1506
UNIVERSITÀ
DEGLI STUDI
DI URBINO
CARLO BO

DISPEA
DIPARTIMENTO DI
SCIENZE PURE E
APPLICATE

Progetto di Architettura degli Elaboratori
“ Ottimizzazione Statica dell’Algoritmo di ricerca Binaria per Array Ordinati “

Autori
Alessio Biagioli - Alessio Visentino - Elia Renzoni

Relazione del progetto dell’insegnamento di Architettura degli Elaboratori, Sessione Estiva, anno accademico 2022/2023. Universitá degli Studi di Urbino, Carlo Bo.

1.0 Specifica del Progetto

1.1 Scopo del Progetto :

Il progetto riguarda l'implementazione di un sistema di recupero password e controllo di accessi a un'applicazione, cercando nel proprio database (file.txt) se il codice del nuovo utente è già in uso; abbiamo deciso di ottimizzare l'algoritmo di ricerca dicotomica per array già ordinati che ci consente di verificare l'esistenza pregressa dei nuovi codici utente; l'algoritmo in questione verrà successivamente tradotto e formalizzato in Assembly, più precisamente secondo l'instruction set definito dal simulatore WinMips64, sul quale abbiamo testato il corretto funzionamento del codice. Un altro importante obiettivo del progetto è quello di ottimizzare l'algoritmo stesso, per ridurre il più possibile il numero di stalli che si presentano e per ridurre, quanto basta, il CPI.
L'algoritmo di Ricerca Binaria è un algoritmo che risolve il problema della ricerca di un dato valore all'interno degli elementi dell'array; questo algoritmo è il più efficace, in termini di complessità asintotica - la quale rappresenta in modo euristico il numero di passi che dovrà svolgere il calcolatore per eseguire il programma - in quanto effettua una ricerca più "intelligente" poiché dimezza lo spazio di ricerca a ogni iterazione *i-esima*, individuando quindi dei sotto-array nei quali proseguire l'attività confrontando il valore dell'elemento mezzo con il dato cercato. Per fare questo deve poter aver tre contatori che indichino la porzione d'array in esame, ovvero l'indice di sinistra, quello di destra e infine quello di mezzo, il quale viene calcolato facendo la media tra i primi due contatori e aggiornando il valore dei primi due ad ogni iterazione. Nel nostro caso l'algoritmo deve cercare su un array di numeri interi, i quali rappresentano dei codici di accesso al sistema, memorizzati in un file e caricati nella struttura dati.

1.2 Specifica Algoritmica :

```c
int ricerca_password(int *password, int codice_utente, int numero_elementi) {

  int sx, dx, mx;

  for (sx = 0, dx = numero_elementi - 1, mx = (sx + dx) / 2;
              ((sx <= dx) && (password[mx] != codice_utente));
       mx = (sx + dx) / 2) {
        if (codice_utente < password[mx])
          dx = mx - 1;
        else
          sx = mx + 1;
  }

  return ((sx <= dx)? codice_utente : 1);

}
```

Figura 1.1 : Algoritmo di Ricerca Binaria per Array Ordinati, formalizzato in C.

1.3 Benchmark :

L'algoritmo che abbiamo implementato risolve in modo efficace il problema della ricerca, in quanto la sua funzione tempo d'esecuzione appartiene alla classe di complessità logaritmica, nel suo caso pessimo, mentre nel caso ottimo appartiene alla classe di complessità costante; ciò significa che a fronte della dimensione dei dati di ingresso *n,* l'algoritmo effettuerà un numero di passi che è logaritmico al crescere di *n.*
Esempio :
$T(n) = O(\log n),\ if\ n = 1000000 \ -> \ T(n) = 6.$
Ciò non accade nel caso di un algoritmo di ricerca sequenziale, in quanto si ha una complessità lineare, compiendo quindi un numero di passi pari alla dimensione di *n.*
Tuttavia questo algoritmo di ricerca binaria non è la soluzione migliore a tutte le casistiche, in quanto esso ha bisogno che l'array sia già ordinato, trascinandosi dietro la complessità asintotica (sia riferita al numero di passi, ma anche al consumo di memoria) dell'algoritmo di ordinamento usato; nel nostro caso abbiamo usato un *select sort* che presenta una complessità quadratica; quindi l'uso di un algoritmo di ricerca binaria è poco consigliato in un contesto in cui i dati cambiano molto di frequente.

la funzione C, da noi implementata, ha i seguenti parametri:

1) Array di interi, il quale contiene i codici di accesso prelevati da un file.txt;
2) Numero degli elementi dell'array, parametro fondamentale quando si lavora con strutture statiche come gli array, il quale ha un valore proporzionale al numero dei dati contenuti nel file.txt;

3) Codice di accesso, il quale rappresenta il valore da cercare.

Abbiamo scelto di lavorare con gli array in quanto l'unico obiettivo del programma è quello di fornire molto velocemente una risposta all'utente, questo non sarebbe stato possibile se avessimo lavorato con delle strutture dati dinamiche, come le Liste, le quali non hanno un accesso diretto al dato in memoria, ma sono formate da "catene" di puntatori da costruire per raggiungere un dato attributo chiave di un elemento *E-esimo*.

2.0 Prima Implementazione in Assembly

```
; @authors Elia Renzoni, Alessio Visentino e Alessio Biagioli
; @date 20/05/2023
; Progetto per la Sessione Estiva di Architettura degli Elaboratori
; @brief Implementazione in MIPS Assembly dell'algoritmo di ricerca dicotomica per array giá
ordinati

          .data

password:         .word 12, 37, 57, 64, 89, 95
codice_utente:    .word 57
indice_sx:        .word 0
indice_dx:        .word 5
indice_mx:        .word 0
divisore:         .word 2
var_supporto:     .word 1

          .text

start:

      lw r1, indice_sx(r0)    ; indice sx di ogni spazio di ricerca
      lw r2, indice_dx(r0)    ; indice dx di ogni spazio di ricerca
      lw r3, indice_mx(r0)    ; indice di mx di ogni spazio di ricerca
      lw r4, var_supporto(r0)  ; variabile per effettuare la sottrazione
      lw r5, codice_utente(r0) ; variabile contenente il codice utente da cercare
      lw r6, divisore(r0)    ; variabile contenente il divisore = 2
      daddi r7, r0, password  ; puntatore al primo elemento dell'array.

ricerca_loop:

      dadd r3, r1, r2  ; somma l'indice di sinistra con quello di destra
      ddiv r3, r3, r6  ; calcola l'indice di mezzo -> mx = sx + dx / 2

imposta_elemento_mezzo:

      slt r11, r12, r3    ; if r12 < r3
      bnez r11, imposta_elemento_mezzo2 ; if r11 != 0 then imposta_elemento_mezzo2
      beqz r11, ultimo_controllo ; if r11 == 0 then ultimo_controllo

ultimo_controllo: ; controlla se r12 è uguale o maggiore di r3

      bne r12, r3, imposta_elemento_mezzo3 ; if r12 != r3 then imposta_elemento_mezzo3
      j indice_impostato  ; if r12 == r3 then indice_impostato

imposta_elemento_mezzo2: ; imposta l'indice if r12 < r3

      daddi r7, r7, 8 ; incrementa il puntatore
      daddi r12, r12, 1 ; incrementa il contatore del ciclo
      j imposta_elemento_mezzo ; ritorna all'inizio del ciclo

imposta_elemento_mezzo3: ; imposta l'indice if r12 > r3

      daddi r7, r7, -8 ; decrementa il puntatore
      daddi r12, r12, -1 ; decrementa il contatore del ciclo
      j imposta_elemento_mezzo ; ritorna all'inizio del ciclo

indice_impostato: ; indice dell'elemento impostato correttamente

      slt r8, r1, r2   ; if r1 < r2
      bnez r8, seconda_validazione ; if r8 != 0 then seconda_validazione
```

```
      beqz r8, controlla_uguaglianza ; if r8 == 0 then controlla_uguaglianza

controlla_uguaglianza: ; controllo l'uguaglianza tra gli indici

      beq r1, r2, seconda_validazione ; if r1 == r2 then seconda_validazione
      bne r1, r2, fine_ricerca  ; if r1 != r2 then fine_ricerca, in quanto r1 > r2

seconda_validazione: ; secondo controllo

      lw r9, 0(r7)   ; carico in r9 il valore di password[mx]
      bne r9, r5, continua_ricerca   ; if password[mx] != codice_utente then continua_ricerca
      j fine_ricerca   ; if password[mx] == codice_utente then fine_ricerca

continua_ricerca:

      slt r10, r5, r9       ; if codice_utente < password[mx]
      bnez r10, ricerca_in_sx   ; if r9 == 1 then ricerca_in_sx
      beqz r10, ricerca_in_dx   ; if r0 == 0 then ricerca_in_dx

ricerca_in_sx: ; cerco r5 verso valori minori

      dsub r2, r3, r4  ; indice_dx = indice_mx - 1
      j ricerca_loop ; ritorno all'inizio del loop

ricerca_in_dx: ; cerco r5 verso valori maggiori

      daddi r1, r3 , 1  ; indice_sx = indice_mx + 1
      j ricerca_loop ; ritorno all'inizio del loop

fine_ricerca:

      bnez r8, trovato   ; if r8 != 0 then trovato -> if r1 < r2
      beqz r8, secondo_controllo  ; if r8 == 0 then secondo_controllo -> if r1 < r2

secondo_controllo: ; controllo se gli indici sono uguali o meno

      beq r1, r2, trovato    ; if r1 == r2 then trovato
      j non_trovato     ; if r1 > r2 then non trovato

trovato:

      lw r11, codice_utente(r0) ; scrivo in r11 il contenuto di passoword[mx]
      j end

non_trovato:

      lw r11, var_supporto(r0) ; scrivo in r11 il contenuto di var_supporto

end:

      halt
```

