RELAZIONE PROGETTO ARCHITETTURE DEGLI ELABORATORI

Cifrario di Cesare

```
cifrarioAsostituzione:
ccInit:
    mv t0, a0   # carico plain text
    mv t1, a1   # carico sost k
    li t3, 26   # per modulo

ccLoop:
    lb t2, 0(t0)
    beqz t2, ccEnd
    blt t2, s7, ccUpdateIndex    # Se < A  → invariato
    ble t2, s8, ccUppercase      # Se ≤ Z  → lettera maiuscola
    blt t2, s9, ccUpdateIndex    # Se < a  → invariato
    ble t2, s10, ccLowercase     # Se ≤ z → lettera minuscola

ccUpdateIndex:
    addi t0, t0, 1
    j ccLoop

ccUppercase:
    mv t4, s7
    j ccCipher

ccLowercase:
    mv t4, s9

ccCipher:
    sub t2, t2, t4    # cif = cod(x) - A oppure cod(x) - a
    add t2, t2, t1    # cif = cif + K
    rem t2, t2, t3    # cif = cif % 26
    add t2, t2, t3    # cif = cif + 26
    rem t2, t2, t3    # cif = cif % 26
    add t2, t2, t4    # cif = cif + A oppure cif + a
    sb t2, 0(t0)
    j ccUpdateIndex

ccEnd:
    jr ra
```

```
string cifrarioAsostituzione(string text, int sostK) {
    int s = 0;

    for (int i = 0; text[i] != '\0'; i++) {

        if (text[i] >= 'A' && text[i] <= 'Z') s = 'A';

        else if (text[i] >= 'a' && text[i] <= 'z') s = 'a';

        else continue;

        text[i] = text[i] - s + sostK;
        text[i] %= 26;
        text[i] += 26;
        text[i] %= 26;
        text[i] += s;
    }
    return text;
}
```

Per la decriptazione il codice è simile
ma invece di aggiungere la key, la rimuovo.

Per il **cifrario a sostituzione** analizzo i vari caratteri della stringa da cifrare e vado ad effettuare il seguente calcolo:

$$cif = (cod(x) - s + sostK)\ mod\ 26$$

Dove “s” corrisponde a **65** oppure a **97** a seconda del case del carattere da cifrare.
Questo però non basta perchè in risc-v usando rem per il modulo, i valori negativi rimangono negativi dopo il modulo, quindi aggiungendo **26** e facendo il modulo di nuovo, “cif” avrà un valore positivo.

Esempio in risc-v:

cod(x) = “**A**” e sostK = **-2**

cif = A - 65 + (-2) = **-2** → cif = -2 mod 26 = **-2** → cif = -2 + 26 = **24** → cif = 24 mod 26 = **24**

Cifrario a Blocchi

```assembly
cifrarioAblocchi:
cbInit:
    addi sp, sp, -8
    sw ra, 4(sp)
    sw a0, 0(sp)

    jal length   # lunghezza stringa in a1 ovvero della key

    mv t0, a0    # t0 = key.length
    mv t1, zero  # i
    lw a0, 0(sp) # ripristino a0
    li t6, 96    # per modulo

cbLoop:
    add t2, t1, a0   # indirizzo plaintext + i
    lb t3, 0(t2)     # carico plaintText[i]
    beqz t3, cbEnd   # plaintext[i] == 0

    rem t4, t1, t0  # pos = i % keyLength
    add t4, t4, a1  # indirizzo key + pos
    lb t4, 0(t4)    # carico key[pos]

    add t5, t4, t3  # a = plaintext[i] + key[pos]
    rem t5, t5, t6  # a = a % 96
    addi t5, t5, 32 # a = a + 32

    sb t5, 0(t2)
    addi t1, t1, 1  # i++
    j cbLoop

cbEnd:
    lw ra, 4(sp)
    addi sp, sp, 8
    jr ra
```

```cpp
string cifrarioAblocchi(string text, string blocKey) {
    int keyLength = blocKey.length();
    int pos = 0;

    for (int i = 0; text[i] != '\0'; i++) {
        pos = i % keyLength;

        text[i] = (text[i] + blocKey[pos]) % 96 + 32;
    }
    return text;
}
```

Per il cifrario a blocchi, per cifrare un carattere basta usare la formula come riportato sopra.

La decifrazione è simile.

