Αλγόριθμος Binary Search in ARMv8 – Assembly Language

Γρηγόρης Γρηγοριάδης
Τμήμα Πληροφορικής
Πανεπιστήμιο Κύπρου
Λευκωσία, Κύπρος
ggrego04@ucy.ac.cy

Γιώργος Αχιλλέως
Τμήμα Πληροφορικής
Πανεπιστήμιο Κύπρου
Λευκωσία, Κύπρος
gachil01@ucy.ac.cy

***Περίληψη*— Η παρούσα εργασία παρουσιάζει τη σχεδίαση και την υλοποίηση του αλγορίθμου Binary Search στη γλώσσα προγραμματισμού Assembly ARMv8. Ο αλγόριθμος binary search κάνει αναζήτηση ενός στοιχείου που θέλει να βρει ο χρήστης σε ένα ταξηνομημένο πίνακα. Μέσα από τη συγκεκριμένη εργασία προσπαθήσαμε να καταλάβαουμε πως λειτουργεί ο κώδικας σε Assembly ARMv8 και πως μπορούμε να κάνουμε τον αλγόριθμο μας να γίνει πιο αποδοτικός και γρήγορος.**

Λέξεις κλειδιά: —Binary Search, ARMv8, αποδοτικός, γρήγορος

ΕΙΣΑΓΩΓΗ

Ο αλγόριθμος ο οποίος επιλέξαμε να ασχοληθούμε είναι ο αλγόριθμος Binary Search. [1] Είναι ένας αλγόριθμος αναζήτησης ο οποίος χρησιμοποιεί ταξινομημένους πίνακες για την αναζήτηση κάποιου στοιχείου. Στις μέρες μας όπου τα δεδομένα και το μέγεθος τους αυξάνονται συνεχώς είναι αναγκαία η γρήγορη αναζήτηση των στοιχείων καθώς και ένας καλός αλγόριθμος αναζήτησης. Καλός αλγόριθμος αναζήτησης μπορεί να θεωρηθεί ο οποιοσδήποτε αλλά εμείς επιλέξαμε ένα αλγόριθμο ο οποίος σε μία δομή δεδομένων όπως ο πίνακας αν είναι ταξινομημένος μπορεί σε ελάχιστο χρόνο αναζήτησης να βρει αν υπάρχει το στοιχείο που ψάχνεις να βρεις ή όχι.

Ο λόγος για τον οποίο επιλέξαμε αλγόριθμο αναζήτησης και όχι αλγόριθμο κάποιου παιχνιδιού αναφέρθηκε νωρίτερα αλλά για να είμαστε πιο σωστοί πρέπει να αναφερθούμε και στο ότι σήμερα με τον αμέτρητο αριθμό αλγορίθμων που υπάρχουν στο διαδίκτυο ένας χρήστης ο οποίος θέλει να βρει δεδομένα του σε μία δομή δεδομένων χρειάζεται ένα αλγόριθμο που μπορεί θα χρησιμοποιήσει αν θα είναι αναδρομικός ακόμη και με βασικές γνώσεις προγραμματισμού και επίσης να κατανοήσει τη λειτουργία του κώδικα και πως θα μπορεί ο ίδιος να τον χρησιμοποιήσει χωρίς τη βοήθεια τρίτων.

Στο συγκεκριμένο Report, επικεντρωθήκαμε αρχικά στο να μετατρέψουμε τον αλγόριθμο μας από τη γλώσσα υψηλού επιπέδου την οποία χρησιμοποιήσαμε, στη περίπτωση μας η C, σε συμβολική γλώσσα προγραμματισμού όπως είναι η ARMv8. Πέραν όμως από τη συγκεκριμένη μετατροπή προσπαθήσαμε με διάφορες τεχνικές που αναφέρονται αργότερα να βελτιστοποιήσουμε τον κώδικα σε συμβολική γλώσσα προγραμματισμού. Επίσης συγκρίναμε τους δύο κώδικες μας, σε συμβολική γλώσσα προγραμματισμού και σε γλώσσα υψηλού επιπέδου.

Τέλος, θεωρούμε ότι το συγκεκριμένο Report αξίζει ενδιαφέροντος αλλά και μελέτης γιατί οι αλγόριθμοι αναζήτησης είναι παντού και ο καθένας μας χρειάζεται να γνωρίζει ένα καλό, απλό και γρήγορο αλγόριθμο ο οποίος να τον βοηθά στο πρόβλημα για το οποίο δημιουργήθηκε χωρίς περαιτέρω πολυπλοκότητα.

