti con le stringhe dei numeri
##### Code segment #####
```

La prima parte del programma introduce i dati da caricare nel segmento dati:

- [titolo](#), è una stringa che verrà utilizzata per stampare nella console un titolo per il programma.
- [input](#), è una stringa che verrà utilizzata per stampare la richiesta di inserimento della stringa che si vuole tradurre.
- [erroreIns](#), è una stringa che verrà utilizzata per segnalare all'utente di aver inserito una stringa completamente vuota e quindi non valida per la traduzione.
- [output](#), è una stringa che verrà utilizzata per segnalare che la traduzione è stata completata.
- [uno](#), ..., [nove](#), sono stringhe che verranno utilizzate nei confronti per verificare se la stringa inserita in input è composta da alcune di esse, come richiesto dal testo dell'esercizio.
- [string](#), verrà utilizzato per identificare lo spazio allocato relativo alla stringa inserita in input dall'utente.
- [result](#), verrà utilizzato per identificare lo spazio allocato per la stringa finale tradotta correttamente come da richiesta.
- [array](#), verrà utilizzato come array di caratteri per poter fungere da appoggio nei confronti fra le sottosequenze della stringa inserita in input ([string](#)) e la varie stringhe che rappresentano i numeri da 1 a 9 ([uno](#), ..., [nove](#)).


```
.text
.globl main

main:
    la $a0, titolo      # stampa stringa titolo
    li $v0, 4
    syscall

stampaS:
    la $a0, input       # stampa stringa input
    li $v0, 4
    syscall

    la $a0, string      # legge la stringa inserita
    li $a1, 101
    li $v0, 8
    syscall
```

La seconda parte del programma contiene la parte relativa al codice, ovvero contiene tutte le varie istruzioni che il programma dovrà seguire per generare correttamente l'output richiesto.

Come prima cosa viene stampata la stringa [titolo](#), utilizzata per mostrare all'utente il titolo del programma, dopodiché viene stampata la stringa [input](#), utilizzata per richiedere all'utente di inserire la stringa da tradurre, e infine viene letta la stringa che l'utente ha appena inserito.

Successivamente viene controllata la lunghezza della stringa appena inserita dall'utente ([string](#)).

```
lunghezzaS:                # controlla la lunghezza della stringa
    li $t1, 0              # t1 conta la lunghezza
    la $t2, string         # t2 punta a un carattere della stringa

nextCh:                    # scorre i caratteri della stringa
    lbu $t0, ($t2)         # legge un carattere della stringa
    beqz $t0, controllo    # controlla il termine della stringa (0=null ASCII)
    addu $t1, $t1, 1       # incrementa il contatore
    addu $t2, $t2, 1       # incrementa la posizione sulla stringa
    j nextCh               # ciclo

controllo:                 # controlla se la stringa e' vuota
    addi $t1, $t1, -1      # viene tolto dal conteggio il carattere "invio"
    bnez $t1, init         # controlla il corretto inserimento della stringa

    la $a0, erroreIns      # stampa stringa erroreIns
    li $v0, 4
    syscall
    j stampaS
```

In questa parte viene, dunque, prima di tutto, il programma scorre ogni carattere della stringa inserita, utilizzando due registri (\$t1, \$t2) per contare la lunghezza e per salvare di volta in volta il carattere esaminato. Il contatore della lunghezza verrà chiaramente incrementato ogni volta che viene letto un carattere e quindi viene instaurato un ciclo che scorrerà tutta la stringa. Una volta che la stringa è stata completamente esaminata viene fatto un controllo finale su di essa. Viene decrementato il contatore della lunghezza della stringa di uno per togliere dal conteggio il carattere "invio", che viene comunque inserito nella stringa in input, e infine il programma controlla il valore della lunghezza è pari a zero. Se la lunghezza è pari a zero vorrà dire che l'utente ha inserito una stringa vuota e quindi verrà stampata un messaggio di errore ([erroreIns](#)) e successivamente verrà richiesto all'utente di reinserire la stringa, altrimenti il programma prosegue normalmente.

```
scorriS:                   # scorre la stringa inserita in input
    lb $s1, string($s0)    # s1 viene utilizzato per leggere, ad ogni ciclo, un carattere della stringa
    sb $s1, array($s2)     # carica il carattere appena letto nell'array di appoggio
    beq $s1, $s3, checkSpazio # controlla se il carattere letto e' uno spazio
    beq $s7, $s0, checkSpazio # controlla se il carattere letto e' l'ultimo della stringa di input
    addi $s2, $s2, 1       # incrementa l'indice dell' array di appoggio
    addi $s0, $s0, 1       # incrementa l'indice della stringa inserita in input
    blt $s0, 101, scorriS  # controlla se il ciclo e' arrivato a fine, ovvero se la stringa input e' stata completamente processata
    j stampaRes            # salta alle istruzioni per la stampa della stringa finale

checkSpazio:               # controlla se lo spazio inserito nell' array di appoggio e' isolato oppure si trova in fondo ad una sotto-stringa di "input"
    beqz $s2, salta        # controlla se l'indice dell' array di appoggio e' uguale a 0 (significa che è uno spazio isolato)
    addi $s2, $s2, -1      # decrementa di uno l'indice dell'array di appoggio per permettere di andare ad "raccolgere" -->
    # --- > solo la sotto-stringa priva di spazi, che verterà utilizzata per i confronti
```


Nella parte successiva del programma verranno inizializzati alcuni registri chiave per il corretto funzionamento del programma e poi verrà instaurato un ciclo che andrà a scansionare ogni carattere della stringa in input. Ogni carattere che viene letto verrà inserito nell'array di appoggio finché non si troverà il carattere “spazio” oppure fino a quando si raggiunge l'ultimo carattere della stringa.

In questo modo l'array di appoggio di volta in volta rappresenterà le varie sottosequenze della stringa in input e quindi sarà formato dai caratteri che le compongono.

Appena si verifica una di queste situazioni, quindi, il programma procederà a confrontare l'array di appoggio, che come abbiamo detto precedentemente rappresenterà la sottosequenza che si vuole analizzare, con le varie stringhe che rappresentano i numeri da 1 a 9 (uno, ..., nove). In particolare dato che l'ultimo carattere che di volta in volta verrà inserito nell'array di appoggio sarà sempre uno “spazio”, tranne nel caso in cui siamo arrivati alla fine della stringa, dovremo controllare se siamo in presenza di un array di appoggio formato solo dal carattere “spazio” oppure formato anche dalla sottosequenza che vogliamo prendere in esame.

```
checkUno:                # confronta la sotto-stringa presente nell'array di appoggio con la stringa "uno"
    la $a0, array         # carica i parametri da passare alla funzione "strcmp", utilizzata per confrontare due stringhe
    la $a1, uno
    jal strcmp            # chiama la funzione per il confronto tra stringhe
    beq $v0,$zero,uguali  # controlla il valore di ritorno della funzione "strcmp" (0-> le due stringhe sono uguali, 1-> s
    addi $s4, $s4, 1      # incrementa di uno il valore contenuto nel registro s4, (il valore contenuto e' interpretato in
```

Per fare i vari confronti tra l'array di appoggio e le varie stringhe che rappresentano i numeri viene chiamata di volta in volta una funzione “strcmp” che ha il compito di verificare se due stringhe sono uguali, e quindi viene utilizzata per verificare se l'array di appoggio è uguale o meno ad una delle stringhe numeriche (uno, ..., nove). La funzione ritorna il valore 0 se le due stringhe sono uguali, altrimenti ritorna il valore 1.

