Relazione sul progetto dell'esame di Architettura degli elaboratori anno accademico 2018-2019

Duccio Serafini – 5790789 – *duccio.serafini@stud.unifi.it*
Andre Cristhian Barreto Donayre – 5364116 – *andre.barreto@stud.unifi.it*

Data di consegna: 29/05/19

Indice:

1. Messaggi Cifrati

Utilizzando QtSpim, scrivere e provare un programma in assembly MIPS che simuli la funzionalità di cifratura e decifratura di un messaggio di testo. In particolare il programma dovrà consentire di:

- Cifrare un messaggio di testo con una combinazione qualsiasi dei seguenti cifrari (algoritmi):

  - **Algoritmo A**: il codice ASCII standard su 8 bit di ciascun carattere del messaggio di testo viene modificato sommandoci una costante decimale K=4, modulo 256. Ovvero, se cod(X)è la codifica ascii standard decimale di un carattere X del messaggio, la cifratura di Xcorrisponderà a: (cod(X)+K) mod 256.

  - **Algoritmo B**: si applica l'Algoritmo A a tutti i caratteri del messaggio di testo che sono in posizione di indice pari (il primo carattere del messaggio avrà indice 0, quindi pari).

  - **Algoritmo C**: si applica l'Algoritmo A a tutti i caratteri del messaggio di testo che sono in posizione di indice dispari (il primo carattere del messaggio avrà indice 0, quindi pari).

  - **Algoritmo D**: il messaggio viene cifrato invertendo l'ordine dei caratteri del messaggio di testo. Ad esempio, se "ciao, prova a cifrarmi" è il contenuto del messaggio di testo da cifrare, allora la cifratura con l'Algoritmo D produrrà come risultato: "imrarfic a avorp ,oaic".

  - **Algoritmo E**: a partire dal primo carattere del messaggio (quello alla posizione 0), il messaggio viene cifrato come una sequenza di stringhe separate da esattamente 1 spazio (codice ascii decimale 32) in cui ciascuna stringa ha la forma "x p1 ... pk", dove x è la prima occorrenza di ciascun carattere presente nel messaggio, p1...pk sono le posizioni in cui il carattere x appare nel messaggio (con p1<...<pk), ed in cui ciascuna posizione è preceduta dal carattere separatore ' ' (per distinguere gli elementi della sequenza delle posizioni). Ad esempio, se "esempio di messaggio criptato 1" è il contenuto del messaggio di testo da cifrare, allora la cifratura con l'Algoritmo E produrrà come risultato: "e 0 2 12 s 1 13 14 m 3 11 p 4 24 i 5 9 18 23 o 6 19 28 7 10 20 29 d 8 a 15 26 g 16 17 c 21 r 22 t 25 27 301 31".

    Note sull'esempio:
    - Nella stringa " 7 10 20 29" il carattere in codifica è lo spazio (' ', codice ascii decimale 32), che appare nelle posizioni 7, 10, 20 e 29 del messaggio.
    - Nella stringa " 30" il carattere in codifica è ' ' (il secondo carattere ' ' è ilcarattere separatore fra gli elementi della sequenza), che appare nella posizione 30 del messaggio.
    - Nella stringa "1 31" il carattere in codifica è '1', che appare nella posizione 31 del messaggio

    Il messaggio di testo verrà cifrato utilizzando una parola chiave che definisce la sequenza delle cifrature da applicare al messaggio. La parola chiave è una stringa S="S1···Sk" formata da al massimo 4 caratteri (quindi con $1 \le k \le 4$), in cui ciascun carattere Si (con $1 \le i \le k$) corrisponde ad uno fra i caratteri 'A', 'B', 'C', 'D' oppure 'E', ed identifica il corrispondente cifrario da applicare al messaggio al passo i mo.
    L'ordine delle cifrature è quindi stabilito dall'ordine in cui appaiono i caratteri nella stringa. A titolo di esempio, si riportano alcuni possibili parole chiave: "C", oppure "AEC", oppure "DEDD", ... Ad esempio, la cifratura del messaggio di testo con la parola chiave "AEC" determinerà l'applicazione dell'algoritmo A, poi dell'algoritmo E (sul messaggio già cifrato con A) ed infine dell'algoritmo C (sul messaggio già cifrato prima con A e poi con E).

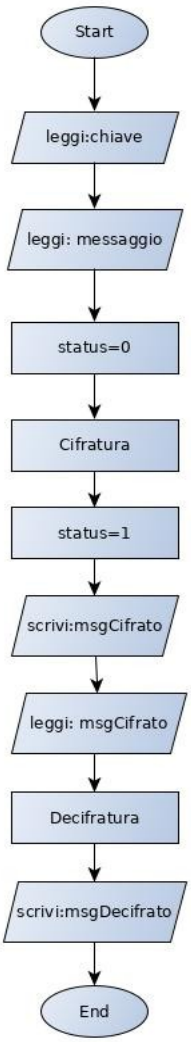
- Decifrare il messaggio di testo cifrato utilizzando la parola chiave invertita Ŝ= “Sk···S1”, ovvero invertendo gli Algoritmi di Cifratura e richiamandoli in ordine inverso (dall’ultimo algoritmo applicato al primo, ovvero da Sk a S1).

Si assuma che il messaggio di testo da cifrare sia disponibile in un file di testo chiamato “messaggio.txt”, composto da un massimo di 128 caratteri, e che la parola chiaveS= “S1···Sk“ con cui cifrare il messaggio sia disponibile in un file di testo chiamato “chiave.txt”. Si supponga inoltre che la chiave nel file “chiave.txt” sia sintatticamente corretta (composta solo dai caratteri ‘A’, ‘B’, ‘C’, ‘D’ oppure ‘E’) e che sia di lunghezza $1 \leq k \leq 4$.

- Il programma dovrà leggere in input il messaggio da cifrare e la parola chiave, e produrre in output: un file di testo chiamato “messaggioCifrato.txt”, contenente il messaggio cifrato utilizzando la parola chiave S=”S1···Sk”;

- Un file di testo chiamato “messaggioDecifrato.txt”, contenente il messaggio decifrato a partire dal messaggio precedentemente cifrato utilizzando la parola chiave invertita Ŝ= “Sk···S1“ e gli Algoritmi di Cifratura invertiti. Nota: il messaggio decifrato correttamente dovrebbe ovviamente corrispondere al messaggio originale contenuto nel file “messaggio.txt”.

2. Descrizione generale



L’obiettivo dell’elaborato è quello di simulare degli algoritmi di **cifratura** e **decifratura** di un **messaggio di testo** attraverso una **chiave** che definisce la sequenza delle cifrature da applicare al messaggio. Questa verrà successivamente usata anche in fase di decifratura, letta però in senso inverso.
La **parola chiave** è una stringa formata da al massimo 4 caratteri, in cui ciascun carattere corrisponde ad uno fra i caratteri ‘**A**’, ‘**B**’, ‘**C**’, ‘**D**’ oppure ‘**E**’, ed identifica il corrispondente **cifrario** da applicare al messaggio al passo *i* mo.
Dato le specifiche dell'esercizio abbiamo deciso di suddividere il programma in varie sezioni, dedicate alla gestione dei tempi di esecuzione degli algoritmi di cifratura.
Ogni algoritmo verrà infatti chiamato solo dopo lo svolgimento delle procedure dedicate al settaggio di stati necessari al corretto funzionamento della cifratura, come verrà spiegato più avanti.
Durante lo sviluppo ci siamo preoccupati di mantenere il codice modulare suddividendolo in procedure il più possibile atomiche, in modo da rendere il codice: pulito, leggibile e facile da mantenere.

3. Strutture dati

- **opCifra**: contiene la stringa “Cifratura in corso...”
- **opDecif**: contiene la stringa “Decifratura in corso...”
- **done**: contiene la stringa “Operazione Terminata.”
- **messaggio**: stringa contenente il percorso del file in cui si trova il messaggio da cifrare
- **chiave**: stringa contenente il percorso del file in cui si trova la chiave di cifratura
- **msgCifrato**: stringa contenente il percorso del file in cui viene scritto il messaggio cifrato
- **msgDecifrato**: contiene il percorso del file in cui viene scritto il messaggio decifrato
- **algorithmJAT**: spazio utilizzato per la tabella dei salti agli algoritmi
- **statusABC**: spazio utilizzato contenere valori utili agli algoritmi A, B e C
- **occurrenceBuffer**: spazio utilizzato dall’algoritmo E per tenere traccia delle occorrenze
- **supportBuffer**: spazio usato come supporto per gli algoritmi D ed E
- **bufferReader**: spazio usato per contenere il messaggio da cifrare
- **bufferKey**: spazio utilizzato per contenere la chiave di cifratura

4. Main

Il programma principale inizia dal label **main** ed ha la responsabilità di eseguire le fasi di cifratura e decifratura, chiamando le apposite **procedure.** Questa sezione di codice verrà quindi chiamata Cifratore.
In questa prima fase esso avrà il compito di inizializzare la tabella dei salti dedicata agli Algoritmi di cifratura, una **JAT** (Jump Address Table), che è il corrispettivo dei linguaggi ad alto livello di uno **switch case** e visto che successivamente nel corso della sua computazione verrà fatto riferimento più di volta agli algoritmi, in questo modo riusciamo ad ottimizzare le sue prestazioni. La tabella verrà riempita con gli indirizzi di riferimento degli algoritmi, cosicché sia possibile saltare direttamente all'algoritmo richiesto rendendo generica la procedura.
Verranno poi letti i file di testo contenenti la chiave di cifratura e il messaggio da cifrare per essere caricati negli appositi buffer.
Una volta fatto ciò il programma chiamerà il metodo **core.**
Nella fase di decifratura, il Cifratore leggerà il file da decifrare. La chiave, ottenuta in fase di cifratura, sarà letta in senso inverso grazie ad un **flag di stato** impostato in $s7, che assume il
valore 0 per la fase di cifratura e viene aggiornato a 1 per la fase di decifratura.
Alla fine di ogni fase il Cifratore avrà il compito di scrivere su file di testo il risultato per mezzo di procedure dedicate.
Le procedure native del linguaggio MIPS che permettono la scrittura e la lettura dei file sono state racchiuse all'interno di procedure più grandi in modo tale da generalizzarle e poterle riutilizzare in maniera indipendente:

1. **writeMessage**: procedura dedicata alla scrittura del file cifrato o decifrato, prende in ingresso il descrittore del file su cui scrivere.
2. **readMessage**: procedura dedicata alla lettura del messaggio da cifrare o decifrare, prende in ingresso il descrittore del file da leggere.
3. **readKey**: procedura dedicata alla lettura della parola chiave da utilizzare nella fase di cifratura o decifratura, prende in ingresso il descrittore del file da leggere.