Figura 2.1 : Prima Implementazione in Assembly, ricerca_binaria.s.

2.1 Spiegazione del Codice Assembly :

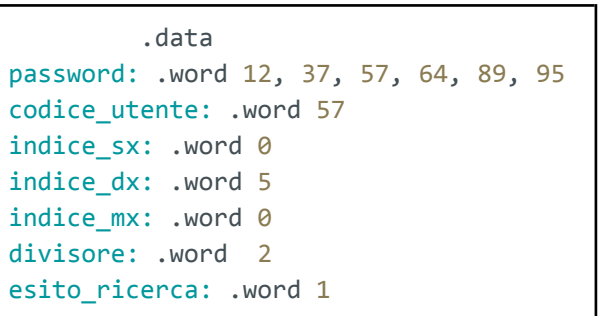
Il brano di codice coincide con la sezione `.data`, la quale è destinata alla dichiarazione delle variabili di cui l'intero algoritmo farà uso. Si trovano le seguenti 5 dichiarazioni : 1) `password`, la quale è l'array nel quale effettuare la ricerca; 2) `codice_utente`, la quale indica il codice inserito dall'utente per fare il login (valore da ricercare); 3) `indice_sx, indice_dx, indice_mx`, variabili che identificano le posizioni degli elementi di ogni dato spazio di ricerca; 4) `divisore`, la quale serve per calcolare il valore dell'indice di mezzo di ogni spazio di ricerca; 5) `esito_ricerca`, che rappresenta l'esito della ricerca.	

Figura 2.2 : segmento di codice, corrispondente alla sezione .data del file ricerca_binaria.s.

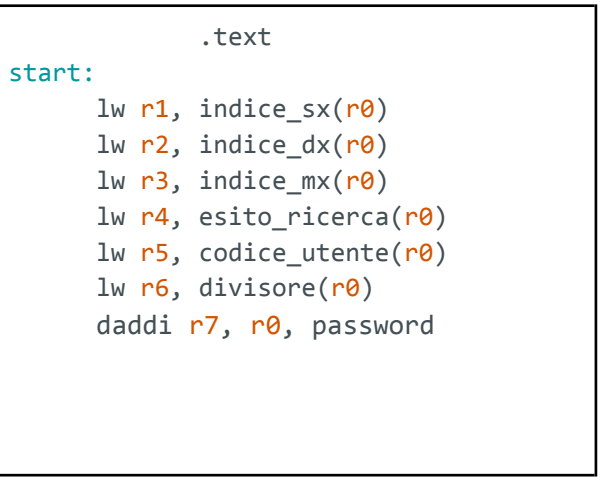
La sezione `.text` è destinata alla scrittura dei passi dell'algoritmo; questa sezione si divide in due etichette di "default", ovvero la `start` e la `end`, le quali conterranno l'algoritmo. Come prime istruzioni troviamo degli indirizzamenti, ovvero istruzioni di `lw` per caricare nel register file i valori delle variabili, attraverso la modalità di indirizzamento indiretto e carichiamo nel registro `r7` un puntatore al primo elemento dell'array.	

Figura 2.3 : segmento di codice corrispondente alla sezione .text del file ricerca_binaria.s.

Il brano di codice a fianco è contenuto all'interno dell'etichetta `ricerca_loop`, la quale corrisponde al loop `for` dell'implementazione in C. Abbiamo spiegato il brano di codice scelto etichetta per etichetta.

1) Sotto l'etichetta `ricerca_loop`, vi sono le istruzioni che servono per calcolare il valore dell'indice di mezzo (mx = (sx + dx) / 2);
2) `imposta_elemento_mezzo`, questa etichetta implementa un loop, il quale serve per incrementare il puntatore `r7`, fino a raggiungere l'elemento che corrisponde alla posizione indicata dall'indice di mezzo `r3`; si usa infatti il registro `r12`, settato a 0, e si verifica se r12 è minore di r3, e se l'esito della verifica è vero allora salto all'etichetta `imposta_elemento_mezzo2`, altrimenti in `ultimo_controllo`;
3) `ultimo_controllo`, si controlla se `r12` è maggiore di `r3`, e in caso di verità si passa all'etichetta `imposta_elemento_mezzo3`, altrimenti sono uguali e quindi r7 punta al corretto valore.
4) `imposta_elemento_mezzo2`, viene incrementato il contatore, ovvero il registro `r12`, di un'unità e il puntatore `r7` viene fatto scorrere verso gli elementi successivi;
5) `imposta_elemento_mezzo3`, viene decrementato il contatore di un'unità e il puntatore viene fatto scorrere verso gli elementi di sinistra;
6) `indice_impostato`, questa etichetta implementa l'algoritmo di ricerca binaria; per prima cosa viene verificato se `r1` (indice di sinistra) è minore di `r2` (indice di destra), se è vero allora si passa a `seconda_validazione`, altrimenti a `controlla_uguaglianza`;
7) `controlla_uguaglianza`, in questa etichetta viene controllato se i

```
ricerca_loop:

      dadd r3, r1, r2
      ddiv r3, r3, r6

imposta_elemento_mezzo:

      slt r11, r12, r3
      bnez r11, imposta_elemento_mezzo2
      beqz r11, ultimo_controllo

ultimo_controllo:

      bne r12, r3, imposta_elemento_mezzo3
      j indice_impostato

imposta_elemento_mezzo2:

      daddi r7, r7, 8
      daddi r12, r12, 1
      j imposta_elemento_mezzo

imposta_elemento_mezzo3:

      daddi r7, r7, -8
      daddi r12, r12, -1
      j imposta_elemento_mezzo

indice_impostato:

      slt r8, r1, r2
      bnez r8, seconda_validazione
      beqz r8, controlla_uguaglianza

controlla_uguaglianza:

      beq r1, r2, seconda_validazione
      bne r1, r2, fine_ricerca

seconda_validazione:

      lw r9, 0(r7)
      bne r9, r5, continua_ricerca
      j fine_ricerca

continua_ricerca:

      slt r10, r5, r9
      bnez r10, ricerca_in_sx
      beqz r10, ricerca_in_dx

ricerca_in_sx:

      dsub r2, r3, r4
      j ricerca_loop
```

due indici sono uguali, e in caso affermativo si passa alla seconda espressione da valutare, altrimenti la ricerca termina in quanto l'indice di sinistra sarà più grande di quello di destra;