```
strcmp:                  # funzione per il confronto fra stringhe( ritorna 0 se sono uguali, 1 se sono diverse)
    add $t1,$zero,$a0    # t1 contiene l'indirizzo di base della prima stringa
    add $t2,$zero,$a1    # t1 contiene l'indirizzo di base della seconda stringa
    add $t5,$zero, $zero # inizializza il contatore per lo scorrimento della sotto-stringa contenuta nell'array di appoggio
    move $t6, $s2        # passa a t6 la dimensione della sotto-stringa da confrontare

loop:
    lb $t3,0($t1)         # carica in t3, ad ogni ciclo, un carattere della prima stringa
    lb $t4,0($t2)         # carica in t4, ad ogni ciclo, un carattere della seconda stringa
    beq $t5,$t6, check    # controlla se siamo arrivati all'ultimo carattere della prima stringa, nel caso passa all'etichetta "check"
    beqz $t4,notEqual     # controlla se siamo arrivati all'ultimo carattere della seconda stringa, nel caso passa all'etichetta "notEqual"
    bne $t3,$t4, notEqual # controlla se i due caratteri letti sono diversi, nel caso passa all'etichetta "notEqual"
    addi $t1,$t1,1        # incrementa di uno il valore del registro t1 in modo tale da passare l'analisi al prossimo carattere della prima stringa
    addi $t2,$t2,1        # incrementa di uno il valore del registro t2 in modo tale da passare l'analisi al prossimo carattere della seconda stringa
    addi $t5, $t5, 1      # incrementa di uno il valore del contatore per lo scorrimento della sotto-stringa contenuta nell'array di appoggio
    j loop

check:
    move $t7, $t2        # \
    addi $t7, $t7, 1     # inserisco nel registro t7 il carattere successivo a quello appena analizzato relativo alla seconda stringa
    lb $t7, 0($t7)       # /
    bnez $t7, notEqual   # controlla se la seconda stringa finira' al prossimo carattere, in tal caso continua con l'istruzione successiva
    beq $t3, $t4, equal  # controlla se i due caratteri letti sono uguali, nel caso passa all'etichetta "equal"
    j notEqual

equal:
    li $v0,0             # entra in questa etichetta solo se le due stringhe sono uguali
    jr $ra               # inserisce il valore 0 nel registro v0

notEqual:
    li $v0, 1            # entra in questa etichetta solo se le due stringhe sono diverse
    jr $ra               # inserisce il valore 1 nel registro v0
```

Successivamente, se l'array di appoggio coincide con una delle stringhe numeriche (uno, ..., nove) allora viene inserito nella stringa finale il valore numerico corrispondente alla stringa, e viene quindi chiaramente anche incrementato di uno il contatore per scorrere la stringa in input e infine ripreso il ciclo per scorrere tale stringa, se invece l'array di appoggio non coincide con nessuna stringa numerica (uno, ..., nove) allora viene inserito nella stringa finale il carattere “?” e anche in questo caso viene incrementato di uno il contatore per lo scorrimento della stringa di input e ripreso il ciclo che scorre la stringa.

```

diversoDaTutto:      # entra in questa etichetta solo se la sotto-stringa analizzata e' diversa da tutte le stringhe "numero" precedenti
    sb $s6, result($s5) # aggiunge il carattere "?" alla stringa finale
    bne $s7, $s0, addSpazio # controlla se deve aggiungere il carattere "spazio" alla stringa finale per separare le sottosequenze
    addi $s5, $s5, 1 # incrementa di uno l'indice della stringa finale
    addi $s0, $s0, 1 # incrementa di uno l'indice della stringa inserita in input
    j reset

salta:               # entra in questa etichetta solo se il carattere "spazio" che è stato inserito nell'array di appoggio e' isolato
    addi $s0, $s0, 1 # incrementa di uno l'indice della stringa inserita in input
    j reset

uguali:             # entra in questa etichetta solo se la sotto-stringa presente nell'array di appoggio e' uguale ad una delle stringhe "numero"
    sb $s4, result($s5) # aggiunge il numero corrispondente compreso tra[1-9] alla stringa finale
    bne $s7, $s0, addSpazio # controlla se deve aggiungere il carattere "spazio" alla stringa finale per separare le sottosequenze
    addi $s5, $s5, 1 # incrementa indice della stringa finale
    addi $s0, $s0, 1 # incrementa indice della stringa inserita in input
    j reset

addSpazio:          # aggiunge il carattere "spazio" alla stringa finale per separare le sottosequenze
    addi $s5, $s5, 1 # incrementa di uno l'indice della stringa finale
    sb $s3, result($s5) # inserisce il carattere "spazio" alla stringa finale
    addi $s5, $s5, 1 # incrementa di uno l'indice della stringa finale
    addi $s0, $s0, 1 # incrementa di uno l'indice della stringa inserita in input

```

Infine quando la stringa in input sarà stata completamente scansionata a questo punto il programma stamperà un stringa che avvisa l'utente che la traduzione è stata effettuata (la stringa **output**) e successivamente verrà stampata la stringa finale completamente tradotta. Il programma si ferma automaticamente una volta che viene raggiunta l'etichetta **“exit”**.

Non viene quindi fatto uso dello Stack Frame.

```

stampaRes:
    la $a0, output      # stampa il messaggio per la stringa di output
    li $v0, 4
    syscall

    la $a0, result      # stampa la stringa finale
    li $v0, 4
    syscall

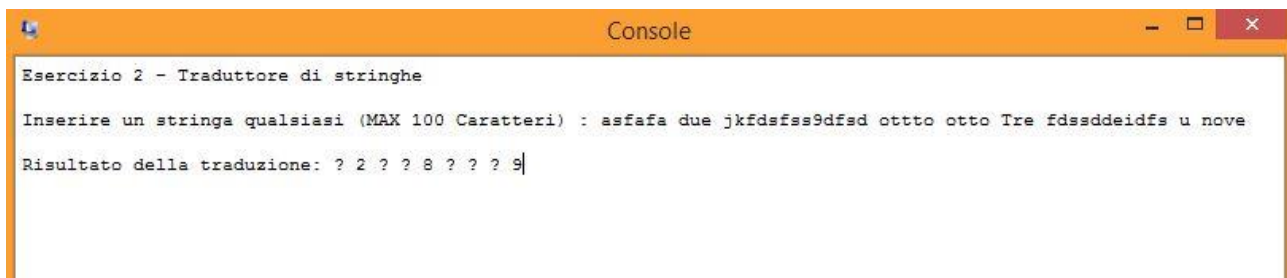
exit:
    li $v0, 10          # uscita dal programma
    syscall

```

Simulazione:

Di seguito viene riportato uno screenshot che mostra il programma in funzione nel caso in cui gli venga passata la seguente stringa in input:

“asfafa due jkfdsfss9dfsd ottto otto Tre fdssddeidfs u nove”



```

Esercizio 2 - Traduttore di stringhe

Inserire un stringa qualsiasi (MAX 100 Caratteri) : asfafa due jkfdsfss9dfsd ottto otto Tre fdssddeidfs u nove

Risultato della traduzione: 2 2 8 9

```

Esercizio di progetto n. 3: Conversione da Binario a Naturale

Testo:

Utilizzando [QtSpim](#), scrivere e provare un programma che legga inizialmente un vettore V di caratteri '0' (zero) o '1' (uno) di n elementi, con $1 \leq n \leq 10$ ('x' per terminarne l'inserimento).

Siano $V[0], \dots, V[n-1]$ gli n caratteri inseriti nel vettore V , e sia $m = n - 1$ (quindi $0 \leq m \leq 9$).

Il programma, interpretando la sequenza di caratteri '0' ed '1' in V come un numero binario puro in cui $V[0]$ è il bit più significativo e $V[m]$ è quello meno significativo, dovrà eseguirne la conversione in decimale implementando le seguenti procedure ricorsive (utilizzando quindi istruzioni jump and link):

$$Conv(V, m, k) = \begin{cases} V[k] & \text{if } k == m \\ Mul2(V[k], m - k) + Conv(V, m, k + 1) & \text{if } k < m \end{cases}$$

$$Mul2(a, b) = \begin{cases} 0 & \text{if } (a == 0) \\ 2 * a & \text{if } (a \neq 0) \text{ AND } (b == 1) \\ Mul2(2 * a, b - 1) & \text{if } (a \neq 0) \text{ AND } (b > 1) \end{cases}$$

Nella chiamata iniziale alla procedura ricorsiva $Conv$ dovrò settare $k = 0$, in quanto k è utilizzato nella procedura come indice per scorrere il vettore V dall'elemento con indice $k = 0$ (bit più significativo) fino all'elemento con indice $k = m$ (bit meno significativo).

Dopo la conversione, il programma dovrà stampare su console il risultato della conversione.

Ad esempio, sia $V = [0,1,1]$ (quindi $m = 2$), richiamando $Conv(V,2,0)$ si dovrà stampare su console:

Binario puro: [011]

Valore decimale corrispondente: 3

Mostrare e discutere l'evoluzione dello stack nel caso specifico in cui $V = [0,1,1]$

Implementazione:

La prima parte del programma introduce i dati da caricare nel segmento dati:

```
##### Data segment #####
.data
head:      .ascii "\nConvertitore di stringhe da binario in decimale."
insert:    .ascii "\nInserire il carattere: "
err:       .ascii "\nIl carattere inserito e' errato, si prega di inserirne un altro."
maxLimitReach: .ascii "\nHai inserito il numero massimo di elementi, procedo con il convertire il numero in decimale"
binary:    .ascii "\nBinario puro: ["
closeQ:    .ascii "]"
decimal:   .ascii "\nNumero decimale: "
arrayB:    .byte 0:10
```

- **head**, è una stringa che verrà utilizzata per stampare nella console il titolo del programma.
- **insert**, è una stringa che verrà utilizzata per stampare la richiesta di inserimento su console.
- **err**, è una stringa che verrà visualizzata per segnalare all'utente che il carattere inserito non è valido. Infatti, vengono accettati solamente i caratteri '0', '1' e 'x'.
- **maxLimitReach**, è una stringa che verrà visualizzata per segnalare all'utente che ha inserito il numero massimo di elementi consentiti.
- **binary**, è una stringa che verrà utilizzata per visualizzare correttamente il numero binario puro che l'utente ha inserito.
- **natural**, è una stringa che verrà utilizzata per visualizzare correttamente il numero decimale corrispondente al binario puro

- `arrayB`, verrà utilizzato come array di caratteri per poter contenere gli elementi inseriti in input dall'utente. Ovviamente conterrà solamente elementi '0' (zero) o '1' (uno)

La seconda parte del programma contiene la parte relativa al codice, ovvero contiene tutte le varie istruzioni che il programma dovrà seguire per generare correttamente l'output richiesto.

Come prima cosa viene stampata la stringa `head`, utilizzata per mostrare all'utente il titolo del programma e vengono inizializzati i registri `$s0`, `$s1` e `$s2`, che verranno utilizzati rispettivamente come contatore dei numeri inseriti fino a quel momento, come risultato finale della conversione e come indice per scorrere il vettore (`k`).

```
.text
.globl main

main:
    li $s0, -1          # s0 contiene il numero massimo di elementi meno uno
    li $s1, 0           # s1 viene utilizzato per caricare il risultato della conversione
    li $s2, 0           # s2 viene utilizzato per scorrere il vettore
    li $a1, 0           # a1 viene utilizzato per scorrere il vettore
    la $a0, head
    li $v0, 4
    syscall
```

Successivamente viene avviato un loop che servirà per chiedere di volta in volta all'utente quale carattere vuole inserire nel vettore e dopodiché viene fatto un controllo per verificare che il carattere sia accettabile. Ricordiamo, infatti, che gli unici caratteri accettati sono '0' (zero), '1' (uno) e 'x' per terminare l'inserimento. Se il carattere non è accettato viene stampato sulla console un messaggio di errore tramite la stringa “`err`” e le fase di input ricomincia, altrimenti si prosegue con l'inserimento del carattere nell'array e l'incremento del numero di elementi inseriti fino a quel momento.

Infine, viene effettuato un controllo sul numero di elementi inseriti fino a quel momento e sul numero massimo di elementi consentiti. Se sono uguali significa che abbiamo raggiunto il numero massimo di elementi e si procede alla fase di conversione, altrimenti la fase di input ricomincia.

```
input:
    beq $s0, 9, maxLimit # controlla se è possibile inserire ulteriori caratteri, nel caso passa all'etichetta "maxLimit"
    la $a0, insert       # stampa la stringa "insert"
    li $v0, 4
    syscall
    li $v0, 12           # legge il carattere in input
    syscall
    jal inputController  # controlla se il carattere inserito è accettabile
    addi $v0, $v0, -48    # essendo $v0 è un carattere, al suo interno è memorizzato il codice ascii di '0' e '1'.
    addi $s0, $s0, 1     # incremento il numero di elementi inseriti
    sb $v0, arrayB($s2)  # memorizzo il carattere all'interno di un vettore
    addi $s2, $s2, 1     # incremento l'indirizzo di base del vettore
    j input              # ripeto il ciclo input

inputController:
    beq $v0, 48, inputOk # se il contenuto di $v0 è uguale a 48 (=0) l'inserimento è corretto
    beq $v0, 49, inputOk # se il contenuto di $v0 è uguale a 49 (=1) l'inserimento è corretto
    beq $v0, 120, inputErr # se il contenuto di $v0 è uguale a 120 (=x) l'inserimento è corretto e l'input viene interrotto

inputErr:
    la $a0, err          # stampa la stringa "err"
    li $v0, 4
    syscall
    j input              # ritorna ad 'input'

inputOk:
    jr $ra               # se l'inserimento è OK, non ci sono problemi e si ritorna all'etichetta input

maxLimit:
    la $a0, maxLimitReach # stampa la stringa "maxLimitReach"
    li $v0, 4
    syscall

inputEnd:
    jal conversion       # viene chiamata la funzione conversion
    j printBinary        # salto all'etichetta printBinary
```

Come da richiesta, la fase di conversione del numero da binario puro a intero è stata implementata ricorsivamente. Per questo motivo abbiamo deciso di utilizzare lo Stack Frame, in modo da poter gestire in modo efficiente i vari contenuti del registro di ritorno `$ra`. Per prima cosa è necessario effettuare un push sullo stack, in modo da poter memorizzare il contenuto di `$ra` per un secondo momento.

Successivamente viene confrontato il registro `$a1` (`k`) con il numero degli elementi inseriti per controllare se siamo arrivati alla fine del vettore: in quel caso è sufficiente aggiungere `V[k]` al risultato finale della conversione e passare alla fase di output. Altrimenti, viene incrementato il valore contenuto in `$a1` e viene chiamata la funzione “`mul2`” che avrà il compito di calcolare il corretto valore in decimale del contenuto di `V[k]`.

Anche questa funzione, avendo struttura ricorsiva, ha bisogno prima di tutto un push sullo stack per salvare la variabile di ritorno `$ra`.

All'interno di “`mul2`” ci sono 2 casi base: il primo è verificato quando il contenuto di `V[k]` è uguale a zero, il secondo quando il contenuto di `V[k]` è diverso da zero e la differenza tra il numero massimo di elementi e l'indice `k` è uguale a 1. In entrambi casi il loop di “`mul2`” termina e il contenuto di `v[k]` viene restituito, tramite la variabile `$v1`, alla funzione “`conversion`”.

Ogniquale volta non si presenta un caso base, la funzione “`mul2`” richiama se stessa decrementando di uno la differenza tra il numero di elementi inseriti e l'indice `k` e raddoppiando il contenuto di `V[k]`. Da notare che, per quanto la differenza tra il numero degli elementi inseriti e `k` possa essere grande, prima o poi si presenterà il caso base 2. Una volta ritornati all'etichetta “`conversion`”, tramite un pop sullo stack per recuperare i vari registri di ritorno, il valore contenuto in `$v1` viene aggiunto alla conversione totale del numero. Quindi, la funzione “`conversion`” richiama se stessa fino a quando l'indice per scorrere il vettore non sarà uguale al numero di elementi inseriti, facendo terminare il ciclo di conversione.