Per favorire la semplicità e la leggibilità del codice è stato preferito implementare due procedure separate per la lettura del messaggio e delle chiave.

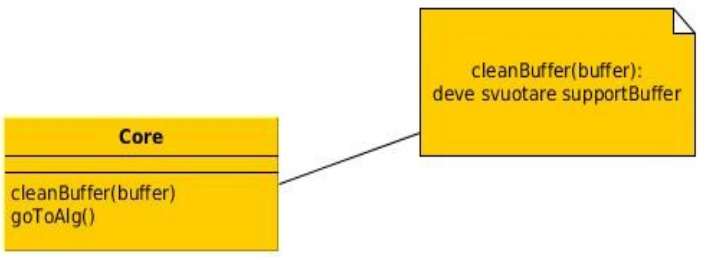
5. Cifratura e Decifratura

Le responsabilità delle fasi di cifratura e decifratura sono state ridotte al minimo per favorire la generalità, infatti entrambe le fasi chiameranno la procedura **core** che eseguirà l'effettiva operazione di cifratura o decifratura.
Dall'analisi del problema è stato definito che gli algoritmi A, B, C hanno degli stati in comune sia durante la cifratura che durante la decifratura. Abbiamo deciso quindi di determinare il loro comportamento grazie alla chiamata di **setStatusABC,** che è la procedura dedicata a impostare l'array degli stati per questi algoritmi chiamandola all'inizio di ogni fase usando il flag di stato in $s7. In questo array verranno inseriti due valori, differenti a seconda dell'algoritmo chiamato e della fase di cifratura/decifratura, quali l'indice dal quale iniziare a cifrare e il passo di scorrimento del messaggio.
Per facilitare la lettura, il settaggio dei singoli stati e della distanza tra questi valori all'interno del buffer sono stati demandati a procedure dedicate che portano il nome dell'algoritmo che a loro interessa.
La fase di cifratura ha la responsabilità di caricare l'indirizzo di partenza per la lettura della chiave essendo la prima fase in esecuzione. Questo però non accade in fase di decifratura,

dato che verrò mantenuto l'indirizzo ottenuto alla fine della fase precedente per scorrere la chiave in senso inverso.

6. Core

Core è il cuore del cifratore, è la procedura usata sia in cifratura sia in decifratura che consente le chiamate ai vari algoritmi eseguendo dei passaggi intermedi. Core ha l'unica responsabilità di chiamare gli algoritmi, a seconda della chiave corrente.
A seconda della fase in cui si trova, core aggiornerà il registro contenente il passo di scorrimento della chiave: questo sarà settato a 1 in fase di cifratura per scorrere in avanti, mentre a -1 in decifratura per tornare indietro.
Prima di ogni chiamata agli algoritmi però verrà eseguita una pulizia del buffer di supporto attraverso l'uso della procedura generica **cleanBuffer**; questo metodo svuota i buffer a lui passati sostituendo con "0" gli elementi al suo interno.



Durante la lettura della chiave, per ogni simbolo trovato al suo interno, **core** chiamerà l'algoritmo associato facendo riferimento alla tabella dei salti **algorithmJAT** inizializzata precedentemente nel main.
Per motivi di chiarezza, il calcolo dell'indirizzo e il salto all'algoritmo desiderato vengono svolti dalla procedura dedicata **goToAlg**: essendo le chiavi rappresentate in formato decimale in un intervallo tra 65 e 69, effettuando la sottrazione tra il valore corrente e la costante di 65 si ottengono valori che vanno da 0 a 4 che, moltiplicati per quattro, permettono di saltare alla posizione corretta nella tabella degli algoritmi. Successivamente, quando il controllo ritornerà a **Core**, questo aggiornerà l'indice della chiave. La lettura della chiave finirà quando verrà trovato lo zero, che indica la fine della stringa.
L'array di riferimento per entrambe le fasi è **bufferReader**. All'inizio del programma contiene il messaggio originario, mentre durante lo svolgimento dei vari algoritmi conterrà la cifratura (o decifratura) parziale fino ad arrivare alla fine della lettura della chiave, quando conterrà il messaggio finale da scrivere nel file di output.

7. Shifter

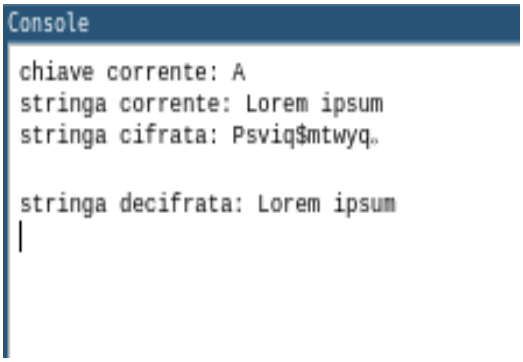
Shifter è la procedura generica associata agli algoritmi A,B,C. L'esecuzione di questo algoritmo determina il cambiamento del contenuto del buffer in ingresso, senza alterarne la grandezza.
Shifter combina le operazioni di cifratura e decifratura in modo da poterle sfruttare nel corso delle varie chiamate.
Queste due operazioni hanno in comune la prima parte di codice, che consiste nel caricare il buffer da analizzare, caricare il "passo" di scorrimento del buffer, e l'indice di partenza da cui iniziare le operazioni.
Per l'algoritmo A il passo di scorrimento sarà di 1, poiché deve iterare su tutti gli elementi, sarà di 2 invece per B e C che iterano su elementi alterni. Allo stesso modo l'algoritmo A e B avranno indice di partenza 0, mentre C avrà indice 1 perché deve lavorare sugli elementi in posizione dispari.
Shifter esegue la cifratura e la decifratura all'interno del ciclo **convertitore**, che si ripete usando un salto incondizionato. Durante il ciclo vengono caricati il valore del modulo e la costante, 4 in cifratura e -4 in decifratura, che permette il cambiamento dei valori presenti nel messaggio.
Shifter distgue le due fasi tramite un controllo sul registro $s7, permettendo di evitare l'esecuzione delle righe che interessano la decifratura. L'algoritmo termina quando nella stringa viene incontrato il valore "0".

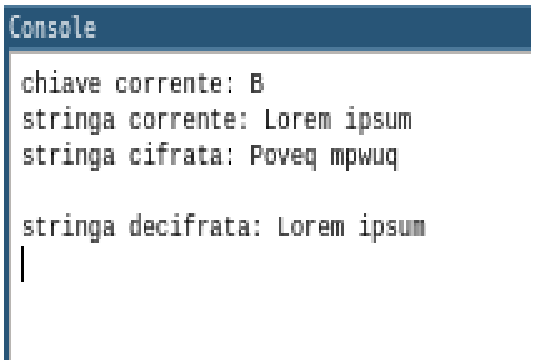
7.1 Funzioni di Shifter

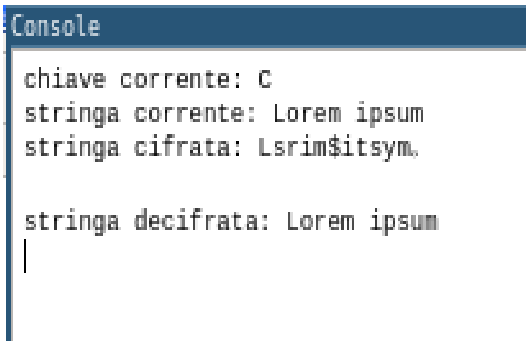
- **shifter**: funzione che carica l'indirizzo del buffer, ottiene il passo di scorrimento, l'indice di partenza e vi avanza
- **convertitore**: carica l'elemento da cifrare, il modulo di 255 e la costante di cifratura 4. Se l'elemento puntato è zero termina il ciclo
- **decriptazione**: controllando il flag di stato questa funzione altera la costante in -4 se lo stato è quello di decifratura, altrimenti va al metodo successivo
- **criptazione**: viene modificato il valore dell'elemento puntato e il puntatore viene spostato al prossimo elemento col passo giusto, poi ricomincia il ciclo tornando a **convertitore**
- **uscitaShifter**: viene caricato l'indirizzo di ritorno del chiamante e il metodo termina

7.2 Prove di funzionamento di Shifter

Per motivi di visibilità è stato riportato solo l'output della console di Qt-Spim. Per tutti e tre gli algoritmi abbiamo usato la frase generica "Lorem ipsum".
Ha una rappresentazione decimale secondo la tabella ASCII:
76-111-114-101-109-32-105-112-115-117-119







8. Algoritmo D

AlgD è la procedura dedicata all'implementazione dell'algoritmo D. L'esecuzione di questo algoritmo ha l'effetto di invertire gli elementi del buffer dato in ingresso, senza alterare il suo contenuto. L'algoritmo usa **supportBuffer**, il buffer di supporto, sul quale verrà scritto il messaggio invertito.
Il principio di inversione si basa sulla lettura in senso opposto del buffer contenente il messaggio e quello di supporto: mentre il primo buffer viene scorso da destra verso sinistra i suoi elementi esaminati vengono salvati all'interno del secondo buffer, che scorre però da sinistra verso destra creando l'inversione desiderata.
Per rendere la procedura più generica possibile **algD** si appoggia a **bufferLength**, il metodo che ritorna la lunghezza del buffer in ingresso che sarà poi utilizzata nel ciclo di **reversal**, che esegue l'effettivo lavoro di inversione della stringa.

