OTTIMIZZAZIONE DI UN CODICE MIPS ASSEMBLY

Progetto d'esame di Architettura degli Elaboratori A.A. 2020/2021

Tommaso Petrelli, Numero di matricola: 305558, Università degli Studi di Urbino Carlo Bo

Specifica

Scopo del progetto

Con questo progetto si vuole tradurre in linguaggio MIPS assembly un dato programma scritto con un linguaggio di programmazione ad alto livello come quello ANSI C per poi applicare delle tecniche di ottimizzazione a basso livello per mitigare dei conflitti logici, strutturali o di controllo del flusso, in modo tale da ottimizzare il tempo di esecuzione e quindi il numero dei cicli di clock. (scrivere le analisi che riguardano il codice).

Analisi del codice

Il programma da cui partiremo per questo progetto sarà scritto nel linguaggio di programmazione ad alto livello ANSI C. L'obbiettivo che si vuole raggiungere con questo programma è prendere una frase o un messaggio come dato di ingresso e poi applicare la tecnica di cifratura di Cesare per ottenere il messaggio cifrato.

Per non aggiungere complessità all'esercizio di traduzione del programma in linguaggio MIPS assembly, il messaggio di input sarà preimpostato nel codice così da non dover gestire la parte di input dei dati.

A questo proposito avremo un messaggio, in questo caso "ARCHITETTURA DEGLI ELABORATORI", che si decide memorizzare su un array di char al quale possiamo anche fissare la dimensione poiché non vi dobbiamo apportare modifiche. Se andiamo a vedere i caratteri che compongono il messaggio possiamo dire che questi sono 30 per cui definiamo e inizializziamo anche un altro array di char di appoggio di 31 elementi (per il carattere terminatore).

Una volta definite queste due strutture dati di 31 elementi allora possiamo definire una variabile per rappresentare la chiave del cifrario di Cesare, e impostiamo per esempio il valore 13.

All'interno della funzione main ci serviamo di un ciclo for per scorrere tutto il messaggio. Dovendo lavorare su un carattere alla volta ci servono 30 iterazioni di tale ciclo. Conoscendo a priori le caratteristiche del messaggio, ad ogni iterazione dovremmo necessariamente due controlli:

- con il costrutto if-else più esterno verifichiamo se il carattere preso in considerazione sia una lettera maiuscola, e se così non fosse copiamo tale carattere senza applicare modifiche nell'array definito per il messaggio cifrato, altrimenti passeremo all'if successivo;
- con il costrutto if-else annidato verifichiamo che il carattere considerato sia minore della lettera che si ottiene sottraendo la chiave al carattere '[' (secondo la codifica ASCII), e se così non fosse si aggiusta il carattere prima di applicare la chiave per cifrarlo, altrimenti se la condizione fosse rispettata applichiamo direttamente la chiave.

A questo punto si stampa il messaggio cifrato.

Implementazione del codice

Per spiegare come sono stati implementati i costrutti di if-else, dobbiamo considerare le seguenti convenzioni adottate:

- il codice di riferimento usato per lavorare con i caratteri è quello ASCII;
- il messaggio è composto da sole lettere maiuscole o spazi, quindi consideriamo valori da 65 a 90 in decimale o da 41 a 5A in esadecimale per le lettere e rispettivamente 32 o 20 per il carattere spazio;
- il carattere subito precedente ad 'A' è '@' con codice 64 in decimale e 40 in esadecimale;
- il carattere subito successivo a 'Z' è '[' con codice 91 in decimale e 5B in esadecimale.

Di seguito abbiamo l'implementazione del programma scritto in linguaggio ANSI C:

```c
#include <stdio.h>

int main() {
    char frase[31] = "ARCHITETTURA DEGLI ELABORATORI";
    char frase_cifrata[31] = "";
    int chiave = 13;
    int i;

    printf("%s", frase);

    for (i = 0; i < 31; i++) {
        if (frase[i] < '[' && frase[i] > '@') {
            if (frase[i] < '[' - chiave)
                frase_cifrata[i] = frase[i] + chiave;
            else
            {
                frase_cifrata[i] = 'A' + chiave - ('Z' - frase[i]) - 1;

            }
        }
        else
            frase_cifrata[i] = frase[i];
    }

    printf("\n%s", frase_cifrata);

    return(0);
}
```

Passaggio al linguaggio MIPS assembly

Partendo dal programma scritto in ANSI C, scriviamo il codice a basso livello e quindi lo si traduce nel linguaggio MIPS assembly come segue:

```
.data
    frase:          .asciiz "ARCHITETTURA DEGLI ELABORATORI"    ;frase da cifrare
    newline:        .asciiz "\n"                                ;carattere newline
    frasecifrata:   .asciiz ""                                  ;frase cifrata
    numcar:         .word   31  ;numero di caratteri della frase
```

```
    chiave:         .word   13  ;chiave del cifrario
    indice:         .word   0   ;indice del ciclo for
    maxcar:         .word   91  ;'[' carattere massimo
    mincar:         .word   64  ;'@' carattere minimo

    CONTROL:        .word   0x10000 ;indirizzo di CONTROL
    DATA:           .word   0x10008 ;indirizzo di DATA

.text
    start: LWU    r28, DATA(r0)    ;carica DATA in r28
        LWU       r29, CONTROL(r0) ;carica CONTROL in r29
        LW        r1, numcar(r0)   ;carica numero di caratteri
        LW        r2, chiave(r0)   ;carica chiave cifratura
        LW        r3, indice(r0)   ;carica indice ciclo
        LW        r6, maxcar(r0)   ;carica carattere '['
        LW        r7, mincar(r0)   ;carica carattere '@'

        DADDI     r4, r0, frase           ;puntatore al primo carattere dell'array
        DADDI     r5, r0, frasecifrata    ;puntatore al primo carattere dell'array

        DADDI     r27, r0, 4      ;imposta DATA per stampare una stringa
        DADDI     r26, r0, frase  ;imposta stringa da stampare
        SD        r26, 0(r28)     ;imposta DATA con la stringa
        SD        r27, 0(r29)     ;stampa la stringa

    ;inizio ciclo for
    loop:  LB     r8,  0(r4)      ;leggi frase[i]
        SLT       r9,  r8,   r6   ;frase[i] < '[' ?
        SLT       r10, r7,   r8   ;'@' < frase[i] ?
        AND       r11, r9,   r10  ;AND tra i due esiti precedenti
        BNEZ      r11, sem        ;if1: salta se diverso da zero

        SB        r8,  0(r5)      ;else1: frasecifrata[i] = frase[i]
        J         endif           ;salto incondizionato

    ;entro in if1
    sem:   DSUB   r12, r6,   r2   ;'[' - chiave
        SLT       r13, r8,   r12  ;frase[i] < ('[' - chiave) ?
        BNEZ      r13, set        ;if2: salta se diverso da zero

        ;else2
        DADDI     r14, r7,   1    ;ottengo 'A'
        DADDI     r15, r6,   -1   ;ottengo 'Z'
        DADD      r14, r14,  r2   ;'A' + chiave in r14
        DADD      r16, r14,  r8   ;r14 + frase[i] in r16
        DSUB      r17, r16,  r15  ;r16 - 'Z' in r17
        DADDI     r17, r17,  -1   ;decremento r17
        SB        r17, 0(r5)      ;frasecifrata[i] = r17
        J         endif           ;salto incondizionato
```

```
    ;entro in if2
    set:   DADD   r20, r8,   r2   ;frase[i] + chiave
        SB        r20, 0(r5)      ;frasecifrata[i] = r20

    ;aggiornamento valori
    endif: DADDI  r4,  r4,   1   ;vai alla posizione successiva di frase
        DADDI     r5,  r5,   1   ;vai alla posizione successiva di frasecifrata
        SLT       r19, r3,   r1  ;indice < numcar ?
        DADDI     r3,  r3,   1   ;incremento indice
        BNEZ      r19, loop      ;salta se diverso da zero
        ;fine ciclo for

        ;stampa carattere newline
        DADDI     r27, r0, 4          ;imposta DATA per stampare una stringa
        DADDI     r26, r0, newline    ;imposta stringa da stampare
        SD        r26, 0(r28)         ;imposta DATA con la stringa
        SD        r27, 0(r29)         ;stampa la stringa

        ;stampa la frase cifrata
        DADDI     r27, r0, 4             ;imposta DATA per stampare una stringa
        DADDI     r26, r0, frasecifrata  ;imposta stringa da stampare
        SD        r26, 0(r28)            ;imposta DATA con la stringa
        SD        r27, 0(r29)            ;stampa la stringa

    end:   HALT
```

