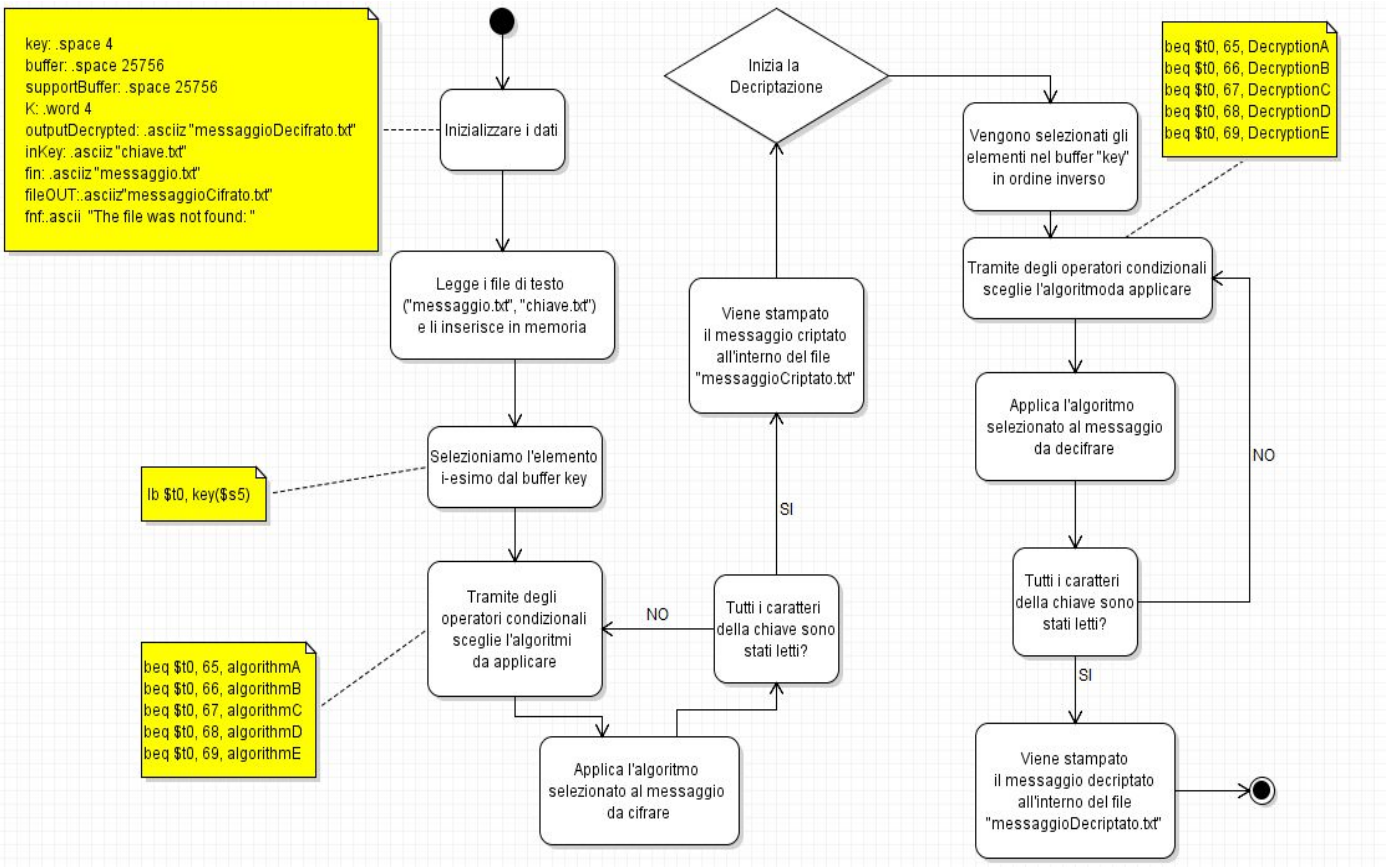
RELAZIONE DEL PROGETTO DI ARCHITETTURE DEGLI ELABORATORI

Autore: *Massimiliano Sirgiovanni*

Matricola: *6357449*

E-mail: *massimiliano.sirgiovanni@stud.unifi.it*

DESCRIZIONE DELLA SOLUZIONE ADOTTATA



Allocazione della memoria e dei dati:

Il programma, non appena viene eseguito, alloca della memoria statica per contenere il messaggio da criptare (**buffer**) e la chiave per la selezione degli algoritmi (**key**). Inoltre viene allocato un terzo buffer, **supportBuffer**, il quale risulterà utile per l'esecuzione degli algoritmi D ed E.

Successivamente vengono inizializzate la costante K, in questo caso pari a 4, e le stringhe contenenti i nomi dei file di testo usati dal programma(*"chiave.txt","messaggio.txt",*

“messaggioCifrato.txt” e “messaggioDecifrato.txt”). Infine, vengono inizializzate due stringhe contenenti dei messaggi di errore (*"The file was not found" e "An incorrect character was entered in the KEY"*). Il primo messaggio viene stampato nel caso in cui i file di input non dovessero essere trovati, mentre il secondo nel caso in cui la chiave dovesse contenere un carattere non corretto.

Lettura e scrittura dei file di testo:

Per poter sviluppare ciò che viene richiesto dal progetto bisogna inizialmente caricare, all’interno di due buffer differenti, la chiave per la selezione degli algoritmi ed il messaggio da criptare. Dopo aver riempito i buffer con le informazioni richieste, si passa alla selezione degli algoritmi.

Selezione degli Algoritmi:

Tramite una serie di operatori condizionali, viene selezionato quale algoritmo dovrà essere applicato al messaggio. La struttura della selezione di un algoritmo X, scritta in linguaggio ad alto livello, è la seguente:

```
int i = 0;
if ( key[i] = X ){
  algoritmoX();
  }
```

Di conseguenza, viene considerato l’elemento i-esimo del buffer key, dove i è compreso tra 0 e 3, e confrontato con il carattere rappresentante dell’algoritmo. All’interno del progetto, nella procedura **selectionAlgorithm**, sono presenti cinque operatori condizionali simili, uno per ogni algoritmo da implementare.

```
algorithmSelection:
                    li $t7, 0   # $t7 = 0;
                    bge $s5, 4, endSelection  # if($s5 >= 4) => jump to endSelection.
                    lb $t0, key($s5)       # $t0 = key[i] with  0<=i<4
                    beq $t0, 65, algorithmA  # if($t0 == A) => algorithmA; An equivale
                    beq $t0, 66, algorithmB  # if($t0 == B) => algorithmB;
                    beq $t0, 67, algorithmC  # if($t0 == C) => algorithmC;
                    beq $t0, 68, algorithmD  # if($t0 == D) => algorithmD;
                    beq $t0, 69, algorithmE  # if($t0 == E) => algorithmE;
                    bnez $t0, printError    # if($t0 != null) => printError
                    # In the event that $ t0 had passed all the "if" statements, or mu
endSelection:
                    move $v1, $s5
                    j print       # Once the algorithms to be applied to the message t

printError:
                    li  $v0, 4          # Print String Syscall
                    la  $a0, errMessage # Load Error String
                    syscall
                    j endSelection
```

A loro volta tutti gli algoritmi sono rappresentati da una o più procedure, le quali modificano il messaggio restituendolo criptato.

La **selezione per gli algoritmi di decriptazione** funziona praticamente allo stesso modo, le uniche differenze sono le procedure chiamate dagli operatori condizionali ed il fatto che i caratteri della chiave vengono letti in ordine inverso.

```
selectionDecryption:
                    li $t7, 0
                    lb $t0, key($s5)
                    beq $t0, 65, DecryptionA
                    beq $t0, 66, DecryptionB
                    beq $t0, 67, DecryptionC
                    beq $t0, 68, DecryptionD
                    beq $t0, 69, DecryptionE
                    j printDecrypted      # =
```

Algoritmo A:

Ora che è stato spiegato il funzionamento della selezione degli algoritmi, procediamo con ordine, analizzando lo sviluppo ed il funzionamento dell'algoritmo A.
L'algoritmo A deve aumentare di 4 il codice ASCII di ogni carattere presente nel messaggio. Quindi, per sviluppare questo algoritmo, si parte creando un ciclo che seleziona, una ad una, tutti i caratteri presenti nel buffer. Questo ciclo nel codice è rappresentato dalla procedura **algorithmA**. In linguaggio ad alto livello potrebbe essere tradotto con un *while:*

```
int i = 0;
while(buffer[i] != 0){
functionA(buffer[i]);
i++ ;
}
```

Come è possibile notare, all'interno del ciclo viene chiamato un metodo, tradotto nel codice in Assembly MIPS con una procedura. La procedura **functionA** viene chiamata solo se viene superata la condizione che l'elemento selezionato all'interno del buffer sia diverso da zero. In questo caso viene passato alla procedura tale elemento e quest'ultima viene chiamata.
La procedura **functionA**, come richiesto dall'algoritmo, aggiunge al codice ASCII, passato alla funzione, il numero 4. Successivamente viene effettuata l'operazione di modulo 256. In linguaggio ad alto livello il codice Assembly MIPS potrebbe essere tradotto in questo modo:

```
double k = buffer[i];
k = k + 4;
k = k % 256;
buffer[i] = k;
```

Per implementare l'operazione di modulo in Assembly MIPS si divide l'elemento del buffer, al quale è stato aggiunto il numero 4, per 256. Successivamente si prende il resto della divisione dal registro *hi*, e lo si carica nel buffer al posto dell'elemento preso in precedenza.

```
algorithmA:
          lb $s0, buffer($t7)    # $s0 = buffer[$t7];
          beqz $s0, endAlgorithmA # if($s0 = null) => endAlgorithmA; The algorithm terminates if the i-th element of the b
          move $a0, $s0     # Pass to the procedure funcitonA the parameter $s0
          jal functionA     # The functionA procedure is called in order to apply the function required by algorithm A
          addi $t7, $t7, 1  # $t7 = $t7 + 1;  The pointer represented by the register $t7 indicates the position within th
          j algorithmA      # Call the "algorithmA" procedure to create a cycle that will allow you to apply the "function

functionA:
          lw $t0, K    # $t0 = K; The constant K is equal to 4
          add $a0, $a0, $t0  # $a0 = $a0 + K;  $a0 was passed by the calling procedure
          li $t1, 256  # $t1 = 256;  We will need it to apply the form function
          divu $a0, $t1  # $a0 : $t1; the quotient is saved in the "lo" register while the rest in the "hi" register
          mfhi $a0   #  $a0 = hi;

          sb $a0, buffer($t7) # buffer[t7] = $a0

          jr $ra   # ou jump through the register and continue to execute the instructions within the "algorithmA" cycle

endAlgorithmA:
# Once the algorithm A is finished, the pointers are canceled
            li $t1, 0     #  $t1 = 0;
            li $t0, 0     # $t0 = 0;
            j endAlgorithm  # => endAlgorithm
```