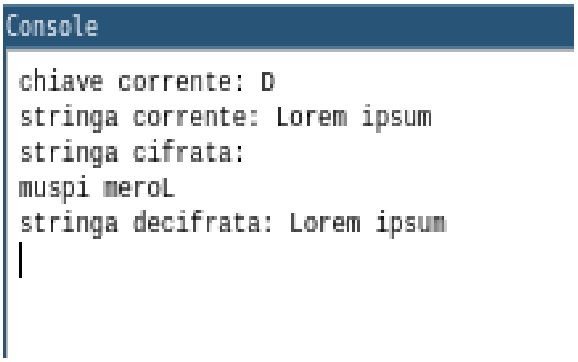
Dato che **reversal** ha effetto su **supportBuffer**, la parte finale dell'algoritmo svuota il buffer in ingresso tramite il metodo **cleanBuffer** e chiama poi una procedura dedicata a copiare il contenuto di **supportBuffer** dentro **bufferReader**, ovvero **overWrite**.
BufferLength è una procedura che aumenta un contatore ogni volta che trova un elemento diverso da "0", in modo tale da trovare il numero di elementi presenti nel buffer passato. Una volta trovato 0 il metodo finisce, dato che il buffer è finito. **OverWrite** si basa sullo stesso principio di **cleanBuffer**, ma scorre gli elementi del buffer dato in ingresso per copiarli in un altro, sovrascrivendo i precedenti elementi.

8.1 Funzioni dell'algoritmo D

- **algD**: funzione che dà inizio all'algoritmo calcolando attraverso **bufferLength** il numero degli elementi del messaggio e inizializza il contatore degli elementi inseriti
- **reversal**: funzione ciclica che salva in **supportBuffer** gli elementi di **bufferReader** parteno dal fondo. Termina quando gli elementi inseriti sono pari a quelli iniziali
- **swapVet**: pulisce il buffer iniziale tramite **cleanBuffer** per poi sovrascriverlo con gli elementi presenti in **supportBuffer** tramite **overWrite**

8.2 Prova di funzionamento dell'algoritmo D

Per motivi di visibilità è stato riportato solo l'output della console di Qt-Spim. Per l'algoritmo abbiamo usato la frase generica "Lorem ipsum".



9. Algoritmo E

Dato che la cifratura dell'algoritmo E e la sua decifratura sono molto diverse tra di loro, esse sono state divise in due procedure distinte: **cifratura_E** e **decifra_E**.

Cifratura-E : Questa procedura è dedicata alla fase di cifratura dell'algoritmo E .
L'esecuzione di questo algoritmo ha l'effetto di cambiare il contenuto del buffer, trovando le occorrenze degli elementi al suo interno e aumentando di conseguenza la dimensione del messaggio.
Viene chiamata inizialmente la procedura **occurrence** che restituisce all'interno di un buffer dedicato, **occurrenceBuffer**, gli elementi di bufferReader ripetuti una volta sola.
Per fare ciò il buffer contenente il messaggio iniziale viene scorso, controllando ogni volta all'interno di **occurrenceBuffer** se l'elemento preso in esame sia già presente oppure no; se l'elemento puntato in **bufferReader** non è ancora presente nel buffer delle occorrenze va inserito.
L'operazione di *costruzione* della **stringa cifrata** viene eseguita poi dalla procedura **writer** che utilizzerà **occurrenceBuffer** e **bufferReader**. Per fare ciò il metodo prende gli elementi di **occurrenceBuffer** come riferimento, uno per uno, e dopo averlo salvato nel buffer di supporto scorre **bufferReader** per controllare quando l'elemento puntato del primo buffer è uguale a quello del secondo; in questo caso l'occorrenza è stata trovata e il contatore delle posizioni viene salvato su **supportBuffer**. Fra le posizioni delle varie occorrenze vengono inseriti dei '-' per separarle, e, una volta finito di scorrere tutto **bufferReader** viene stampato uno spazio per separare gli elementi uno dall'altro.

L’algoritmo E ha necessità inoltre al suo interno di un metodo che permette di rappresentare cifre di posizioni superiori al 9, **storeDigits**. Questo particolare metodo prima conta da quante cifre è effettivamente composto il numero indicante le posizioni attraverso ripetute divisioni intere per 10, poi avanza di tale numero su **supportBuffer** e a ritroso ottiene le singole cifre della posizione, tramite resto modulo 10, per salvarle nell’ordine corretto.
L’esecuzione di **writer** ha effetto su **supportBuffer**, quindi la parte finale dell'algoritmo chiamerà la procedura che di **overWrite** per scrivere il contenuto del medesimo buffer dentro **bufferReader**.

Decifra-E : Questa procedura è dedicata alla fase di decifratura dell'algoritmo E.
L’esecuzione di questo algoritmo ha l’effetto di cambiare il contenuto e diminuendo di conseguenza la dimensione del messaggio in entrata.
La decifratura viene svolta dal ciclo **findPos** che distingue tra gli spazi “ ” e “-” per distinguere tra simboli e occorrenze. Quando vengono trovati gli spazi significa che è il momento di piazzare un nuovo elemento nella stringa di output, mentre quando vengono trovati i trattini va piazzato ancora l’elemento trovato in precedenza nella posizione corretta.
L’esecuzione del ciclo **findPos** ha effetto su **supportBuffer**, quindi la parte finale dell'algoritmo chiamerà la procedura che di **overWrite** per scrivere il contenuto del medesimo buffer dentro **bufferReader**.

9.1 Funzioni dell’algoritmo E

- Cifratura -

- **cifratura_E**: funzione che gestisce l’ordine di esecuzione degli altri metodi all’interno dell’algoritmo. Inizialmente chiama **occurrence**, una volta finito inizializza un contatore per gli elementi inseriti per poi chiamare **writer**. Al suo termine calcola il numero degli elementi presenti in **supportBuffer** tramite **bufferLength**, sovrascrive i suoi elementi in **bufferReader** e pulisce sia il buffer di supporto che quello delle occorrenze tramite **cleanBuffer**
- **occurrence**: inizio del ciclo che trova le singole occorrenze, termina quando viene trovato il valore 0
- **control**: controlla se è la prima volta che appare un certo elemento di **bufferReader**, in tal caso va a **firstOccurrence**
- **firstOccurrence**: salva l’elemento in **occurrenceBuffer**
- **ignore**: metodo che gestisce la comparsa di un elemento già presente in **occurrenceBuffer** andando all’elemento successivo
- f**inish_occurrence**: finito il metodo torna a **cifratura_E**
- **writer**: inizio del metodo che crea la cifratura e inizializza il contatore delle posizioni
- **elements**: funzione che scorre il buffer delle occorrenze per esaminare uno specifico elemento di **bufferReader**
- **positions**: stampa le posizioni in cui si trova l’elemento esaminato, separati da ‘.’
- **storeDigit**: metodo che salva le posizioni con una sola cifra
- **nextControl**: metodo che scorre all’elemento successivo di **bufferReader**
- **nextElement**: funzione che scorre al prossimo elemento di **occurrenceBuffer**
- **digitsCounter**: metodo che conta da quante cifre è composta la posizione
- **storeDigits**: metodo che avanza del numero di posizioni giusto rispetto al numero delle cifre
- **storeCicle**: ciclo di salvataggio di cifre multiple
- **offset**: metodo che imposta nelle giuste posizioni i puntatori ai buffer e aumenta i contatori del valore corretto
- **end_writer**: termina la scrittura su **supportBuffer** e torna a **cifratura_E**

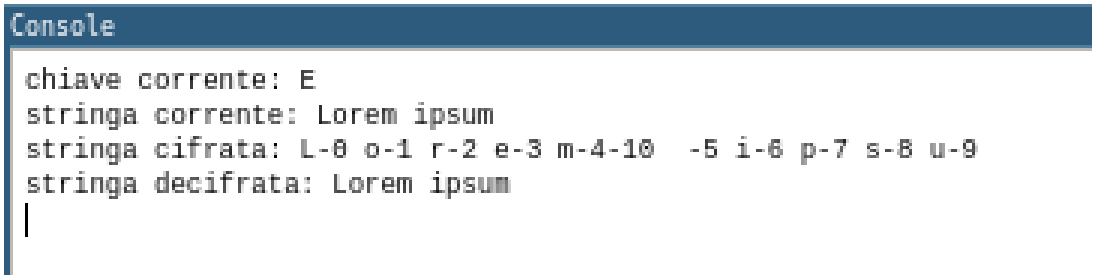
- Decifratura -

- **decifra_E**: inizio del metodo di decifratura che imposta la costante per la creazione di posizioni superiori alle unità
- **itemToPlace**: metodo che trova l’elemento da piazzare e avanza di due posizioni dato che fra l’elemento e la posizione dell’occorrenza ci sarà un ‘-’
- **findPos**: ciclo che trova la posizione giusta in cui inserire l’elemento. Se dopo aver trovato una cifra ne viene trovata un’altra significa che quella cifra era una decina e viene moltiplicata per 10 sommandola alla cifra appena trovata e così via.