8) `seconda_validazione`, si carica in `r9` il valore puntato da `r7` e se questo è diverso dal valore da cercare allora si prosegue verso l'attività di ricerca, altrimenti si salta all'etichetta `fine_ricerca`, in quanto si è trovato il valore da cercare;
9) `continua_ricerca`, in questa etichetta si determina in che spazio di ricerca continuare; si controlla infatti se il valore da cercare è minore all'elemento di mezzo oppure meno;
10) `ricerca_in_sx`, si prosegue la ricerca verso elementi più piccoli, in tal senso si aggiorna il registro `r2`, diminuendolo, inserendoci la differenza tra l'indice di mezzo e 1;
11) `ricerca_in_dx`, si prosegue la ricerca verso elementi più grandi, aggiornando il registro `r1`, ovvero l'indice di sinistra, inserendoci il valore dell'indice di mezzo incrementato di un'unità.

```
ricerca_in_dx:

      daddi r1, r3 , 1
      j ricerca_loop
```

Figura 2.4 : segmento di codice corrispondente all'etichetta ricerca_loop del file ricerca_binaria.s.

Il seguente brano di codice scrive in `r4` l'esito della ricerca. Nel caso in cui la ricerca è avvenuta con successo e quindi se l'indice di sinitra è minore o uguale a quello di destra, scrive in `r4` il valore trovato, altrimenti un qualsiasi valore, che nel nostro caso è 1.

```
fine_ricerca:

      bnez r8, trovato
      beqz r8, secondo_controllo

secondo_controllo:

       beq r1, r2, trovato
```

	`j non_trovato`
	
	`trovato:`
	
	`lw r11, codice_utente(r0)`
	`j end`
	
	`non_trovato:`
	
	`lw r11, var_supporto(r0)`

Figura 2.5 : segmento di codice corrispondente all'etichetta fine_ricerca, del file ricerca_binaria.s.

Con il seguente brano di codice l'esecuzione dell'algoritmo termina	`end:` `halt`

Figura 2.6 : segmento di codice corrispondente all'etichetta end, del file ricerca_binaria.s

2.2 Esito della Simulazione :

1) CPUC e CPI

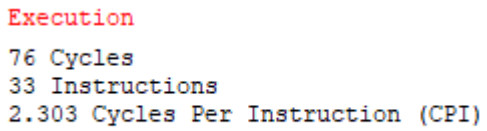


Figura 27 : Prestazioni della prima implementazione in Assembly.

2) Stalli Generati

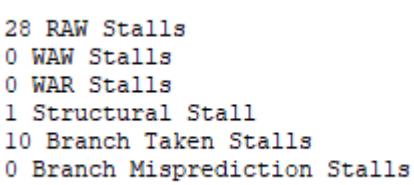


Figura 2.8 : Stalli della prima implementazione in Assembly.

3.0 Ottimizzazione Statica

Data la prima implementazione in Assembly si intende ottimizzare il sorgente, attraverso ottimizzazioni statiche, le quali hanno lo scopo di migliorare le performance, in termini di tempo di esecuzione, diminuendo gli stalli presenti.
Il programma presenta 28 RAW (Read After Write) stalls, 10 Branch Taken Stalls e uno stallo strutturale, nel caso in cui si cerchi il numero 57.
Le ottimizzazioni statiche che abbiamo implementato sono le seguenti due, nel corretto ordine di presentazione:

1) Instruction Reordering, la quale consiste nel riordinare le istruzioni del programma che creano delle dipendenze logiche, ovvero inerenti ai dati, risolti tramite gli stalli, i quali sono dei cicli di clock "senza effetto";
2) Loop Unrolling, che riguarda lo srotolamento dei cicli, ovvero scrivere il loop tante volte quante sono le iterazioni che compie.

3.1 Risoluzione degli stalli con Instruction Reordering :

Per prima cosa abbiamo individuato i principali stalli RAW i quali si presentavano principalmente in 4 punti.
Le prime istruzioni che generano gli stalli RAW erano in corrispondenza del punto nel quale si impostava il valore dell'indice di mezzo; siccome la divisione è un'operazione che ha latenza molto lunga, circa 24 cicli di clock per la fase di execute, e siccome le istruzioni successive fanno uso del suo registro destinazione si creano 20 stalli RAW; tuttavia in seguito ad alcune prove è emerso che avremmo potuto risolvere pochi stalli, in quanto i principali problemi erano dati dalla latenza dell'operazione di divisione e dalla scarsa possibilità di cambiare l'ordine delle istruzioni senza modificare la semantica dell'algoritmo.
Ci siamo concentrati nella riduzione degli stalli nei seguenti quattro punti :

```
1) slt r11, r12, r3
   bnez r11, imposta_elemento_mezzo2
   beqz r11, ultimo_controllo
2) lw r9, 0(r7)
   bne r9, r5, continua_ricerca
3) slt r8, r1, r2
   bnez r8, seconda_validazione
   beqz r8, controlla_uguaglianz

4) ddiv r3, r3, r6

   imposta_elemento_mezzo:
```

```
slt r11, r12, r3
bnez r11, imposta_elemento_mezzo2
beqz r11, ultimo_controllo
```

I primi tre brani di codice esposti generano complessivamente 8 stalli, quando si cerca il numero 57, in quanto la loro esecuzione viene ripetuta più volte.
Abbiamo creato quattro diverse versioni per ottimizzare gli stalli appena esposti.