L'algoritmo di decriptazione A risulta essere estremamente simile al suo alter ego per la criptazione. Infatti, l'unica differenza tra i due algoritmi è da ritrovarsi nella funzione che va applicata all'algoritmo. Poiché nell'algoritmo di criptazione è stata aggiunta al codice ASCII di ogni carattere la costante k, pari a quattro, nell'algoritmo di decriptazione sarà necessario sottrarla. Però questo non basta, di fatto, poiché si tratta di una sottrazione, la quale potrebbe come risultato un numero negativo, bisogna aggiungere al codice ASCII il numero 256. Tuttavia, questo non cambierà il risultato finale, poiché è necessario eseguire l'operazione di modulo 256 sul codice ASCII, il che significa che l'aggiunta di 256 sarebbe come aggiungere il numero 0 al codice ASCII *(256 mod 256 = 0)*.

```
DecryptionA:
# The decryption algorithm A is very similar to the decryption algorithm, in fact the ASCII code of each character of the message,
          lb $s0, buffer($t7)  # $s0 = buffer[$t7];
          beqz $s0, endDecryption  # if($s0 == null) => endDecryption;
          move $a0, $s0  # $a0 = $s0; The value of the register $a0 is passed to the "decrypA" procedure
          jal decrypA    # => decrypA; The "decrypA" procedure is called, which is the function to apply to the message
          addi $t7, $t7, 1  # $t7++;
          j DecryptionA     # => DecryptionA; The cycle is created which allows the decryption algorithm to be applied to all the


decrypA:
          lw $t0, K    # $t0 = K;  The constant K is equal to 4
          sub $a0, $a0, $t0  # $a0 = $a0 - $t0;
          li $t1, 256  # $t1 = 256; The number 256 is saved in the register $ t1, which will be used to apply the module function
          add $a0, $a0, $t1 # $a0 = $a0 + $t1;
          divu $a0, $t1   # $a0 : $t1; the quotient is saved in the "lo" register while the rest in the "hi" register
          mfhi $a0   # $a0 = hi; The result of the module function, that is the rest of the division made in the previous instruct

          sb $a0, buffer($t7)  # buffer[$t7]=$a0;

          jr $ra
```

Algoritmi B e C:

Gli algoritmi B e C sono molto simili all'algoritmo A. Infatti la funzione da applicare risulta essere la stessa, ed infatti nel codice viene richiamata la stessa procedura, mentre il ciclo da applicare risulta leggermente diverso.

Algoritmo B:

```
int i = 0;
while(buffer[i] != 0){
functionA(buffer[i]);
i= i + 2 ;
}
```

Algoritmo C:

```
int i = 1;
while(buffer[i] != 0){
functionA(buffer[i]);
i = i + 2;
}
```

E' possibile notare che il ciclo usato per l'algoritmo B risulta essere equivalente a quello che andrebbe usato per l'algoritmo C, per questo motivo, si evita ripetizione di codice e si richiama direttamente l'algoritmo B. L'unica differenza tra i due algoritmi è l'inizializzazione del puntatore. Infatti, nel momento in cui viene chiamata la procedura **algorithmC,** nel codice Assembly MIPS, viene inizializzato il puntatore ad uno e successivamente si chiama la procedura **"algorithmB".**

```
algorithmB:

        lb $s0, buffer($t7)       # $s0 = buffer[$t7];
        beqz $s0,endAlgorithmA    # if($s0 == null) => endAlgorithmA; The algorithm terminates if the i-th element of the buffer is null
        move $a0, $s0          # Pass to the procedure funcitonA the parameter $s0
        jal functionA     # The functionA procedure is called in order to apply the function required by algorithm A
        addiu $t7, $t7, 2  # $t7 = $t7 + 2; the algorithm should be applied only to characters present in even positions
        j algorithmB      # The loop is started to apply the algorithm B to all the characters of the message

############################################################

algorithmC:
        li $t7, 1      # $t7 = 1; The algorithm C requires to apply the algorithm A to the characters present in the odd positions. For t
        j algorithmB            # => algorithmB; The cycle used for the algorithm B is equivalent to that used for the algorithm C, code
```

Come per l'algoritmo A, le **decriptazioni di B e C** sono praticamente identiche alle loro controparti di criptazione. L'unica differenza è dovuta al fatto che la funzione richiamata dai due algoritmi è la funzione dell'algoritmo di decriptazione A (invece di aggiungere al codice ASCII la costante k, quest'ultima va sottratta).

```
DecryptionB:
        lb $s0, buffer($t7)    # $s0 = buffer[$t7]
        beqz $s0, endDecryption  # if($s0 == null) => endDecryption; The algorithm terminates if the i-th element of the buffer
        move $a0, $s0    # $a0 = $s0; The value of the register $a0 is passed to the "decrypA" procedure
        jal decrypA      # => decrypA; The "decrypA" procedure is called, which is the function to apply to the message
        addi $t7, $t7, 2  # $t7 = $t7 + 2; the algorithm should be applied only to characters present in even positions
        j DecryptionB  # => DecryptionB; The cycle is created which allows the decryption algorithm to be applied to all the cha

################################################################

DecryptionC:
        li $t7, 1 # $t7 = 1; The algorithm C requires to apply the algorithm A to the characters present in the odd positions.
        j DecryptionB  # => DecryptionB; The cycle used for the algorithm B is equivalent to that used for the algorithm C, cod
```

Algoritmo D:

Procedendo con l'analisi delle soluzioni adottate, si passa all'algoritmo D, consiste nell'invertire il messaggio da criptare. Per sviluppare questo algoritmo è stato usato un buffer di supporto, il quale ha la stessa identica grandezza del buffer usato per salvare il messaggio. Per prima cosa è necessario calcolare quale sia il numero di caratteri presenti nel messaggio da cifrare. Tale calcolo viene effettuato attraverso una procedura "count", la quale continua ad aumentare un puntatore finché non trova un elemento nel buffer pari a zero.

```
count:
            li $v1, 0  # $v1 = 0;
loopCount:
            lb $t2, buffer($v1) # $t2 = buffer[$v1];
            beqz $t2, endCount  # if($t2 == null) => endCount;
            addiu $v1, $v1, 1    # $v1 = $v1 + 1; The pointer
            j loopCount    # => loopCount; A cycle is created
endCount:
           jr $ra
```

La procedura *count* restituisce il numero di elementi presenti nel buffer, che verrà utilizzato dall'algoritmo D.
Per prima cosa, tramite un operatore condizionale, si controlla se il numero dei caratteri del messaggio è pari a zero. In questo caso, termina il ciclo per la criptazione e si iniziano ad eseguire le istruzioni necessarie per terminare l'algoritmo. Se, invece, si dovesse superare la condizione, allora si prenderebbero dal buffer, nel quale è contenuto il messaggio, l'elemento i-esimo e si caricherebbe nella posizione j-esima. Come è possibile notare, per eseguire questo algoritmo servono due puntatori, il puntatore *i* inizializzato dalla procedura count, ovvero pari al numero di elementi presenti nel buffer, ed il puntatore *j* inizializzato ad uno. Il puntatore *i*, durante il ciclo, viene fatto retrocedere, al contrario viene fatto aumentare il puntatore *j*. In linguaggio ad alto livello si potrebbe tradurre come:

```
int i = buffer.lenght;
int j = 0;
int k = 0;
while(i != 0){
k = buffer[ i ];
supportBuffer[ j ] = k
i -- ;
j ++ ;
}
```

IN ASSEMBLY MIPS:

```
funzioneD:
            beqz  $s3, reloadBuffer   # if($s3==0) => reloadBuffer; If the register $ s3, that is the pointer that starts
            sub $s3, $s3, 1          # $s3 = $s3 - 1;
            lb $t2, buffer($s3)       # $t2 = buffer[$t3];
            sb $t2, supportBuffer($t8)   #supportBuffer[$t8] = $t2; Starting from position 0, the buffer elements are loa
            addi $t8, $t8, 1          # $t8 = $t8 + 1
            j funzioneD               # A loop is created to apply the D algorithm to all the characters of the message
```

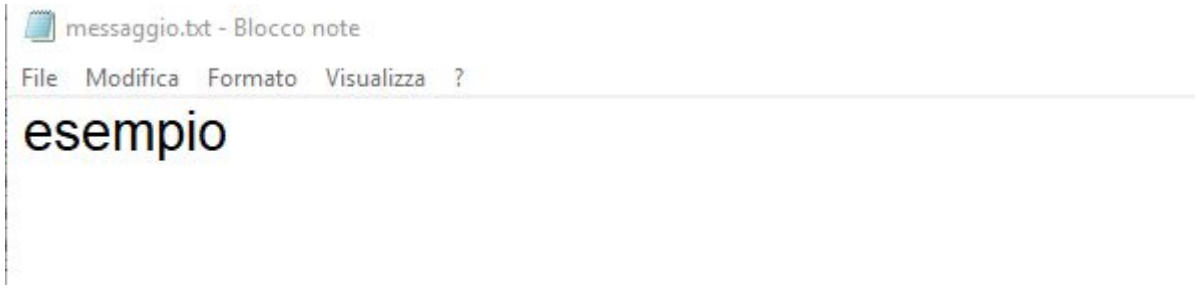
Una volta terminato il ciclo si prosegue a ricaricare il messaggio criptato sul buffer originale e a reinizializzare il buffer di supporto. Infine si azzerano tutti i registri precedentemente usati. Dato che l'algoritmo D risulta essere identico sia per la criptazione e che per la **decriptazione**, per evitare ripetizione di codice, si usa un operatore condizionale per stabilire quale dei due algoritmi si sta eseguendo. Per questo viene usato un registro ***$s7***, il quale viene inizializzato ad uno nel momento in cui dovesse partire la decriptazione.

beqz $s7, endAlgorithm
j FineDecriptazione

Algoritmo E:

Come è stato per l'algoritmo D, si inizia calcolando il numero di elementi presenti all'interno del buffer. Si richiama la procedura *"count"* attraverso un'istruzione jump and link. Dopo aver salvato nel registro $s3 il risultato ottenuto dalla procedura count, l'idea è quella di creare più procedure che lavorino assieme e che gestiscano i diversi tipi di caratteri da usare per l'algoritmo E.
Per rendere più chiaro il funzionamento dell'algoritmo E si prenda in considerazione questo esempio.