- **placeItem**: una volta trovato uno spazio o un ‘-’ questa funzione salva l’elemento in posizione giusta. Una volta arrivato alla fine della stringa il metodo svuota **bufferReader** e vi sovrascrive gli elementi di **supportBuffer**

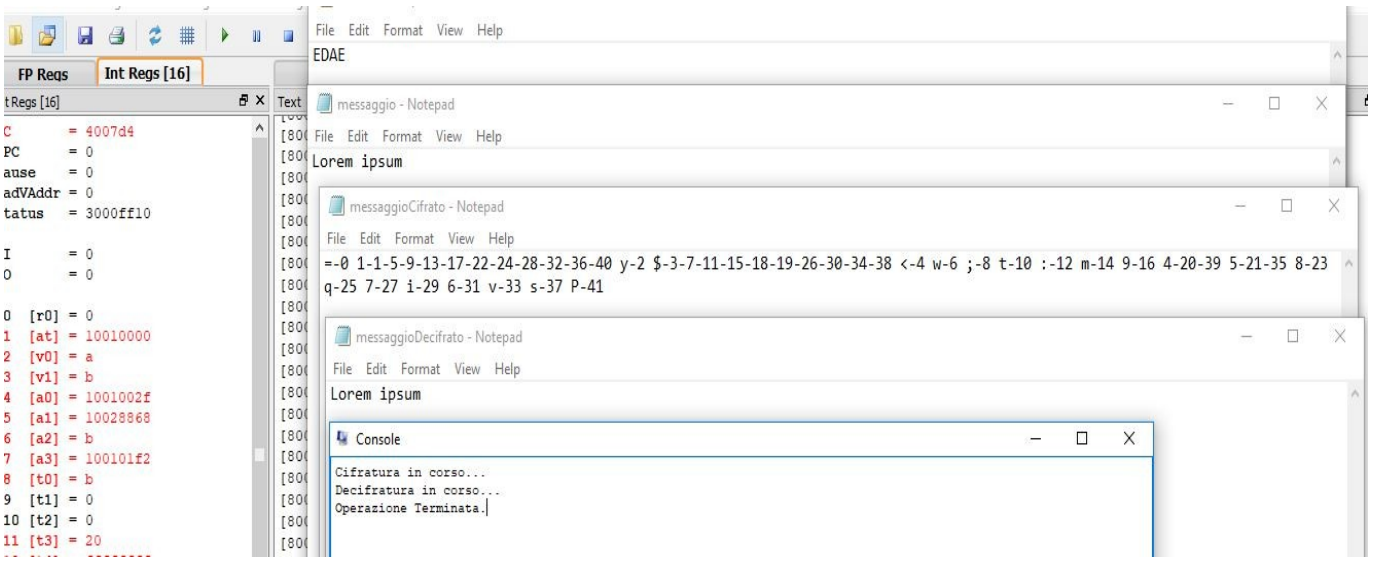
9.2 Prova di funzionamento dell’algoritmo E

Per motivi di visibilità è stato riportato solo l’output della console di Qt-Spim. Per tutti e tre gli algoritmi abbiamo usato la frase generica “Lorem ipsum”.



10. Prova di funzionamento degli algoritmi

Per la prove di funzionamento generale è stata usata nuovamente la stringa di 11 caratteri : Lorem ipsum .



11. Scelte implementative

Visti i problemi di saturazione della memoria riscontrati precedentemente e la seconda revisione del testo, a bufferReader e supportBuffer sono stati assegnati una dimensione di 100000 byte.
Questa scelta è stata determinata dal calcolo del caso peggiore che possa avvenire in fase di cifratura, ovvero un messaggio di 128 caratteri su cui viene eseguita una cifratura in chiave “EEEE”. Supponendo poi che tutti gli elementi siano differenti tra di loro, alla prima chiamata di E il messaggio creatosi avrà almeno 4 nuovi elementi per ogni occorrenza presentatasi nel messaggio iniziale: questi nuovi elementi sono il carattere di cui si trova l’occorrenza, il “-” seguente, la posizione in cui l’elemento viene trovato e lo spazio che divide le occorrenze tra di loro. Considerato ciò, e considerato anche il fatto che nelle successive chiamate verranno trovati elementi anche in posizioni superiori al migliaio, i caratteri aumentano esponenzialmente. Abbiamo così deciso di calcolare approssimativamente lo spazio necessario attraverso più operazioni di moltiplicazione: 128 x 4 (dati i caratteri ipotizzati sopra) x 5 (date le posizioni che saranno sicuramente sopra il centinaio) x 6 x 6 (date le posizioni che saranno sicuramente sopra il migliaio).

Il risultato di tale operazione è 92160, che abbiamo deciso di arrotondare a 100000 nel caso in cui il calcolo da noi effettuato non copra tutte le possibilità.

Dal testo dell'esercizio poi sappiamo che la stringa iniziale sarà composta da un massimo di 128 elementi, ma abbiamo deciso di lasciare la dimensione del buffer delle occorrenze (occurenceBuffer) a 256, perché, dato che la tabella ASCII contiene 256 elementi, non possiamo escludere il fatto che possano comparire tutti almeno una volta durante le chiamate dei vari algoritmi.

Per ottimizzare la funzionalità della procedura Core() è stata introdotta sin dal main una JAT-table dedicata agli algoritmi. In alternativa poteva essere adoperata una serie di beq, che però rendevano il codice meno mantenibile all'occorrenza.

Per favorire la leggibilità del codice sono stati implementate due procedure differenti per la lettura del messaggio e della chiave (readMessage e readKey).

Nel codice è stato riscontrata la necessità di introdurre in punti ben precisi un procedura dedicata alla pulizia dei buffer che vengono utilizzati (cleanBuffer), dato che, ad esempio in presenza di messaggi molto brevi, venivano stampati caratteri rimasti dalle operazioni precedenti.

Visto che in più occasioni è stata riscontrata la necessità di calcolare la lunghezza si una stringa è stato aggiunta la procedura dedicata al calcolo della lunghezza dei buffer, bufferLength(buffer), che ritorna la lunghezza del vettore in ingresso.