3.1.1 Prima Versione Assembly :

```
; @authors Elia Renzoni, Alessio Visentino e Alessio Biagioli
; @date 20/05/2023
; Progetto per la Sessione Estiva di Architettura degli Elaboratori
; @brief Implementazione in MIPS Assembly dell'algoritmo di ricerca dicotomica per array giá
ordinati

          .data

password:         .word 12, 37, 57, 64, 89, 95
codice_utente:    .word 57
indice_sx:        .word 0
indice_dx:        .word 5
indice_mx:        .word 0
divisore:         .word 2
var_supporto:     .word 1

          .text

start:

      lw r1, indice_sx(r0)    ; indice sx di ogni spazio di ricerca
      lw r2, indice_dx(r0)    ; indice dx di ogni spazio di ricerca
      lw r3, indice_mx(r0)    ; indice di mx di ogni spazio di ricerca
      lw r4, var_supporto(r0)  ; variabile per effettuare la sottrazione
      lw r5, codice_utente(r0) ; variabile contenente il codice utente da cercare
      lw r6, divisore(r0)    ; variabile contenente il divisore = 2
      daddi r7, r0, password  ; puntatore al primo elemento dell'array.

ricerca_loop:

      dadd r3, r1, r2  ; somma l'indice di sinistra con quello di destra
      ddiv r3, r3, r6  ; calcola l'indice di mezzo -> mx = sx + dx / 2

imposta_elemento_mezzo:

      slt r11, r12, r3  ; if r12 < r3
      bnez r11, imposta_elemento_mezzo2 ; if r11 != 0 then imposta_elemento_mezzo2
      beqz r11, ultimo_controllo ; if r11 == 0 then ultimo_controllo

ultimo_controllo: ; controlla se r12 é uguale o maggiore di r3

      bne r12, r3, imposta_elemento_mezzo3 ; if r12 != r3 then imposta_elemento_mezzo3
      j indice_impostato  ; if r12 == r3 then indice_impostato
```

```
imposta_elemento_mezzo2: ; imposta l'indice if r12 < r3

      daddi r7, r7, 8 ; incrementa il puntatore
      daddi r12, r12, 1 ; incrementa il contatore del ciclo
      j imposta_elemento_mezzo ; ritorna all'inizio del ciclo

imposta_elemento_mezzo3: ; imposta l'indice if r12 > r3

      daddi r7, r7, -8 ; decrementa il puntatore
      daddi r12, r12, -1 ; decrementa il contatore del ciclo
      j imposta_elemento_mezzo ; ritorna all'inizio del ciclo

indice_impostato: ; indice dell'elemento impostato correttamente

      lw r9, 0(r7) ; carica in r9 il puntatore
      slt r8, r1, r2   ; if r1 < r2
      bnez r8, seconda_validazione ; if r8 != 0 then seconda_validazione
      beqz r8, controlla_uguaglianza ; if r8 == 0 then controlla_uguaglianza

controlla_uguaglianza: ; controllo l'uguaglianza tra gli indici

      beq r1, r2, seconda_validazione ; if r1 == r2 then seconda_validazione
      bne r1, r2, fine_ricerca  ; if r1 != r2 then fine_ricerca, in quanto r1 > r2

seconda_validazione: ; secondo controllo

      bne r9, r5, continua_ricerca   ; if password[mx] != codice_utente then continua_ricerca
      j fine_ricerca   ; if password[mx] == codice_utente then fine_ricerca

continua_ricerca:

      slt r10, r5, r9       ; if codice_utente < password[mx]
      bnez r10, ricerca_in_sx   ; if r9 == 1 then ricerca_in_sx
      beqz r10, ricerca_in_dx   ; if r0 == 0 then ricerca_in_dx

ricerca_in_sx: ; cerco r5 verso valori minori

      dsub r2, r3, r4  ; indice_dx = indice_mx - 1
      j ricerca_loop ; ritorno all'inizio del loop

ricerca_in_dx: ; cerco r5 verso valori maggiori

      daddi r1, r3 , 1  ; indice_sx = indice_mx + 1
      j ricerca_loop ; ritorno all'inizio del loop

fine_ricerca:

      bnez r8, trovato   ; if r8 != 0 then trovato -> if r1 < r2
      beqz r8, secondo_controllo  ; if r8 == 0 then secondo_controllo -> if r1 < r2

secondo_controllo: ; controllo se gli indici sono uguali o meno

      beq r1, r2, trovato    ; if r1 == r2 then trovato
      j non_trovato     ; if r1 > r2 then non trovato

trovato:

      lw r11, codice_utente(r0) ; scrivo in r11 il contenuto di passoword[mx]
      j end

non_trovato:

      lw r11, var_supporto(r0) ; scrivo in r11 il contenuto di var_supporto
```

```
end:
      halt
```

Nella prima versione abbiamo risolto gli stalli che si presentavano nell'etichetta `seconda_validazione`, andando a cambiare di posizione l'istruzione `lw r9, 0(r7)`, in questo modo non si presenta più lo stallo per risolvere il problema corrispondente al secondo punto del precedente elenco. Abbiamo posizionato l'istruzione di caricamento sotto l'etichetta `indice_impostato` senza cambiare la semantica dell'algoritmo, in quanto effettuiamo il caricamento in `r9` in un punto diverso non pregiudicando l'output.

Tramite la prima versione abbiamo eliminato due stalli e il CPI è diminuito.

```
Execution
74 Cycles
33 Instructions
2.242 Cycles Per Instruction (CPI)

Stalls
26 RAW Stalls
0 WAW Stalls
0 WAR Stalls
1 Structural Stall
10 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size
172 Bytes
```

Figura 3.1: Performance dell'Algoritmo in seguito alla prima ottimizzazione statica.

3.1.2 Seconda Versione Assembly :

```
; @authors Elia Renzoni, Alessio Visentino e Alessio Biagioli
; @date 20/05/2023
; Progetto per la Sessione Estiva di Architettura degli Elaboratori
; @brief Implementazione in MIPS Assembly dell'algoritmo di ricerca dicotomica per array giá
ordinati

          .data

password:         .word 12, 37, 57, 64, 89, 95
codice_utente:    .word 57
indice_sx:        .word 0
indice_dx:        .word 5
indice_mx:        .word 0
divisore:         .word 2
var_supporto:     .word 1

          .text

start:

      lw r1, indice_sx(r0)    ; indice sx di ogni spazio di ricerca
      lw r2, indice_dx(r0)    ; indice dx di ogni spazio di ricerca
      lw r3, indice_mx(r0)    ; indice di mx di ogni spazio di ricerca
      lw r4, var_supporto(r0)  ; variabile per effettuare la sottrazione
      lw r5, codice_utente(r0) ; variabile contenente il codice utente da cercare
      lw r6, divisore(r0)    ; variabile contenente il divisore = 2
      daddi r7, r0, password  ; puntatore al primo elemento dell'array.

ricerca_loop:

      dadd r3, r1, r2  ; somma l'indice di sinistra con quello di destra
      ddiv r3, r3, r6  ; calcola l'indice di mezzo -> mx = sx + dx / 2

imposta_elemento_mezzo:

      slt r11, r12, r3  ; if r12 < r3
      bnez r11, imposta_elemento_mezzo2 ; if r11 != 0 then imposta_elemento_mezzo2
      beqz r11, ultimo_controllo ; if r11 == 0 then ultimo_controllo

ultimo_controllo: ; controlla se r12 é uguale o maggiore di r3

      bne r12, r3, imposta_elemento_mezzo3 ; if r12 != r3 then imposta_elemento_mezzo3
      j indice_impostato  ; if r12 == r3 then indice_impostato

imposta_elemento_mezzo2: ; imposta l'indice if r12 < r3

      daddi r7, r7, 8 ; incrementa il puntatore
      daddi r12, r12, 1 ; incrementa il contatore del ciclo
      j imposta_elemento_mezzo ; ritorna all'inizio del ciclo
```

```
imposta_elemento_mezzo3: ; imposta l'indice if r12 > r3

      daddi r7, r7, -8 ; decrementa il puntatore
      daddi r12, r12, -1 ; decrementa il contatore del ciclo
      j imposta_elemento_mezzo ; ritorna all'inizio del ciclo

indice_impostato: ; indice dell'elemento impostato correttamente

      slt r8, r1, r2   ; if r1 < r2
      lw r9, 0(r7)  ;  caricamento in r9 il valore del puntatore
      bnez r8, seconda_validazione ; if r8 != 0 then seconda_validazione
      beqz r8, controlla_uguaglianza ; if r8 == 0 then controlla_uguaglianza

controlla_uguaglianza: ; controllo l'uguaglianza tra gli indici

      beq r1, r2, seconda_validazione ; if r1 == r2 then seconda_validazione
      bne r1, r2, fine_ricerca  ; if r1 != r2 then fine_ricerca, in quanto r1 > r2

seconda_validazione: ; secondo controllo

      bne r9, r5, continua_ricerca   ; if password[mx] != codice_utente then continua_ricerca
      j fine_ricerca   ; if password[mx] == codice_utente then fine_ricerca

continua_ricerca:

      slt r10, r5, r9       ; if codice_utente < password[mx]
      bnez r10, ricerca_in_sx   ; if r9 == 1 then ricerca_in_sx
      beqz r10, ricerca_in_dx   ; if r0 == 0 then ricerca_in_dx

ricerca_in_sx: ; cerco r5 verso valori minori

      dsub r2, r3, r4  ; indice_dx = indice_mx - 1
      j ricerca_loop ; ritorno all'inizio del loop

ricerca_in_dx: ; cerco r5 verso valori maggiori

      daddi r1, r3 , 1  ; indice_sx = indice_mx + 1
      j ricerca_loop ; ritorno all'inizio del loop

fine_ricerca:

      bnez r8, trovato   ; if r8 != 0 then trovato -> if r1 < r2
      beqz r8, secondo_controllo  ; if r8 == 0 then secondo_controllo -> if r1 < r2

secondo_controllo: ; controllo se gli indici sono uguali o meno

      beq r1, r2, trovato    ; if r1 == r2 then trovato
      j non_trovato     ; if r1 > r2 then non trovato

trovato:

      lw r11, codice_utente(r0) ; scrivo in r11 il contenuto di passoword[mx]
      j end

non_trovato:

      lw r11, var_supporto(r0) ; scrivo in r11 il contenuto di var_supporto

end:

      halt
```