Cifratura occorrenza

```
cifraturaOccorrenze:
coInit:
    addi sp, sp, -4
    sw ra, 0(sp)
    jal get_string_address_cif
    lw ra, 0(sp)
    addi sp, sp, 4

    mv t0, a1 # t0 = indirizzo cipherText
    mv t1, a0 # t1 = indirizzo plainText
    li a2, 1  # per marcare i caratteri già visitati
    li a3, 0x7fffff0  #indirizzo stack
    li a4, 10 # per divisione
    li a5, 45 # trattino

coLoop1:
    lb t2, 0(t1)
    beqz t2, coEnd
    beq t2, a2, cpUpdateIndexLoop1
    # char di già criptato

    sb t2, 0(t0)  #inserisco carattere cyperText
    addi t0, t0, 1
    mv t3, t1 #indirizzo carattere plainText per loop2

coLoop2:
    lb t4, 0(t3)
    beqz t4, coEndLoop2
    bne t4, t2, cpUpdateIndexLoop2

    sb a2, 0(t3)   # marco il carattere corrente con 1
    sb a5, 0(t0)   # aggiungo trattino in cypherText
    addi t0, t0, 1 # aggiorno indirizzo cyperText

    # recupero la posizione del carattere
    sub t5, t3, a0
    blt t5, a4, coPosSingleDigit  # pos < 10

coPushPosDigit:
    beqz t5, coPopPosDigit

    rem t6, t5, a4 # digit di pos
    addi t6, t6, 48

    addi sp, sp, -1  # alloco spazio per stack
    sb t6, 0(sp)  # metto digit nello stack

    div t5, t5, a4
    j coPushPosDigit

coPopPosDigit:
    beq sp, a3, cpUpdateIndexLoop2

    lb t5, 0(sp)   # pop stack
    addi sp, sp, 1  # cifra successiva

    sb t5, 0(t0)  # carico la cifra in cipherText
    addi t0, t0, 1

    j coPopPosDigit

coPosSimgleDigit:
    addi t5, t5, 48
    sb t5, 0(t0)
    addi t0, t0, 1

cpUpdateIndexLoop2:
    addi t3, t3, 1 # aggiorno indirizzo plainText
    j coLoop2

coEndLoop2:
    sb s6, 0(t0)   # metto lo spazio in cipherText
    addi t0, t0, 1 # aggiorno indirizzo cyperText

cpUpdateIndexLoop1:
    addi t1, t1, 1 # aggiorno indirizzo plainText
    j coLoop1

coEnd:
    addi t0, t0, -1
    sb zero, 0(t0)  # carattere di fine stringa
    mv a0, a1
    jr ra
```

Esempio di stringa cifrata a occorrenza:

“occorrenza” → “o-0-3 c-1-2 r-4-5 e-6 n-7 z-8 a-9”

I caratteri del plainText vengono sostituiti con le loro relative posizioni nella stringa.

Per iniziare chiamo la funzione “get_string_address_cif”, che ritorna l’indirizzo dove salvare la stringa cifrata. Faccio questo perchè se venisse chiamata due volte consecutivamente la funzione, e se per prendere l’indirizzo dove viene salvata la stringa cifrata avessi fatto “la a1, cipherText”, quando chiamo per la seconda volta la funzione, a0 contiene l’indirizzo del plainText (che corrisponde a cipherText) e a1 avrebbe di nuovo l’indirizzo di cipherText, quindi a0 e a1 punteranno alla stessa stringa che però durante l’esecuzione viene cambiata più volte. Grazie a “get_string_address_cif” ho ad ogni chiamata della funzione un indirizzo diverso per salvare le stringhe.

Attraverso “loop1” seleziono il carattere per cui trovare le posizioni all’interno del plainText. Quindi nel caso dell’esempio precedente, viene presa in considerazione la ‘o’. Nel loop2 trovo una posizione del carattere ‘o’, e la salvo nello stack.

Dopodiché marco il carattere in modo tale da evitare di cifrarlo nuovamente.

Per salvare nello stack la posizione prendo le singole cifre e le trasformo in ascii.
Una volta salvate nello stack, posso estrarre le varie cifre e posizionarle all’ interno di cipherText.