12. Codice

```
# GRUPPO DI LAVORO :
# DUCCIO SERAFINI                          E-MAIL: duccio.serafini@stud.unifi.it
# ANDRE CRISTHIAN BARRETO DONAYRE  E-MAIL: andre.barreto@stud.unifi.it
#
# DATA DI CONSEGNA:  29/05/19
#

.data

# STRINGHE DEDICATE PER LA VISUALIZZAZIONE DELLA OPERAZIONE IN CORSO:
        opCifra:        .asciiz         "Cifratura in corso..."
        opDecif:        .asciiz         "\nDecifratura in corso..."
        done:           .asciiz         "\nOperazione Terminata."
# DESCRITTORI DEI FILE IN INGRESSO:
        messaggio:      .asciiz         "messaggio.txt"
        chiave:         .asciiz         "chiave.txt"
# DESCRITTORI DEI FILE IN USCITA:
        msgCifrato:     .asciiz         "messaggioCifrato.txt"
        msgDecifrato:   .asciiz         "messaggioDecifrato.txt"

.align 2

# BUFFER DECICATI AL SUPPORTO DELLE PROCEDURE:
        algorithmJAT:   .space          20
        statusABC:      .space          36
        supportInvert:  .space          4
        occurrenceBuffer:       .space          256
        supportBuffer:  .space          100000

# BUFFER DEDICATI ALLA LETTURA DEI DATI DEI FILE IN INPUT:
        bufferReader:   .space          100000
```

```
                bufferKey:          .space          4
.align 2

.text
.globl main

main:           addi    $sp, $sp, -16
                jal     algorithmTable      # Creo una JAT table per chiamare gli algoritmi
                sw      $ra, 0($sp)         # Salvo nello stack l'indirizzo di ritorno del chiamante
                sw      $s0, 4($sp)
                sw      $s1, 8($sp)
                sw      $s2, 12($sp)

################ AVVIO FASE CIFRATURA ################

                la      $a0, chiave         # Carico l'indirizzo del file che contiene la chiave
                jal     readKey                     # Vado alla procedura che la legge

                la      $a0, messaggio      # Carico l'indirizzo del file che contiene il messaggio
                jal     readMessage         # Vado alla procedura che lo legge

                li      $s7, 0              # Variabile di stato : settata per la CIFRATURA

                jal     cifratura           # Fase di CIFRATURA

                la      $a0, msgCifrato     # Carico l'indirizzo del file in cui verra' scritto il messaggio
cifrato
                jal     writeMessage        # E vado alla procedura che lo scrive

                li      $v0, 16
                la      $a0, msgCifrato     # Chiusura del file del messaggio cifrato
                syscall

################ AVVIO FASE DECIFRATURA ################

                li      $s7, 1              # VARIABILE DI STATO : settata per la DECIFRATURA

                la      $a0, msgCifrato     # Metto in $a0 l'indirizzo del file che contiene il messaggio
cifrato
                jal     readMessage         # Vado al metodo di lettura del messaggio

                jal     decrifratura        # Fase di DECIFRATURA

                la      $a0, msgDecifrato   # Carico l'indirizzo del file in cui verra' scritto il messaggio
decifrato
                jal     writeMessage        # Vado al metodo che lo scrive

                li      $v0, 16
                la      $a0, msgDecifrato   # Chiusura del file contenente il messaggio decifrato
                syscall

                j       exit                # Vado alla fine del programma

#-------------------------------------------------------------------------------------------------------------#
cifratura:      addi    $sp, $sp,-4         # Alloco spazio dello stack per una parola
                sw      $ra, 0($sp)         # Salvo il ritorno del chiamante del main
```

```
                li      $v0, 4                  # Stampa del messaggio della cifratura
                la      $a0, opCifra            # "Cifratura in corso..."
                syscall

                la      $a1, statusABC          # Salvo in $a1 l'indirizzo del'array degli stati per A-B-C
                jal     setStatusABC            # Salto alla procedura di inizializzazione

                la      $s6, bufferKey          # Metto l'indirizzo del buffer chiave in $s6
                jal     core                    # Vado alla parte centrale del programma

uscita:         lw      $ra, 0($sp)             # Riprendo l'indirizzo di ritorno del chiamante
                addi    $sp, $sp, 4             # Dealloco lo spazio che lo conteneva
                jr      $ra                     # Torno al chiamante che era nel main

#-------------------------------------------------------------------------------------------------------------------#
decrifratura:   addi    $sp, $sp,-4             # Alloco spazio nello stack per una parola
                sw      $ra, 0($sp)             # Salvo l'indirizzo di ritorno del chiamante

                li      $v0, 4                  # Stampa del messaggio della decifratura
                la      $a0, opDecif            # "Decifratura in corso..."
                syscall

                la      $a1, statusABC          # Salvo in $a1 l'indirizzo del'array degli stati per A-B-C
                jal     setStatusABC            # Salto alla procedura di inizializzazione

                jal     core                    # Vado alla parte centrale del programma

                lw      $ra, 0($sp)             # Carico l'indirizzo di ritorno del chiamante
                addi    $sp, $sp, 4             # Dealloco spazio dello stack
                jr      $ra                     # Torno al main

#-------------------------------------------------------------------------------------------------------------------#

core:           addi    $sp, $sp, -4            # Alloco spazio nello stack per una parola
                sw      $ra, 0($sp)             # Ci salvo l'indirizzo di ritorno del chiamante

                beqz    $s7, offsetChiave       # Se il flag e' 0 siamo in cifratura e vado a scorriChiave
                li      $s3, -1                 # Imposto $s3 a -1 per scorrere la chiave al contrario
                addi    $s6, $s6, -1            # Torno indietro di una posizione perche' la cifratura ha portato
                                                # il puntatore della chiave fuori dal buffer
                j       prossimoAlg             # Vado direttamente alla procedura che chiama gli algoritmi

offsetChiave:   li      $s3, 1                  # Imposto $s3 a 1 per scorrere la chiave in avanti

prossimoAlg:    lb      $t0, ($s6)              # Carico l'elemento puntato della chiave
                beqz    $t0, fineCore           # Se e' zero allora sono arrivato alla fine della stringa
                blt     $t0, 65, scorriChiave   # Controlli per evitare ogni altro simbolo presente nella chiave
                bgt     $t0, 69, scorriChiave   # con valori minori o maggiori rispetto, relativamente, ad A ed
E

                li      $t1, 65                 # I vari algoritmi da chiamare vengono riconosciuti
                sub     $t0, $t0, $t1           # attraverso una operazione di sottrazione con 65
                move    $a0, $t0                # Salvo il risultato  della sottrazione in $a0

                la      $a1, supportBuffer      # Carico il buffer di support
                jal     cleanBuffer             # per pulirlo attraverso cleanBuffer
```

```
                jal     goToAlg                 # Vado al metodo che chiama l'algoritmo scelto dalla
chiave

scorriChiave:   add     $s6, $s6, $s3           # Scorro la chiave di 1 se siamo in cifratura, di -1 in decifratura
                j       prossimoAlg             # Vado all'algoritmo successivo

fineCore:       lw      $ra, 0($sp)             # Carico l'indirizzo di ritorno del chiamante
                addi    $sp, $sp, 4             # Delloco spazio dello stack
                jr      $ra                     # Torno a cifratura/decifratura

# goToAlg: Procedura che calcola la posizione in cui saltare nella tabella degli algoritmi
# Parametri: $a0 <-- Sottrazione ottenuta in precedenza fra il valore ascii della chiave e 65
#
# In questo modo verra' restituito un valore che, moltiplicato nel seguente metodo per 4, servira' a trovare
# la posizione corretta nella tabella degli algoritmi da cui verra' chiamato l'algoritmo di cifratura richiesto

goToAlg:        addi    $sp, $sp,-4             # Salvo il registro $ra corrente per potere tornare
                sw      $ra, 0($sp)             # al main a fine alla fine della procedura

                li      $t2, 4                  # Costante di default per il calcolo dell'indirizzo in cui saltare
                mult    $t2, $a0                # Moltiplico la costante per la sottrazione ottenuta in
precedenza
                mflo    $t2                     # Riprendo il risultato dal regristro dedicato alla
moltiplicazione


                lw      $a0, algorithmJAT($t2)  # Carico l'indirizzo contenuto nella JAT alla posizione
specificata
                jr      $a0                     # Viene eseguito il salto all'algoritmo richiesto

ritorno_scelta: lw      $ra, 0($sp)             # Carico l'indirizzo di ritorno
                addi    $sp, $sp, 4             # Dealloco spazio nello stack
                jr      $ra                     # Torno a CORE

#-------------------------------------------------------------------------------------------------------------------#
# cleanBuffer: Procedura dedicata alla pulizia del contenuto di qualsiasi buffer in ingresso
# Parametri :   $a1 <-- buffer da pulire
cleanBuffer:    lb      $t0, ($a1)              # Carico in $t0 l'elemento puntato
                beqz    $t0, endClean           # Se e' zero sono arrivato alla fine della stringa
                move    $t0, $zero              # Altrimento svuoto la variabile
                sb      $t0, 0($a1)             # Per caricarla nella stringa, cancellando il precedente elemento
                addi    $a1, $a1, 1             # Vado al prossimo elemento

                j       cleanBuffer             # Ripeto

endClean:       jr      $ra                     # Torno dove e' stato chiamato cleanBuffer

# shifter: Procedura generica che svolge la cifratura e la decifratura degli algoritmi A, B e C
# Il suo comportamento e' definito da procedure che settano dei flag prima di ogni chiamata
# Parametri:    $s0 <--  offset di inizio di scorrimento del buffer
#               $s1 <--  flag distinzione tra operazione di CRIFRATURA e DECIFRATURA
#               $s2 <--   valore dedicato al passo di scorrimento del buffer
#
# VALORE DI RITORNO:    VOID

shifter:        addi    $sp, $sp, -4            # Salvo il registro $ra corrente per potere tornare
                sw      $ra, 0($sp)             # al main a fine alla fine della procedura
```

```
                la      $a3, bufferReader       # Carico l'indirizzo del buffer che contiene il messaggio
                lb      $s2, 8($a0)             # $s2: passo per lo scorrimento all'elemento successivo
                lb      $s0, 0($a0)             # $s0: e' l'indice di partenza per la lettura
                add     $a3, $a3, $s0           # Vado all' indice giusto

convertitore:   lb      $t0, 0($a3)             # Metto in $t0 l'elemento da cifrare
                beqz    $t0, uscitaShifter # Se ha valore 0 allora siamo arrivati alla fine della stringa
                li      $t1, 255                # Viene definito il valore del modulo
                li      $t2, 4                  # E la costante di cifratura

decriptazione:  beqz    $s7, criptazione        # Se $s7 ha valore 0 allora siamo in cifratura
                                                # altrimenti siamo in decifratura