L'ambiente in cui si simula l'esecuzione di questo codice si chiama WinMIPS64 che a fine esecuzione ci fornisce le seguenti prestazioni:

```
Execution

743 Cycles
556 Instructions
1.336 Cycles Per Instruction (CPI)


Stalls

92 RAW Stalls
0 WAW Stalls
0 WAR Stalls
0 Structural Stalls
91 Branch Taken Stalls
0 Branch Misprediction Stalls


Code size

188 Bytes
```

Al termine dell'esecuzione del programma sono stati effettuati 734 cicli di clock (Cycles) e abbiamo ottenuto un CPI accettabile pari 1.336. Infatti, se andiamo ad analizzare i parametri successivi nella sezione "Stalls" vediamo che il numero degli stalli dovuti alla dipendenza RAW ("RAW Stalls") è 92 e quelli dovuti agli abort delle istruzioni in seguito alla valutazione di una condizione di salto ("Branch Taken Stalls") sono 91.

Possiamo valutare il codice come accettabile ma provare comunque ad abbassare il numero di cicli di clock per migliorarne l'efficienza.

Per preimpostare in memoria le variabili che ci servono usiamo la sezione .data del file che contiene il programma in MIPS assembly che costituisce l'input al simulatore WinMIPS64.

Il simulatore va ad allocare dello spazio in memoria alle variabili attribuendo ad ogni zona di memoria in cui sono stati scritti dei dati un'etichetta che possiamo paragonare ad un nome che diamo ad una variabile quando lavoriamo ad alto livello. Questa etichetta verrà trasformata dal loader nell'indirizzo effettivo di

quella zona di memoria consentendo di usare i dati nel programma semplicemente facendogli riferimento attraverso le etichette.

Nella sezione .text dello stesso file andremo a scrivere invece le istruzioni vere e proprie per dire al processore quali operazioni deve compiere in modo tale da concorrere al raggiungimento dell'obbiettivo definito tra le specifiche di progetto.

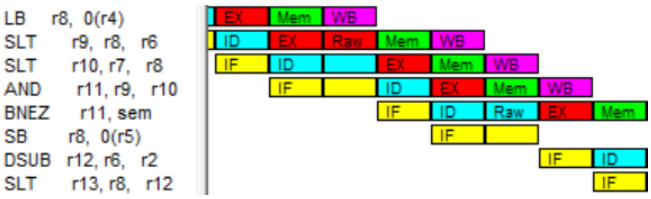
Notiamo fin da ora che completezza si è scelto di implementare anche la parte di codice relativa all'output. Per fare ciò, seguendo la documentazione di WinMIPS64, ci serviamo di due etichette speciali che chiamiamo CONTROL e DATA che fanno rispettivamente riferimento agli indirizzi di memoria 0x10000 e 0x10008. Allora quello che viene fatto per scrivere i messaggi nel terminale si deve impostare l'indirizzo di memoria DATA con il valore che vogliamo stampare a schermo e impostare il giusto operando all'indirizzo di memoria CONTROL (gli operandi sono specificati nella documentazione).

Un'altra cosa che ci interessa osservare è che i registri su cui lavoriamo sono tutti interi poiché non si utilizzano valori floating point e quindi possiamo già escludere operazioni con latenza maggiore di uno (anche perché con l'architettura Harvard separiamo la memoria delle istruzioni da quella dei dati) e quindi conflitti di tipo architetturale.

Nei paragrafi successivi andremo ad applicare le tecniche di ottimizzazione per migliorare l'efficienza del codice. Prima però vale la pena osservare come i segmenti di codice che non sono racchiusi all'interno di cicli non andranno presi in considerazione durante le ottimizzazioni per la ragione che essendo istruzioni che vengono eseguite una volta soltanto non ha senso lavorarci su. Infatti, se proviamo ad eseguire il codice ci accorgiamo che la pipeline lavora a pieno regime finché non entra per la prima volta nel loop.

Ottimizzazione 1

Continuando ad eseguire il codice incontriamo fin da subito degli stalli e delle dipendenze fra dati di tipo RAW (Read After Write):



Durante tutta l'esecuzione del codice incontreremo solamente stalli di questo tipo, tutti causati da delle dipendenze fra dati (considereremo nel paragrafo successivo gli stalli dovuti agli abort).

Per prima cosa allora andiamo a vedere se abbiamo delle opportunità di Instruction Reordering (IR) e Register Renaiming (RR) per tentare di eliminare alcuni degli stalli presenti nel codice dovuti a conflitti di dati o false dipendenze.

Consideriamo il primo segmento di codice che provoca degli stalli:

```
;inizio ciclo for
loop:  LB      r8,  0(r4)      ;leggi frase[i]
    SLT        r9,  r8,   r6   ;frase[i] < '[' ?
    SLT        r10, r7,   r8   ;'@' < frase[i] ?
    AND        r11, r9,   r10  ;AND tra i due esiti precedenti
    BNEZ       r11, sem        ;if1: salta se diverso da zero
```

In questo caso non riusciamo ad applicare la tecnica di IR poiché cambiare la posizione nel codice di una qualsiasi di queste istruzioni vorrebbe dire perdere la correttezza del codice poiché non possiamo sovvertire l'ordine senza andare a violare le dipendenze tra i dati. L'unica operazione che possiamo fare è il RR ma

solamente per impiegare meno risorse della CPU e quindi invece di usare il registro r11 possiamo sovrascrivere il registro r9 con il risultato dell'istruzione AND e poi usarlo come parametro dell'istruzione BNEZ.

Passiamo allora al prossimo segmento di codice che causa degli stalli:

```
;entro in if1
sem:   DSUB   r12, r6,   r2   ;'[' - chiave
       SLT    r13, r8,   r12  ;frase[i] < ('[' - chiave) ?
       BNEZ   r13, set        ;if2: salta se diverso da zero
```

Anche in questo caso non riusciamo a scambiare l'ordine di queste tre istruzioni perché andremmo a violare le dipendenze tra i dati di RAW. Possiamo però di nuovo applicare la tecnica di RR per diminuire l'utilizzo di risorse del processore. Immaginiamo quindi di sostituire il registro r12 con il registro r10 (o r9) utilizzato in precedenza (che possiamo permetterci di sovrascrivere dato che non ci servirà più per il resto del ciclo) e andiamo ancora a sostituire il registro r13 sempre con quello chiamato r10. A questo punto avremo liberato due registri che il processore può utilizzare per altre operazioni.