ΠΕΡΙΣΓΡΑΦΗ ΤΟΥ ΑΛΟΡΙΘΜΟΥ ΣΕ ΓΛΩΣΣΑ ΥΨΗΛΟΥ ΕΠΙΠΕΔΟΥ

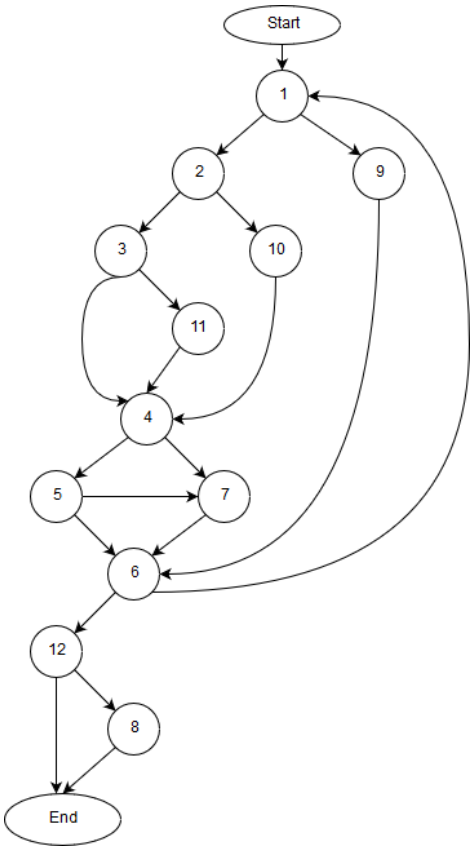
[2] Η υλοποίηση του αλγορίθμου μας σε γλώσσα υψηλού επιπέδου γίνεται αναδρομικά. Αρχικά έχουμε ένα πίνακα ενός εκατομμυρίου θέσεων μέσα στον οποίο τοποθετούνται οι ένα εκατομμύριο αριθμοί του αρχείου μας. Όταν τελειώσει η εισαγωγή των αριθμών από το αρχείο στον πίνακα, το πρόγραμμα μας ζητά από το χρήστη να επιλέξει τον αριθμό που θέλει να βρει και καλείται η αναδρομική συνάρτηση BinarySearch. Η συνάρτηση μας παίρνει ως παραμέτρους την τιμή του χρήστη, την πρώτη και τελευταία θέση του πίνακα και τον πίνακα. Βρίσκει το μέσο του πίνακα και ελέγχει αν η τιμή του πίνακα ισούται με την τιμή στη μεσαία θέση του πίνακα. Αν ισχύει επιστρέφει το μέσο του πίνακα. Ελέγχει αν η πρώτη θέση του πίνακα είναι μεγαλύτερη από την τελευταία και επιστρέφει -1 αν είναι αληθές. Συνεχίζει αν δεν βρήκε το στοιχείο ελέγχοντας αν η τιμή που έδωσε ο χρήστης είναι μικρότερη από τον αριθμό στη μεσαία θέση του πίνακα, αν ισχύει καλεί την BinarySearch μέχρι να βρει τον αριθμό με παραμέτρους την τιμή που έδωσε ο χρήστης, την πρώτη θέση του πίνακα αλλά για τελευταία θέση του πίνακα την μεσαία θέση – 1 και τέλος τον πίνακα. Αν ο προηγούμενος έλεγχος δεν ισχύει ελέγχει αν η τιμή που έδωσε ο χρήστης είναι μεγαλύτερη από τον αριθμό στη μεσαία θέση του πίνακα, αν ισχύει καλεί την BinarySearch μέχρι να βρει τον αριθμό με παραμέτρους την τιμή που έδωσε ο χρήστης, αρχική θέση το μέσο του πίνακα + 1, την τελική θέση του πίνακα και τέλος τον πίνακα.

```
#include <stdio.h>
#include <stdlib.h>
int BinarySearch(int value, int first,int last,int data[]){
```

```
    int mid=first+(last-first)/2;
    if(value == data[mid]){
      return(mid);
    }else if (first>last){
      return -1;
    }else{
      if (value<data[mid])
        return BinarySearch(value,first,mid-1,data);
      else if (value>data[mid])
        return BinarySearch(value,mid+1,last,data);
            }
}
int main(){
  int data[1000000];
  FILE *fp=NULL;
  fp = fopen("numbers_1mil.txt", "r");
  int i=0; int value=1; int position;
  while(1){
    if(feof(fp))
            break;
    fscanf(fp,"%d",&data[i]);
    i++;
  }
  printf("Give me the number you want to find: ");
  scanf("%d",&value);
  position=BinarySearch(value,0,1000000,data);
  (position == -1)? printf("The number is not found in the
table!\n"): printf("The number is found in place %d of the
table!\n",position );
  return 0;
}
```