```
conversion:
    addi $sp, $sp, -4          #push sullo stack per salvare le variabili
    sw $ra, 0($sp)            #salvo $ra per il ritorno
    lb $a3, arrayB($a1)        #salvo il contenuto di v[k] in $a3
    beq $a1, $s0, prereturn    #controllo se siamo arrivati all'ultimo elemento del vettore, nel caso passa all'etichetta "prereturn"
    sub $a2, $s0, $a1          #calcolo a2 sottraendo l'elemento corrente al numero massimo di elementi (m-k)
    addi $a1, $a1, 1           #incremento k
    jal mul2                   #chiamo la funzione mul2
    add $s1, $s1, $v1          #sommo al risultato il valore di ritorno della funzione mul2
    jal conversion             #chiamo la funzione conversion
    j return2                  #salto a return2

prereturn:
    add $s1, $s1, $a3          #aggiungo v[k] al totale

return2:
    lw $ra, 0($sp)            #riprendo il ritorno
    addi $sp, $sp, 4           #pop sullo stack per liberare le variabili
    jr $ra

mul2:
    addi $sp, $sp, -4          #funzione necessaria per calcolare il corretto valore in decimale di v[k]
    sw $ra, 0($sp)            #salvo $ra per il ritorno
    beq $a3, $zero, mul2base1  #controllo se il valore di v[k] è uguale a zero, nel caso passa all'etichetta "mul2base1"
    li $t0, 1
    beq $a2, $t0, mul2base2    #controllo se il valore di 'm-k' è uguale a uno, nel caso passa all'etichetta "mul2base2"
    add $a3, $a3, $a3          #calcolo il nuovo v[k] come 2*v[k]
    addi $a2, $a2, -1          #decremento il valore di 'm-k' di uno
    jal mul2                   #chiamo la funzione mul2
    j mul2End                  #salto all'etichetta mul2End

mul2base1:
    li $v1, 0                  #primo caso base, ci entra solo se v[k] = 0
    j mul2End                  #setta $v1 (il valore di ritorno) uguale a zero
    #salta all'etichetta mul2End

mul2base2:
    add $v1, $a3, $a3          #secondo caso base, ci entra solo se $a2 = 1
    #calcolo $v1 (il valore di ritorno) come 2*v[k]

mul2End:
    lw $ra, 0($sp)            #funzione necessaria per il ritorno
    addi $sp, $sp, 4           #riprendo il ritorno
    jr $ra                    #pop sullo stack per liberare le variabili
    #salto al valore/loce del registro $ra
```

Infine quando la stringa in input sarà stata completamente convertita in decimale il programma stamperà il numero in binario puro e il numero in decimale corrispondente. Il programma si ferma automaticamente una volta che viene raggiunta l'etichetta “`exit`”.

```

loopBinary:
    addi $s2, $s2, 1          #funzione necessaria per visualizzare i singoli elementi del vettore
    lb $a0, arrayB($s2)       # incrementa l'indirizzo di base del vettore
    li $v0, 1                 # stampa il contenuto di v[k]
    syscall
    bne $s2, $s0, loopBinary  # controllare se siamo arrivati all'ultimo elemento del vettore, nel caso contrario richiama se stessa
    jr $ra                   # salto al valore del registro $ra

printBinary:
    la $a0, binary           #funzione necessaria per la stampa del numero in binario puro
    li $v0, 4                # stampa la stringa "binary"
    syscall
    li $s2, -1               # setta il contenuto di $s2 a -1
    jal loopBinary           # chiama la funzione loopBinary
    la $a0, closeq           # stampa la stringa "closeq"
    li $v0, 4
    syscall

printConv:
    la $a0, decimal          #funzione necessaria per la stampa del numero naturale corrispondente al binario puro
    li $v0, 4                # stampa la stringa "decimal"
    syscall
    move $a0, $s1            # stampa il contenuto di $s1, ovvero il risultato dell'etichetta "conversion"
    li $v0, 1
    syscall

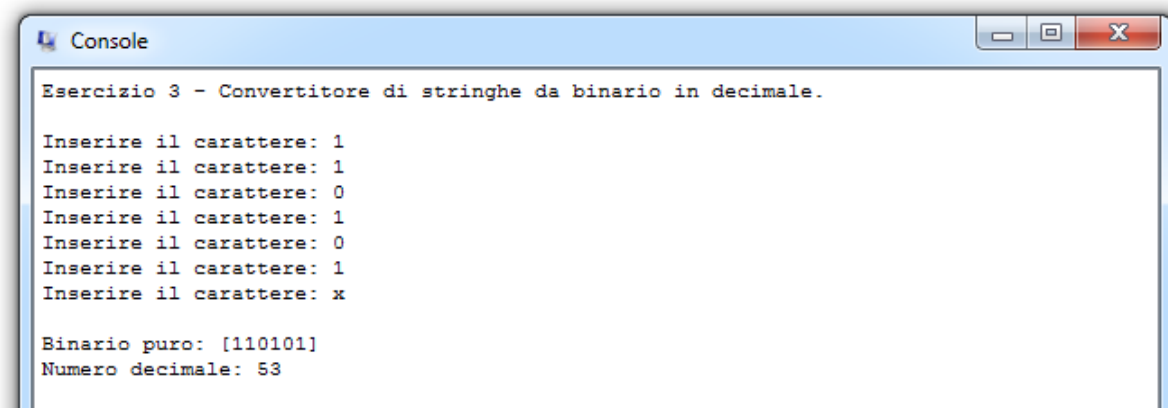
exit:
    li $v0, 10              #uscita dal programma
    syscall

```

Simulazione:

Di seguito viene riportato uno screenshot che mostra il programma in funzione nel caso in cui gli venga passata la seguente sequenza in input:

“1 1 0 1 0 1 x”



In particolare, vediamo come viene gestita lo Stack Frame quando viene ricevuto in input il vettore [0,1,1].

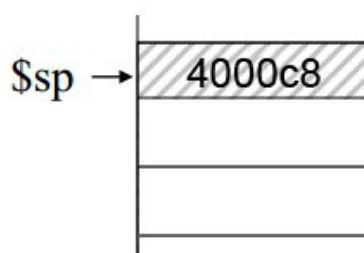
Non appena la fase di input termina, la funzione “[inputEnd](#)” chiama tramite un jump and link la funzione “[conversion](#)”, quindi crea sullo Stack lo spazio necessario per memorizzare il contenuto del registro [\\$ra](#)(=4000c8) e lo carica immediatamente.

```

addi $sp, $sp, -4          # push sullo stack per salvare le variabili
sw $ra, 0($sp)             # salvo $ra per il ritorno

```

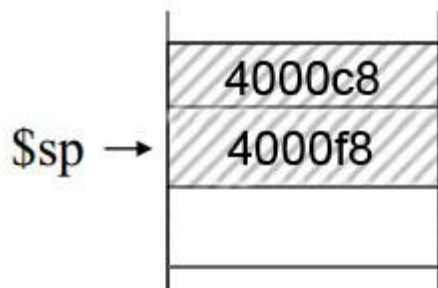
Il contenuto nello Stack Frame è,
al momento, così composto
(cresce verso il basso):



Successivamente confronta [\\$a1](#)(=0) con [\\$s0](#)(=2) e visto che sono diversi tra loro viene chiamata la funzione “[mul2](#)” tramite il comando jump and link. Anche qui, viene caricato il contenuto del registro

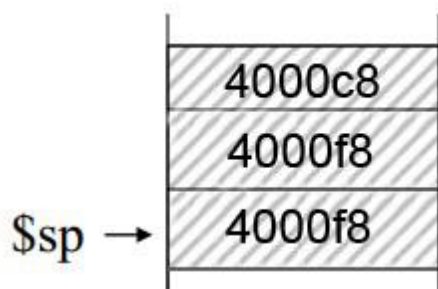
`$ra(=4000f0)` sullo Stack. Tuttavia, visto che `v[0]` è uguale a zero si presenta immediatamente un caso base, viene estratto l'ultimo elemento inserito nello Stack e si ritorna (tramite `jr $ra`) al loop “`conversion`”. Quindi, dopo aver incrementato la variabile `k` di uno, richiama se stesso tramite un jump and link.

Lo stack viene aggiornato aggiungendo il contenuto di `$ra(=4000f8)`.



Ancora una volta, essendo `$a1(=1)` diverso da `$s0(=2)` viene richiamato “`mul2`” con conseguente push sullo Stack, ma uscirà immediatamente (tornando al loop “`conversion`”) effettuando un pop in quanto la differenza tra il numero degli elementi e `k` è uguale a 1 (secondo caso base). Quindi, dopo aver incrementato la variabile `k` di uno, la funzione “`conversion`” richiama se stesso.

Lo stack viene aggiornato aggiungendo il contenuto di `$ra(=4000f8)`.



Questa volta `$a1` è uguale a `$s0`, quindi il ciclo di conversione è finito, il numero è stato trovato e sarà compito dell'etichetta “`return2`” effettuare dei pop sullo Stack in modo da ricavare l'indirizzo di partenza.

In primo luogo salta all'indirizzo `4000f8`, che ricordiamo essere:

```
j return2                                # salto a return2
```

Questo procedimento si ripete un'altra volta e successivamente si salta all'etichetta `4000c8`, che ricordiamo essere:

```
j printBinary                            # salto all'etichetta printBinary
```

Passando quindi alla fase di output.

Esercizio di progetto n. 4: Split di liste

Testo:

Utilizzando [QtSpim](#), scrivere e provare un programma che visualizzi all'utente un menu di scelta con le seguenti quattro opzioni:

- a) Creazione di una nuova lista di interi. Si crea dinamicamente una lista doppiamente concatenata di interi inseriti dall'utente (0 per terminare l'inserimento). Il nome della lista verrà assegnato automaticamente nel formato *List-i*.

Per esempio, se ho già creato fino a quel momento 2 liste (List-1 e List-2) e richiamo questa opzione inserendo gli interi 33 -44 55 123 12, dovrò creare una nuova lista chiamata List-3 che conterrà:

List-3: [33 -44 55 123 12]

In seguito si visualizzano su console tutte le liste create fino a quel momento con gli elementi che le compongono separati da spazi e si ritorna al menù (visualizzare su console la stringa "[Nil](#)" nel caso di lista vuota). Ad esempio, supponendo di aver già creato le liste List-1 e List-2 e List-3, dovrò visualizzare su console:

List-1: [44 -3 10 1 1 2]

List-2: [1 2 6 3]

List-3: [33 -44 55 123 12]

- b) Split due. Per ogni lista precedentemente creata (partendo da List-1, quindi List-2, List-3, ... etc.) si effettuano le seguenti operazioni:
 - i. Sia n il numero di elementi di cui è composta la lista, tale lista verrà modificata e dovrà contenere solo i primi $\lfloor n/2 \rfloor$ (ovvero $n \div 2$) valori interi.
 - ii. Gli interi rimossi dalla lista al punto i. precedente andranno a costituire gli elementi di una nuova lista il cui nome sarà assegnato automaticamente nel formato *List-i*.

Lo split non ha alcun effetto se applicato ad una lista vuota ("[Nil](#)"), ovvero ad una lista non contenente alcun valore intero.

Per esempio, se List-1: [33 -44 55 123 12] e List-2: [11 13] e List-3: Nil e List-4: [1] sono le liste esistenti fino a questo momento, eseguendo le operazioni ai punti i. e ii. si ottiene:

List-1: [33 -44] <-- lista già esistente ma modificata
List-5: [55 123 12] <-- nuova lista creata dallo split di List-1

List-2: [11] <-- lista già esistente ma modificata
List-6: [13] <-- nuova lista creata dallo split di List-2

List-3: Nil <-- lista già esistente ma non modificata in quanto vuota

List-4: Nil <-- lista già esistente ma modificata
List-7: [1] <-- nuova lista creata dallo split di List-4

In seguito si visualizzano su console tutte le liste create fino a quel momento con gli elementi che le compongono separati da spazi e si ritorna al menù (visualizzare su console la stringa “Nil” nel caso di lista vuota). Ad esempio, supponendo di aver già creato le liste List-1, List-2 ..., List-7 dovrò visualizzare su console:

```
List-1: [33 -44]
List-2: [11]
List-3: Nil
List-4: Nil
List-5: [55 123 12]
List-6: [13]
List-7: [1]
```

c) Split tre. Per ogni lista precedentemente creata (partendo da List-1, quindi List-2, List-3, ... etc.) si effettuano le seguenti operazioni:

- i. Sia n il numero di elementi di cui è composta la lista, tale lista verrà modificata e dovrà contenere solo i primi $x = \lfloor n/3 \rfloor$ (ovvero $n \div 3$) valori interi.
- ii. Gli interi rimossi dalla lista al punto i. precedente andranno a costituire gli elementi di due nuove liste i cui nomi saranno assegnati automaticamente nel formato List-i. Alla prima nuova lista saranno assegnati i primi x (calcolato al punto i.) valori interi degli elementi rimossi dalla lista al punto i., mentre alla seconda nuova lista saranno assegnati tutti gli interi rimanenti.

Lo split non ha alcun effetto se applicato ad una lista vuota ("Nil"), ovvero ad una lista non contenente alcun valore intero.

Per esempio, se List-1: [33 -44 55 123 12] e List-2: [11 13] e List-3: Nil e List-4: [1] sono le liste create fino a questo momento, eseguendo le operazioni ai punti i. e ii. si ottiene:

```
List-1: [33]          <-- lista già esistente ma modificata
List-5: [-44]         <-- prima nuova lista creata dallo split di List-1
List-6: [55 123 12]   <-- seconda nuova lista creata dallo split di List-1

List-2: Nil           <-- lista già esistente ma modificata
List-7: Nil           <-- prima nuova lista creata dallo split di List-2
List-8: [11 13]       <-- seconda nuova lista creata dallo split di List-2

List-3: Nil           <-- lista già esistente ma non modificata in quanto vuota

List-4: Nil           <-- lista già esistente ma modificata
List-9: Nil           <-- prima nuova lista creata dallo split di List-4
List-10: [1]          <-- seconda nuova lista creata dallo split di List-4
```

In seguito si visualizzano su console tutte le liste create fino a quel momento con gli elementi che le compongono separati da spazi e si ritorna al menù (visualizzare su console la stringa “Nil” nel caso di lista vuota). Ad esempio, supponendo di aver già creato le liste List-1, List-2, ..., List-10 dovrò visualizzare su console:

List-1: [33]
List-2: Nil
List-3: Nil
List-4: Nil
List-5: [-44]
List-6: [55 123 12]
List-7: Nil
List-8: [11 13]
List-9: Nil
List-10: [1]

d) Stampa un messaggio di uscita e esce dal programma.

Alle opzioni a), b), c) corrisponderanno le chiamate alle opportune procedure utilizzando l'istruzione di jump and link. Alla scelta d) corrisponderà l'uscita dal programma. Le liste dovranno essere allocate dinamicamente in memoria utilizzando la system call “sbrk” del MIPS.

Implementazione:

```
##### Data segment #####
.data
titolo:      .asciiz "Esercizio 4 - Split di Liste\n"
menu0:       .asciiz "\n-Menu-"
menu1:       .asciiz "\n1 - Creazione di una nuova lista di interi. "
menu2:       .asciiz "\n2 - Split due. "
menu3:       .asciiz "\n3 - Split tre. "
menu4:       .asciiz "\n4 - Esci dal programma.\n"
sceltaMenu:  .asciiz "\nInserisci un comando: "
sceltaErrata: .asciiz "\nHai inserito un comando non valido, inserirne un' altro.\n"
uscita:      .asciiz "\nUscita dal programma"
lista0:      .asciiz "\nList-"
lista1:      .asciiz ": "
lista2:      .asciiz "["
lista3:      .asciiz "]"
inserimento: .asciiz "\nInserisci un numero intero nella lista (0 per terminare): "
spazio:      .asciiz " "
nil:         .asciiz "Nil"
newLine:     .asciiz "\n"
zeroListe:   .asciiz "\nNon e' stata inserita nessuna lista, lo split richiesto non puo' essere eseguito\n"
listeCreate: .asciiz "\nListe create fino ad ora:"
splitEseguito: .asciiz "\nLo split richiesto e' stato eseguito:"

##### Code segment #####
```

La prima parte del programma introduce i dati da caricare nel segmento dati:

- **titolo**, è una stringa che verrà utilizzata per stampare all'utente un titolo per il programma.
- **menu0**, **menu1**, **menu2**, **menu3**, **menu4**, sono delle stringhe che verranno utilizzate per stampare il menu principale del programma.
- **sceltaMenu**, è una stringa che verrà utilizzata dal programma per segnalare all'utente la necessita di inserire un comando.
- **sceltaErrata**, è una stringa che verrà utilizzata per segnalare che il comando scelto dall'utente è errato.
- **uscita**, è una stringa che segnala l'uscita dal programma.
- **lista0**, **lista1**, **lista2**, **lista3**, sono delle stringhe che verranno utilizzate per stampare correttamente le varie liste che l'utente ha inserito.
- **inserimento**, verrà utilizzata per richiedere all'utente l'inserimento di un intero in una lista.
- **spazio**, verrà utilizzata per rappresentare il carattere “**spazio**”.
- **nil**, verrà utilizzata per rappresentare la stringa “**Nil**”.
- **newLine**, stringa utilizzata per andare a capo.
- **zeroListe**, viene utilizzata per segnalare all'utente che non è stata ancora inserita nessuna lista.
- **listeCreate**, viene utilizzata per segnalare all'utente la serie di liste create fino a quel momento.

- **splitEseguito**, viene utilizzata per segnalare all'utente che lo split scelto è stato eseguito correttamente.

La seconda parte del programma contiene la parte relativa al codice, ovvero contiene tutte le varie istruzioni che il programma dovrà seguire per generare correttamente l'output richiesto.

Come prima cosa viene stampata la stringa “**titolo**” e vengono inizializzati i registri **\$s0**, **\$s1** che verranno utilizzati, rispettivamente, come contatore delle liste create fino ad un dato momento e come contatore del numero di elementi inseriti in una lista.