Un'alternativa che invece potremmo adottare è quella di eseguire le due istruzioni DSUB e SLT prima che venga effettuato il salto a questa etichetta. Quello che si intende dire è: applichiamo la tecnica di IR ma considerando anche il segmento di codice precedente (";inizio ciclo for") e quindi andando ad eseguire le due istruzioni DSUB e SLT andandole a posizionare subito dopo l'istruzione "`LB r8, 0(r4)`". Ovviamente in questo modo dovremmo utilizzare un registro diverso da r10 poiché questo verrà utilizzato subito dopo per memorizzare il risultato di un'altra condizione. Immaginiamo allora di utilizzare il registro r11. In questo modo possiamo garantire la correttezza del codice perché non andiamo a sporcare altri registri. A questo punto possiamo di nuovo applicare l'IR per andare spostare l'istruzione "`SLT r11, r8, r11`" in mezzo alle istruzioni `AND` e `BNEZ`. Quello che otteniamo sarà il codice seguente:

```
;inizio ciclo for
loop:  LB     r8,  0(r4)      ;leggi frase[i]
       DSUB   r11, r6,   r2   ; <- '[' - chiave
       SLT    r9,  r8,   r6   ;frase[i] < '[' ?
       SLT    r10, r7,   r8   ;'@' < frase[i] ?
       AND    r9,  r9,   r10  ;AND tra i due esiti precedenti
       SLT    r11, r8,   r11  ; <- frase[i] < ('[' - chiave) ?
       BNEZ   r9,  sem        ;if1: salta se diverso da zero

       SB     r8,  0(r5)      ;else1: frasecifrata[i] = frase[i]
       J      endif           ;salto incondizionato

;entro in if1
sem:   BNEZ   r11, set        ;if2: salta se diverso da zero
```

Con questi accorgimenti possiamo apprezzare un notevole grado di ottimizzazione del codice. Infatti, applicando l'IR in questo modo siamo capaci di eliminare tutte le dipendenze RAW tra i dati. Questo è possibile perché nel codice di partenza non avevamo abbastanza distanza tra le istruzioni LB e SLT per utilizzare il registro r8 senza stalli e non avevamo abbastanza distanza tra le istruzioni AND e BNEZ per utilizzare il registro r9. Siccome tutte queste istruzioni hanno latenza 1 allora è bastato inserire tra le due l'istruzione DSUB, nel primo caso, e SLT, nel secondo caso.

Consideriamo però che tutto questo lo facciamo sapendo di andare ad eseguire due istruzioni che potenzialmente sono inutili e vanno ad impegnare il processore inutilmente. Si preferisce ugualmente questa soluzione perché si risente molto dal punto di vista della crescita delle prestazioni ed inoltre avendo il

messaggio di input preimpostato sappiamo che la gran maggior parte delle volte dovremmo comunque effettuare il salto all'etichetta "sem" e quindi quasi ad ogni iterazione dovremmo comunque eseguire le due istruzione DSUB e SLT.

Se ora andiamo ad eseguire di nuovo il codice ottimizzato otteniamo le seguenti prestazioni:

```
Execution

659 Cycles
564 Instructions
1.168 Cycles Per Instruction (CPI)


Stalls

0 RAW Stalls
0 WAW Stalls
0 WAR Stalls
0 Structural Stalls
91 Branch Taken Stalls
0 Branch Misprediction Stalls


Code size

188 Bytes
```

Come vediamo nell'immagine il numero dei cicli di clock effettuati per eseguire l'intero programma è sceso a 659 contro i 743 che avevamo prima. Altri due parametri che sono scesi di molto sono il CPI, che ha raggiunto un valore molto vicino a 1 (CPI ideale), e il numero di stalli dovuti ad una dipendenza RAW, che sono scesi a zero.

Queste prestazioni ci dicono che il codice è ottimizzato quasi al massimo, almeno per quanto il numero di stalli e il CPI. Allora con le prossime ottimizzazioni cerchiamo di far scendere ancora il numero dei cicli di clock.

Proseguendo ad analizzare il codice per vedere se si possono applicare ancora le tecniche di RR e IR consideriamo il seguente pezzo di codice:

```
        ;else2
        DADDI     r14, r7,   1    ;ottengo 'A'
        DADDI     r15, r6,   -1   ;ottengo 'Z'
        DADD      r14, r14,  r2   ;'A' + chiave in r14
        DADD      r16, r14,  r8   ;r14 + frase[i] in r16
        DSUB      r17, r16,  r15  ;r16 - 'Z' in r17
        DADDI     r17, r17,  -1   ;decremento r17
        SB        r17, 0(r5)      ;frasecifrata[i] = r17
        J         endif           ;salto incondizionato
```

Anche in questo caso possiamo applicare entrambe le tecniche, in particolare dovremmo applicare l'IR e di conseguenza ci servirà applicare anche il RR. Notiamo che in r14 e r15 vengono memorizzate essenzialmente due costanti: 'A' e 'Z'. Per il momento questi vengono calcolati ad ogni iterazione che entra in questo else e quindi spesso. Per diminuire il numero di istruzioni allora potremmo dedicare i due registri r14 e r15 per contenere le due costanti e quindi posizionare queste due istruzioni fuori dal loop. Questa scelta comporta di rinominare tutti quei registri r14 e r15 che vengono usati come destinazione altrimenti andremo a rompere l'efficacia del programma. Come ultimo accorgimento circa questo segmento di codice dobbiamo dire che le due istruzioni con cui ricaviamo 'A' e 'Z' non possono essere messe a caso fuori dal ciclo: possiamo posizionarle dopo aver calcolato il valore che contiene r6 e quello che contiene r7 altrimenti violeremo la dipendenza tra i dati di tipo RAW; tra queste due istruzioni e quelle che calcolano r6 e r7 ci deve essere almeno una istruzione in mezzo (poiché la latenza è 1) così da non dover utilizzare degli stalli.