Messaggio da criptare:



Messaggio criptato:

Con la prima procedura, ***"startE"*** andremo a gestire gli elementi del messaggio da criptare, nel nostro caso le lettere della parola *"esempio"*. Per prima cosa, tramite operatore condizionale, si verificherà che il puntatore scelto per gestire il buffer abbia valore diverso dal numero massimo di elementi presenti nel buffer. In caso contrario l'algoritmo terminerebbe. Successivamente, ad uno ad uno, vengono caricate all'interno di un registro i caratteri che non sono stati già caricati in precedenza. Una volta caricato un elemento i-esimo del buffer in un registro, si verifica che questo elemento sia diverso da zero. Nel caso in cui questo elemento dovesse essere pari a zero, invece di terminare l'algoritmo, come fatto in precedenza con gli altri, si salta la posizione in cui è presente l'elemento i-esimo. Questo perché, come si potrà vedere anche nelle successive procedure, una volta utilizzato l'elemento i-esimo, ed i caratteri equivalenti, la sua posizione all'interno del buffer viene inizializzata a zero. Questo risulterà essere molto utile per evitare che uno stesso carattere venga selezionato più volte. Successivamente l'elemento selezionato viene inserito all'interno di un buffer di supporto (come la lettera "e" nell'esempio sopracitato).

```
algorithmE:
            jal count   # The "count" procedure is called which returns the number of elements present in the buffer
            move $s3, $v1  # $s3 = $v1; The number of elements returned by the procedure count in the register $s3 is saved
            li $t5, 0   # $t5 = 0; The registers to be used during the algorithm are reset
            li $t1, 0   # $t1 = 0;
            li $t9, 0   # $t9 = 0;
startE:
            beq $t1, $s3, endE    # if($t1 == $s3) => endE; $s3 is equal to buffer.length
            lb $s4, buffer($t1)   # $s4 = buffer[$t1];
            beqz $s4, skipPosition  # if($s4 == null) => skipPosition; If the i-th element is null, the current position is skipped.
            sb $s4, supportBuffer($t9) # supportBuffer($t9) = $s4;
            addiu $t9, $t9, 1  # $t9 = $t9 + 1; Move the pointer forward one position
            move $t5, $t1     # A copy of the pointer $t1 is created inside the register $t5 to be able to use the pointer, without r
```

La procedura *"searchSameChar"*, come suggerisce anche il nome, ha il compito di trovare, all'interno del buffer, i caratteri equivalenti a quello selezionato dalla procedura *"startE"*.
Il ciclo è, in realtà, molto semplice, infatti, oltre all'operazione per caricare un elemento i-esimo del buffer in un registro e all'operazione per fare avanzare il puntatore, contiene solo due operatori condizionali.
Il primo operatore verifica se l'elemento appena selezionato sia equivalente al carattere trovato nella procedura startE, in tal caso si avvierebbero le procedure per l'inserimento del carattere, secondo le regole stabilite dall'algoritmo E.

beq $t2, $s4, addDash *#Nel registro $s4 è contenuto il carattere trovato nella procedura startE e in $t2 è contenuto il carattere appena caricato dal buffer*

.

D’altra parte, il secondo operatore condizionale verifica se il buffer è già stato controllato interamente, per cui, necessariamente, non vi possono essere più caratteri simili. In questo caso si ritorna alla procedura “startE”. Ritornando alla procedura iniziale, passando però da una procedura che ci permetta di aggiungere uno spazio, viene selezionato un nuovo carattere e da lì riparte la ricerca di elementi equivalenti, ovvero la procedura *“searchSameChar”.*

bge $t5, **$s3**, **addSpace** *#$t5 rappresenta il puntatore per il buffer, mentre $s3 rappresenta il numero massimo di elementi presenti nel buffer, restituiti in precedenza dalla procedura “count”*

La procedura **addSpace** semplicemente inserisce nel buffer di supporto una spazio (con codice ASCII pari a 32) e aumenta di uno i puntatori del buffer e del supportBuffer. Fatte queste operazioni, tramite un’istruzione jump, riporta il programma alla procedura **startE**.

```
addSpace:
        addiu $s0, $zero, 32
        sb $s0, supportBuffer($t9)
        addiu $t9, $t9, 1
        addiu $t1, $t1, 1
        j startE
```

Dopo aver verificato cosa accade se il buffer viene interamente controllato, è necessario effettuare un passo indietro e tornare al primo operatore condizionale della procedura **searchSameChar**, ovvero nel caso in cui sia stato trovato un carattere equivalente all’elemento selezionato dal buffer. Secondo ciò che richiede l’algoritmo, non serve inserire il carattere ridondante appena trovato, ma bisogna inserire la sua posizione. Come visto in precedenza, se la condizione viene soddisfatta, ovvero che i due caratteri siano equivalenti, il programma proseguirà all’interno della procedura **addDash**. Questa procedura non fa altro che inserire un trattino (“ - “ con codice ASCII pari a 45) all’interno del buffer, nella prima posizione vuota, e da qui avvia la procedura per il calcolo della posizione.

```
addDash:
#This procedure is called when a character similar to the one pr
        li $t8, 45   # $t8 = 45; In the register $ t8 the ASCII
        sb $t8, supportBuffer($t9) # supportBuffer[$t9] = $t8;
        addiu $t9, $t9, 1          # $t9 = $t9 + 1;
```

Si prosegue con la procedura “AddNumber” nella quale si partirà a calcolare ed inserire nel buffer di supporto le cifre facenti parte della posizione del carattere.
L’algoritmo E ogni volta che viene applicato al messaggio ne aumenta la grandezza. Effettuiamo un breve calcolo per scoprire fino a che cifra può arrivare la grandezza del messaggio cifrato. Consideriamo il caso peggiore, ovvero quando l’algoritmo E viene applicato quattro volte ad un messaggio di 128 caratteri tutti distinti tra loro (*Si considerano i caratteri distinti poiché quando questi ultimi si ripetono non è necessario riscrivere, prima delle posizioni, il simbolo del carattere*). Per effettuare questo calcolo bisogna considerare che all’aumento delle cifre numeriche della posizione del carattere, aumentano anche le

celle occupate nel buffer. Per esempio se la posizione della lettera x dovesse essere zero allora occuperà quattro celle(e-0), mentre se dovesse essere la posizione 10 allora occuperebbe cinque celle (e-10). Inoltre c'è da considerare che al totale andrebbe sottratto il numero 1 poiché l'ultima sequenza di celle non deve contenere lo spazio finale.

Applicazione dell'algoritmo E	Numero di caratteri del messaggio da cifrare	Calcolo delle cifre
Prima	128	4*10 = 40 5*90 = 450 6*28 = 168 tot. = 658 - 1= 657
Seconda	657	4*10 = 40 5*90 = 450 6*557 = 3.342 tot. = 3.832 - 1 = 3.831
Terza	3.831	4*10 = 40 5*90 = 450 6*900 = 5.400 7*2831 = 19.817 tot. = 25.707 - 1 = 25.706
Quarta	25.706	4*10 = 40 5*90 = 450 6*900 = 5.400 7*24.706 = 172.942 tot. 178.832 - 1= 178.831

In realtà, il calcolo è approssimato per eccesso, poiché dopo la prima iterazione dell'algoritmo, e anche per le successive, è impossibile che tutti caratteri siano diversi tra loro. Questo perché, essendo già stato applicato l'algoritmo, all'interno del messaggio saranno presenti trattini, spazi e cifre numeriche che sicuramente si ripeteranno più volte. Perciò la lunghezza del buffer e del buffer di supporto dovrà essere pari a 178.831. Tuttavia non è necessario, per il calcolo delle cifre, arrivare fino alle centinaia di migliaia. Infatti nell'ultima applicazione dell'algoritmo si prenderà in esame il messaggio criptato dall'algoritmo E nella sua terza iterazione, per cui la posizione massima che potrà raggiungere sarà 25.706, ovvero le cifre da trascrivere nel messaggio raggiungeranno al massimo le decine di migliaia.
Ora che abbiamo ottenuto l'informazione di cui avevamo bisogno, possiamo proseguire con il calcolo delle cifre. Quindi, per ottenere i numeri da inserire nel bufffer, si divide la cifra che rappresenta la posizione dell'elemento selezionato per i seguenti numeri: 10000, 1000, 100, 10 e 1 con il medesimo ordine. Da queste divisioni otteniamo tutte i numeri che da inserire nel buffer di supporto per indicare la posizione dell'elemento preso dal buffer.

```
addNumber:
            move $t4, $t5   # $t4 = $t5;
            li $t2, 10000   # $t2 = 10000;
            div $t4, $t2    # $t4 : $t2 =>
            jal operAddNumb # => operAddNu
            move $s6, $v1   # $s6 = $v1; T
            beq $s6, 48, calculateNumber
            addi $t9, $t9, 1  # $t9 = $t9

calculateNumber:
#This procedure allows, starting from the
             divu $t2, $t2, 10  # $t2 = $t
             div $t4, $t2   # The position
             jal operAddNumb   # We call t
             move $s6, $v1     # $s6 = $v1
             beq $t2, 1, endCalculate  # i
             bne $s6, 48 addPosition  # if
             sub $s2, $t9, 1   # $s2 = $t9
             lb $s1, supportBuffer($s2)
             beq $s1, 45, calculateNumber
             addiu $t9, $t9, 1 # $t9 = $t9
             j calculateNumber # => calcul
```

Tuttavia questi numeri non possono essere inseriti, così come sono, all'interno del buffer di supporto, ma vanno tradotti in ASCII. E qui che entra in azione la procedura ***"operAddNumb"***, che come si può osservare dal codice sopra riportato viene chiamata con un istruzione ***jump and link***. Tale procedura viene chiamata dopo la divisione della posizione per uno dei numeri sopracitati e come prima cosa sposta nel registro **$v1** il quoziente della divisione, presente nel registro **lo**. Successivamente a questo numero viene aggiunto il numero 48, poiché rappresenta il codice ASCII dello zero, in questo modo si ottiene il codice ASCII del quoziente della divisione, salvando il tutto nel registro $v1. Dopo aver inserito nel buffer il contenuto del registro $v1 si sposta il resto della divisione, salvato nel registro **hi**, all'interno del registro **$t4** per poter continuare con le divisioni.

```
operAddNumb:
# Operation that allows us to insert the figures just found
         mflo $v1     # $v1 = lo; The result of the previously ma
         addi $v1, $v1, 48   # $v1 = $v1 + 4; The number 48 is ad
         sb $v1, supportBuffer($t9)  # supportBuffer[$t9] = $v1;
         mfhi $t4    # $t4 = hi; We retrieve the rest of the divi
         jr $ra
```

Terminati i calcoli da effettuare, all'interno della procedura ***"calcolateNumber"***, un operatore condizionale farà spostare il programma nella procedura ***"endCalculate"***. In questa procedura, oltre a gestire il puntatore ed a reinizializzare i registri usati nelle procedure di calcolo, viene azzerata nel buffer la cella dove era presente l'elemento precedentemente selezionato nella procedura ***"searchSameChar"***. Questa istruzione è importante perché rende più semplice la ricerca di caratteri, permettendo, come già visto in

precedenza, nella procedura “startE”, di inserire un operatore condizionale che salti le celle con all’interno valore numerico pari a zero.

```
endCalculate:
# All the digits of the position
            addiu $t9, $t9, 1
            li $s1, 0  # $s1 = 0;
            li $s2, 0  # $s2 = 0;
            li $t2, 0  # $t2 = 0;
            li $s6, 0  # $s6 = 0;

replaceWithZero:
          sb $zero, buffer($t5)
          j searchSameChar
```