Nella seconda versione abbiamo posizionato l'istruzione `lw r9, 0(r7)` dopo `slt r8, r1, r2`; in questo modo abbiamo risolto lo stallo RAW che si verificava tra `slt r8, r1, r2` e `bnez r8, seconda_validazione`.

Tramite la seconda versione abbiamo eliminato uno stallo RAW e il CPI è diminuito

```
Execution

73 Cycles
33 Instructions
2.212 Cycles Per Instruction (CPI)


Stalls

25 RAW Stalls
0 WAW Stalls
0 WAR Stalls
1 Structural Stall
10 Branch Taken Stalls
0 Branch Misprediction Stalls


Code size

172 Bytes
```

Figura 3.2 : Performance dell'Algoritmo in seguito alla seconda ottimizzazione statica.

3.1.3 Terza Versione Assembly :

```
; @authors Elia Renzoni, Alessio Visentino e Alessio Biagioli
; @date 20/05/2023
; Progetto per la Sessione Estiva di Architettura degli Elaboratori
; @brief Implementazione in MIPS Assembly dell'algoritmo di ricerca dicotomica per array giá
ordinati

          .data

password:         .word 12, 37, 57, 64, 89, 95
codice_utente:    .word 57
indice_sx:        .word 0
indice_dx:        .word 5
indice_mx:        .word 0
divisore:         .word 2
var_supporto:     .word 1

          .text

start:

      lw r1, indice_sx(r0)    ; indice sx di ogni spazio di ricerca
      lw r2, indice_dx(r0)    ; indice dx di ogni spazio di ricerca
      lw r3, indice_mx(r0)    ; indice di mx di ogni spazio di ricerca
      lw r5, codice_utente(r0) ; variabile contenente il codice utente da cercare
      lw r6, divisore(r0)    ; variabile contenente il divisore = 2
      daddi r7, r0, password  ; puntatore al primo elemento dell'array.

ricerca_loop:

      dadd r3, r1, r2  ; somma l'indice di sinistra con quello di destra
      ddiv r3, r3, r6  ; calcola l'indice di mezzo -> mx = sx + dx / 2

imposta_elemento_mezzo:

      slt r11, r12, r3  ; if r12 < r3
      lw r4, var_supporto(r0)  ; caricamento della variabile di supporto
      bnez r11, imposta_elemento_mezzo2 ; if r11 != 0 then imposta_elemento_mezzo2
      beqz r11, ultimo_controllo ; if r11 == 0 then ultimo_controllo

ultimo_controllo: ; controlla se r12 é uguale o maggiore di r3

      bne r12, r3, imposta_elemento_mezzo3 ; if r12 != r3 then imposta_elemento_mezzo3
      j indice_impostato  ; if r12 == r3 then indice_impostato

imposta_elemento_mezzo2: ; imposta l'indice if r12 < r3

      daddi r7, r7, 8 ; incrementa il puntatore
      daddi r12, r12, 1 ; incrementa il contatore del ciclo
      j imposta_elemento_mezzo ; ritorna all'inizio del ciclo

imposta_elemento_mezzo3: ; imposta l'indice if r12 > r3

      daddi r7, r7, -8 ; decrementa il puntatore
      daddi r12, r12, -1 ; decrementa il contatore del ciclo
      j imposta_elemento_mezzo ; ritorna all'inizio del ciclo

indice_impostato: ; indice dell'elemento impostato correttamente

      slt r8, r1, r2   ; if r1 < r2
      lw r9, 0(r7)  ;  caricamento in r9 del valore del puntatore
```

```
      bnez r8, seconda_validazione ; if r8 != 0 then seconda_validazione
      beqz r8, controlla_uguaglianza ; if r8 == 0 then controlla_uguaglianza

controlla_uguaglianza: ; controllo l'uguaglianza tra gli indici

      beq r1, r2, seconda_validazione ; if r1 == r2 then seconda_validazione
      bne r1, r2, fine_ricerca  ; if r1 != r2 then fine_ricerca, in quanto r1 > r2

seconda_validazione: ; secondo controllo

      bne r9, r5, continua_ricerca   ; if password[mx] != codice_utente then continua_ricerca
      j fine_ricerca   ; if password[mx] == codice_utente then fine_ricerca

continua_ricerca:

      slt r10, r5, r9       ; if codice_utente < password[mx]
      bnez r10, ricerca_in_sx   ; if r9 == 1 then ricerca_in_sx
      beqz r10, ricerca_in_dx   ; if r0 == 0 then ricerca_in_dx

ricerca_in_sx: ; cerco r5 verso valori minori

      dsub r2, r3, r4  ; indice_dx = indice_mx - 1
      j ricerca_loop ; ritorno all'inizio del loop

ricerca_in_dx: ; cerco r5 verso valori maggiori

      daddi r1, r3 , 1  ; indice_sx = indice_mx + 1
      j ricerca_loop ; ritorno all'inizio del loop

fine_ricerca:

      bnez r8, trovato   ; if r8 != 0 then trovato -> if r1 < r2
      beqz r8, secondo_controllo  ; if r8 == 0 then secondo_controllo -> if r1 < r2

secondo_controllo: ; controllo se gli indici sono uguali o meno

      beq r1, r2, trovato    ; if r1 == r2 then trovato
      j non_trovato     ; if r1 > r2 then non trovato

trovato:

      lw r11, codice_utente(r0) ; scrivo in r11 il contenuto di passoword[mx]
      j end

non_trovato:

      lw r11, var_supporto(r0) ; scrivo in r11 il contenuto di var_supporto

end:

      halt
```

Nella terza versione abbiamo risolto gli stalli che si verificavano tra `slt r11, r12, r3` e `bnez r11, imposta_elemento_mezzo2`, inserendo l'istruzione di `lw r4, var_supporto(0)`; in questo modo non si verificano le interferenze logiche tra le due istruzioni, in quanto carico ad ogni iterazione in `r4` il valore di `var_supporto`.