Il codice MIPS assembly che si ottiene dopo questa prima ottimizzazione è il seguente:

```
.data
    frase:          .asciiz "ARCHITETTURA DEGLI ELABORATORI"    ;frase da cifrare
    newline:        .asciiz "\n"                                ;carattere newline
    frasecifrata:   .asciiz ""                                  ;frase cifrata
    numcar:         .word   31  ;numero di caratteri della frase
    chiave:         .word   13  ;chiave del cifrario
    indice:         .word   0   ;indice del ciclo for
```

```
    maxcar:         .word   91  ;'[' carattere massimo
    mincar:         .word   64  ;'@' carattere minimo

    CONTROL:        .word   0x10000 ;indirizzo di CONTROL
    DATA:           .word   0x10008 ;indirizzo di DATA

.text
    start: LWU    r28, DATA(r0)    ;carica DATA in r28
        LWU       r29, CONTROL(r0) ;carica CONTROL in r29
        LW        r1, numcar(r0)   ;carica numero di caratteri
        LW        r2, chiave(r0)   ;carica chiave cifratura
        LW        r3, indice(r0)   ;carica indice ciclo
        LW        r6, maxcar(r0)   ;carica carattere '['
        LW        r7, mincar(r0)   ;carica carattere '@'

        DADDI     r4, r0, frase           ;puntatore al primo carattere dell'array
        DADDI     r5, r0, frasecifrata    ;puntatore al primo carattere dell'array

        DADDI     r14, r7,   1    ;ottengo 'A'
        DADDI     r15, r6,   -1   ;ottengo 'Z'

        DADDI     r27, r0, 4      ;imposta DATA per stampare una stringa
        DADDI     r26, r0, frase  ;imposta stringa da stampare
        SD        r26, 0(r28)     ;imposta DATA con la stringa
        SD        r27, 0(r29)     ;stampa la stringa

    ;inizio ciclo for
    loop:  LB     r8,  0(r4)      ;leggi frase[i]
        DSUB      r11, r6,   r2   ;'[' - chiave
        SLT       r9,  r8,   r6   ;frase[i] < '[' ?
        SLT       r10, r7,   r8   ;'@' < frase[i] ?
        AND       r9,  r9,   r10  ;AND tra i due esiti precedenti
        SLT       r11, r8,   r11  ;frase[i] < ('[' - chiave) ?
        BNEZ      r9,  sem        ;if1: salta se diverso da zero

        SB        r8,  0(r5)      ;else1: frasecifrata[i] = frase[i]
        J         endif           ;salto incondizionato

    ;entro in if1
    sem:   BNEZ   r11, set        ;if2: salta se diverso da zero

        ;else2
        DADD      r12, r14,  r2   ;'A' + chiave in r14
        DADD      r16, r12,  r8   ;r12 + frase[i] in r16
        DSUB      r16, r16,  r15  ;r16 - 'Z' in r16
        DADDI     r16, r16,  -1   ;decremento r16
        SB        r16, 0(r5)      ;frasecifrata[i] = r16
        J         endif           ;salto incondizionato

    ;entro in if2
```

```
set:   DADD   r20, r8,   r2   ;frase[i] + chiave
       SB     r20, 0(r5)      ;frasecifrata[i] = r20

;aggiornamento valori
endif: DADDI  r4,  r4,   1   ;vai alla posizione successiva di frase
       DADDI  r5,  r5,   1   ;vai alla posizione successiva di frasecifrata
       SLT    r19, r3,   r1  ;indice < numcar ?
       DADDI  r3,  r3,   1   ;incremento indice
       BNEZ   r19, loop      ;salta se diverso da zero
       ;fine ciclo for

       ;stampa carattere newline
       DADDI  r27, r0, 4          ;imposta DATA per stampare una stringa
       DADDI  r26, r0, newline    ;imposta stringa da stampare
       SD     r26, 0(r28)         ;imposta DATA con la stringa
       SD     r27, 0(r29)         ;stampa la stringa

       ;stampa la frase cifrata
       DADDI  r27, r0, 4             ;imposta DATA per stampare una stringa
       DADDI  r26, r0, frasecifrata  ;imposta stringa da stampare
       SD     r26, 0(r28)            ;imposta DATA con la stringa
       SD     r27, 0(r29)            ;stampa la stringa

end:   HALT
```

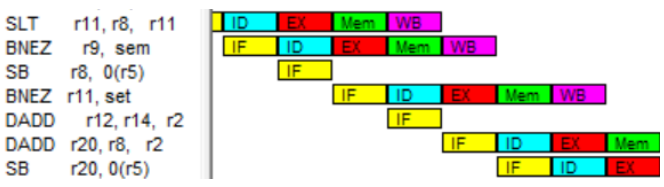
Le prestazioni ottenute sono le seguenti:

```
Execution
639 Cycles
544 Instructions
1.175 Cycles Per Instruction (CPI)

Stalls
0 RAW Stalls
0 WAW Stalls
0 WAR Stalls
0 Structural Stalls
91 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size
188 Bytes
```

Confrontando queste prestazioni con quelle ottenute subito prima notiamo che il CPI in realtà è leggermente aumentato ma questo ci va comunque bene perché abbiamo anche abbassato ancora il numero di cicli di clock di 20 raggiungendo 639 cicli di clock.

Ottimizzazione 2

Possiamo dire che a questo punto ci sono pochi casi in cui vedremo la pipeline non lavorare a pieno regime e questi sono quei casi in cui si verificano degli stalli dovuti agli abort delle istruzioni successive alle istruzioni di salto condizionato. Infatti, eseguendo il codice ci accorgiamo che il comportamento che assume è il seguente:



Dopo aver usufruito il più possibile dell'IR e del RR, ora proviamo ad applicare le tecniche di Loop Unrolling (LU) e Partial Loop Unrolling (PLU) con l'obbiettivo di abbassare il numero degli stalli Branch Taken e di creare nuove opportunità di utilizzo dell'IR così da provare a far diminuire ancora il numero dei cicli di clock e il CPI.

Sapendo a priori che il numero dei cicli che vengono effettuati è 30, quindi abbiamo un multiplo di 3, se srotolassimo il ciclo a passi che non sono multipli di 3 allora il PLU non funziona perché ci ritroveremo in una condizione per cui non usciamo mai dal ciclo. Allora quello che si vuole fare è scrivere all'interno del loop tre volte lo stesso codice, e quindi la variabile di conteggio ci dirà che il loop avanza a passi di 3.

Immaginiamo allora di applicare tale accorgimento ed eseguire il codice parzialmente srotolato a passi di 3. Inoltre, avendo dei costrutti if-else all'interno del loop dobbiamo modificare opportunamente le etichette a cui fanno riferimento i salti. Dopodiché le prestazioni che otteniamo eseguendo questo nuovo codice sono le seguenti:

```
Execution

611 Cycles
536 Instructions
1.140 Cycles Per Instruction (CPI)


Stalls

0 RAW Stalls
0 WAW Stalls
0 WAR Stalls
0 Structural Stalls
71 Branch Taken Stalls
0 Branch Misprediction Stalls


Code size

364 Bytes
```

Quindi applicando il PLU siamo riusciti ad abbassare ancora il numero di cicli di clock ed il CPI, ma questa volta anche il numero degli stalli Branch Taken che sono scesi a 71. Allora questo ci indica che il PLU è stato applicato correttamente in quanto ci aspettavamo che il numero di stalli scendesse di 20, cioè di 2/3 rispetto alle 30 iterazioni precedenti.

A questo punto sappiamo che potrebbe esserci la possibilità di riapplicare l'IR al nuovo codice parzialmente srotolato:

- Come prima cosa andiamo a vedere come si comporta il contatore di ciclo insieme a tutte quelle istruzioni che definiscono la condizione di uscita dal ciclo. Avendo il codice replicato tre volte, vuol dire che andiamo ad incrementare di 1 il contatore per tre volte, e per tre volte andiamo a controllare se tale contatore sia minore del numero degli elementi dell'array di input. Quello che possiamo fare in questo caso è fondere queste 6 istruzioni in 4, ossia:

```
SLT     r19, r3,   r1  ;indice < numcar ?
DADDI   r3,  r3,   1   ;incremento indice
SLT     r19, r3,   r1  ;indice < numcar ?
DADDI   r3,  r3,   1   ;incremento indice
SLT     r19, r3,   r1  ;indice < numcar ?
DADDI   r3,  r3,   1   ;incremento indice

BNEZ    r19, loop      ;salta se diverso da zero
```

```
SLT     r19, r3,   r1  ;indice < numcar ?
DADDI   r3,  r3,   1   ;incremento indice
SLT     r19, r3,   r1  ;indice < numcar ?
DADDI   r3,  r3,   2   ;incremento indice

BNEZ    r19, loop      ;salta se diverso da zero
```

  In questo modo facciamo scendere i cicli di clock a 589 a costo di far salire il CPI a 1.143.

- Un altro aspetto che posso considerare è quello del caricamento degli elementi dell'array nei registri e l'incremento che facciamo ad ogni ciclo per passare all'elemento successivo. Potremmo provare anche in questo caso a spostarci ad ogni iterazione di tre elementi con una sola istruzione a patto però che aggiustiamo l'offset nelle istruzioni di caricamento nei registri interni. In questo modo otteniamo un CPI pari a 1.152 ma miglioriamo dal punto di vista dell'efficienza del codice perché scendiamo a 567 cicli di clock.