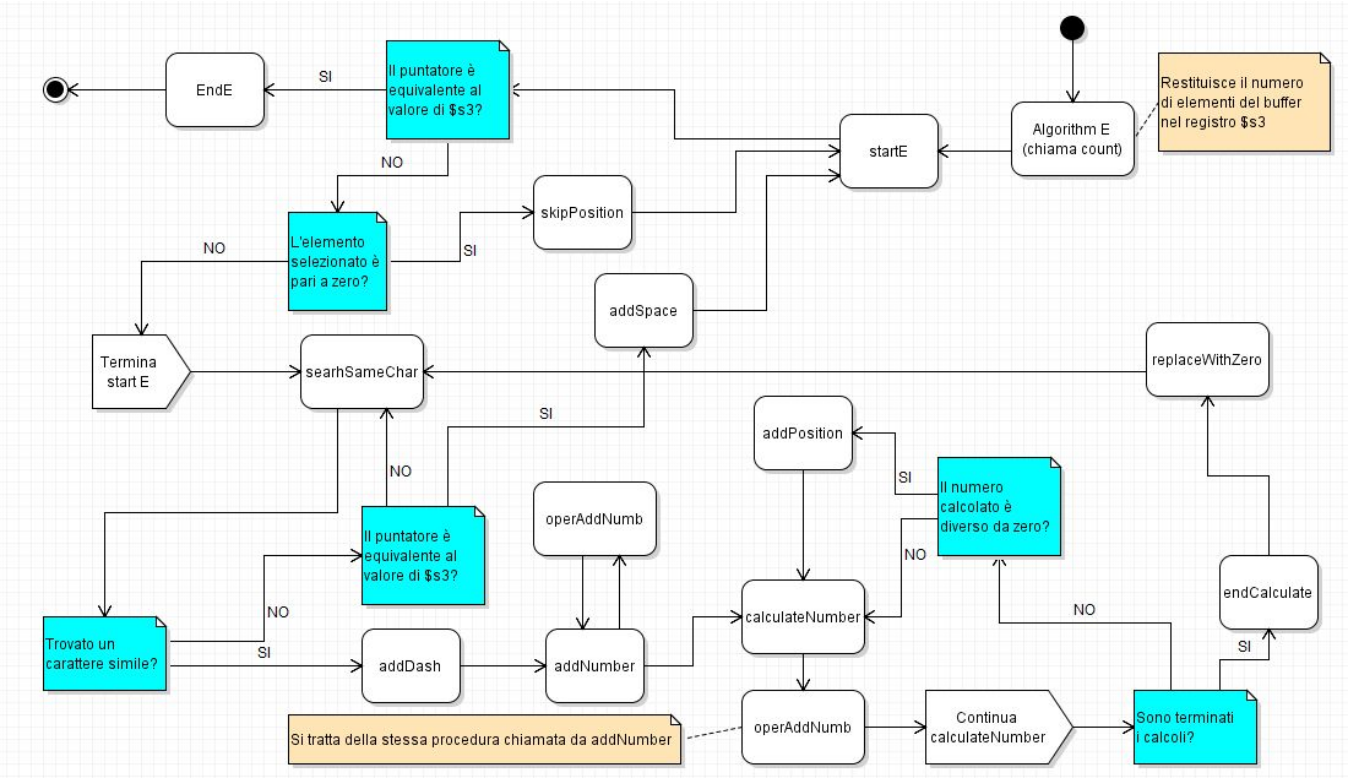
Una volta che l’algoritmo E è stato applicato a tutti i caratteri del messaggio, ovvero il puntatore del buffer è uguale al totale degli elementi presenti in quest’ultimo, può terminare. L’algoritmo termina con un operatore condizionale presente nella procedura ***“startE”*** che nel momento in cui dovesse essere rispettata la condizione fa proseguire il programma dalla procedura ***“endE”***.

beq $t1, $s3, endE

La procedura endE elimina l’ultimo spazio all’interno del buffer di supporto, poiché ogni volta che viene applicato l’algoritmo E ad un determinato carattere va inserito uno spazio, ma non è necessario per l’ultima iterazione. Inoltre endE reinizializza i registri usati e richiama le procedure per l’inserimento del messaggio criptato nel buffer e per ripulire il buffer di supporto.

```
endE:
      sub $t9, $t9, 1    # $t9--; The pointer is moved back one position
      sb $zero, supportBuffer($t9) # supportBuffer[$t9] = 0;  Since an extra
      li $t9, 0  # $t9 = 0; The registers used for the algorithm E are reset
      li $t8, 0  # $t8 = 0
      li $t5, 0  # $t5 = 0
      li $s3, 0  # $s3 = 0
      li $t1, 0  # $t1 = 0
      li $v1, 0  # $v1 = 0
      j reloadBuffer     # => reloadBuffer
```

Infine, tutti gli algoritmi terminano nella procedura ***“endAlgorithm”,*** la quale aumenta il puntatore della chiave e chiama un’istruzione *jump* per tornare alla selezione degli algoritmi.



i riquadri in azzurro rappresentano gli operatori condizionali

Passiamo ora **all'algoritmo di decriptazione E,** il quale, proprio come il suo gemello di criptazione, risulta essere piuttosto complesso. Si parte selezionando il primo elemento dal buffer, il quale è sicuramente un carattere da inserire e non una posizione o un trattino. Successivamente si fa partire una procedura, ***"loopE"***, che seleziona un elemento dal buffer e nel caso sia un numero in codice ASCII lo trasforma nel numero decimale corrispondente.

```
loopE:
        addiu $s4, $s4, 1
        lb $s0, buffer($s4)
        move $a1, $s0
        jal checkEndLoop
        move $s0, $a1
        sub $s0, $s0, 48
        j checkDecimal
```

Per controllare se si tratta di un numero si chiama la procedura ***"checkEndLoop"***. In quest'ultima procedura vengono effettuati tre controlli. Se l'elemento è un trattino viene fatta ripartire dall'inizio la procedura loopE.

beq $a1, 45, **loopE**

Se invece l'elemento salvato nel registro $a1 è uno spazio allora si fa ripartire l'intero algoritmo scegliendo un nuovo carattere per il loop.

beq $a1, 32**, restartLoop**

Se l'elemento è pari a zero allora termina l'algoritmo.

beqz $a1, endDecrypE

La procedura ***"restartLoop"*** aumenta solo il puntatore e richiama la procedura ***"startDecrypE".*** Invece la procedura ***"endDecrypE"*** reinizializza i registri usati dall'algoritmo e chiama la procedura ***"clearBuffer",*** che serve per reinizializzare il buffer col messaggio da decriptare per poi riempirlo col messaggio decriptato.
Tornando alla procedura loopE, una volta superati tutti gli operatori condizionali, la procedura chiama, attraverso un'istruzione *jump*, la procedura ***"checkDecimal".***
Tale procedura, insieme alla procedura ***"loopCheckDec"*** direttamente collegata, serve per calcolare la posizione del buffer in cui salvare il carattere, selezionato dal buffer di supporto all'inizio dell'algoritmo. Precedentemente, nella procedura ***"loopE"*** era stato selezionato un numero, il quale si trovava subito dopo un trattino. Nella procedura *"loopCheckDec"* si verifica se quel numero è un'unità o un altro tipo di cifra (decine, centinaia, migliaia o decine di migliaia). Nel caso in cui si tratti di un'unità, il carattere salvato nel registro $s2, dalla procedura "startDecrypE", viene salvato nella posizione indicata dalla cifra unitaria e viene richiamata la procedura "loopE". In caso contrario si applica questa formula finché non si trova la cifra unitaria.

char j = buffer[i-2];
int k = buffer[i];

```
while (isNumber(buffer[i+1])){
  k = k*10 + buffer[i+1];
  i++;
}
```

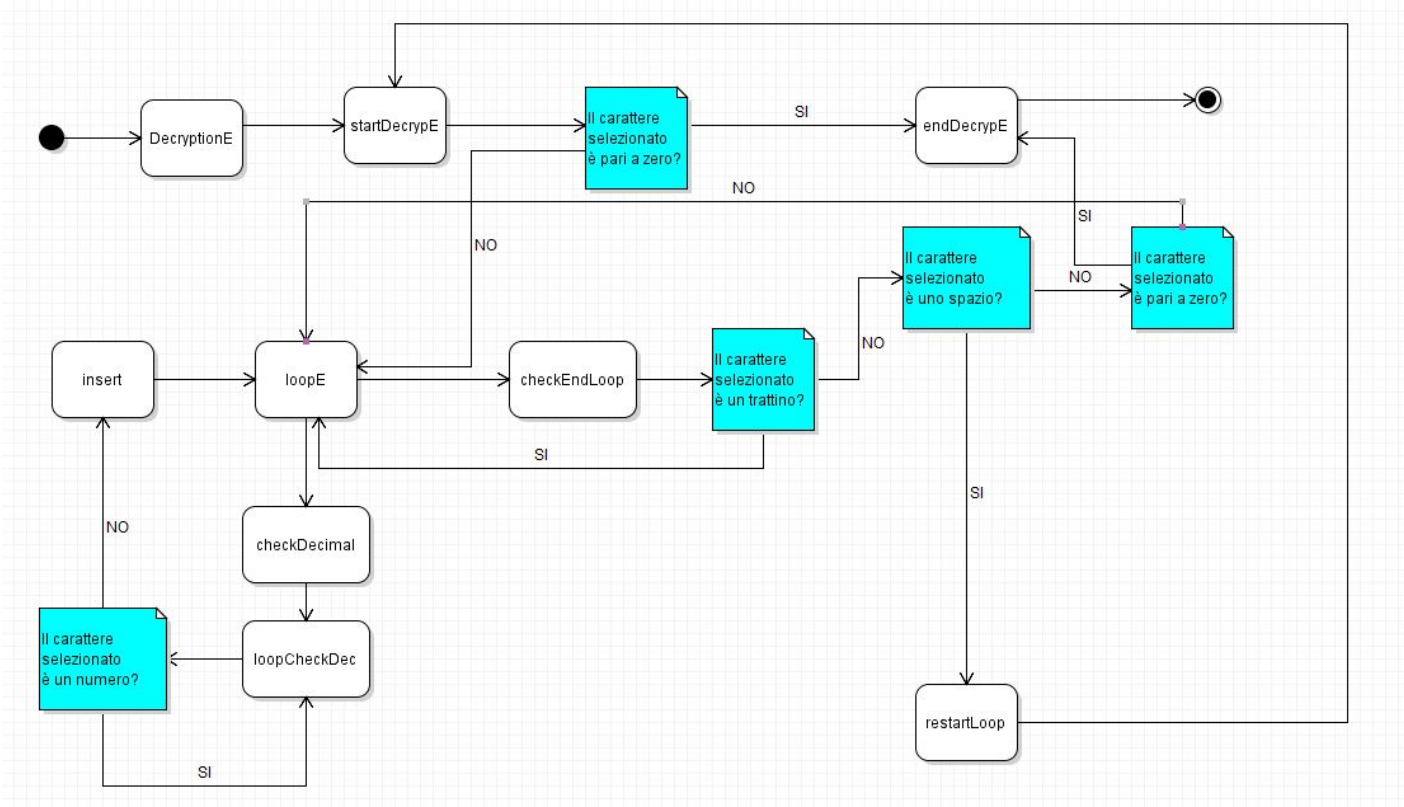
supportBuffer[k] = j;

```
checkDecimal:
            move $t9, $s4     # $t9 = $s4; Save the pointer in another register

loopCheckDec:
            addiu $t9, $t9, 1    # $t9++; The pointer is advanced one position
            lb $s1, buffer($t9)  # $s1 = buffer[$t9];
            bge $s1, 58, insert  # if($s1 >= 58) => insert; If the character sav
            ble $s1, 47, insert  # if($s1 <= 47) => insert; If the character sav
            sub $s1, $s1, 48     # $s1 = $s1 - 48; Since $ s1 has exceeded the t
            mul $s0, $s0, 10     # $s0 = $s0 * 10;
            add $s0, $s0, $s1    # $s0 = $s0 + $s1;  Remember that the number in
            addiu $s4, $s4, 1     # $s4++; The pointer is advanced one position
            j loopCheckDec        # A cycle is created

insert:
            sb $s2, supportBuffer($s0)  # supportBuffer[$s0] = $s2;
            j loopE        # Return to the previous cycle
```

Una volta decriptati tutti gli elementi del buffer, l'algoritmo termina grazie a uno di due operatori condizionali, il primo presente nella procedura startDecrypE ed il secondo nella procedura checkEndLoop. Entrambi gli operatori chiamano la procedura endDecrypE, la quale reinizializza i registri usati e chiama le procedure necessarie per reinizializzare i buffer e ricaricare il messaggio decriptato nel buffer originale. Infine, come tutti gli algoritmi di decriptazione, viene chiamata la procedura ***"endDecryption",*** la quale fa arretrare il puntatore usato per il buffer key.