```
main:
    li $s0, 0          # contatore delle liste create
    li $s1, 0          # contatore del numero di elementi inseriti in una lista

    la $a0, titolo     # stampa il titolo
    li $v0, 4
    syscall

menu:
    la $a0, menu0      # stampa il menu0
    li $v0, 4
    syscall

    la $a0, menu1      # stampa il menu1
    li $v0, 4
    syscall
```

Successivamente viene stampato il menu principale del programma e in seguito tramite la stringa “**sceltaMenu**” viene richiesto all'utente di inserire un comando per decidere che tipo di operazione dovrà eseguire il programma. Dopodiché viene fatto un controllo sul comando appena inserito dall'utente: se il comando è compreso fra 1 e 4 allora viene eseguita la richiesta corrispondente (1: creazione di una nuova lista, 2: split due, 3: split tre, 4: uscita dal programma) altrimenti viene stampato un messaggio di errore (stringa “**sceltaErrata**”), che segnala all'utente di aver inserito un comando errato invitandolo a reinserirne uno corretto, e in seguito viene ristampato il menu principale.

```
controlloMenu:
    beq $v0, 1, nuovaLista    # viene controllato se e' stata richiesta la creazione di una nuova lista
    beq $v0, 2, splitDue     # viene controllato se e' stato richiesto lo split due sulle liste create
    beq $v0, 3, splitTre     # viene controllato se e' stato richiesto lo split tre sulle liste create
    beq $v0, 4, exit         # viene controllato se e' stata richiesta l'uscita dal programma

    la $a0, sceltaErrata     # viene stampato un messaggio di errore che avverte l'utente
    li $v0, 4
    syscall

    j menu                  # torna a stampare il menu
```

Se il comando inserito dall'utente è il numero intero 1 allora il programma, come accennato precedentemente, si occuperà di gestire la creazione di una nuova lista. Viene quindi chiamata la funzione “**creazioneLista**” che ha il compito di gestire correttamente tale richiesta.

Per prima cosa vengono inizializzati a zero il registro **\$s1** e i registri **\$t8** e **\$t9**, che verranno utilizzati, rispettivamente, per salvare il riferimento alla testa e alla coda della lista che verrà creata.

```

creazioneLista:
    move $t8, $zero      # t8 viene utilizzato come riferimento alla Testa della lista e posto a 0
    move $t9, $zero      # t9 viene utilizzato come riferimento alla Coda della lista e posto a 0
    li $s1, 0            # s1 viene inizializzato a 0

```

Successivamente viene avviato il loop principale che servirà per chiedere di volta in volta all'utente quale numero intero vuole inserire nella lista (per terminare gli inserimenti l'utente dovrà inserire l'intero 0) e dopodiché viene allocato in un blocco di 12 byte il record che si vuole inserire nella lista, che sarà quindi formato da 3 campi: i 4 byte del primo campo verranno utilizzati per salvare il puntatore all'elemento precedente, i 4 byte del secondo campo verranno utilizzati per salvare il numero intero specificato dall'utente e i 4 byte dell'ultimo campo verranno utilizzati per salvare il puntatore all'elemento successivo. Viene quindi in seguito controllato se il record che si sta inserendo è il primo o meno della lista per poi andare ad aggiornare i registri `$t8` e `$t9` che contengono il riferimento alla testa e alla coda della lista che si sta creando e per eventualmente collegare l'ultimo record allocato a tale lista.

```

inputloop:                # inizio loop di input;
    li $v0, 4
    la $a0, inserimento
    syscall                # stampa messaggio di inserimento

    li $v0, 5
    syscall                # legge un intero inserito dall'utente

    beq $v0, $zero, finelista # se l'intero letto e' zero, procedera' a stampare tutte le liste inserite fino ad ora
    move $t1, $v0
    addi $s1, $s1, 1        # incremento il numero degli elementi inseriti

    li $v0, 9
    li $a0, 12
    syscall                # restituisce un blocco, puntato da v0 di 12 byte (4 byte per l'intero, 4 byte per il
                            # puntatore al precedente, 4 byte per il puntatore al successivo

    sw $zero, 0($v0)        # campo dedicato all'elemento-precedente = nil
    sw $t1, 4($v0)         # campo dedicato all'intero = t1
    sw $zero, 8($v0)        # campo dedicato all'elemento-successivo = nil

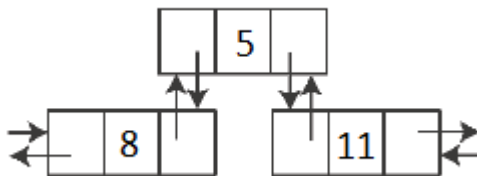
    bne $t8, $zero, linkLast # se t8!=nil (coda non vuota) vai a linkLast
    move $t8, $v0           # altrimenti (prima inserzione) Testa=v0
    move $t9, $v0           # altrimenti (prima inserzione) Coda=v0

    j inputloop            # torna all'inizio del loop di input

linkLast:
    sw $v0, 8($t9)         # se la coda e' non vuota, collega l'ultimo elemento della lista, puntato da Coda (t9) al nuovo record;
    sw $t9, 0($v0)         # dopodiche' modifica Coda per farlo puntare al nuovo record
    move $t9, $v0          # il campo elemento successivo dell'ultimo del record prende v0
                            # Coda = v0
    j inputloop            # salta all'inizio del loop

```

Qui di seguito viene riportato un esempio di lista che l'utente può creare:



Una volta completata la creazione della lista, per permettere al programma di memorizzarla, e per poi quindi andare a riesaminarla in seguito, viene fatto uso dello stack frame per poter salvare i principali riferimenti e le informazioni che saranno sufficienti a poter riottenere la lista creata. Viene dunque creato uno spazio di tre **words** nello stack frame per potervi salvare i riferimenti alla testa e alla coda della lista e inoltre il numero di elementi che la lista contiene.

Quando la creazione e la memorizzazione della lista viene terminata viene infine incrementato il contatore del numero delle liste inserite dall'utente e successivamente viene chiamata la funzione che gestisce la stampa di tutte le liste create fino a quel momento.

La funzione di stampa delle liste si basa su un loop principale che consiste prima di tutto nel calcolare l'indirizzo di base dello stack frame da cui prelevare i riferimenti relativi alla testa e alla coda della lista che deve essere stampata, dopodiché viene esaminato e quindi stampato ogni elemento della lista grazie ai puntatori all'elemento successivo di cui dispone ogni singolo record di una lista, facendo partire lo scorrimento dalla testa fino ad arrivare all'elemento che funge da coda, separando ogni numero intero con uno spazio e racchiudendo tutti gli interi stampati all'interno di una coppia di parentesi quadre.

Una volta che la stampa delle liste si è conclusa viene dunque ristampato il menu principale e viene richiesto all'utente un nuovo comando.