- Un discorso analogo a quello precedente lo possiamo fare per le istruzioni di caricamento in memoria dei risultati calcolati e quello che otteniamo sono delle prestazioni che aumentano in quanto, anche se il CPI è salito ancora a 1.160, il numero di cicli di clock è sceso a 545.

Il codice MIPS assembly che si ottiene dopo questa seconda ottimizzazione è il seguente:

```
.data
    frase:          .asciiz "ARCHITETTURA DEGLI ELABORATORI"    ;frase da cifrare
    newline:        .asciiz "\n"                                ;carattere newline
    frasecifrata:   .asciiz ""                                  ;frase cifrata
    numcar:         .word   31  ;numero di caratteri della frase
    chiave:         .word   13  ;chiave del cifrario
    indice:         .word   0   ;indice del ciclo for
    maxcar:         .word   91  ;'[' carattere massimo
    mincar:         .word   64  ;'@' carattere minimo

    CONTROL:        .word   0x10000 ;indirizzo di CONTROL
    DATA:           .word   0x10008 ;indirizzo di DATA

.text
    start: LWU    r28, DATA(r0)    ;carica DATA in r28
        LWU       r29, CONTROL(r0) ;carica CONTROL in r29
        LW        r1, numcar(r0)   ;carica numero di caratteri
        LW        r2, chiave(r0)   ;carica chiave cifratura
        LW        r3, indice(r0)   ;carica indice ciclo
        LW        r6, maxcar(r0)   ;carica carattere '['
        LW        r7, mincar(r0)   ;carica carattere '@'

        DADDI     r4, r0, frase           ;puntatore al primo carattere dell'array
        DADDI     r5, r0, frasecifrata    ;puntatore al primo carattere dell'array

        DADDI     r14, r7,   1    ;ottengo 'A'
        DADDI     r15, r6,   -1   ;ottengo 'Z'

        DADDI     r27, r0, 4      ;imposta DATA per stampare una stringa
        DADDI     r26, r0, frase  ;imposta stringa da stampare
        SD        r26, 0(r28)     ;imposta DATA con la stringa
        SD        r27, 0(r29)     ;stampa la stringa

    ;inizio ciclo for
    loop:  LB     r8,  0(r4)      ;leggi frase[i]
        DSUB      r11, r6,   r2   ;'[' - chiave
        SLT       r9,  r8,   r6   ;frase[i] < '[' ?
        SLT       r10, r7,   r8   ;'@' < frase[i] ?
        AND       r9,  r9,   r10  ;AND tra i due esiti precedenti
        SLT       r11, r8,   r11  ;frase[i] < ('[' - chiave) ?
        BNEZ      r9,  sem        ;if1: salta se diverso da zero

        SB        r8,  0(r5)      ;else1: frasecifrata[i] = frase[i]
```

```
    J         endif           ;salto incondizionato

;entro in if1
sem:   BNEZ   r11, set        ;if2: salta se diverso da zero

    ;else2
    DADD      r12, r14,  r2   ;'A' + chiave in r14
    DADD      r16, r12,  r8   ;r12 + frase[i] in r16
    DSUB      r16, r16,  r15  ;r16 - 'Z' in r16
    DADDI     r16, r16,  -1   ;decremento r16
    SB        r16, 0(r5)      ;frasecifrata[i] = r16
    J         endif          ;salto incondizionato

;entro in if2
set:   DADD   r20, r8,   r2   ;frase[i] + chiave
    SB        r20, 0(r5)      ;frasecifrata[i] = r20

endif:

2:     LB     r8,  1(r4)      ;leggi frase[i]
    DSUB      r11, r6,   r2   ;'[' - chiave
    SLT       r9,  r8,   r6   ;frase[i] < '[' ?
    SLT       r10, r7,   r8   ;'@' < frase[i] ?
    AND       r9,  r9,   r10  ;AND tra i due esiti precedenti
    SLT       r11, r8,   r11  ;frase[i] < ('[' - chiave) ?
    BNEZ      r9,  sem2       ;if1: salta se diverso da zero

    SB        r8,  1(r5)      ;else1: frasecifrata[i] = frase[i]
    J         endif2          ;salto incondizionato

;entro in if1
sem2:  BNEZ   r11, set2       ;if2: salta se diverso da zero

    ;else2
    DADD      r12, r14,  r2   ;'A' + chiave in r14
    DADD      r16, r12,  r8   ;r12 + frase[i] in r16
    DSUB      r16, r16,  r15  ;r16 - 'Z' in r16
    DADDI     r16, r16,  -1   ;decremento r16
    SB        r16, 1(r5)      ;frasecifrata[i] = r16
    J         endif2          ;salto incondizionato

;entro in if2
set2:  DADD   r20, r8,   r2   ;frase[i] + chiave
    SB        r20, 1(r5)      ;frasecifrata[i] = r20

endif2:

3:     LB     r8,  2(r4)      ;leggi frase[i]
    DSUB      r11, r6,   r2   ;'[' - chiave
    SLT       r9,  r8,   r6   ;frase[i] < '[' ?
```

```
        SLT       r10, r7,   r8   ;'@' < frase[i] ?
        AND       r9,  r9,   r10  ;AND tra i due esiti precedenti
        SLT       r11, r8,   r11  ;frase[i] < ('[' - chiave) ?
        BNEZ      r9,  sem3        ;if1: salta se diverso da zero

        SB        r8,  2(r5)      ;else1: frasecifrata[i] = frase[i]
        J         endif3          ;salto incondizionato

    ;entro in if1
    sem3:  BNEZ   r11, set3        ;if2: salta se diverso da zero

        ;else2
        DADD      r12, r14,  r2   ;'A' + chiave in r14
        DADD      r16, r12,  r8   ;r12 + frase[i] in r16
        DSUB      r16, r16,  r15  ;r16 - 'Z' in r16
        DADDI     r16, r16,  -1   ;decremento r16
        SB        r16, 2(r5)      ;frasecifrata[i] = r16
        J         endif3          ;salto incondizionato

    ;entro in if2
    set3:  DADD   r20, r8,   r2   ;frase[i] + chiave
        SB        r20, 2(r5)      ;frasecifrata[i] = r20

    ;aggiornamento valori
    endif3: DADDI r4,  r4,   3   ;vai alla posizione successiva di frase
        DADDI     r5,  r5,   3   ;vai alla posizione successiva di frasecifrata

        SLT       r19, r3,   r1  ;indice < numcar ?
        DADDI     r3,  r3,   1   ;incremento indice
        SLT       r19, r3,   r1  ;indice < numcar ?
        DADDI     r3,  r3,   2   ;incremento indice

        BNEZ      r19, loop      ;salta se diverso da zero
        ;fine ciclo for

        ;stampa carattere newline
        DADDI     r27, r0, 4          ;imposta DATA per stampare una stringa
        DADDI     r26, r0, newline    ;imposta stringa da stampare
        SD        r26, 0(r28)         ;imposta DATA con la stringa
        SD        r27, 0(r29)         ;stampa la stringa

        ;stampa la frase cifrata
        DADDI     r27, r0, 4             ;imposta DATA per stampare una stringa
        DADDI     r26, r0, frasecifrata  ;imposta stringa da stampare
        SD        r26, 0(r28)            ;imposta DATA con la stringa
        SD        r27, 0(r29)            ;stampa la stringa

    end:   HALT
```

Le prestazioni sono:

```
Execution
545 Cycles
470 Instructions
1.160 Cycles Per Instruction (CPI)

Stalls
0 RAW Stalls
0 WAW Stalls
0 WAR Stalls
0 Structural Stalls
71 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size
340 Bytes
```

Vediamo che siamo riusciti grazie ad un PLU a riutilizzare la tecnica di IR per migliorare le prestazioni del codice. Infatti, rispetto al codice del paragrafo precedente abbiamo abbassato il CPI da 1.175 a 1.160, il numero di cicli di clock da 639 a 545 e questa volta siamo anche riusciti ad abbassare il numero degli stalli da 91 a 71.