ΔΙΑΓΡΑΜΜΑ ΡΟΗΣ ΤΗΣ ΕΠΗΛΥΣΗΣ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ ΣΕ ΣΥΜΒΟΛΙΚΗ ΓΛΩΣΣΑ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ ΚΑΙ ΠΕΡΙΓΡΑΦΗ ΤΟΥ

Το διάγραμμα ροής μας αρχίζει από τον κόμβο Start. Όταν φθάνει στον κόμβο 1 έχει δύο κόμβους για να μετακινηθεί. Επιλέγοντας τον κόμβο 2 του δίνονται και πάλι δύο κόμβοι για να μετακινηθεί. Αν επιλέξει τον κόμβο 3 μπορεί να πηδήσει στον κόμβο 4 αντί του κόμβου 11. Στον κόμβο 4 φθάνουμε και μέσω του κόμβου 10. Στον κόμβο 4 υπάρχουν και πάλι δύο επιλογές. Με την επιλογή του κόμβου 5 μπορούμε να επιλέξουμε τον κόμβο 6 ή 7. Όμως στον κόμβο 6 φθάνεις και μέσω του 7 αλλά και μέσω του 9 αν επιλεγεί στην αρχή. Ο συγκεκριμένος κόμβος (6) επιστρέφει πίσω στον κόμβο 1, αλλά μπορούμε από τον κόμβο 6 να πάμε στον κόμβο 12 και απευθείας στο τέλος. Υπάρχει όμως και μονοπάτι μετά τον κόμβο 12 που οδηγεί στον κόμβο 8 και μετά στο τέλος.



ΠΕΡΙΓΡΑΦΗ ΚΩΔΙΚΑ ΣΥΜΒΟΛΙΚΗΣ ΓΛΩΣΣΑΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ

BB1: Αυτό το Block ξεκινά από την αρχή του function BinarySearch. Σε αυτό το Block μεταφέρεται ο πίνακας (γραμμή 94), το νούμερο που αναζητεί ο χρήστης (γρ.97), το flag μας που ισούτε με -2 (γρ.99), το αρχικό σημείο του πίνακα που ισούτε με 0 (γρ.96) και το τελευταίο σημείο του που ισούτε με 1,000,000 (γρ.95) από τα saved registers στα result. Μεταφέρθηκαν επίσης και δύο αριθμοί σε καταχωρητές για να μπορέσουμε να εκτελέσουμε πράξεις (γρ.102-104). Οι πράξεις αυτές γίνονται αργότερα για τον υπολογισμό του μέσου του πίνακα και γαίνονται πιο κάτω:

mid=first+(last-first)/2

Όπου mid το μέσο του πίνακα, first το πρώτο του σημείο και last το τελευταίο. Μετά πολλαπλασιάζεται με τον αριθμό 4 (γρ.108) ο οποίος αποθηκεύτηκε καταχωρητή (γρ.99) για να βρεθεί αργοτερα η θέση του σημείου αυτού στον πίνακα (γρ.109). Όταν βρεθεί τότε συγκρίνεται με το νούμερο που ψάχνουμε (γρ.111). Αν είναι μικρότερο, τότε οδηγούμαστε στο Block 9 (γρ.112). Αν δεν είναι τότε πάμε στο Block 2.

```
89   //************[bb1]*************
90   binarySearch:
91           stp     x29,x30,[sp,-32]!
92           add     x29,sp,0
93
94           mov     x0,x19
95           mov     x2,x21
96           mov     x3,x22
97           mov     x6,x25
98           mov     x5,2
99           mov     x7,x26
100          mov     x27,4
101
102          subs    x4,x2,x3
103          sdiv    x4,x4,x5
104          add     x4,x4,x3
105          mov     x24,x4
106
107          //comparison with arr[mid]
108          mul     x4,x4,x27
109          ldr     w1,[x0,x4]
110
111          cmp     x1,x6
112          b.lt    move_right
```