```
loopStampa:
    beq $t4, $zero, fineStampa    # se t4 = 0 si e' raggiunta la fine della lista e si esce

    li $v0, 1                     # altrimenti si stampa l'elemento corrente. Cioe':
    lw $a0, 4($t4)                # a0 = valore del campo intero dell'elemento corrente (puntato da t4)
    syscall                       # stampa valore intero dell'elemento corrente

    lw $t4, 8($t4)                # t4 = valore del campo elemento-successivo dell'elemento corrente (puntato da t4)
    beq $t4, $zero, loopStampa    # se t4 punta a 0 significa che l'elemento processato e' l'ultimo e quindi non viene
                                   # stampato lo spazio

    li $v0, 4
    la $a0, spazio
    syscall                       # stampa spazio

    j loopStampa                  # salta all'inizio del ciclo di stampa
```

Se il comando inserito dall'utente è il numero intero 2 allora il programma, come accennato precedentemente, si occuperà di gestire lo split due di tutte liste create fino a quel momento. Viene quindi chiamata la funzione “[splitDueListe](#)” che ha il compito di gestire correttamente tale richiesta.

Per prima cosa vengono inizializzati i registri [\\$t0](#) e [\\$t1](#) che verranno utilizzati rispettivamente per sapere il numero di liste rimanenti su cui eseguire lo split e per calcolare correttamente la posizione nello stack da cui prelevare i riferimenti alla testa e la coda delle varie liste. Successivamente viene avviato il loop principale della funzione che consiste prima di tutto nel calcolare la posizione corretta dello stack da cui prelevare le informazioni necessarie per lo split: la testa della lista da processare che sarà salvata nel registro [\\$t8](#), la coda che sarà salvata nel registro [\\$t7](#) e il numero di elementi presenti che sarà salvato nel registro [\\$t3](#). Viene poi eseguita una divisione fra il numero di elementi presenti nella lista e il numero 2 e il quoziente di tale divisione andrà a determinare da quale posizione far partire lo split e quindi da che posizione creare una nuova lista. Dopodiché, partendo dalla testa, la lista da processare viene fatta scorrere fino alla posizione calcolata precedentemente.

```
scorrimentoLista1:
    beq $t6, $t4, creaSplitDue    # se il contatore t6 e' uguale al numero di elementi che non dovranno essere modificati
    lw $t5, 8($t5)                # t5 = valore del campo elemento-successivo dell'elemento corrente (puntato da t5)
    addi $t6, $t6, 1              # incremento il contatore t6 per sapere quanti elementi ho scansionato
    j scorrimentoLista1

creaSplitDue:
    move $t9, $t5                 # salvo il valore di t5 in t9
    lw $t5, 0($t5)                # metto in t5 il puntatore all'elemento precedente della lista

    bne $t5, $zero, notNil2       # se t5=0 allora la lista dovra' essere modificata in "nil", altrimenti salta a "notNil2"
    sw $zero, -12($t2)            # setto il num. di elementi a 0
    sw $zero, -8($t2)             # setto la coda a 0
    sw $zero, -4($t2)             # setto la testa a 0
    j nuovaListaSplit2

notNil2:
    sw $t5, -8($t2)               # cambio la coda della lista esaminata con il puntatore in t5
    sw $t4, -12($t2)              # setto il num. di elementi che saranno presenti nella lista dopo la modifica
    sw $zero, 8($t5)              # cambio l'elemento successivo di t5 con 0
```

Se la lista esaminata è “**Nil**” chiaramente non verrà fatta alcuna modifica e quindi si passerà a processare la lista successiva, altrimenti, i primi **n** elementi (con **n** pari al quoziente della divisione fra il numero di elementi e il numero 2) faranno sempre parte della lista originale ma il riferimento alla coda della lista verrà aggiornato con il riferimento del record in posizione **n**, mentre tutti gli elementi rimanenti dopo quella posizione andranno a formare una nuova lista, che avrà come testa il record che sarà presente nella posizione successiva a **n** e come coda la stessa della lista originale (prima che iniziasse lo split).

```
nuovaListaSplit2:
    sw $zero, 0($t9)          # pongo il puntatore al precedente del primo elemento della nuova lista a 0
    addi $sp,$sp, -12         # crea spazio per tre words nello stack frame
    sw $t9, 8($sp)            # salvo la testa, che sarà t9, della nuova lista appena creata nello stack frame
    sw $t7, 4($sp)            # salvo la coda della lista appena creata nello stack frame
    sub $t7, $t3, $t6         # uso t7 come appoggio per conoscere il num. di elementi della nuova lista(t3-t6)
    sw $t7, 0($sp)            # salvo il numero di elementi della lista appena creata nello stack frame
    addi $s0, $s0, 1          # aumento di uno il contatore delle liste create
    j fineSplitDue

splitDueNil:
    addi $t1, $t1, -1         # decremento di 1 t1 per calcolare correttamente la posizione nello stack da cui prelevare le info
                                # in questo caso infatti non viene creata nessuna nuova lista

fineSplitDue:
    addi $t0, $t0, -1         # decremento di 1 il numero delle liste da splittare
    bne $t0, $zero, loopSplitDue # se ci sono ancora liste da splittare torna a "loopSplitDue"
    jr $ra                   # altrimenti lo split due di tutte le liste è concluso
```

Una volta che lo split viene eseguito verrà stampato all’utente una stringa (**splitEseguito**) per avvisarlo che l’operazione è andata a buon fine e successivamente viene richiamata la funzione per la stampa delle liste per poter mostrare all’utente il risultato dello split appena eseguito. Infine, una volta che la stampa delle liste si è conclusa, viene dunque ristampato il menu principale e viene richiesto all’utente un nuovo comando.

Se il comando inserito dall’utente è il numero intero 3 allora il programma, come accennato precedentemente, si occuperà di gestire lo split tre di tutte le liste create fino a quel momento. Viene quindi chiamata la funzione “**splitTreListe**” che ha il compito di gestire correttamente tale richiesta. Anche qui, come nel caso dello split due vengono utilizzati i registri \$t0 e \$t1 per sapere il numero di liste rimanenti su cui eseguire lo split e per calcolare correttamente la posizione nello stack da cui prelevare la testa, la coda e il numero di elementi delle varie liste.

Come nel caso dello split due il loop principale della funzione consiste prima di tutto nel calcolare la posizione corretta dello stack da cui prelevare le informazioni necessarie per lo split: la testa della lista da processare che sarà salvata nel registro **\$t5**, la coda che sarà salvata nel registro **\$t7** e il numero di elementi presenti che sarà salvato nel registro **\$t3**. Viene poi eseguita una divisione fra il numero di elementi presenti nella lista e il numero 3 e il quoziente di tale divisione andrà a determinare da quale posizione far partire lo split e quindi da che posizione creare due nuove liste. Dopodiché, partendo dalla testa, la lista da processare viene fatta scorrere fino alla posizione calcolata precedentemente.


```

scorrimentoLista2:
    beq $t6, $t4, creaSplitTre    # se il contatore t6 e' uguale al numero di elementi che non dovranno essere modificati
    lw $t5, 8($t5)                # t5 = valore del campo elemento-successivo dell'elemento corrente (puntato da t4)
    addi $t6, $t6, 1              # incremento il contatore t6 per sapere quanti elementi ho scansionato
    j scorrimentoLista2

creaSplitTre:
    li $t6, 2                      # uso t6 per caricare il valore 2
    addi $t8, $t8, 1              # incremento t8 per sapere se la prossima mossa da fare e' creare una nuova lista
    beq $t8, $t6, creaNuoveListeSplit3 # se t8 e' maggiore di 1 allora crea nuove liste
    move $t9, $t5                 # salvo il puntatore all'elemento della lista in t9
    lw $t5, 0($t5)                # metto in t5 il puntatore all'elemento precedente della lista

    bne $t5, $zero, notNil3       # se t5=0 allora la lista dovra' essere modificata in "nil", altrimenti salta a "notNil3"
    sw $zero, -12($t2)            # setto il num. di elementi a 0
    sw $zero, -8($t2)             # setto la coda a 0
    sw $zero, -4($t2)             # setto la testa a 0
    j nuovaListaSplit3

notNil3:
    sw $t5, -8($t2)               # cambio la coda della lista esaminata con il puntatore in t5
    sw $t4, -12($t2)              # setto il num. di elementi che saranno presenti nella lista dopo la modifica
    sw $zero, 8($t5)              # cambio l'elemento successivo di t5 con 0

nuovaListaSplit3:
    li $t6, 0                      # uso t6 come contatore per scorre la lista, per sapere cosa eliminare
    move $t5, $t9                 # salvo il puntatore all'elemento della lista in t5
    j scorrimentoLista2

```

Se la lista esaminata è “**Nil**” chiaramente non verrà fatta alcuna modifica e quindi si passerà a processare la lista successiva, altrimenti, i primi **n** elementi (con **n** pari al quoziente della divisione fra il numero di elementi e il numero 3) faranno sempre parte della lista originale ma il riferimento alla coda della lista verrà aggiornato con il riferimento del record in posizione **n**, mentre i successivi **n** elementi rimanenti dopo quella posizione andranno a formare una nuova lista che avrà come testa il record che sarà presente nella posizione successiva a **n** e come coda il record che sarà in posizione **2*n**, ed infine verrà creata un ultima lista che avrà come testa il record presente nella posizione successiva a **2*n** e come coda la stessa della lista originale (prima che iniziasse lo split).