Ottimizzazione 3

Adesso, prendendo in considerazione le prestazioni che abbiamo ottenuto finora, ci accorgiamo che la finestra di ottimizzazione si è ristretta in quanto abbiamo quasi raggiunto un CPI ideale, abbiamo eliminato quasi tutti gli stalli eliminabili, ma soprattutto abbiamo raggiunto un numero di cicli di clock quasi 200 cicli più basso rispetto a quello del codice originale con cui siamo partiti.

Quello che possiamo fare adesso è provare ad utilizzare di nuovo PLU o LU e IR. Applichiamo allora uno srotolamento totale, quindi il LU. Quello che ci si aspetta è che non avendo più un ciclo, vengano eliminati tutti gli stalli Branch Taken che non sono dovuti ai costrutti if-else. Il LU verrà applicato al codice già ottimizzato e quindi in questo LU rendiamo implicita anche la tecnica di IR.

Il simulatore WinMIPS64 utilizzato per questo progetto non supporta file di dimensione superiore ai 1028 byte, allora nel menù configurazione dobbiamo cambiare l'architettura e specificare che il campo "Code Address Bus" sia pari a 12 perché otterremmo $2^{12} = 4096$ byte di memoria per caricare il codice. Questo campo, infatti, ci permette di modificare il numero dei cavi del bus degli indirizzi.

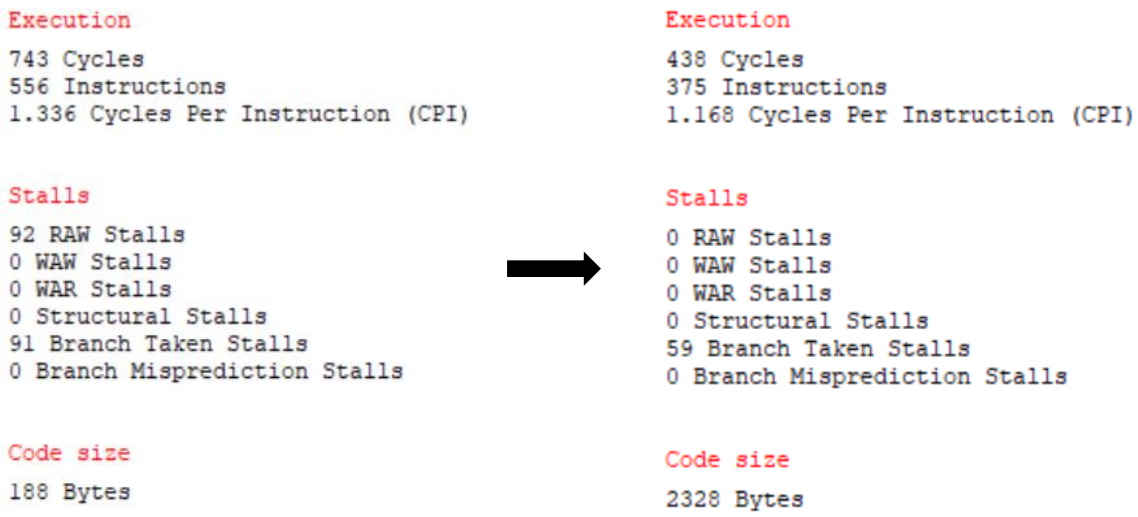
Le prestazioni che otteniamo applicando il LU, e quindi eliminando completamente il loop, insieme all'IR sono le seguenti:

```
Execution
438 Cycles
375 Instructions
1.168 Cycles Per Instruction (CPI)

Stalls
0 RAW Stalls
0 WAW Stalls
0 WAR Stalls
0 Structural Stalls
59 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size
2328 Bytes
```

Ora possiamo dire di aver aumentato l'efficienza di questo codice ancora di più in quanto il numero dei cicli clock è sceso di oltre 100 cicli. Il CPI è leggermente salito ma comunque rimane vicino a quel valore ideale che è 1. Infine, vediamo che il numeri degli stalli Branch Taken è sceso a 59. Questo accade perché abbiamo eliminato completamente la reiterazione del codice. Quindi, ovviamente non si portano gli stalli a zero perché abbiamo dei salti condizionati per controllare il flusso del codice, ma possiamo dire di aver rimosso tutti quegli stalli che erano causati dall'implementazione del ciclo for e quindi di aver rimosso degli abort.

Ottimizzazioni ed osservazioni conclusive

In seguito alle prestazioni che abbiamo ottenuto nelle varie versioni di codice dello stesso programma abbiamo visto che facendo un LU totale otteniamo l'efficienza migliore. Quindi in conclusione possiamo dire di aver ottenuto un'ottimizzazione del codice importante e lo possiamo vedere mettendo a confronto i parametri del codice con cui siamo partiti con quelli che abbiamo ottenuto applicando LU e IR:



Si fa notare che per ottenere queste prestazioni abbiamo dovuto sacrificare la generalità del codice in quanto questo è stato specializzato il più possibile in base ai dati di input su cui dovevamo lavorare.

Se volessimo mantenere il codice in una forma più generale che funzioni anche con dati di input differenti e quindi con messaggi definiti dall'utente dovremmo perdere tali prestazioni ottime. Come prima cosa, infatti, non potremmo sicuramente utilizzare il LU dato che questo presuppone la conoscenza della dimensione (numero dei caratteri) del messaggio e quindi dell'input.

Torniamo allora a quella versione di codice che garantiva comunque delle buone prestazioni e che sfruttava insieme il PLU e l'IR. In questa versione, presupponendo che il messaggia fosse composto da 31 caratteri, abbiamo srotolato il codice a passi di 3. Per rendere questo codice da subito più generale potremmo invertire l'incremento del contatore, ossia lo possiamo far partire da n e farlo arrivare fino a zero. In questo modo faremmo funzionare il codice per messaggi di dimensione pari a multipli di 3, in più elimineremmo tutte le istruzioni di controllo di diseguaglianza e quindi diminuiremmo anche il numero di cicli di clock per eseguire il programma (a patto di non introdurre degli stalli).

```
;aggiornamento valori
endif3: DADDI r3,  r3,  -3   ;decremento indice
    DADDI     r4,  r4,   3   ;vai alla posizione successiva di frase
    DADDI     r5,  r5,   3   ;vai alla posizione successiva di frasecifrata
    BNEZ      r3, loop      ;salta se diverso da zero
```