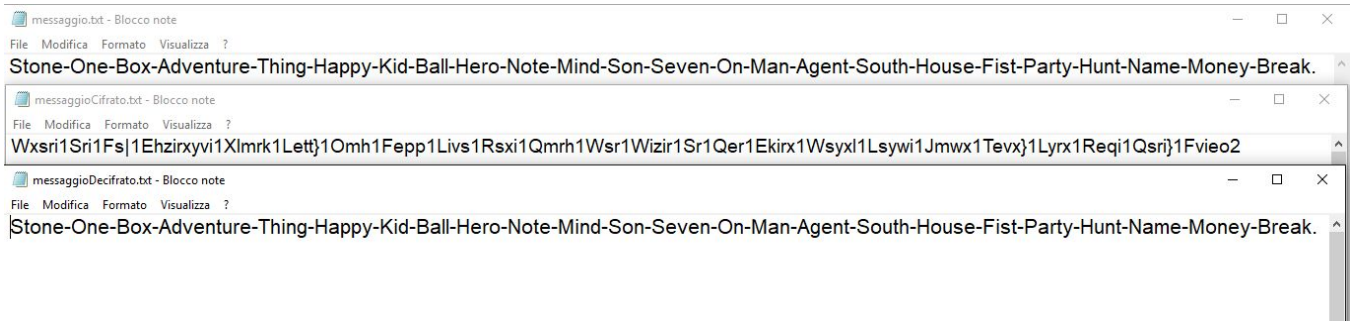
Scrittura nei file di output

Le ultime operazioni che il programma deve eseguire sono quelle di scrittura all'interno dei due file di output ("messaggioCifrato.txt" e "messaggioDecifrato.txt"). La scrittura del messaggio cifrato viene effettuata subito dopo la fine del processo di criptazione, ma prima del processo di decriptazione. Tramite l'utilizzo delle syscall fornite da QTSPIM si trascrive il contenuto del buffer nel file di testo. Al termine di tali istruzioni viene chiamata, tramite funzione jump, la procedura "startDecryption". Il processo di scrittura del messaggio decifrato è equivalente, però viene effettuato alla fine del processo di decriptazione e, una volta terminato, chiama la procedura ***"exit"***, che termina il programma.

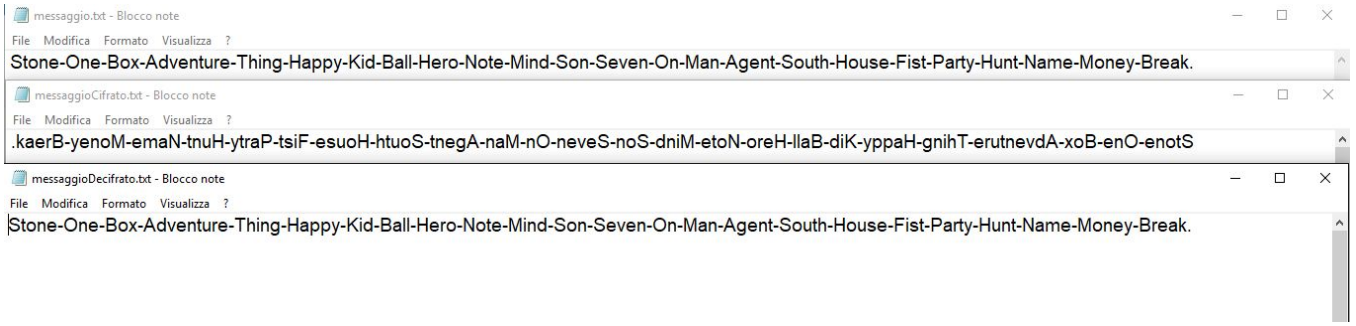
Test di correttezza

Effettuiamo alcuni test sul programma per verificarne l'effettiva correttezza. Per effettuare questi test è stato scelto un messaggio di 128 caratteri, numero massimo di caratteri richiesti dal testo del progetto, tuttavia è possibile decifrare messaggi anche molto più grandi. Se si decide di applicare esclusivamente gli algoritmi A, B, C e D si può arrivare ad un massimo di 178.831 caratteri. Il problema potrebbe sorgere quando viene applicato l'algoritmo E, il quale, come visto anche grazie alla tabella a pagina 10, aumenta esponenzialmente la grandezza del messaggio. Tuttavia dato che il calcolo effettuato per stabilire la grandezza del buffer è approssimato per eccesso, anche nel caso in cui venga applicato quattro volte l'algoritmo E, si potrebbe sforare il limite dei 128 caratteri. Partiamo con alcuni casi semplici, nei quali viene eseguito un solo algoritmo per volta.

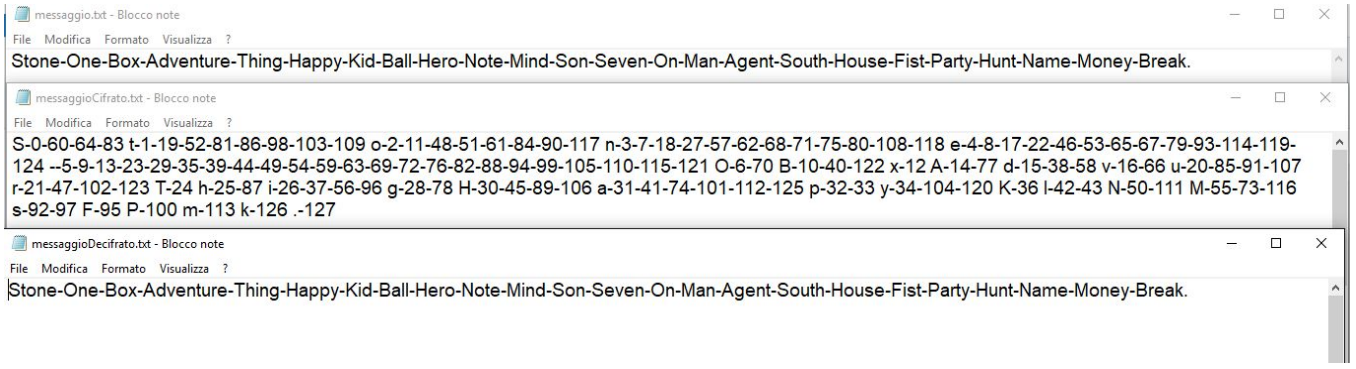
Chiave: A



Chiave: D



Chiave: E



Passiamo successivamente ad alcune prove nel caso medio, ovvero l'applicazione di più algoritmi diversi tra loro.

Chiave: ADEB

messaggio.txt - Blocco note di Windows

File Modifica Formato Visualizza ?

```
Stone-One-Box-Adventure-Thing-Happy-Kid-Ball-Hero-Note-Mind-Son-Seven-On-Man-Agent-South-House-Fist-Party-Hunt-Name-Money-Break.
```

messaggioCifrato.txt - Blocco note di Windows

File Modifica Formato Visualizza ?

```
6-4 s-5 i-6-5512:-9318:-=6$i1318117-7414<-:0166-;4185-509-514-51=-527 z-8-65184-50: J-9-<71157$116116-57126-68137-79149-91159-98168-:8177
-;8187-<8196-=81144115411581162$}17127-=3$r1911=-87156-9615=-:5174-504-50=-524-528 w-5013;-8316:-;617=-51:-529 U-51158-;2$q118 V-5617; |-
58128-69145-86179-50<-52: }-6013:-821147$L125-78186-=7$T12; {-70139 q-71175-=01141$J136 p-801142$W148-:316;-52; o-8919= l-901153$S15;-525
~-:11151$h16=-<91152$p188-<5$O195 x-=4199 \-507 €-519
```

messaggioDecifrato.txt - Blocco note di Windows

File Modifica Formato Visualizza ?

```
Stone-One-Box-Adventure-Thing-Happy-Kid-Ball-Hero-Note-Mind-Son-Seven-On-Man-Agent-South-House-Fist-Party-Hunt-Name-Money-Break.
```

Chiave: CDAE

messaggio.txt - Blocco note di Windows

File Modifica Formato Visualizza ?

```
Stone-One-Box-Adventure-Thing-Happy-Kid-Ball-Hero-Note-Mind-Son-Seven-On-Man-Agent-South-House-Fist-Party-Hunt-Name-Money-Break.
```

messaggioCifrato.txt - Blocco note di Windows

File Modifica Formato Visualizza ?

```
6-0 o-1 i-2-3-13-26-81-86-96-105-119-123 z-4-61-80-106-111 F-5-87-117 5-6-12-22-28-58-64-68-78-88-92-98-104-114-118-122 }-7-20-23-36-4
-34-48-60-62-71-74-101-110 r-9-19-47-59-65-109 w-10-35-66-76-116 Q-11 u-14 e-15-53 V-16 1-17-33-39-45-51-55-73-83 |-18-24-46-108-115-
v-25-52-56-70-100-120-124 T-27 x-29-41-75-94 {-30 N-32 s-37-43-79-125 P-38-82 p-40-85-102 [-44 k-49-99 I-50 U-54-72 S-57-121 W-63-67-1
R-77 t-84-95 q-90 O-91 X-103 y-107 l-112 E-113
```

messaggioDecifrato.txt - Blocco note di Windows

File Modifica Formato Visualizza ?

```
Stone-One-Box-Adventure-Thing-Happy-Kid-Ball-Hero-Note-Mind-Son-Seven-On-Man-Agent-South-House-Fist-Party-Hunt-Name-Money-Break.
```

Chiave: EEBC

messaggio.txt - Blocco note di Windows

File Modifica Formato Visualizza ?

```
Stone-One-Box-Adventure-Thing-Happy-Kid-Ball-Hero-Note-Mind-Son-Seven-On-Man-Agent-South-House-Fist-Party-Hunt-Name-Money-Break.
```

messaggioCifrato.txt - Blocco note di Windows

File Modifica Formato Visualizza ?

```
W14$115171:1=15815:15=16616916<17517918518718:18=19619919<1:51:;1:=1;51;81;;1<41<71<:1<=1=61=91==1549154;154=15561559155<156515
68156;15741577157;1585158:158;158=15951598159;15:415:715::15:=15;615;915;<15<515<815<;15=415=715=:15==1646164:1654165816641666166;
16741677167=1688168;16961699169<16:716::16;516;816;;16<416<:16<=16=616=:1746174;17541759175<17651768176=1776177;17841787178:1796
1799179<17:517:917:=17;917;<17<717<:17=417=:18451848184=1856185<18651868187418771 87<1887188=189918:5$4161917717;1:41=81=;164816
4=1668166=167616;716<616=8177=178<17:717<<17=718551889188:
$:181;16;1971<51<8155;1566156915;=15<615<=166516:916:;16:<175;1767176:178=17=<186;189<
$81<18;19;154:155:157:158815:;15:<15;415;815=<1675168:16=417481785179:17:417<917<=18461849$<15416716:17418<19:1;71<91=71=<1546154<
15=515=815=9169;16:416;91755177517781788$71551781:<156415761597159:15:515:815<4169:16==175=177<179717;:17;=17<417<817=;184:18671
896$
$15617=1:915471589165<1669167;1686169416:516:=16<8174417491757176;1779179417;717<517=817==184;185:186<187:1885188;1897189=
$x157$515915;16817617:1881891951981:61:71;61<<1=:1544154515541578157 9157<157=158615961647164;164<16551656165 9165;166<16781684
1689169716:816;=16<516<<16=716=;178;1798179;17:617:817::17:;17;417<;17=518571858185918691 86:188818941895189:18:6$=15<16=17<19=156=
157515841594159=15:915;515;;15<715=;1644164516;<1769178918751878187=
$91641941;<1=5155=1567158<15:615;715;:164916571698169=16;:174=1766178617;61854185=18641884$61651861;91<61557155815871599159<15
<:15=6165:1679167:168516;616<;16=916=<1747174<175:177417:<17;517;;17=618471876189;18:7$s184$;1:81;41;:1;=1<;1=41555156:156<15<915<<1
667168<168=16<716=5175617641777179=1866187918:8$r1::$i1548$S165=$F166:$|167<$E1687$h1695$z16:6$y16;4$v16<9$X1745$l174:
$m1758$k176<$L177:$e1795$t17;8$}17<6$O17=9$p1844$R184<$Q185;$w186=$J187;$T1886$q188<$o1898$218:4
```

messaggioDecifrato.txt - Blocco note di Windows

File Modifica Formato Visualizza ?