Utilizzo lo stack perchè per isolare le cifre della posizione divido e utilizzo il modulo 10, quindi ottengo sempre la cifra più a destra della posizione. Però le cifre da inserire all’interno di cipherText devono essere da sinistra a destra. Quindi utilizzando lo stack per salvare le cifre da destra verso sinistra, estraendo dallo stack avrò le cifre nell’ordine corretto.

Alla fine del Loop 1 inserisco il carattere di fine stringa.

Decifratura occorrenza

```
decifraturaOccorrenze:
doInit:
    addi sp, sp, -4
    sw ra, 0(sp)
    jal get_string_address_decif

    # a0 = indirizzo cipherText (input)
    # a1 = indirizzo plainText (output)
    mv t0, a0
    li a2, 0  # per posizionare carattere
    li a3, 45 # trattino
    li t4, 0  # conta cifre posizione
    li t6, 0  # contatore caratteri di plainText


doLoop1:
    lb t1, 0(t0)
    beqz t1, doEnd
    addi t2, t0, 2 # + 2 perchè c'è un trattino

doLoop2:
    lb t3, 0(t2)
    beq t3, a3, doAsciiToInt #salta se trattino
    beq t3, s6, doAsciiToInt #salta se spazio
    beqz t3, doAsciiToInt
    addi t4, t4, 1 # aggiorno contatore cifre della
posizione
    j doUpdateIndexLoop2

doAsciiToInt:
    sub t2, t2, t4

doCalcPos:
    beqz t4, doStoreChar

    lb t5, 0(t2) # prendo la cifra
    addi t5, t5, -48
    addi t2, t2, 1 # sposto il puntatore alla prossima cifra
    addi t4, t4, -1

    mv a0, t4
    jal pow # calcolo 10^t4
    mul a0, t5, a0 # moltiplico per la potenza
    add a2, a2, a0 # sommo la cifra al totale
    j doCalcPos

doStoreChar:
    add a2, a1, a2 #indirizzo plainText + posizione
    sb t1, 0(a2)
    addi t6, t6, 1 #aggiorno contatore caratteri plainText

doUpdateIndexLoop2:
    addi t2, t2, 1
    beq t3, s6, doUpdateIndexLoop1 #se spazio salta
    beqz t3, doUpdateIndexLoop1
    li a2, 0
    j doLoop2

doUpdateIndexLoop1:
    mv t0, t2
    j doLoop1

doEnd:
    add t6, a1, t6  # prendo pos ultimo carattere
    sb zero, 0(t6)  # inserisco il fine stringa

    mv a0, a1   # output in a0
    lw ra, 0(sp) # reset ra e stack
    addi sp, sp, 4
    jr ra
```

Per lo stesso motivo della cifratura chiamo la funzione “get_string_address_cif”, che mi ritorna l’indirizzo dove salvare la stringa decifrata.

Nel loop1 verifico quale carattere andrà messo nelle varie posizioni all’interno della stringa.

Nel loop 2 parto dalla prima cifra della posizione, aumento il contatore e vado avanti, finchè non trovo uno spazio, un trattino oppure il carattere di fine stringa.

Dopodichè sposto il puntatore della stringa cifrata e lo riposiziono alla prima cifra della posizione.
Diminuisco la cifra di 48 almeno ho un numero decimale e non in ascii.
Poi moltiplico la cifra per 10^(num cifre - i) dove i all’inizio è uno, e alla fine è uguale al numero di cifre.
Alla fine sommo il risultato di questa moltiplicazione al totale. Esegue questa procedura per tutte le cifre della posizione.

In questo modo ho la posizione dove andare ad inserire il carattere del loop1, nella stringa decifrata.

Una volta inserito il carattere aumento il contatore dei caratteri della stringa decifrata. Grazie a questo contatore, alla fine di entrambi i loop ho la posizione finale per inserire lo zero che segnala la fine della stringa.

Dizionario

```
DIZIONARIO:
dizInit:
    mv t0, a0
    li t3, 187 # z + a - 32
    li t4, 105 # 0 in ascii + 9 in ascii
    li t5, 48
    li t6, 57

dizLoop:
    lb t1, 0(t0)
    beqz t1, dizEnd
    blt t1, t5, dizUpdateIndexLoop # Se < di 0 in ascii → sys
    ble t1, t6, dizNumber # Se ⩽ di 9 in ascii → numero
    blt t1, s7, dizUpdateIndexLoop # Se < A → sys
    ble t1, s8, dizLetter # Se ⩽ Z → lettera
    blt t1, s9, dizUpdateIndexLoop # Se < a → sys
    ble t1, s10, dizLetter # Se ⩽ z → lettera

dizUpdateIndexLoop:
    addi t0, t0, 1
    j dizLoop

dizNumber:
    sub t1, t4, t1 # 105 - numero
    sb t1, 0(t0)
    j dizUpdateIndexLoop

dizLetter:
    sub t1, t3, t1 # 187 - lettera
    sb t1, 0(t0)
    j dizUpdateIndexLoop

dizEnd:
    jr ra
```