Nel nostro caso però il messaggio è composto da un numero di caratteri che non è un multiplo di 3. Quello che possiamo fare è dire che se il messaggio di input è un multiplo di 3 allora possiamo utilizzare questa tecnica appena descritta, ma se così non fosse allora avremmo un resto e questo dobbiamo gestirlo con un ulteriore ciclo al quale non possiamo applicare PLU o LU non conoscendo il resto a priori.

```
;ciclo per gestire il resto
loopr: BEQZ   r3, stampa      ;salta se non c'è resto
    LB        r8,  0(r4)      ;leggi frase[i]
```

```
        DSUB      r11, r6,   r2   ;'[' - chiave
        SLT       r9,  r8,   r6   ;frase[i] < '[' ?
        SLT       r10, r7,   r8   ;'@' < frase[i] ?
        AND       r9,  r9,   r10  ;AND tra i due esiti precedenti
        SLT       r11, r8,   r11  ;frase[i] < ('[' - chiave) ?
        BNEZ      r9,  semr       ;if1: salta se diverso da zero

        SB        r8,  0(r5)      ;else1: frasecifrata[i] = frase[i]
        J         endifr          ;salto incondizionato

    ;entro in if1
    semr:   BNEZ   r11, setr      ;if2: salta se diverso da zero

        ;else2
        DADD      r12, r14,  r2   ;'A' + chiave in r14
        DADD      r16, r12,  r8   ;r12 + frase[i] in r16
        DSUB      r16, r16,  r15  ;r16 - 'Z' in r16
        DADDI     r16, r16,  -1   ;decremento r16
        SB        r16, 0(r5)      ;frasecifrata[i] = r16
        J         endifr          ;salto incondizionato

    ;entro in if2
    setr:   DADD   r20, r8,   r2  ;frase[i] + chiave
        SB         r20, 0(r5)     ;frasecifrata[i] = r20

    ;aggiornamento valori
    endifr: DADDI r3,  r3,  -1   ;decremento indice
        DADDI     r4,  r4,   1   ;vai alla posizione successiva di frase
        DADDI     r5,  r5,   1   ;vai alla posizione successiva di frasecifrata
        J         loopr          ;iterazione
    ;fine ciclo for
```

Quello che ci rimane da gestire sono i casi in cui il messaggio sia composto da zero elementi e quello in cui questo sia composto da un numero inferiore di 3 elementi. Per ovviare a queste due situazioni ci basta spostare il controllo e l'istruzione di salto all'inizio del primo loop e iterarlo aggiungendo una istruzione di salto incondizionato in fondo.

Il codice che otteniamo nella versione più generale e comunque ottimizzata è il seguente:

```
.data
    frase:         .asciiz "ARCHITETTURA DEGLI ELABORATORI"  ;frase da cifrare
    newline:       .asciiz "\n"                              ;carattere newline
    frasecifrata:  .asciiz ""                                ;frase cifrata
    chiave:        .word   13  ;chiave del cifrario
    indice:        .word   31  ;indice del ciclo for
    maxcar:        .word   91  ;'[' carattere massimo
    mincar:        .word   64  ;'@' carattere minimo

    CONTROL:       .word   0x10000 ;indirizzo di CONTROL
    DATA:          .word   0x10008 ;indirizzo di DATA
```

```
.text
    start: LWU    r28, DATA(r0)     ;carica DATA in r28
        LWU       r29, CONTROL(r0)  ;carica CONTROL in r29
        LW        r2,  chiave(r0)   ;carica chiave cifratura
        LW        r3,  indice(r0)   ;carica indice ciclo
        LW        r6,  maxcar(r0)   ;carica carattere '['
        LW        r7,  mincar(r0)   ;carica carattere '@'

        DADDI     r4, r0, frase         ;puntatore al primo carattere dell'array
        DADDI     r5, r0, frasecifrata  ;puntatore al primo carattere dell'array

        DADDI     r14, r7,   1    ;ottengo 'A'
        DADDI     r15, r6,   -1   ;ottengo 'Z'
        SLTI      r30, r3,   3    ;set r30 se r3 < 3

        DADDI     r27, r0, 4      ;imposta DATA per stampare una stringa
        DADDI     r26, r0, frase  ;imposta stringa da stampare
        SD        r26, 0(r28)     ;imposta DATA con la stringa
        SD        r27, 0(r29)     ;stampa la stringa

    ;inizio ciclo for
    loop:  BNEZ   r30, loopr      ;salta se diverso da zero
                                  ;se abbiamo meno di 3 lettere
        LB        r8,  0(r4)      ;leggi frase[i]
        DSUB      r11, r6,   r2   ;'[' - chiave
        SLT       r9,  r8,   r6   ;frase[i] < '[' ?
        SLT       r10, r7,   r8   ;'@' < frase[i] ?
        AND       r9,  r9,   r10  ;AND tra i due esiti precedenti
        SLT       r11, r8,   r11  ;frase[i] < ('[' - chiave) ?
        BNEZ      r9,  sem        ;if1: salta se diverso da zero

        SB        r8,  0(r5)      ;else1: frasecifrata[i] = frase[i]
        J         endif           ;salto incondizionato

    ;entro in if1
    sem:   BNEZ   r11, set        ;if2: salta se diverso da zero

        ;else2
        DADD      r12, r14,  r2   ;'A' + chiave in r14
        DADD      r16, r12,  r8   ;r12 + frase[i] in r16
        DSUB      r16, r16,  r15  ;r16 - 'Z' in r16
        DADDI     r16, r16,  -1   ;decremento r16
        SB        r16, 0(r5)      ;frasecifrata[i] = r16
        J         endif          ;salto incondizionato

    ;entro in if2
    set:   DADD   r20, r8,   r2   ;frase[i] + chiave
        SB        r20, 0(r5)      ;frasecifrata[i] = r20
```

```
endif:

2:      LB      r8,  1(r4)      ;leggi frase[i]
        DSUB    r11, r6,   r2   ;'[' - chiave
        SLT     r9,  r8,   r6   ;frase[i] < '[' ?
        SLT     r10, r7,   r8   ;'@' < frase[i] ?
        AND     r9,  r9,   r10  ;AND tra i due esiti precedenti
        SLT     r11, r8,   r11  ;frase[i] < ('[' - chiave) ?
        BNEZ    r9,  sem2       ;if1: salta se diverso da zero

        SB      r8,  1(r5)      ;else1: frasecifrata[i] = frase[i]
        J       endif2          ;salto incondizionato

;entro in if1
sem2:   BNEZ    r11, set2       ;if2: salta se diverso da zero

        ;else2
        DADD    r12, r14,  r2   ;'A' + chiave in r14
        DADD    r16, r12,  r8   ;r12 + frase[i] in r16
        DSUB    r16, r16,  r15  ;r16 - 'Z' in r16
        DADDI   r16, r16,  -1   ;decremento r16
        SB      r16, 1(r5)      ;frasecifrata[i] = r16
        J       endif2          ;salto incondizionato

;entro in if2
set2:   DADD    r20, r8,   r2   ;frase[i] + chiave
        SB      r20, 1(r5)      ;frasecifrata[i] = r20

endif2:

3:      LB      r8,  2(r4)      ;leggi frase[i]
        DSUB    r11, r6,   r2   ;'[' - chiave
        SLT     r9,  r8,   r6   ;frase[i] < '[' ?
        SLT     r10, r7,   r8   ;'@' < frase[i] ?
        AND     r9,  r9,   r10  ;AND tra i due esiti precedenti
        SLT     r11, r8,   r11  ;frase[i] < ('[' - chiave) ?
        BNEZ    r9,  sem3        ;if1: salta se diverso da zero

        SB      r8,  2(r5)      ;else1: frasecifrata[i] = frase[i]
        J       endif3          ;salto incondizionato

;entro in if1
sem3:   BNEZ    r11, set3        ;if2: salta se diverso da zero

        ;else2
        DADD    r12, r14,  r2   ;'A' + chiave in r14
        DADD    r16, r12,  r8   ;r12 + frase[i] in r16
        DSUB    r16, r16,  r15  ;r16 - 'Z' in r16
        DADDI   r16, r16,  -1   ;decremento r16
        SB      r16, 2(r5)      ;frasecifrata[i] = r16
```