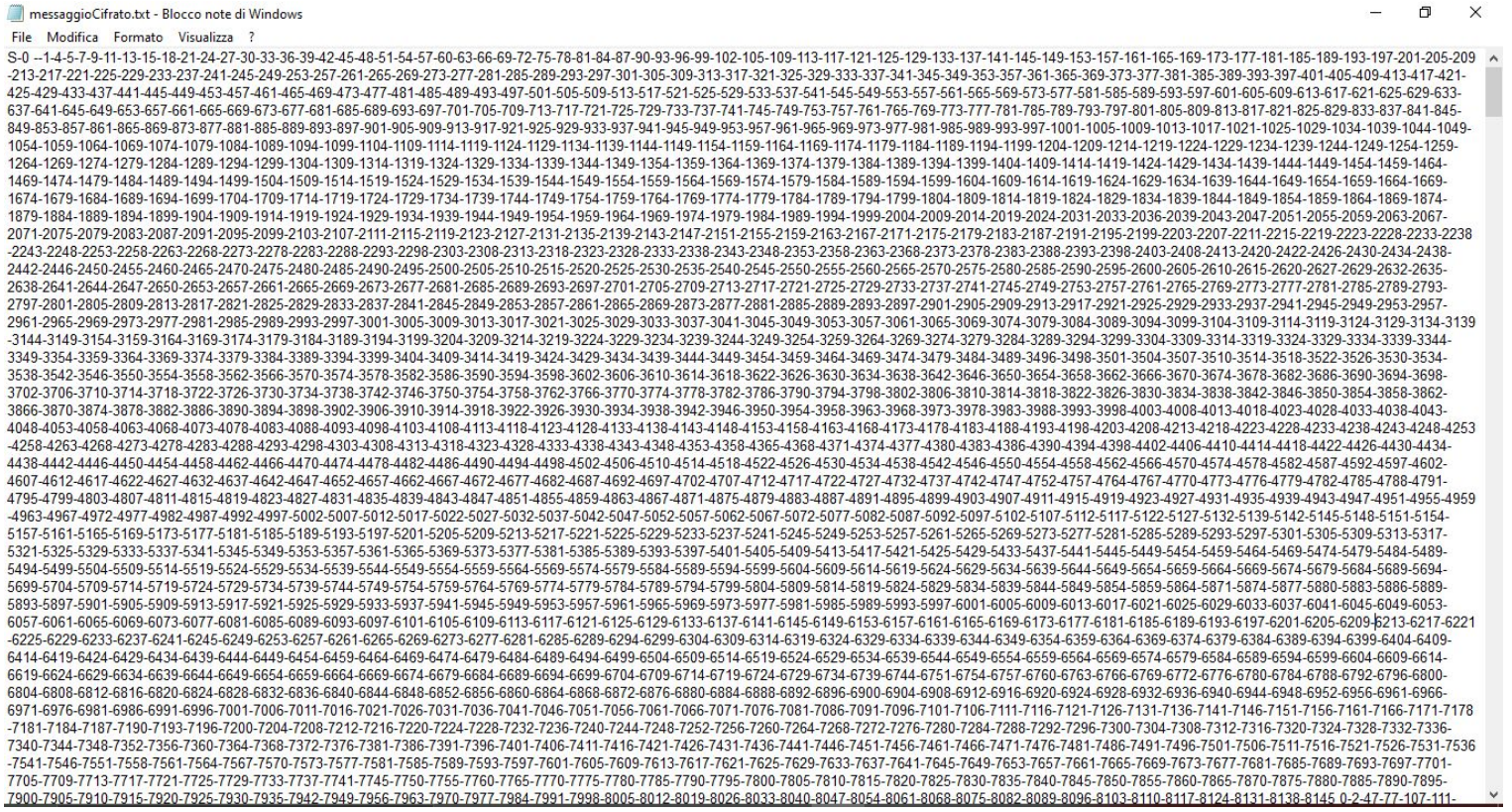
```
Stone-One-Box-Adventure-Thing-Happy-Kid-Ball-Hero-Note-Mind-Son-Seven-On-Man-Agent-South-House-Fist-Party-Hunt-Name-Money-Break.
```

Chiave: ABCD

Ed infine vediamo il caso pessimo, ovvero quando viene applicato quattro volte l'algoritmo E.

Chiave: EEEE

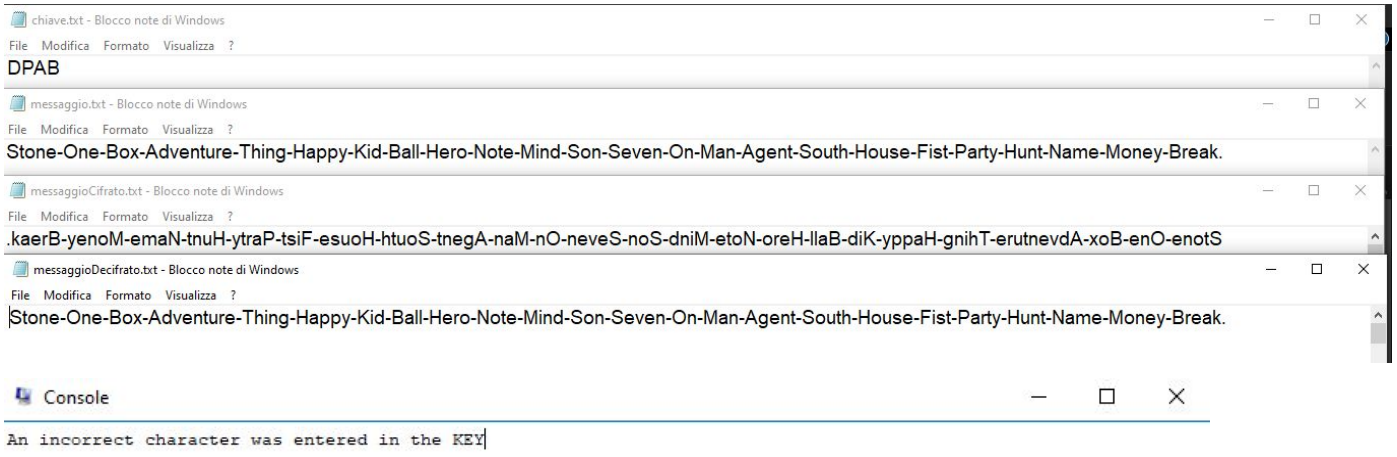
Poiché il messaggio criptato risultava essere troppo grande è stato necessario rimpicciolire la grandezza dei caratteri. Tuttavia qui di seguito è possibile vedere, più in grande, la parte iniziale del messaggio cifrato e la parte finale.



L’ultima funzionalità che vorrei far notare è la seguente: eseguendo un test con una chiave valida fino ad un certo punto, ma successivamente illegale, il programma eseguirà la criptazione fino al punto in cui la chiave è ancora legittima. Per quanto riguarda la decriptazione, il programma eseguirà la decriptazione solo per gli algoritmi che sono stati applicati al messaggio, perciò solo quelli presenti prima del carattere illegittimo. Inoltre

stampa un messaggio di errore *("An incorrect character was entered in the KEY")* per avvisare l'utente di questa anomalia.