Tramite la terza versione abbiamo diminuito gli stalli a 22 e il CPI a 2.057

```
Execution

72 Cycles
35 Instructions
2.057 Cycles Per Instruction (CPI)


Stalls

22 RAW Stalls
0 WAW Stalls
0 WAR Stalls
1 Structural Stall
10 Branch Taken Stalls
0 Branch Misprediction Stalls


Code size

172 Bytes
```

Figura 3.3 : Performance dell'Algoritmo in seguito alla terza ottimizzazione statica.

3.1.4 Quarta Versione Assembly :

```
; @authors Elia Renzoni, Alessio Visentino e Alessio Biagioli
; @date 20/05/2023
; Progetto per la Sessione Estiva di Architettura degli Elaboratori
; @brief Implementazione in MIPS Assembly dell'algoritmo di ricerca dicotomica per array giá
ordinati

          .data

password:         .word 12, 37, 57, 64, 89, 95
codice_utente:    .word 57
indice_sx:        .word 0
indice_dx:        .word 5
indice_mx:        .word 0
divisore:         .word 2
var_supporto:     .word 1

          .text

start:

      lw r1, indice_sx(r0)    ; indice sx di ogni spazio di ricerca
      lw r2, indice_dx(r0)    ; indice dx di ogni spazio di ricerca
      lw r3, indice_mx(r0)    ; indice di mx di ogni spazio di ricerca
      lw r6, divisore(r0)    ; variabile contenente  il divisore = 2
      daddi r7, r0, password  ; puntatore al primo elemento dell'array.

ricerca_loop:

      dadd r3, r1, r2  ; somma l'indice di sinistra con quello di destra
      ddiv r3, r3, r6  ; calcola l'indice di mezzo -> mx = sx + dx / 2

imposta_elemento_mezzo:

      lw r5, codice_utente(r0) ; caricamento del valore da ricercare
      slt r11, r12, r3  ; if r12 < r3
      lw r4, var_supporto(r0)  ; caricamento della variabile di supporto
      bnez r11, imposta_elemento_mezzo2 ; if r11 != 0 then imposta_elemento_mezzo2
      beqz r11, ultimo_controllo ; if r11 == 0 then ultimo_controllo

ultimo_controllo: ; controlla se r12 é uguale o maggiore di r3

      bne r12, r3, imposta_elemento_mezzo3 ; if r12 != r3 then imposta_elemento_mezzo3
      j indice_impostato  ; if r12 == r3 then indice_impostato

imposta_elemento_mezzo2: ; imposta l'indice if r12 < r3

      daddi r7, r7, 8 ; incrementa il puntatore
      daddi r12, r12, 1 ; incrementa il contatore del ciclo
      j imposta_elemento_mezzo ; ritorna all'inizio del ciclo

imposta_elemento_mezzo3: ; imposta l'indice if r12 > r3

      daddi r7, r7, -8 ; decrementa il puntatore
      daddi r12, r12, -1 ; decrementa il contatore del ciclo
      j imposta_elemento_mezzo ; ritorna all'inizio del ciclo

indice_impostato: ; indice dell'elemento impostato correttamente

      slt r8, r1, r2   ; if r1 < r2
      lw r9, 0(r7)  ;  caricamento in r9 del valore del puntatore
```

```
      bnez r8, seconda_validazione ; if r8 != 0 then seconda_validazione
      beqz r8, controlla_uguaglianza ; if r8 == 0 then controlla_uguaglianza

controlla_uguaglianza: ; controllo l'uguaglianza tra gli indici

      beq r1, r2, seconda_validazione ; if r1 == r2 then seconda_validazione
      bne r1, r2, fine_ricerca  ; if r1 != r2 then fine_ricerca, in quanto r1 > r2

seconda_validazione: ; secondo controllo

      bne r9, r5, continua_ricerca   ; if password[mx] != codice_utente then continua_ricerca
      j fine_ricerca   ; if password[mx] == codice_utente then fine_ricerca

continua_ricerca:

      slt r10, r5, r9       ; if codice_utente < password[mx]
      bnez r10, ricerca_in_sx   ; if r9 == 1 then ricerca_in_sx
      beqz r10, ricerca_in_dx   ; if r0 == 0 then ricerca_in_dx

ricerca_in_sx: ; cerco r5 verso valori minori

      dsub r2, r3, r4  ; indice_dx = indice_mx - 1
      j ricerca_loop ; ritorno all'inizio del loop

ricerca_in_dx: ; cerco r5 verso valori maggiori

      daddi r1, r3 , 1  ; indice_sx = indice_mx + 1
      j ricerca_loop ; ritorno all'inizio del loop

fine_ricerca:

      bnez r8, trovato   ; if r8 != 0 then trovato -> if r1 < r2
      beqz r8, secondo_controllo  ; if r8 == 0 then secondo_controllo -> if r1 < r2

secondo_controllo: ; controllo se gli indici sono uguali o meno

      beq r1, r2, trovato    ; if r1 == r2 then trovato
      j non_trovato     ; if r1 > r2 then non trovato

trovato:

      lw r11, codice_utente(r0) ; scrivo in r11 il contenuto di passoword[mx]
      j end

non_trovato:

      lw r11, var_supporto(r0) ; scrivo in r11 il contenuto di var_supporto

end:

      halt
```

In questa versione abbiamo diminuito gli stalli RAW che si verificavano tra l'istruzione `ddiv r3, r3, r6` e `slt r11, r3, r2` aggiungendo l'istruzione di caricamento in `r5`, del codice utente da ricercare, senza interferire con la semantica dell'algoritmo.

Tramite questa versione abbiamo eliminato uno stallo RAW e abbassato il CPI a 1.973

```
Execution

73 Cycles
37 Instructions
1.973 Cycles Per Instruction (CPI)


Stalls

21 RAW Stalls
0 WAW Stalls
0 WAR Stalls
1 Structural Stall
10 Branch Taken Stalls
0 Branch Misprediction Stalls


Code size

172 Bytes
```

Figura 3.4 : Performance dell'Algoritmo in seguito

3.2 Risoluzione degli Stalli tramite Loop Unrolling :

L'algoritmo nel caso in cui si cerchi il valore 57 non effettua alcuna iterazione sul primo loop, ovvero quello principale che inizia con l'etichetta `ricerca_loop`, in quanto una volta impostato il valore dell'indice di mezzo e raggiunto il corrispettivo elemento nell'array, la chiave di ricerca viene subito trovata. Tuttavia il ciclo annidato, ovvero quello che inizia con l'etichetta `imposta_elemento_mezzo`, viene ripetuto tre volte. Quindi abbiamo deciso di applicare il Loop Unrolling totale sul seguente ciclo :

```
imposta_elemento_mezzo:

      lw r5, codice_utente(r0) ; caricamento del valore da ricercare
      slt r11, r12, r3  ; if r12 < r3
      lw r4, var_supporto(r0)  ; caricamento della variabile di supporto
      bnez r11, imposta_elemento_mezzo2 ; if r11 != 0 then imposta_elemento_mezzo2
      beqz r11, ultimo_controllo ; if r11 == 0 then ultimo_controllo

ultimo_controllo: ; controlla se r12 é uguale o maggiore di r3

      bne r12, r3, imposta_elemento_mezzo3 ; if r12 != r3 then imposta_elemento_mezzo3
      j indice_impostato  ; if r12 == r3 then indice_impostato

imposta_elemento_mezzo2: ; imposta l'indice if r12 < r3

      daddi r7, r7, 8 ; incrementa il puntatore
      daddi r12, r12, 1 ; incrementa il contatore del ciclo
      j imposta_elemento_mezzo ; ritorna all'inizio del ciclo

imposta_elemento_mezzo3: ; imposta l'indice if r12 > r3

      daddi r7, r7, -8 ; i--, decrementa il puntatore
      daddi r12, r12, -1 ; decrementa il contatore del ciclo
      j imposta_elemento_mezzo ; ritorna all'inizio del ciclo
```

Figura 3.5 : Brano di Codice da Ottimizzare, ricerca_binaria-4.s.