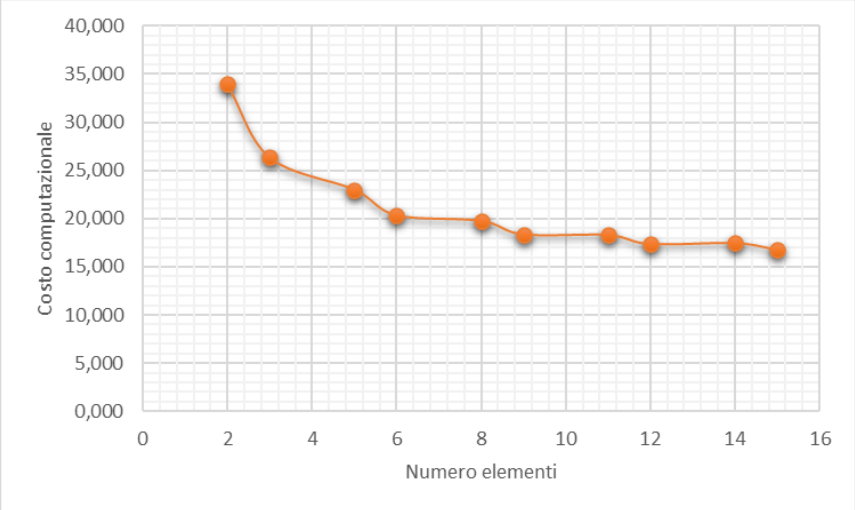
```
    J         endif3          ;salto incondizionato

;entro in if2
set3:  DADD   r20, r8,   r2   ;frase[i] + chiave
    SB        r20, 2(r5)      ;frasecifrata[i] = r20

;aggiornamento valori
endif3: DADDI r3,  r3,  -3   ;decremento indice
    DADDI     r4,  r4,   3   ;vai alla posizione successiva di frase
    DADDI     r5,  r5,   3   ;vai alla posizione successiva di frasecifrata
    SLTI      r30, r3,   3   ;set r30 se r3 < 3
    J         loop           ;iterazione: ho più di 3 elementi
    ;fine ciclo for

;ciclo per gestire il resto
loopr: BEQZ   r3, stampa      ;salta se non c'è resto
    LB        r8,  0(r4)      ;leggi frase[i]
    DSUB      r11, r6,   r2   ;'[' - chiave
    SLT       r9,  r8,   r6   ;frase[i] < '[' ?
    SLT       r10, r7,   r8   ;'@' < frase[i] ?
    AND       r9,  r9,   r10  ;AND tra i due esiti precedenti
    SLT       r11, r8,   r11  ;frase[i] < ('[' - chiave) ?
    BNEZ      r9,  semr       ;if1: salta se diverso da zero

    SB        r8,  0(r5)      ;else1: frasecifrata[i] = frase[i]
    J         endifr          ;salto incondizionato

;entro in if1
semr:   BNEZ   r11, setr      ;if2: salta se diverso da zero

    ;else2
    DADD      r12, r14,  r2   ;'A' + chiave in r14
    DADD      r16, r12,  r8   ;r12 + frase[i] in r16
    DSUB      r16, r16,  r15  ;r16 - 'Z' in r16
    DADDI     r16, r16,  -1   ;decremento r16
    SB        r16, 0(r5)      ;frasecifrata[i] = r16
    J         endifr          ;salto incondizionato

;entro in if2
setr:   DADD   r20, r8,   r2  ;frase[i] + chiave
    SB         r20, 0(r5)     ;frasecifrata[i] = r20

;aggiornamento valori
endifr: DADDI r3,  r3,  -1   ;decremento indice
    DADDI     r4,  r4,   1   ;vai alla posizione successiva di frase
    DADDI     r5,  r5,   1   ;vai alla posizione successiva di frasecifrata
    J         loopr          ;iterazione
;fine ciclo for

;stampa carattere newline
```

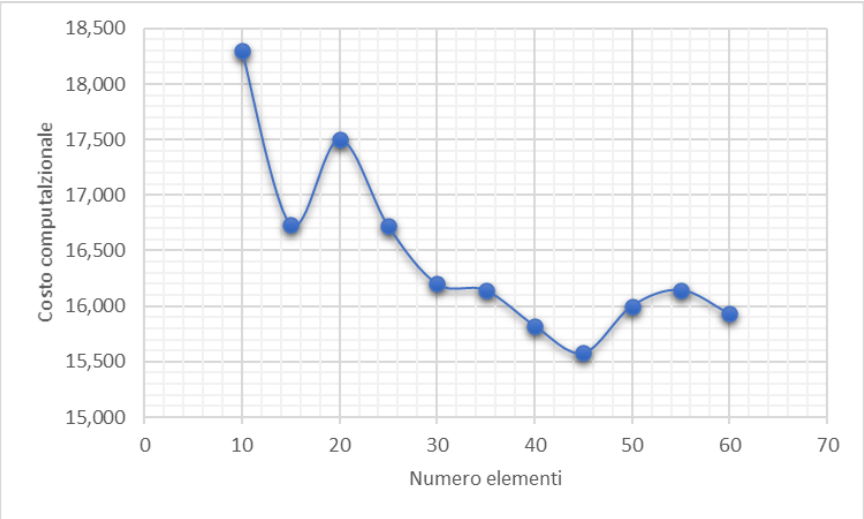
```
stampa: DADDI r27, r0, 4          ;imposta DATA per stampare una stringa
    DADDI     r26, r0, newline    ;imposta stringa da stampare
    SD        r26, 0(r28)         ;imposta DATA con la stringa
    SD        r27, 0(r29)         ;stampa la stringa

    ;stampa la frase cifrata
    DADDI     r27, r0, 4             ;imposta DATA per stampare una stringa
    DADDI     r26, r0, frasecifrata  ;imposta stringa da stampare
    SD        r26, 0(r28)            ;imposta DATA con la stringa
    SD        r27, 0(r29)            ;stampa la stringa

end:   HALT
```

Le prestazioni che si ottengono senza perdere l'efficacia del codice sono:

```
Execution
518 Cycles
442 Instructions
1.172 Cycles Per Instruction (CPI)

Stalls
0 RAW Stalls
0 WAW Stalls
0 WAR Stalls
0 Structural Stalls
72 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size
428 Bytes
```

Infine, possiamo andare a vedere qual è il costo computazionale per ogni elemento aggiunto rispetto a questo programma. Questo lo possiamo fare perché abbiamo aggiunto tutti quegli accorgimenti che rendono il codice più efficace e generale. Se andiamo a considerare come parametri il numero degli elementi che stiamo elaborando (n) e il numero dei cicli di clock (CPUC) che servono per terminare l'esecuzione possiamo calcolare il costo computazionale di un elemento seguendo la relazione $\frac{CPUC}{n}$. Ora costruiamo dei grafici per vedere come si comporta questo parametro e quello che otteniamo viene mostra qui di seguito:





Questi due grafici ci mostrano che aggiungere degli elementi che ci portano ad avere un numero di caratteri pari a un multiplo di 3 comporta un costo computazionale minore rispetto a quando questo non accade. Questo lo possiamo apprezzare particolarmente nel secondo grafico poiché presenta dei massimi in prossimità di valori che sono "vicini" a multipli di 3 e dei minimi in prossimità di valori che invece sono proprio multipli di 3. Con il primo grafico vediamo anche aggiungendo elementi abbassiamo il costo computazionale perché le istruzioni che riguardano l'inizializzazione dei puntatori, indici e altre variabili o le istruzioni per stampare il messaggio di output vengono diluite tanto più quanti sono gli elementi.