Chiave: DPAB



CODICE SORGENTE IN ASSEMBLY MIPS

Le parole contrassegnate in verde rappresentano i commenti

```
# Author: Massimiliano Sirgiovanni                    Date: 04/09/2019
# email: massimiliano.sirgiovanni@stud.unifi.it

.data

    #Buffers
    key: .space 4
    buffer: .space 178831
    supportBuffer: .space 178831

    K: .word 4

    #File di Input e Output
    outputDecrypted: .asciiz "messaggioDecifrato.txt"
    inKey: .asciiz "chiave.txt"
    fin: .asciiz "messaggio.txt"
    fileOUT: .asciiz   "messaggioCifrato.txt"
    fnf:        .ascii  "The file was not found"
    errMessage: .ascii "An incorrect character was entered in the KEY"

.text

.globl main
main:

start:
        jal inputKey   # Using the jump and link instruction, select the procedure that will read the key from the text file
```

```
        jal inputFile  # As previously done for the key, the message to be encrypted is read from the text file

        li $t6, 0     # $t6 = 0; Reset the registers $ t7 and $ t6 used previously to load the message on the buffer


algorithmSelection:
                li $t7, 0   # $t7 = 0;
                bge $s5, 4, endSelection  # if($s5 >= 4) => jump to endSelection. The maximum length allowed for the key is 4
                lb $t0, key($s5)       # $t0 = key[i] with  0<=i<4
                beq $t0, 65, algorithmA  # if($t0 == A) => algorithmA; An equivalent instruction must be executed for the other
algorithms
                beq $t0, 66, algorithmB  # if($t0 == B) => algorithmB;
                beq $t0, 67, algorithmC  # if($t0 == C) => algorithmC;
                beq $t0, 68, algorithmD  # if($t0 == D) => algorithmD;
                beq $t0, 69, algorithmE  # if($t0 == E) => algorithmE;
                bnez $t0, printError    # if($t0 != null) => printError
                # In the event that $ t0 had passed all the "if" statements, or must be zero, because the message is finished, or there is
an error in the key
endSelection:
                move $v1, $s5
                j print       # Once the algorithms to be applied to the message to be encrypted are finished, the message is printed in
the "messageCriptato.txt" output file

printError:
                li          $v0, 4                # Print String Syscall
              la          $a0, errMessage   # Load Error String
              syscall
              j endSelection

##################################################
algorithmA:
        lb $s0, buffer($t7)    # $s0 = buffer[$t7];
        beqz $s0, endAlgorithmA # if($s0 = null) => endAlgorithmA; The algorithm terminates if the i-th element of the buffer is null
        move $a0, $s0     # Pass to the procedure funcitonA the parameter $s0
        jal functionA     # The functionA procedure is called in order to apply the function required by algorithm A
        addi $t7, $t7, 1  # $t7 = $t7 + 1;  The pointer represented by the register $t7 indicates the position within the message
        j algorithmA      # Call the "algorithmA" procedure to create a cycle that will allow you to apply the "functionA" function to
the whole message

functionA:
        lw $t0, K    # $t0 = K; The constant K is equal to 4
        add $a0, $a0, $t0  # $a0 = $a0 + K;  $a0 was passed by the calling procedure
        li $t1, 256  # $t1 = 256;  We will need it to apply the form function
        divu $a0, $t1  # $a0 : $t1; the quotient is saved in the "lo" register while the rest in the "hi" register
        mfhi $a0   #  $a0 = hi;

        sb $a0, buffer($t7) # buffer[t7] = $a0

        jr $ra   # ou jump through the register and continue to execute the instructions within the "algorithmA" cycle

endAlgorithmA:
# Once the algorithm A is finished, the pointers are canceled
          li $t1, 0     #  $t1 = 0;
```

```
        li $t0, 0     # $t0 = 0;
        j endAlgorithm  # => endAlgorithm
############################################################
algorithmB:

        lb $s0, buffer($t7)      # $s0 = buffer[$t7];
        beqz $s0,endAlgorithmA    # if($s0 == null) => endAlgorithmA; The algorithm terminates if the i-th element of the buffer is
null
        move $a0, $s0         # Pass to the procedure funcitonA the parameter $s0
        jal functionA     # The functionA procedure is called in order to apply the function required by algorithm A
        addiu $t7, $t7, 2  # $t7 = $t7 + 2; the algorithm should be applied only to characters present in even positions
        j algorithmB      # The loop is started to apply the algorithm B to all the characters of the message

################################################################

algorithmC:
        li $t7, 1      # $t7 = 1; The algorithm C requires to apply the algorithm A to the characters present in the odd positions. For
this reason the pointer of a position is immediately advanced.
        j algorithmB          # => algorithmB; The cycle used for the algorithm B is equivalent to that used for the algorithm C, code
repetition is avoided and the algorithm B is recalled directly

#####################################################################
algorithmD:
        jal count      # The "count" procedure is called which returns the number of elements present in the buffer
        move $s3, $v1          # $s3 = $v1; The number of elements returned by the procedure count in the register $s3 is saved
        li $t8, 0      # $t8 = 0; The register $ t8 is reinitialized to use it as a pointer for the algorithm D
funzioneD:
        beqz  $s3, reloadBuffer   # if($s3==0) => reloadBuffer; If the register $ s3, that is the pointer that starts from the final
position of the buffer, is equal to zero the algorithm ends.
        sub $s3, $s3, 1        # $s3 = $s3 - 1;
        lb $t2, buffer($s3)       # $t2 = buffer[$t3];
        sb $t2, supportBuffer($t8)   #supportBuffer[$t8] = $t2; Starting from position 0, the buffer elements are loaded into a
support buffer. A new pointer is used.
        addi $t8, $t8, 1         # $t8 = $t8 + 1
        j  funzioneD             # A loop is created to apply the D algorithm to all the characters of the message

fineD:
# The registers that were used are reset
    li $s3, 0   # $s3 = 0;
    li $t8, 0   # $t8 = 0
    li $t4, 0   # $t4 = 0
    beqz $s7, endAlgorithm  # if($s7 = null) => endAlgorithm; The register $ s7 is used to avoid repetition of code, and to allow the
use of algorithm D both for encryption and decryption.
    j endDecryption    # else => endDecryption

#########################################################
algorithmE:
        jal count   # The "count" procedure is called which returns the number of elements present in the buffer
        move $s3, $v1  # $s3 = $v1; The number of elements returned by the procedure count in the register $s3 is saved
        li $t5, 0   # $t5 = 0; The registers to be used during the algorithm are reset
        li $t1, 0   # $t1 = 0;
        li $t9, 0   # $t9 = 0;
startE:
        beq $t1, $s3, endE    # if($t1 == $s3) => endE; $s3 is equal to buffer.length
```

```
        lb $s4, buffer($t1)     # $s4 = buffer[$t1];
        beqz $s4, skipPosition  # if($s4 == null) => skipPosition; If the i-th element is null, the current position is skipped.
        sb $s4, supportBuffer($t9)   # supportBuffer($t9) = $s4;
        addiu $t9, $t9, 1     # $t9 = $t9 + 1; Move the pointer forward one position
        move $t5, $t1       # A copy of the pointer $t1 is created inside the register $t5 to be able to use the pointer, without modifying it, in the next cycle

searchSameChar:
# This procedure creates a cycle that looks for, inside the buffer, all the characters equivalent to the one previously saved in register $t2
        lb $t2, buffer($t5)          # $t2 = buffer[$t5];
        beq $t2, $s4, addDash    # if($t2 == $s4) => addDash;
        bge $t5, $s3, addSpace  # if($t5 >= $s3) => addSpace; If the pointer is greater than or equal to the maximum position of the buffer, it means that equivalent characters are no longer present
        addiu $t5, $t5, 1   # $t5 = $t5 + 1; Move the pointer forward one position
        j searchSameChar     # => searchSameChar; A cycle is created that re-runs the "searchSameChar" procedure

addDash:
#This procedure is called when a character similar to the one previously loaded in the register $s4 is found inside the buffer
      li $t8, 45   # $t8 = 45; In the register $ t8 the ASCII code of "-" is inserted to be able to use it in the algorithm E
      sb $t8, supportBuffer($t9) # supportBuffer[$t9] = $t8; It is inserted in the support buffer, in which the encrypted message is inserted, the dash "-"
      addiu $t9, $t9, 1        # $t9 = $t9 + 1;

addNumber:
       move $t4, $t5   # $t4 = $t5;
       li $t2, 10000   # $t2 = 10000; The number 10000 is inserted in the register $t2 in order to make the divisions necessary for the success of the algorithm
       div $t4, $t2           # $t4 : $t2 => The quotient and the rest sell saved in the "lo" and "hi" registers
       jal operAddNumb   # => operAddNumb; We call the "operAddNumb" procedure to enter the tens of thousands
       move $s6, $v1    # $s6 = $v1; The result of the procedure called is kept in $ s6
       beq $s6, 48, calculateNumber   # if($s6 == 0) => calculateNumber;
       addi $t9, $t9, 1    # $t9 = $t9 + 1;

calculateNumber:
#This procedure allows, starting from the units of thousands, to calculate, and insert in the buffer, all the digits of the position saved in $t4
        divu $t2, $t2, 10    # $t2 = $t2 : 10; Starting from register $t2 = 10000 the register is gradually reduced up to $t2 = 1
        div $t4, $t2    # The position previously saved in the register $t4 is divided by a number between {1000, 100, 10, 1} in order to obtain all the necessary figures
        jal operAddNumb   # We call the "operAddNumb" procedure to enter the digits
        move $s6, $v1     # $s6 = $v1; The result of the procedure called is kept in $ s6
        beq $t2, 1, endCalculate  # if($t2 == 1) => endCalculate;
        bne $s6, 48 addPosition  # if($s6 != 0) => addPosition;
        sub $s2, $t9, 1   # $s2 = $t9 - 1; In the register $ s2 the position preceding the one indicated by $ t9 is loaded
        lb $s1, supportBuffer($s2)    # $s1 = supportBuffer($s2);
        beq $s1, 45, calculateNumber  # if($s1== "-") => calculateNumber; The cycle is restarted
        addiu $t9, $t9, 1   # $t9 = $t9 + 1; If the previous instruction is false, the pointer is advanced
        j calculateNumber   # => calculateNumber; The procedure is restarted by creating a cycle

addPosition:
        addiu $t9, $t9, 1  # $t9 = $t9 + 1
        j calculateNumber  # => calculateNumber
```

```
endCalculate:
# All the digits of the position of the character selected in the support buffer have been calculated and entered
        addiu $t9, $t9, 1    # $t9 = $t9 + 1;
        li $s1, 0  # $s1 = 0; All the registers used are reinitialized
        li $s2, 0  # $s2 = 0;
        li $t2, 0  # $t2 = 0;
        li $s6, 0  # $s6 = 0;

replaceWithZero:
      sb $zero, buffer($t5)  # buffer[$t5} = 0; Replace the character in the original buffer with a zero
      j searchSameChar       # => searchSameChar; It continues with the search for characters equivalent to the one selected

operAddNumb:
# Operation that allows us to insert the figures just found
      mflo $v1     # $v1 = lo; The result of the previously made division is recovered from the register
      addi $v1, $v1, 48   # $v1 = $v1 + 4; The number 48 is added to that result, which represents the zero in the ascii code
      sb $v1, supportBuffer($t9)  # supportBuffer[$t9] = $v1;
      mfhi $t4    # $t4 = hi; We retrieve the rest of the division just made and save it in the register $t4
      jr $ra

addSpace:
      li $s0, 32      # $s0 = " "; 32 is the ascii code of space
      sb $s0, supportBuffer($t9)   # supportBuffer[$t9]= $s0;
      addiu $t9, $t9, 1          # $t9++; The pointers are advanced

skipPosition:
        addiu $t1, $t1, 1  #$t1++;
        j startE        # => startE; The algorithm E is restarted to choose another character

endE:
    sub $t9, $t9, 1    # $t9--; The pointer is moved back one position
    sb $zero, supportBuffer($t9) # supportBuffer[$t9] = 0;  Since an extra space has been inserted at the end of algorithm E it is
eliminated by replacing it with zero
    li $t9, 0  # $t9 = 0; The registers used for the algorithm E are reset
    li $t8, 0  # $t8 = 0
    li $t5, 0  # $t5 = 0
    li $s3, 0  # $s3 = 0
    li $t1, 0  # $t1 = 0
    li $v1, 0  # $v1 = 0
    j reloadBuffer     # => reloadBuffer
############################################################
endAlgorithm:
           addiu $s5, $s5, 1   #$t5++;
           j algorithmSelection  # => algorithmSelection; Return to algorithm selection

############################################################
count:
        li $v1, 0  # $v1 = 0;
loopCount:
        lb $t2, buffer($v1) # $t2 = buffer[$v1];
        beqz $t2, endCount  # if($t2 == null) => endCount;  In this case the count procedure is terminated