Per il dizionario quando trovo una lettera devo sostituirla con il suo opposto e invertire il case. Per i numeri devo trovare l'opposto mentre i caratteri di sistema vengono lasciati invariati.

Per trovare l'opposto di una lettera minuscola utilizzo la seguente formula:

$$z - cif(x) + a - 32$$

Dove il meno 32 serve per cambiare il case della lettera. Riscrivendo la formula si ottiene:

$$z + a - 32 - cif(x)$$

ovvero:

$$187 - cif(x)$$

Per i caratteri maiuscoli la formula è la seguente:

$$Z - cif(x) + A + 32$$
$$Z + A + 32 - cif(x)$$
$$187 - cif(x)$$

Quindi se il carattere da cifrare è una lettera maiuscola o minuscola per cifrare basta fare 187 meno il carattere da cifrare.

Allo stesso modo per i numeri la formula è la seguente:

$9 - cif(x) + 0$ (0 e 9 sono in ascii)
$9 + 0 - cif(x)$ (0 e 9 sono in ascii)
$105 - cif(x)$ (105 è in decimale)

Per decifrare il dizionario basta eseguire gli stessi calcoli della cifrazione perché con il dizionario vengono generati gli opposti.

Inversione

```
INVERSIONE:
invInit:
    addi sp, sp, -4
    sw ra, 0(sp)
    jal get_string_address_for_inv
    mv t0, a0    # carico plaintext
    mv t2, a1    # carico cipherText
    sb zero, 0(t2) # fine stringa

invLoop:
    lb t1, 0(t0)
    beqz t1, invEnd

    addi t0, t0, 1
    addi t2, t2, -1
    sb t1, 0(t2)
    j invLoop
invEnd:
    mv a0, t2  # a0 output
    lw ra, 0(sp) # reset stack e ra
    addi sp, sp, 4
    jr ra
```

Per lo stesso motivo del cifratura occorrenza utilizz una funzione che ritorna un puntatore ad un'area di memoria per salvare la stringa cifrata.

Per invertire la stringa, con un

```
#include <stdio.h>

int main()
{
    int len = 8;

    char plainText[] = "Assembly";
    char cipherText[len];
    cipherText[len] = '\0';


    for(int i=0, j = len-1; plainText[i] != '\0'; i++, j--) {
        cipherText[j] = plainText[i];
    }

    return 0;
}
```

puntatore alla stringa plainText scorro in avanti, mentre con un puntatore a cipherText, scorro all'indietro.

In C corrisponde all'incirca a questo:

Generazione indirizzi per salvare stringhe

```
    # 0x10000404 - 0x10003000 sezione per
    # memorizzare stringa cifrata in cifra occorrenza
    li a0, 0x10000404
    li s0, 0x10000400
    sw a0, 0(s0)

    # 0x100043FC - 0x100053FC sezione per
    # memorizzare stringa cifrata in cifra occorrenza
    li a0, 0x10005400
    li s0, 0x100053FC
    sw a0, 0(s0)

get_string_address_cif:
    li a2, 0x10000400
    lw a1, 0(a2)
    addi a1, a1, 1200
    sw a1, 0(a2)
    jr ra

get_string_address_for_inv:
    li a2, 0x10005400
    lw a1, 0(a2)
    addi a1, a1, -1200
    sw a1, 0(a2)
    jr ra
```

Per generare gli indirizzi dove salvare le stringhe, prima di eseguire le varie cifrature salvo in 0x1000400, l'indirizzo da utilizzare per salvare le stringhe.

Quando viene chiamata la funzione "get_string_address_cif" aggiorno l'indirizzo all'interno di 0x1000400, aumentandolo di un numero che ho scelto casualmente. Per "get_string_address_cif_for_inv" aggiorno l'indirizzo diminuendolo, necessario per come ho implementato l'algoritmo inversione.