                li      $t4, -1                 # In questo caso metto in $t4 -1
                mult    $t2, $t4                # Per moltiplicarlo a $t2 (contiene 4)
                mflo    $t2                     # E ottenere -4 per poi sottrarlo all'elemento da decifrare
                                                # AGGIUNGERE CONTROLLO SUI NEGATIVI

criptazione:    add     $t0, $t0, $t2           # Sommo all'elemento +4 o -4
                div     $t0, $t1                # Poi effettuo la divisione per 255
                mfhi    $t0                     # E salvo il modulo in $t0
                sb      $t0, 0($a3)             # Salvo il valore ottenuto al posto di quello precedente nel
buffer
                add     $a3, $a3, $s2           # Avanzo di 1 o 2 posizioni a seconda dell'algoritmo chiamato

                j       convertitore            # Torno al convertitore

uscitaShifter:  lw      $ra, 0($sp)             # Carico l'indirizzo di ritorno del chiamante
                addi    $sp, $sp, 4             # Dealloco lo spazio dello stack
                jr      $ra                     # Fine di shifter e ritorno a core

# algD: Procedura che inverte i buffer dati in input
# Parametri :   $a2 <--- bufferReader , buffer contenente la stringa a invertire
#               $a3 <--- buffer di support alla procedura di inversione
algD:           add     $sp, $sp, -4            # Alloco spazio nello stack per una parola
                sw      $ra, 0($sp)             # Salvo l'indirizzo di ritorno del chiamante
                                                # $a2 <-- bufferReader
                jal     bufferLength            # Vado alla procedura che calcola la lunhgezza del buffer
                addi    $a2, $a2, -1            # Dato che il puntatore e' fuori dal buffer lo faccio tornare
                                                # indietro di una posizione
                move    $s0, $v1                        # Recupero il valore di ritorno : lunghezza del buffer
corrente
                move    $t0, $zero              # Riinizializzo $t0 per contare gli elementi inseriti
                                                # Ciclo di inversione:
reversal:       beq     $t0, $s0, swapVet       # Se il numero dei caratteri inseriti e' pari alla lunghezza del
buffer
                                                # allora posso uscire dalla procedura
                lbu     $t1, ($a2)              # Carico l'elemento puntato del buffer
                sb      $t1, ($a3)              # E lo salvo nel buffer di uscita
                addi    $a2, $a2, -1            # Vado al carattere precedente del buffer di input
                addi    $a3, $a3, 1             # Scorro alla posizione successiva del buffer di output
                addi    $t0, $t0, 1             # Aumento di 1 il contatore dei caratteri inseriti
                j       reversal                # Ricomincio ilciclo

swapVet:        la      $a1, bufferReader       # Metto in $a1 l'indirizzo di bufferReader
                jal     cleanBuffer             # Per chiamare la procedura di pulizia del buffer
```

```
                la      $a2, bufferReader       # Carico il buffer che va sovrascritto
                la      $a3, supportBuffer      # Carico il buffer contenente gli elementi da scrivere in
bufferReader
                jal     overWrite               # Vado alla procedura di sovrascrittura

                lw      $ra, 0($sp)             # Carico l'indirizzo di ritorno del chiamante
                add     $sp, $sp, 4             # Dealloco spazio dello stack
                jr      $ra                     # Fine dell'algoritmo D

# overWrite: Procedura che sovrascrive il contenuto di qualasiasi vettore
# Parametri:    $a2 <-- Vettore da sovrascrivere
#               $a3 <-- Vettore con i dati da scrivere
#               $s0 <-- Lunghezza dell'array da scrivere
overWrite:      move    $t0, $zero              # Inizializzo il contatore degli elementi inseriti

loop_overWrite: beq     $t0, $s0, EXIT_loopOW # Se ho inserito il numero giusto di elementi, esco da overWrite
                lb      $t1, 0($a3)             # Altrimenti carico l'elemento puntato di supportBuffer
                sb      $t1, 0($a2)             # E lo salvo in bufferReader
                addi    $a2, $a2, 1             # Avanzo di una posizione su bufferReader
                addi    $a3, $a3, 1             # Avanzo di una posizione su supportBuffer
                addi    $t0, $t0, 1             # Aumento il numero degli elementi inseriti di 1
                j       loop_overWrite          # Ricomincio il ciclo

EXIT_loopOW:    jr      $ra                     # Torno al chiamante

# bufferLength: Procedura che conta il numero dei caratteri nel buffer in ingresso
# Parametri:    $a2 <-- La stringa di cui contare la lunghezza
bufferLength:   move    $t0, $zero              # Inizializzo contatore degli elementi della stringa a 0

counterLoop:    lbu     $t1, 0($a2)             # Carico il carattere puntato in $t1
                beqz    $t1, endCounter         # Se sono arrivato alla fine della stringa il metodo termina
                addi    $t0, $t0, 1             # Altrimenti aumento il contatore di 1
                addi    $a2, $a2, 1             # Scorro alla posizione successiva del buffer
                j       counterLoop             # Inizio un nuovo ciclo

endCounter:     move    $v1, $t0                # Valore di ritorno in $v1
                jr $ra                          # Torno al chiamante

# cifratura_E : Algoritmo che conta le occorence della stringa data in input
# Parametri:    $a2 <-- bufferReader
#               $a3 <-- supportBuffer
cifratura_E:    add     $sp, $sp, -4            # Alloco spazio nello stack per una parola
                sw      $ra, 0($sp)             # Salvo l'indirizzo di ritorno del chiamante

                la      $a2, occurrenceBuffer   # Occurrence buffer verra' riempito con gli elementi di
bufferReader
                                                # ripetuti una sola volta
                jal     occurrence              # Salto al metodo che riempe tale buffer

                la      $a2, occurrenceBuffer   # Rimetto il puntatore all'inizio del buffer
                move    $t5, $zero              # Inizializazzione del contatore degli elementi inseriti in
supportBuffer

                jal     writer                  # Salto al metodo che produce l'output di cifratura

                la      $a2, supportBuffer      # Rimetto il puntatore all'inizio del buffer
```

```
                jal     bufferLength            # Vado al metodo che conta il numero di elementi presenti in
supportBuffer

                addi    $s0, $v1, -1            # Diminuisco la lunghezza di 1 perche' va ignorato l'ultimo
elemento
                la      $a2, bufferReader       # Rimetto il puntatore all'inizio di bufferReader
                la      $a3, supportBuffer      # e di supportBuffer
                jal     overWrite               # Vado alla sovrascrittura di supportBuffer in bufferReader

                la      $a1, supportBuffer      # Rimetto il puntatore all'inizio del buffer
                jal     cleanBuffer             # e lo svuoto per un possibile riutilizzo

                la      $a1, occurrenceBuffer   # Rimetto il puntatore all'inizio del buffer
                jal     cleanBuffer             # e lo svuoto per un possibile riutilizzo

                lw      $ra, 0($sp)             # Carico l'indirizzo di ritorno del chiamante
                add     $sp, $sp, 4             # Dealloco spazio dello stack
                jr      $ra                     # Torno al chiamante
                # FINE ALGORITMO E

# Inizio del metodo che riempe occurrenceBuffer
occurrence:     lbu     $t1, ($a1)              # Carico in $t1 il carattere puntato di bufferReader
                beqz    $t1, finish_occurence   # Se sono alla fine della stringa il metodo termina
# Frammento di metodo che riconosce se un elemento e' gia' stato inserito
control:        lbu     $t2, ($a2)              # Carico l'elemento puntato di occurrenceBuffer in $t2
                beqz    $t2, firstOccurrence    # Se in quella posizione non e' presente alcun elemento allora e'
la
                                                # prima volta che viene trovato. Vado quindi a
"firstOccurrence"
                beq     $t1, $t2, ignore        # Se gli elementi sono uguali invece vado a "ignore"
                addi    $a2, $a2, 1             # Altrimenti se sono diversi scorro di una posizione il buffer
delle
                j       control                 # occorrenze per controllare se l'elemento e' gia' stato trovato
prima
                                                # oppure e' la prima volta
# Metodo che gestisce la prima occorrenza di un elemento
firstOccurrence: sb     $t1, 0($a2)             # Salvo l'elemento che ho trovato nel buffer delle occorrenze
                addi    $a1, $a1, 1             # Vado alla posizione successiva di bufferReader
                la      $a2, occurrenceBuffer   # Rimetto il puntatore all'inizio del buffer delle occorrenze
                j       occurrence              # e inizio nuovamente a cercare le prime occorrenze degli
elementi
# Frammento di metodo che ignora un elemento se e' gia' presente in occurrenceBuffer
ignore:         addi    $a1, $a1, 1             # Scorro alla posizione successiva di bufferReader
                la      $a2, occurrenceBuffer   # Rimetto il puntatore all'inizio del buffer delle occorrenze
                j       occurrence              # e inizio nuovamente a cercare le prime occorrenze degli
elementi

finish_occurence:jr     $ra                     # Torno a cifratura_E

# Metodo che inizia il ciclo di cifratura del messaggio
writer:         la      $a1, bufferReader       # Torno all'inizio di bufferReader per leggere il messaggio
                move    $t0, $zero              # Inizializzo il contatore delle posizioni
# Metodo che scorre il buffer delle occorrenze per esaminare uno specifico elemento di bufferReader
elements:       lbu     $t2, ($a2)              # Carico l'elemento puntato di occurrenceBuffer in $t2
                beqz    $t2, end_writer         # Se sono arrivato alla fine del buffer allora l'algoritmo termina
                sb      $t2, 0($a3)             # Altrimenti salvo $t2 all'interno di supportBuffer, in modo tale
che
```

```
                                        # evidenzi l'elemento preso in esame in occurrenceBuffer
                addi    $a3, $a3, 1     # Vado alla posizione successiva di supportBuffer
                addi    $t5, $t5, 1     # Dato che ho inserito un elemento aumento il contatore $t5 di
1
# Metodo che stampa le posizioni in cui si trova l'elemento esaminato
positions:      lbu     $t1, ($a1)      # Carico l'elemento puntato di bufferReader in $t1
                beqz    $t1, nextElement        # Se sono alla fine del buffer allora vuol dire che ho
controllato tutte le
                                        # occorrenze dell'elemento puntato in occurrenceBuffer, e
posso andare al prossimo
                bne     $t1, $t2, nextControl   # Se $t1 e $t2 sono diversi allora vado al metodo che scorre al
controllo
                                        # dell'elemento successivo di bufferReader
                li      $t3, '-'        # Altrimenti carico il simbolo '-'
                sb      $t3, 0($a3)     # e lo salvo in supportBuffer per separare le occorrenze

                addi    $t5, $t5, 1     # Dato che ho inserito un elemento in supportBuffer aumento