        addiu $v1, $v1, 1    # $v1 = $v1 + 1; The pointer $s3 is increased, which at the end of the procedure will be equal to the
number of elements present in the buffer
        j loopCount    # => loopCount; A cycle is created
```

```
endCount:
        jr $ra

#############################################################

reloadBuffer:
        li $t7, 0   # $t7 = 0; The register $t7 is reset to zero, to use it as a pointer

loopReload:
        lb $t2, supportBuffer($t7) # $t2 = supportBuffer[$t7];
        beqz $t2, clearSupportBuffer  # if($t2 == null) => clearSupportBuffer;
        sb $t2, buffer($t7)     # buffer[$t7] = $t2;
        addi $t7, $t7, 1        # $t7 = $t7 + 1; The pointer identified by the register $ t7 is increased
        j loopReload            # => loopReload; The cycle is restarted

clearSupportBuffer:
# The support buffer is reinitialized so that it can be reused
        li $t7, 0   # $t7 = 0; The register $ t7 is reset to zero, to use it as a pointer
loopClear:
        lb $t2, supportBuffer($t7)  # $t2 = supportBuffer[$t7];
        beqz $t2, fineD    # if($t2 == null) => fineD;
        sb $zero, supportBuffer($t7)  # supportBuffer[$t7] = 0; The zero is inserted in the support buffer to reinitialize it
        addi $t7, $t7, 1    # $t7++;
        j loopClear         # => loopClear;



##################Decriptazione##################

startDecryption:
            move $s5, $v1      # $s5 = $v1; The pointer to the "key" buffer is retrieved
            sub $s5, $s5, 1    # $s5--; It is necessary to decrease the pointer of a unit, since no algorithm is called in the last
execution of the "selectionAlgorithm" procedure.
            li $s7, 1  # $s7 = 1; The register $s7 is used only to allow the program to understand if an encryption or decryption is
taking place (Useful to avoid duplication of code for algorithm D)

selectionDecryption:
              li $t7, 0           # $t7 = 0; The $t7 register is reinitialized so that it can be used as a pointer
              lb $t0, key($s5)          # $t0 = key($s5); with  0<=i<4
              beq $t0, 65, DecryptionA  # if($t0 == A) => DecryptionA; Si seleziona, tramite una serie di operatori condizionali,
l'algoritmo di decriptazione da eseguire
              beq $t0, 66, DecryptionB  # if($t0 == B) => DecryptionB;
              beq $t0, 67, DecryptionC  # if($t0 == C) => DecryptionC;
              beq $t0, 68, DecryptionD  # if($t0 == D) => DecryptionD;
              beq $t0, 69, DecryptionE  # if($t0 == E) => DecryptionE;
              j printDecrypted      # => printDecrypted; Once the algorithms to be applied to the message are finished, the decrypted
message must be printed
#############################################################

DecryptionA:
# The decryption algorithm A is very similar to the decryption algorithm, in fact the ASCII code of each character of the message,
instead of added, is subtracted 4
        lb $s0, buffer($t7)  # $s0 = buffer[$t7];
        beqz $s0, endDecryption  # if($s0 == null) => endDecryption;
```

```
        move $a0, $s0  # $a0 = $s0; The value of the register $a0 is passed to the "decrypA" procedure
        jal decrypA    # => decrypA; The "decrypA" procedure is called, which is the function to apply to the message
        addi $t7, $t7, 1  # $t7++;
        j DecryptionA     # => DecryptionA; The cycle is created which allows the decryption algorithm to be applied to all the characters of the message


decrypA:
        lw $t0, K    # $t0 = K;  The constant K is equal to 4
        sub $a0, $a0, $t0  # $a0 = $a0 - $t0;
        li $t1, 256  # $t1 = 256; The number 256 is saved in the register $ t1, which will be used to apply the module function
        add $a0, $a0, $t1 # $a0 = $a0 + $t1;
        divu $a0, $t1   # $a0 : $t1; the quotient is saved in the "lo" register while the rest in the "hi" register
        mfhi $a0   # $a0 = hi; The result of the module function, that is the rest of the division made in the previous instruction, is saved in the register $ t2

        sb $a0, buffer($t7)  # buffer[$t7]=$a0;

        jr $ra

#####################################################################

DecryptionB:
        lb $s0, buffer($t7)   # $s0 = buffer[$t7]
        beqz $s0, endDecryption  # if($s0 == null) => endDecryption; The algorithm terminates if the i-th element of the buffer is null
        move $a0, $s0   # $a0 = $s0; The value of the register $a0 is passed to the "decrypA" procedure
        jal decrypA     # => decrypA; The "decrypA" procedure is called, which is the function to apply to the message
        addi $t7, $t7, 2  # $t7 = $t7 + 2; the algorithm should be applied only to characters present in even positions
        j DecryptionB  # => DecryptionB; The cycle is created which allows the decryption algorithm to be applied to all the characters of the message

#####################################################################

DecryptionC:
         li $t7, 1 # $t7 = 1; The algorithm C requires to apply the algorithm A to the characters present in the odd       positions. For this reason the pointer of a position is immediately advanced.
         j DecryptionB  # => DecryptionB; The cycle used for the algorithm B is equivalent to that used for the algorithm C, code repetition is avoided and the algorithm B is recalled directly

#####################################################################

DecryptionD:

        j algorithmD   # => algorithmD; Since the algorithm D is equivalent for both encryption and decryption, the encryption algorithm is called directly

#####################################################################

DecryptionE:
         li $s4, 0   # $s4 = 0; The register $ s4 is reset because it will be used as a pointer

startDecrypE:
          lb $s2, buffer($s4)  # $s2 = buffer[$s4];
          beqz $s2, endDecrypE # if($s2 == null) => endDecrypE;
```

```
loopE:
        addiu $s4, $s4, 1  # $s4 = $s4 + 1;
        lb $s0, buffer($s4)  # $s0 = buffer($s4); This is the next element compared to the one entered in the register $s2
        move $a1, $s0      # $a1 = $s0; The value of $s0 is passed to the procedure
        jal checkEndLoop   # => checkEndLoop; It's called a procedure that allows you to understand if the cycle is over
        move $s0, $a1      # $s0 = $a1; The value returned by the procedure is resumed
        sub $s0, $s0, 48   # $s0 = $s0 - 48; If the character passed to the "checkEndLoop" procedure has passed the procedure
itself, necessarily it is a ascii code number and for this reason it subtracts 48 (ascii code of 0)
        j checkDecimal     # => checkDecimal; Let's move on to adding the digits in the support buffer

checkEndLoop:
#The procedure is used to carry out checks and act accordingly
        beq $a1, 45, loopE      # if($a1 == "-") => loopE; If the figure passed to the procedure is equal to 45, or the Ascii code of "-",
the cycle is restarted
        beq $a1, 32, restartLoop   # if($a1 == " ") => restartLoop; If the figure is equal to 32, or the Ascii code of the space, the
algorithm E is restarted by choosing a new digit
        beqz $a1, endDecrypE     # if($a1 != null) => endDecrypE; If the digit is equal to 0, the algorithm ends
        jr $ra

restartLoop:
        addiu $s4, $s4, 1  # $s4 = $s4 + 1;
        j startDecrypE     # =>startDecrypE;  The algorithm is restarted after increasing the pointer

checkDecimal:
        move $t9, $s4     # $t9 = $s4; Save the pointer in another register to use two different versions of the same pointer

loopCheckDec:
        addiu $t9, $t9, 1    # $t9++; The pointer is advanced one position
        lb $s1, buffer($t9)  # $s1 = buffer[$t9];
        bge $s1, 58, insert  # if($s1 >= 58) => insert; If the character saved in register $ s1 should be greater than or equal to 58, or
greater than the Ascii code of 9, the "insert" procedure is called
        ble $s1, 47, insert  # if($s1 <= 47) => insert; If the character saved in register $s1 should be less than or equal to 47, or less
than the Ascii code of 0, the "insert" procedure is called
        sub $s1, $s1, 48     # $s1 = $s1 - 48; Since $ s1 has exceeded the two conditional operators it means for sure that it is the
ascii code of a number.
        mul $s0, $s0, 10     # $s0 = $s0 * 10;
        add $s0, $s0, $s1   # $s0 = $s0 + $s1;  Remember that the number in the register $s0 has been multiplied by 10.
        addiu $s4, $s4, 1     # $s4++; The pointer is advanced one position
        j loopCheckDec       # A cycle is created

insert:
        sb $s2, supportBuffer($s0)  # supportBuffer[$s0] = $s2;
        j loopE       # Return to the previous cycle

endDecrypE:
        li $s4, 0   # $s4 = 0; The registers used for the decryption algorithm E are reinitialized
        li $s0, 0   # $s0 = 0;
        li $t4, 0    # $t4 = 0;
        li $a1, 0   # $a1 = 0;
        li $s2, 0   # $s2 = 0;
        j clearBuffer

clearBuffer:
```

```
# This procedure is used to reinitialize the buffer
        lb $t2, buffer($t4)    # $t2 = buffer[$t4];
        beqz $t2, reloadBuffer # if($t2 == null) => reloadBuffer; If $ t2 is 0 (ASCII code of the null) then the procedure ends
         sb $zero, buffer($t4)  # buffer[$t4] = 0;
         addiu $t4, $t4, 1   # $t4 = $t4 + 1; Move the pointer forward one position
         j clearBuffer    # A cycle is created

#########################################################################

endDecryption:

        sub $s5, $s5, 1            # $s5 = $s5 - 1; The pointer of the key of a position is moved back
        j selectionDecryption   # => selectionDecryption; Return to the selection of the decryption algorithms


 exit:
    li $v0, 10   # End the program
    syscall



#####################Operazioni di Input#################

inputFile:
open:
     li   $v0, 13       # system call for open file
     la   $a0, fin      # board file name
     li   $a1, 0        # Open for reading
     li   $a2, 0
     syscall            # open a file (file descriptor returned in $v0)
     move $s6, $v0      # save the file descriptor
     blt $v0, 0, err    # Goto Error

read:
    li   $v0, 14       # system call for read from file
    move $a0, $s6      # file descriptor
    la   $a1, buffer   # address of supportBuffer to which to read
    li   $a2, 178831   # hardcoded supportBuffer length
    syscall            # read from file

closeInputFile:
            li   $v0, 16       # system call for close file
            move $a0, $s6      # file descriptor to close
            syscall            # close file
            jr $ra



inputKey:
openKey:
     li   $v0, 13           # system call for open file
     la   $a0, inKey        # board file name
     li   $a1, 0            # Open for reading
     li   $a2, 0
```

```
    syscall             # open a file (file descriptor returned in $v0)
    move $s6, $v0       # save the file descriptor
    blt $v0, 0, err     # Goto Error

readKey:
    li   $v0, 14       # system call for read from file
    move $a0, $s6      # file descriptor
    la   $a1, key      # address of supportBuffer to which to read
    li   $a2, 4        # hardcoded supportBuffer length
    syscall            # read from file

closeInputFileKey:
            li   $v0, 16       # system call for close file
            move $a0, $s6      # file descriptor to close
            syscall            # close file
            jr $ra

####################Operazioni di Output####################
# Write Data
print:
    li          $v0, 13             # Open File Syscall
    la          $a0, fileOUT        # Load File Name
        li $a1, 1
        li $a2, 0
    syscall
    move    $t1, $v0    # Save File Descriptor
    li          $v0, 15             # Write File Syscall
    move    $a0, $t1            # Load File Descriptor
    la          $a1, buffer         # Load Buffer Address
    li          $a2, 178831         # Buffer Size
    syscall


# Close File
close:
    li          $v0, 16             # Close File Syscall
    move    $a0, $t1            # Load File Descriptor
    syscall
    j           startDecryption             # Goto Decryption


#Output Decrypted

printDecrypted:

    li          $v0, 13                     # Open File Syscall
    la          $a0, outputDecrypted        # Load File Name
        li          $a1, 1
        li          $a2, 0

    syscall
    move    $t1, $v0            # Save File Descriptor

    li          $v0, 15             # Write File Syscall
```

```
    move    $a0, $t1            # Load File Descriptor
    la      $a1, buffer         # Load Buffer Address
    li      $a2, 178831         # Buffer Size
    syscall


# Close File
closeDecryptation:
    li      $v0, 16             # Close File Syscall
    move    $a0, $t1 # Load File Descriptor
    syscall
    j       exit                # Goto End

# Error
err:
    li      $v0, 4              # Print String Syscall
    la      $a0, fnf          # Load Error String
    syscall
```