```

nuovaListaSplit3:
    li $t6, 0                      # uso t6 come contatore per scorre la lista, per sapere cosa eliminare
    move $t5, $t9                 # salvo il puntatore all'elemento della lista in t5
    j scorrimentoLista2

creaNuoveListeSplit3:
    sw $zero, 0($t9)               # pongo il puntatore al precedente del primo elemento della nuova lista a 0
    addi $sp, $sp, -12             # crea spazio per tre words nello stack frame
    lw $t8, 0($t5)                # uso t8 per salvare l'elemento precedente a quello puntato da t5 (che sara' la coda della nuova lista)
    bne $t8, $zero, notNewNil3    # se t8=0 allora la lista dovra' essere modificata in "nil", altrimenti salta a "notNewNil3"

    sw $zero, 0($sp)              # setto il num. di elementi a 0
    sw $zero, 4($sp)              # setto la coda a 0
    sw $zero, 8($sp)              # setto la testa a 0
    addi $s0, $s0, 1              # aumento di uno il contatore delle liste create
    j prossimalista

notNewNil3:
    sw $t9, 8($sp)                # salvo la testa, che sara' t9, della nuova lista appena creata nello stack frame
    sw $zero, 8($t8)              # rendo il puntatore-succesivo della coda nullo=0
    sw $t8, 4($sp)                # salvo la coda della lista appena creata nello stack frame
    sw $t4, 0($sp)                # salvo il numero di elementi della lista appena creata nello stack frame
    addi $s0, $s0, 1              # aumento di uno il contatore delle liste create

```

Una volta che lo split viene eseguito verrà stampato all’utente una stringa ([splitEseguito](#)) per avvisarlo che l’operazione è andata a buon fine e successivamente viene richiamata la funzione per la stampa delle liste per poter mostrare all’utente il risultato dello split appena eseguito. Infine, una volta che la stampa delle liste si è conclusa, viene dunque ristampato il menu principale e viene richiesto all’utente un nuovo comando.

```

exit:
    mul $t0, $s0, 12              # utilizzo t0 per sapere quanto spazio deve essere liberato
    add $sp, $sp, $t0             # libera lo spazio occupato dello stack

    la $a0, uscita
    li $v0, 4
    syscall

    li $v0, 10
    syscall

```

Se il comando inserito dall'utente è il numero intero 4 allora il programma, come accennato precedentemente, prima di tutto libererà nello stack frame lo spazio occupato da tutta la serie di liste create nel corso della sue esecuzione e successivamente provvederà a terminarsi, avvisando l'utente con la stampa a video della stringa "uscita".

Simulazione:

Di seguito viene riportato uno screenshot che mostra il programma in funzione nel caso in cui l'utente voglia inserire una nuova lista: [34 -12 3 77 -6]



```
Esercizio 4 - Split di Liste

-Menu-
1 - Creazione di una nuova lista di interi.
2 - Split due.
3 - Split tre.
4 - Esci dal programma.

Inserisci un comando: 1

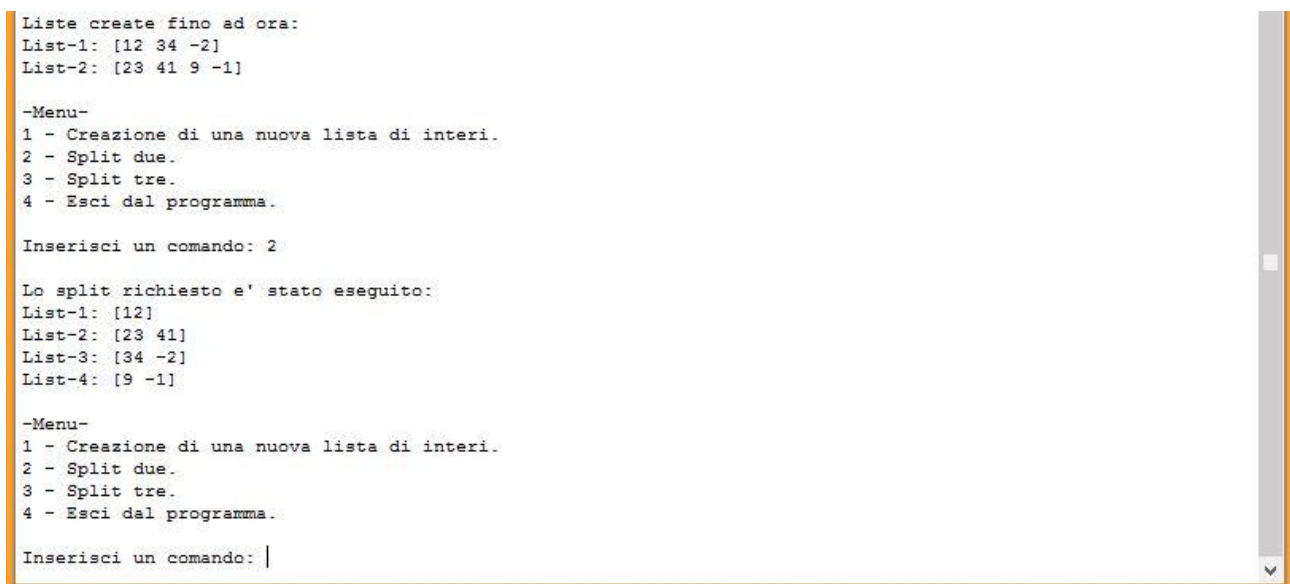
Inserisci un numero intero nella lista (0 per terminare): 34
Inserisci un numero intero nella lista (0 per terminare): -12
Inserisci un numero intero nella lista (0 per terminare): 3
Inserisci un numero intero nella lista (0 per terminare): 77
Inserisci un numero intero nella lista (0 per terminare): -6
Inserisci un numero intero nella lista (0 per terminare): 0

Liste create fino ad ora:
List-1: [34 -12 3 77 -6]

-Menu-
1 - Creazione di una nuova lista di interi.
2 - Split due.
3 - Split tre.
4 - Esci dal programma.

Inserisci un comando: |
```

Il seguente screenshot invece mostra il programma in funzione nel caso in cui l'utente voglia eseguire lo split due di due liste: [12 34 -2] e [23 41 9 -1]



```
Liste create fino ad ora:
List-1: [12 34 -2]
List-2: [23 41 9 -1]

-Menu-
1 - Creazione di una nuova lista di interi.
2 - Split due.
3 - Split tre.
4 - Esci dal programma.

Inserisci un comando: 2

Lo split richiesto e' stato eseguito:
List-1: [12]
List-2: [23 41]
List-3: [34 -2]
List-4: [9 -1]

-Menu-
1 - Creazione di una nuova lista di interi.
2 - Split due.
3 - Split tre.
4 - Esci dal programma.

Inserisci un comando: |
```

Il seguente screenshot invece mostra il programma in funzione nel caso in cui l'utente voglia eseguire lo split tre di due liste: [12 34 -2] e [23 41 9 -1]

```
Liste create fino ad ora:
List-1: [12 34 -2]
List-2: [23 41 9 -1]

-Menu-
1 - Creazione di una nuova lista di interi.
2 - Split due.
3 - Split tre.
4 - Esci dal programma.

Inserisci un comando: 3

Lo split richiesto e' stato eseguito:
List-1: [12]
List-2: [23]
List-3: [34]
List-4: [-2]
List-5: [41]
List-6: [9 -1]

-Menu-
1 - Creazione di una nuova lista di interi.
2 - Split due.
3 - Split tre.
4 - Esci dal programma.

Inserisci un comando: |
```

User Stack:

Lo stack frame viene utilizzato dal programma per salvare i riferimenti di testa e coda della varie liste che l'utente e il programma possono creare a seconda degli algoritmi eseguiti. Ogni qual volta che viene creata una nuova lista viene infatti allocato uno spazio di 12 byte nello stack frame, che quindi crescerà verticalmente di 12 byte ad ogni nuova lista. I primi 4 byte allocati serviranno per salvare il riferimento alla testa della lista, i successivi 4 byte saranno utilizzati per il riferimento alla coda della lista mentre gli ultimi 4 byte verranno utilizzati per salvare il numero di elementi che la lista contiene (il numero di interi che contiene).