Il precedente brano di codice, in seguito al suo srotolamento, si presenta nella seguente forma, da notare il fatto che abbiamo eliminato l'etichetta `imposta_elemento_mezzo3`, in quanto il valore 57 viene subito trovato e quindi non vi è la necessità di decrementare il puntatore :

```
; @authors Elia Renzoni, Alessio Visentino e Alessio Biagioli
; @date 20/05/2023
; Progetto per la Sessione Estiva di Architettura degli Elaboratori
; @brief Implementazione in MIPS Assembly dell'algoritmo di ricerca dicotomica per array
giá ordinati

          .data

password:         .word 12, 37, 57, 64, 89, 95
codice_utente:    .word 57
indice_sx:        .word 0
indice_dx:        .word 5
indice_mx:        .word 0
divisore:         .word 2
var_supporto:     .word 1

          .text

start:

      lw r1, indice_sx(r0)    ; indice sx di ogni spazio di ricerca
      lw r2, indice_dx(r0)    ; indice dx di ogni spazio di ricerca
      lw r3, indice_mx(r0)    ; indice di mx di ogni spazio di ricerca
      lw r6, divisore(r0)    ; variabile contenente il divisore = 2
      daddi r7, r0, password  ; puntatore al primo elemento dell'array.

ricerca_loop:

      dadd r3, r1, r2  ; somma l'indice di sinistra con quello di destra
      ddiv r3, r3, r6  ; calcola l'indice di mezzo -> mx = sx + dx / 2



imposta_elemento_mezzo:

      lw r5, codice_utente(r0) ; caricamento del valore da ricercare
      slt r11, r12, r3  ; if r12 < r3
      lw r4, var_supporto(r0)  ; caricamento della variabile di supporto
      bnez r11, imposta_elemento_mezzo2 ; if r11 != 0 then imposta_elemento_mezzo2
      beqz r11, ultimo_controllo ; if r11 == 0 then ultimo_controllo

ultimo_controllo: ; controlla se r12 é uguale o maggiore di r3

      j indice_impostato  ; if r12 == r3 then indice_impostato

imposta_elemento_mezzo2: ; imposta l'indice if r12 < r3

      daddi r7, r7, 8 ; incrementa il puntatore
      daddi r12, r12, 1 ; incrementa il contatore del ciclo
```

```
imposta_elemento_mezzo:

      lw r5, codice_utente(r0) ; caricamento del valore da ricercare
      slt r11, r12, r3  ; if r12 < r3
      lw r4, var_supporto(r0)  ; caricamento della variabile di supporto
      bnez r11, imposta_elemento_mezzo2 ; if r11 != 0 then imposta_elemento_mezzo2
      beqz r11, ultimo_controllo ; if r11 == 0 then ultimo_controllo

ultimo_controllo: ; controlla se r12 é uguale o maggiore di r3

      j indice_impostato  ; if r12 == r3 then indice_impostato

imposta_elemento_mezzo2: ; imposta l'indice if r12 < r3

      daddi r7, r7, 8 ; incrementa il puntatore
      daddi r12, r12, 1 ; incrementa il contatore del ciclo

imposta_elemento_mezzo:

      lw r5, codice_utente(r0) ; caricamento del valore da ricercare
      slt r11, r12, r3  ; if r12 < r3
      lw r4, var_supporto(r0)  ; caricamento della variabile di supporto
      bnez r11, imposta_elemento_mezzo2 ; if r11 != 0 then imposta_elemento_mezzo2
      beqz r11, ultimo_controllo ; if r11 == 0 then ultimo_controllo

ultimo_controllo: ; controlla se r12 é uguale o maggiore di r3

      j indice_impostato  ; if r12 == r3 then indice_impostato

imposta_elemento_mezzo2: ; imposta l'indice if r12 < r3

      daddi r7, r7, 8 ; incrementa il puntatore
      daddi r12, r12, 1 ; incrementa il contatore del ciclo

indice_impostato: ; indice dell'elemento impostato correttamente

      slt r8, r1, r2   ; if r1 < r2
      lw r9, 0(r7)  ;  caricamento in r9 il valore del puntatore
      bnez r8, seconda_validazione ; if r8 != 0 then seconda_validazione
      beqz r8, controlla_uguaglianza ; if r8 == 0 then controlla_uguaglianza

controlla_uguaglianza: ; controllo l'uguaglianza tra gli indici

      beq r1, r2, seconda_validazione ; if r1 == r2 then seconda_validazione
      bne r1, r2, fine_ricerca  ; if r1 != r2 then fine_ricerca, in quanto r1 > r2

seconda_validazione: ; secondo controllo

      bne r9, r5, continua_ricerca   ; if password[mx] != codice_utente then
continua_ricerca
      j fine_ricerca   ; if password[mx] == codice_utente then fine_ricerca

continua_ricerca:

      slt r10, r5, r9       ; if codice_utente < password[mx]
      bnez r10, ricerca_in_sx   ; if r9 == 1 then ricerca_in_sx
```

```
      beqz r10, ricerca_in_dx   ; if r0 == 0 then ricerca_in_dx

ricerca_in_sx: ; cerco r5 verso valori minori

      dsub r2, r3, r4  ; indice_dx = indice_mx - 1
      j ricerca_loop ; ritorno all'inizio del loop

ricerca_in_dx: ; cerco r5 verso valori maggiori

      daddi r1, r3 , 1  ; indice_sx = indice_mx + 1
      j ricerca_loop ; ritorno all'inizio del loop

fine_ricerca:

      bnez r8, trovato   ; if r8 != 0 then trovato -> if r1 < r2
      beqz r8, secondo_controllo  ; if r8 == 0 then secondo_controllo -> if r1 < r2

secondo_controllo: ; controllo se gli indici sono uguali o meno

      beq r1, r2, trovato    ; if r1 == r2 then trovato
      j non_trovato     ; if r1 > r2 then non trovato

trovato:

      lw r11, codice_utente(r0) ; scrivo in r11 il contenuto di passoword[mx]
      j end

non_trovato:

      lw r11, var_supporto(r0) ; scrivo in r11 il contenuto di var_supporto

end:

      halt
```

Figura 3.6 : Algoritmo in seguito al Loop Unrolling, ricerca_binaria-loopUnrolling-1.s

Tramite lo srotolamento abbiamo eliminato due Branch Taken Stalls e il CPI è aumentato

```
Execution

68 Cycles
34 Instructions
2.000 Cycles Per Instruction (CPI)


Stalls

21 RAW Stalls
0 WAW Stalls
0 WAR Stalls
1 Structural Stall
8 Branch Taken Stalls
0 Branch Misprediction Stalls


Code size

216 Bytes
```

Figura 3.7 : Performance dell'Algoritmo in seguito al Loop Unrolling.