$t5 di 1
                move    $t4, $t0        # Metto in $t4 il contatore delle posizioni
                move    $t8, $zero      # Inizializzo il contatore delle cifre

                sgt     $t7, $t4, 9     # Se il contatore e' superiore a 9 imposto $t7 a 1
                beq     $t7, 1, digitsCounter   # In tal caso vado al metodo che conta da quante cifre e'
composto
                                        # il contatore
# Metodo che salva una sola cifra (se il contatore $t4 e' ancora minore di 10)
storeDigit:     addi    $a3, $a3, 1     # Avanzo di uno perche' sto puntando a "-"
                addi    $t0, $t0, 48    # Aggiungo 48 a $t0 per convertirlo in ASCII
                sb      $t0, 0($a3)     # Salvo il valore ottenuto in supportBuffer
                addi    $t0, $t0, -48   # Faccio tornare il contatore di posizioni al valore precedente
                addi    $a3, $a3, 1     # Avanzo di una posizione su supportBuffer
                addi    $t5, $t5, 1     # Aumento di uno il contatore degli elementi

# Metodo che passa al prossimo controllo
nextControl:    addi    $a1, $a1, 1     # Vado all'elemento successivo di bufferReader
                addi    $t0, $t0, 1     # Aumento di 1 il contatore delle posizioni
                j       positions       # Torno al controllo delle posizioni

# Metodo che permette di passare al prossimo controllo degli elementi basato sul buffer delle occorrenze
nextElement:    li      $t3, ' '        # Carico in $t3 uno spazio per separare le varie occorrenze
degli elementi
                sb      $t3, 0($a3)     # Lo salvo all'interno di supportBuffer
                addi    $a3, $a3, 1     # Avendo inserito questo elelemento avanzo alla posizone
successiva
                addi    $t5, $t5, 1     # E aumento di 1 il contatore degli elementi inseriti
                addi    $a2, $a2, 1     # Passo al prossimo elemento di confronto presente in
occurrenceBuffer
                j       writer          # E ricomincio il ciclo

# Metodo che conta da quante cifre e' formata l'occorrenza
digitsCounter:  beqz    $t4, storeDigits        # Se ho contato tutte le cifre allora vado al metodo che salva
cifre multiple
                li      $t9, 10         # Metto in $t9 il valore 10
                div     $t4, $t9        # Per effettuare la divisione ed eliminare l'ultima cifra
                mflo    $t4             # Salvo il quoziente in $t4
                addi    $t8, $t8, 1     # Aumento di 1 il contatore delle cifre
                j       digitsCounter   # Ricomincio il ciclo di divisione
```

```
# Metodo che salva in supportBuffer occorrenze con piu' di una cifra (se il contatore $t4 e' maggiore di 10)
storeDigits:    move    $s0, $t8            # Salvo il numero delle cifre nella costante $s0
                add     $a3, $a3, $s0       # Avanzo del numero di cifre corretto su supportBuffer
                move    $t4, $t0            # Metto in $t4 il contatore delle posizioni
# Ciclo di salvataggio di cifre multiple
storeCicle:     div     $t4, $t9            # Divido il contatore delle posizioni per 10
                mflo    $t4                 # Salvo in $t4 il quoziente
                mfhi    $t8                 # Salvo in $t8 il resto
                                            # per poi salvare la cifra ottenuta nella giusta posizione
                addi    $t8, $t8, 48        # Aggiungo 48 a $t8 per convertirlo in ASCII
                sb      $t8, ($a3)          # Salvo il valore cosi' ottenuto nella giusta posizione
                beqz    $t4, offset         # Se il numero e' stato stampato completamente allora termino
il ciclo
                addi    $a3, $a3, -1        # Altrimenti vado alla posizione precedente del buffer
                                            # per salvare la cifra precedente della posizione
                j       storeCicle          # Ricomincio il ciclo di salvataggio

# Metodo che imposta nelle giuste posizioni i puntatori ai buffer e aumenta i contatori del valore corretto
offset:         add     $a3, $a3, $s0       # Avanzo nuovamente in supportBuffer del numero di cifre
appena salvate
                add     $t5, $t5, $s0       # Il contatore dei caratteri inseriti aumenta del numero di cifre
salvate
                addi    $a1, $a1, 1         # Avanzo di 1 in bufferReader
                addi    $t0, $t0, 1         # Aumento di 1 il contatore delle posizioni
                j       positions           # Torno a cercare le posizioni degli elementi
end_writer:     jr      $ra                 # Torno al metodo principale

# decifra_E: Algoritmo che decifra il messaggio cifrato da E
# Parametri:    $a2 <-- bufferReader
#               $a3 <-- supportBuffer
decifra_E:      addi    $sp, $sp -4         # Alloco spazio nello stack per una parola
                sw      $ra, 0($sp)         # Salvo l'indirizzo di ritorno del chiamante
                li      $s1, 10             # $s1 e' la costante che servira' per formare le posizioni
superiori al 9
# Metodo che trova l'elemento da piazzare
itemToPlace:    la      $a2, supportBuffer  # Imposto l'indirizzo iniziale del buffer di supporto in $a2
                lbu     $t0, ($a1)          # Carico il primo elemento della frase in $t0 (che sara'
l'elemento
                                            # che dovra' essere inserito per formare la frase originaria)
                addi    $a1, $a1, 2         # Scorro avanti di 2 dato che dopo questo elemento ci
sara'sicuramente '-'
                move    $t1, $zero          # Inizializzo la variabile che formera' la posizione
# Ciclo che trova la posizione in cui piazzare l'elemento
findPos:        lbu     $t2, ($a1)          # Carico l'elemento puntato in $t2
                beq     $t2, '-', placeItem # Se tale elemento e' '-'
                beq     $t2, ' ', placeItem # O uno spazio
                beqz    $t2, placeItem      # Oppure la fine della stringa
                                            # Allora ho trovato la posizione giusta dove collocare
l'elemento
                mult    $t1, $s1            # Altrimenti vuol dire che la posizione non e' completa
                mflo    $t1                 # Salvo il risultato della moltiplicazione di $t1 per 10 in $t1
                addi    $t2, $t2, -48       # Converto l'elemento da ASCII a numero
                add     $t1, $t1, $t2       # Sommo la cifra per formare la posizione
                addi    $a1, $a1, 1         # Scorro di 1 il buffer
                j       findPos             # Ricomincio il ciclo
# Metodo che piazza l'elemento una volta trovata la sua posizione
placeItem:      add     $a2, $a2, $t1       # Mi sposto alla posizione indicata
```

```
                sb      $t0, 0($a2)             # Inserisco l'elemento in posizione corretta

                addi    $a1, $a1, 1             # Avanzo di 1 sul buffer
                beq     $t2, ' ', itemToPlace   # Se prima ho trovato uno spazio allora devo trovare
                                                # il prossimo elemento da piazzare
                la      $a2, supportBuffer      # Se invece ho trovato un '-' significa che devo piazzare
l'elemento
                                                # altre volte, torno quindi all'inizio del buffer codificato
                move    $t1, $zero              # Metto nuovamente a 0 il contatore delle posizioni
                beq     $t2, '-', findPos   # E torno al metodo che trova le posizioni
                                                # Alrimenti vuol dire che sono arrivato alla fine della stringa
                                                # e il programma puo' terminare
                la      $a1, bufferReader       # Rimetto il puntatore all'inizio del buffer
                jal     cleanBuffer             # e lo svuoto

                la      $a2, supportBuffer      # Rimetto il puntatore all'inizio del buffer
                jal     bufferLength            # Per calcolarne il numero di elementi

                move    $a1, $v1                        # Salvo il valore di ritorno in $a1
                la      $a2, bufferReader       # Rimetto il puntatore all'inizio di bufferReader
                la      $a3, supportBuffer      # e all'inizio di supportBuffer
                jal     overWrite               # Per sovrascrivere supportBuffer in bufferReader

                la      $a1, supportBuffer      # Rimetto il puntatore all'inizio dle buffer
                jal     cleanBuffer             # e lo svuoto

                lw      $ra, 0($sp)             # Carico l'indirizzo di ritorno del chiamante
                addi    $sp, $sp, 4             # Dealloco spazio dello stack
                jr      $ra                     # Torno al chiamante
                # FINE DECIFRATURA ALGORITMO E
#-------------------------------------------------------------------------------------------------------------------#

# Procedura che inizializza la tabella dedicata agli algoritmi
algorithmTable: la      $t7, algorithmJAT       # Salvo l'indirizzo della JAT in $t7
                la      $t6, algoritmo_A                # In $t6 metto l'indirizzo all'algoritmo A
                sw      $t6, 0($t7)             # E lo salvo nella JAT
                la      $t6, algoritmo_B                # In $t6 metto l'indirizzo all'algoritmo B
                sw      $t6, 4($t7)             # E lo salvo nella JAT
                la      $t6, algoritmo_C                # In $t6 metto l'indirizzo all'algoritmo C
                sw      $t6, 8($t7)             # E lo salvo nella JAT
                la      $t6, algoritmo_D                # In $t6 metto l'indirizzo all'algoritmo D
                sw      $t6, 12($t7)            # E lo salvo nella JAT
                la      $t6, algoritmo_E                # In $t6 metto l'indirizzo all'algoritmo E
                sw      $t6, 16($t7)            # E lo salvo nella JAT

                move    $v0, $t7                # Restituisco l'indirizzo della JAT in $v0
                jr      $ra                     # Torno nel main


# setStatusABC: Imposta l'array degli stati dedicati alle procedure A, B e C
# Offset per lettura dello stato : 0 e' A , 12 e' B, 24 e' C
#
setStatusABC:   addi    $sp, $sp, -4            # Faccio spazio nello stack per una parola
                sw      $ra, 0($sp)             # Salvo l'indirizzo di ritorno del chiamante

                jal     algAStatus              # Imposto gli stati per l'algoritmo A
```

```
                addi    $a1, $a1, 12            # Vado avanti di 3 spazi
                jal     algBStatus              # Imposto gli stati per l'algoritmo B

                addi    $a1, $a1, 12            # Vado avanti di 3 spazi
                jal     algCStatus              # Imposto gli stati per l'algoritmo B

                lw      $ra, 0($sp)             # Carico l'indirizzo di ritorno del chiamante
                addi    $sp, $sp, 4             # Dealloco spazio dello stack
                jr      $ra                     # Torno al chiamante in Cifratura/Decifratura

# algAStatus: Procedura dedicata al settaggio dei flag dedicati all'algoritmo A
algAStatus:     addi    $sp, $sp, -4            # Faccio spazio nello stack per una parola
                sw      $ra, 0($sp)             # Salvo l'indirizzo di ritorno del chiamante

                li      $t0, 0                  # Carico l'offset di partenza
                sb      $t0, 0($a1)             # E lo salvo nel buffer
                li      $t2, 1                  # Carico il passo di lettura
                sb      $t2, 8($a1)             # E lo salvo nel buffer