3.2.1 Seconda Versione Assembly :

Nella seconda versione abbiamo diminuito la replicazione del ciclo, da tre a due, in quanto abbiamo eliminato le istruzioni superflue e dato più importanza all'incremento del puntatore, ovvero l'istruzione `daddi r7, r7, 8`, che avveniva solo due volte. Così facendo abbiamo eliminato gli stalli derivanti dal conflitto di dati tra l'operazione di divisione, `ddiv`, e quella di controllo effettuato tramite l'istruzione slt per impostare il ciclo.

```
; @authors Elia Renzoni, Alessio Visentino e Alessio Biagioli
; @date 20/05/2023
; Progetto per la Sessione Estiva di Architettura degli Elaboratori
; @brief Implementazione in MIPS Assembly dell'algoritmo di ricerca dicotomica per array
giá ordinati

          .data

password:         .word 12, 37, 57, 64, 89, 95
codice_utente:    .word 57
indice_sx:        .word 0
indice_dx:        .word 5
indice_mx:        .word 0
divisore:         .word 2
var_supporto:     .word 1

          .text

start:

      lw r1, indice_sx(r0)    ; indice sx di ogni spazio di ricerca
      lw r2, indice_dx(r0)    ; indice dx di ogni spazio di ricerca
      lw r3, indice_mx(r0)    ; indice di mx di ogni spazio di ricerca
      lw r6, divisore(r0)    ; variabile contenente il divisore = 2
      daddi r7, r0, password  ; puntatore al primo elemento dell'array.

ricerca_loop:

      dadd r3, r1, r2  ; somma l'indice di sinistra con quello di destra
      ddiv r3, r3, r6  ; calcola l'indice di mezzo -> mx = sx + dx / 2

imposta_elemento_mezzo:

      lw r5, codice_utente(r0) ; caricamento del valore da ricercare
      lw r4, var_supporto(r0)  ; caricamento della variabile di supporto
      daddi r7, r7, 8 ; incrementa il puntatore

imposta_elemento_mezzo:

      daddi r7, r7, 8 ; incrementa il puntatore

indice_impostato:

      slt r8, r1, r2   ; if r1 < r2
```

```
      lw r9, 0(r7)  ;  caricamento in r9 del valore del puntatore
      bnez r8, seconda_validazione ; if r8 != 0 then seconda_validazione
      beqz r8, controlla_uguaglianza ; if r8 == 0 then controlla_uguaglianza

controlla_uguaglianza: ; controllo l'uguaglianza tra gli indici

      beq r1, r2, seconda_validazione ; if r1 == r2 then seconda_validazione
      bne r1, r2, fine_ricerca  ; if r1 != r2 then fine_ricerca, in quanto r1 > r2

seconda_validazione: ; secondo controllo

      bne r9, r5, continua_ricerca   ; if password[mx] != codice_utente then
continua_ricerca
      j fine_ricerca   ; if password[mx] == codice_utente then fine_ricerca

continua_ricerca:

      slt r10, r5, r9       ; if codice_utente < password[mx]
      bnez r10, ricerca_in_sx   ; if r9 == 1 then ricerca_in_sx
      beqz r10, ricerca_in_dx   ; if r0 == 0 then ricerca_in_dx

ricerca_in_sx: ; cerco r5 verso valori minori

      dsub r2, r3, r4  ; indice_dx = indice_mx - 1
      j ricerca_loop ; ritorno all'inizio del loop

ricerca_in_dx: ; cerco r5 verso valori maggiori

      daddi r1, r3 , 1  ; indice_sx = indice_mx + 1
      j ricerca_loop ; ritorno all'inizio del loop

fine_ricerca:

      bnez r8, trovato   ; if r8 != 0 then trovato -> if r1 < r2
      beqz r8, secondo_controllo  ; if r8 == 0 then secondo_controllo -> if r1 < r2

secondo_controllo: ; controllo se gli indici sono uguali o meno

      beq r1, r2, trovato    ; if r1 == r2 then trovato
      j non_trovato     ; if r1 > r2 then non trovato

trovato:

      lw r11, codice_utente(r0) ; scrivo in r11 il contenuto di passoword[mx]
      j end

non_trovato:

      lw r11, var_supporto(r0) ; scrivo in r11 il contenuto di var_supporto

end:

      halt
```

Figura 3.8 : Algoritmo in seguito al Loop Unrolling, ricerca_binaria-LoopUnrolling-2.s

In seguito all'ultima ottimizzazione abbiamo eliminato gli stalli RAW, tuttavia si presentano ancora 4 aborti, i quali però sono relativi al ciclo che effettua la ricerca, che non abbiamo potuto srotolare in quanto viene ripetuto una sola volta, nel caso in cui si cerchi il valore 57. Inoltre il CPI e i cicli di clock complessivi sono diminuiti.

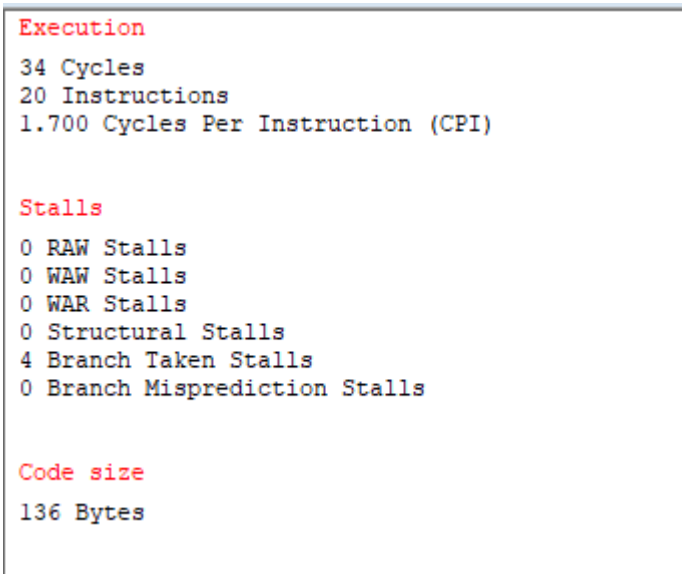


Figura 3.9 : Performance dell'Algoritmo in seguito all'ultima ottimizzazione, ricerca_binaria-LoopUnrolling-3.s

4.0 Conclusioni

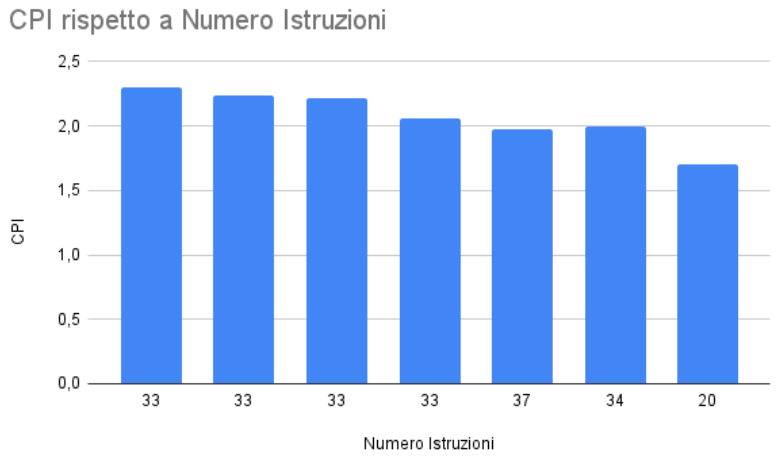


Figura 4.1 : valutazione della performance dell'algoritmo.

Il grafico a colonne ci comunica che il CPI diminuisce sensibilmente con il diminuire del numero delle istruzioni, tuttavia si presentano due casi particolari, ovvero quando il numero delle istruzioni é pari a 37, quindi rappresenta un'interruzione rispetto al trend di decrescita, nel quale il CPI diminuisce, e quando il numero delle istruzioni è pari a 34, nel quale si registra un sensibile aumento del CPI.

Concludendo abbiamo ottimizzato l'algoritmo diminuendo totalmente gli stalli RAW e quelli strutturali, in seguito al loop unrolling, tuttavia rimangono ancora quattro Branch Stalls che derivano dal fetch dei salti, condizionati e non, utili per effettuare i controlli delle espressioni.