                beqz    $s7, stepA              # Se il flag e' 0 allora siamo in cifratura
                li      $t1, 1                  # Altrimenti siamo in decifratura e imposto il flag a 1
                sb      $t1, 4($a1)             # E lo salvo nel buffer
                j       fineStatusA             # Vado alla fine del metodo

stepA:          li      $t1, 0                  # Essendo in cifratura imposto il flag a 0
                sb      $t1, 4($a1)             # E lo salvo nel buffer

fineStatusA:    lw      $ra, 0($sp)             # Carico indirizzo di ritorno del chiamante
                addi    $sp, $sp, 4             # Dealloco spazio dello stack
                jr $ra                          # Torno a setStatusABC

# algBStatus: Procedura dedicata al settaggio dei flag dedicati all'algoritmo B
algBStatus:     addi    $sp, $sp, -4            # Faccio spazio nello stack per una parola
                sw      $ra, 0($sp)             # Salvo l'indirizzo di ritorno del chiamante

                li      $t0, 0                  # Carico l'offset di partenza
                sb      $t0, 0($a1)             # E lo salvo nel buffer
                li      $t2, 2                  # Carico il passo di lettura
                sb      $t2, 8($a1)             # E lo salvo nel buffer

                beqz    $s7, stepB              # Se il flag e' 0 allora siamo in cifratura
                li      $t1, 1                  # Altrimenti siamo in decifratura e imposto il flag a 1
                sb      $t1, 4($a1)             # E lo salvo nel buffer
                j       fineStatusB             # Vado alla fine del metodo

stepB:          li      $t1, 0                  # Essendo in cifratura imposto il flag a 0
                sb      $t1, 4($a1)             # E lo salvo nel buffer

fineStatusB:    lw      $ra, 0($sp)             # Carico indirizzo di ritorno del chiamante
                addi    $sp, $sp, 4             # Dealloco spazio dello stack
                jr $ra                          # Torno a setStatusABC

# algCStatus: Procedura dedicata al settaggio dei flag dedicati all'algoritmo C
algCStatus:     addi    $sp, $sp, -4            # Faccio spazio nello stack per una parola
                sw      $ra, 0($sp)             # Salvo l'indirizzo di ritorno del chiamante

                li      $t0, 1                  # Carico l'offset di partenza
```

```
                sb      $t0, 0($a1)             # E lo salvo nel buffer
                li      $t2, 2                  # Carico il passo di lettura
                sb      $t2, 8($a1)             # E lo salvo nel buffer

                beqz    $s7, stepC              # Se il flag e' 0 allora siamo in cifratura
                li      $t1, 1                  # Altrimenti siamo in decifratura e imposto il flag a 1
                sb      $t1, 4($a1)             # E lo salvo nel buffer
                j       fineStatusC             # Vado alla fine del metodo

stepC:          li      $t1, 0                  # Essendo in cifratura imposto il flag a 0
                sb      $t1, 4($a1)             # E lo salvo nel buffer

fineStatusC:    lw      $ra, 0($sp)             # Carico indirizzo di ritorno del chiamante
                addi    $sp, $sp, 4             # Dealloco spazio dello stack
                jr $ra                          # Torno a setStatusABC

#-------------------------------------------------------------------------------------------------------------------#
# Chiamata agli algoritmi di cifratura e decifratura

algoritmo_A:    la      $a0, statusABC          # Carico l'indirizzo del buffer degli stati per A B o C
                jal     shifter                 # Chiamo shifter settato per l'algoritmo A

                j       ritorno_scelta          # Torno indietro

algoritmo_B:    la      $a0, statusABC          # Carico l'indirizzo del buffer degli stati per A B o C
                addi    $a0, $a0, 12            # Avanzo di 12 posizioni nel buffer per trovare gli stati di B
                jal     shifter                 # Chiamo shifter settato per l'algoritmo B

                j       ritorno_scelta          # Torno indietro

algoritmo_C:    la      $a0, statusABC          # Carico l'indirizzo del buffer degli stati per A B o C
                addi    $a0, $a0, 24            # Avanzo di 24 posizioni nel buffer per trovare gli stati di C
                jal     shifter                 # Chiamo shifter settao per l'algoritmo C

                j       ritorno_scelta          # Torno indietro

algoritmo_D:    la      $a2, bufferReader       # Carico l'indirizzo del buffer che contiene il messaggio in $a2
                la      $a3, supportBuffer      # Carico l'indirizzo del buffer di supporto in $a3
                jal     algD                    # Vado all'algoritmo D

                j       ritorno_scelta          # Torno indietro

algoritmo_E:    la      $a1, bufferReader       # Carico l'indirizzo del buffer che contiene il messaggio in $a2
                la      $a3, supportBuffer      # Carico l'indirizzo del buffer di supporto in $a3

                beq     $s7, 1, decifratura_E   # Se siamo in fase di decifratura vado a decifra_E
                jal     cifratura_E             # Altrimenti vado al criptaggio di E
                j       salta_decifra           # Ignorando la decifratura

decifratura_E:  jal     decifra_E               # Chiamo il metodo che decifra E

salta_decifra:  j       ritorno_scelta          # Torno indietro

#-------------------------------------------------------------------------------------------------------------------#
# readMessage : Procedura dedicata alla lettura del file che deve essere CIFRATO o DECIFRATO
# parametri :   $a0 <-- descrittore del file da leggere
#
```

```
# valore di ritorno:        void
# il suo effetto e' quello di riempire il buffer da trattare
readMessage:    addi    $sp, $sp, -4            # Apro spazio nello stack per una parola
                sw      $ra, 0($sp)             # Salvo l'indirizzo di ritorno del chiamante

                jal     openFile_read           # Apre il file in solo lettura, il descrittore lo riceve dal main


                move    $a0, $v0                        # Passo il descrittore del file
                la      $a1, bufferReader       # Carico il buffer che conterra' il messaggio
                li      $a2, 255                        # Imposto la dimensione del buffer
                jal     readFile                # Leggo il file e carico il buffer dedicato

                lw      $ra, 0($sp)             # Reimposto il registro del chiamante
                addi    $sp, $sp, 4             # Dealloco spazio dello stack
                jr $ra

# readKey: Procedura dedicata alla lettura del file che contiene la CHIAVE di cifratura(decifratura)
# PARAMETRI :                   $a0 <-- DESCRITTORE DEL FILE
#
# Valore di ritorno:        void
# Il suo effetto e' quello di riempire il buffer con la chiave
readKey:                addi    $sp, $sp, -4            # Alloco spazio nel buffer per una parola
                sw      $ra, 0($sp)             # Salvo il rigistro di ritorno del chiamante

                jal     openFile_read           # Apro il file in lettura

                move    $a0, $v0                        # Salvo il descrittore del file per la prossima procedura
                la      $a1, bufferKey          # Carico il buffer che conterra' la chiave
                li      $a2, 4                  # Imposto la dimensione del buffer
                jal     readFile                # Vado alla procedura di lettura da file

                lw      $ra, 0($sp)             # Carico il registro di ritorno
                addi    $sp, $sp, 4             # Dealloco lo spazio della pila
                jr $ra                          # Torno al precedente Jal

# writeMessage : Procedura dedicata alla scrittura del file CIFRATO o DECIFRATO
# Parametri :   $a0 <-- descrittore del file da scrivere (l'etticheta che conitiene il percorso)
#
# Valore di ritorno:        void
writeMessage:   addi    $sp, $sp, -4            # Alloco spazio nello stack per una parola
                sw      $ra, 0($sp)             # Salvo il registro di ritorno del chiamante
                                                # $a0<--DESCRITTORE DEL FILE
                jal     openFile_write
                move    $a0, $v0                        # Passo il descrittore del file in $a0

                la      $a2, bufferReader       # Salvo l'indirizzo di bufferReader in $a2
                jal     bufferLength            # Per poi trovarne la lunghezza

                move    $a2, $v1                        # Recupero in $a2 il valore restituito da bufferLength
                la      $a1, bufferReader       # Salvo l'indirizzo di bufferReader in $a1
                jal     writeFile               # Per poi scrivere il buffer nel file

                li      $v0, 16                 # Chiamata a sistema di chiusura file
                move    $a0, $a1                        # Passo in $a0 l'indirizzo del file
                syscall
```

```
                lw      $ra, 0($sp)             # Reimposto il registro del chiamante
                addi    $sp, $sp, 4             # Dealloco spazio dello stack
                jr $ra                          # Torno al main

# openFile_read: Procedura che permette di aprire un file in solo lettura
# $a0: Descrittore del file
#
# Valore di ritorno :   $v0 <-- Indirizzo di memoria del buffer con i dati letti
openFile_read:  li      $v0, 13                 # Chiamata a sistema per apertura file
                li      $a1, 0                  # Flag di lettura
                li      $a2, 0                  # (Ignorato)
                syscall

                jr $ra

# openFile_write: Procedura che permette di aprire un file in solo scrittura
# $a0: Descrittore del file
#
# Valore di ritorno :   $v0 <-- Indirizzo di memoria del buffer con i dati letti
openFile_write: li      $v0, 13                 # Chiamata a sistema per apertura file
                li      $a1, 1                  # Flag di scrittura
                li      $a2, 0                  # (Ignorato)
                syscall

                move $v1, $v0                   # Salvo il percorso del file in $v1
                jr $ra

# readFile: Procedura per la lettura dei file
# $a0: Descrittore del file
# $a1: Registro che contiene l'indirizzo di partenza del buffer di riferimento
# $a2: Grandezza del buffer di riferimento
# VALORE DI RITORNO:        $v0 <-- REGISTRO DEL BUFFER CON I DATI LETTI
readFile:       li      $v0, 14
                syscall

                jr $ra

# WRITE-FILE: PROCEDURA PER SCRIVERE IL CONTENUTO NEL FILE
# $a0: Descrittore del file
# $a1: Registro che contiene l'indirizzo di partenza del buffer di riferimento
# $a2: Grandezza del buffer di riferimento
writeFile:      li      $v0, 15
                syscall

                jr $ra
#-------------------------------------------------------------------------------------------------------------------#
exit:           lw      $ra, 0($sp)
                lw      $s0, 4($sp)
                lw      $s1, 8($sp)
                lw      $s2, 12($sp)
                addi    $sp, $sp, 16            # Dealloco spazio dello stack per chiuderlo definitivamente

                li      $v0, 4                  # Visualizza il messaggio di terminazione del programma
                la      $a0, done               # "Operazione Terminata."

                syscall
```

```
li          $v0,10
syscall                         # terminazione del programma
```