BB2: Αν ο έλεγχος του block 1 δεν είναι αληθής τότε γίνεται ένας άλλος ο οποίος ελέγχει αν το μέσο του πίνακα είναι μεγαλύτερο από το νούμερο που ψάχνουμε. Αν ισχύει ο έλεγχος αυτός τότε μεταφερόμαστε στο Block 10 (γρ.115). Για αντίθετο αποτέλεσμα μεταφερόμαστε στο Block 3.

```
114  //************[bb2]*************
115          b.gt    move_left
```

BB3: Μετά τους ελέγχους των πιο πάνω Blocks, γίνεται και ένας τελευταίος έλεγχος αν το νούμερο του μέσου είναι ίσο με το νούμερο που αναζητούμε. Αν ισχύει, τότε μεταφερόμαστε στο Block 11 (γρ.118). Αν δεν ισχύει μεταφερόμαστε στο Block 4.

```
117  //************[bb3]*************
118          b.eq    found
```

BB4: Όταν εκτελεστεί το Block 2 ή 3 το νούμερο που ψάχνουμε (γρ.124), το καινούργιο αρχικό (γρ.123) και τελικό σημείο του πίνακα (γρ.122) φεύγουν από τα saved registers και πάνε στα result registers. Μετά ελέγχουμε αν το τελευταίο σημείο του πίνακα είναι μικρότερο από το νούμερο που ψάχνουμε (γρ.126). Αν ισχύει, συνεχίζει στο Block 7 (γρ.127) . Αν δεν ισχύει, συνεχίζει στο Block 5.

```
120  //************[bb4]*************
121  return:
122          mov     x2,x21
123          mov     x3,x22
124          mov     x6,x25
125
126          cmp     w2,w6
127          b.lt    false_return
```

BB5: Όταν ο έλεγχος στο Block 4 αποτύχει, γίνεται ακόμη ένας ο οποίος ελέγχει αν το αρχικό σημείο του πίνακα είναι μεγαλύτερο ή ίσο με το νούμερο που αναζητούμε (γρ.130). Αν ισχύει τότε οδηγούμαστε στο Block 7 (γρ.131). Αν δεν ισχύει οδηγούμαστε στο Block 6.

```
129  //************[bb5]*************
130          cmp     w3,w6
131          b.ge    false_return
```

BB6: Σε αυτό το Block μεταφέρεται ο counter μας από το saved register σε result register (γρ.135). Ακολούθως συγκρίνουμε το counter αυτό με τον αριθμό -2 (γρ.137). Αν ισούνται τότε πάει στην αρχή του function δηλαδή στο Block 1 για να ξαναεκτελστεί από την αρχή (γρ.138). Αν δεν ισχύει συνεχίζει στο Block 12.

```
132  //************[bb6]*************
133
134  check:
135          mov     w7,w26
136
137          cmp     w7,-2
138          b.eq    binarySearch
139
140          ldp     x29,x30,[sp],32
141          ret
```

BB7: Σ'αυτό το Block μεταφέρεται ο αριθμός -1 στο counter για να υποδείξει ότι ο αριθμός δε βρέθηκε στον πίνακα (γρ.145). Μετά επιστρέφει στο Block 6 (γρ.147).

```
143  //************[bb7]*************
144  false_return:
145          mov     w26,-1
146
147          b       check
```

BB8: Αυτό το Block τείθεται σε λειτουργία όταν το counter είναι ίσο με -1 δηλαδή όταν ο αριθμός δεν βρίσκεται στον πίνακα. Τυπώνει έξω ότι ο αριθμός δε βρέθηκε στον πίνακα και το πρόγραμμα τερματίζεται (γρ.150-155).

```
149    //************[bb8]*************
150    error:
151            adr     x0,error_str
152            bl      printf
153
154            ldp     x29,x30,[sp],32
155            ret
```

BB9: Εδώ μεταφέρεται ο αριθμός του μέσου από το saved register στο result register (γρ.159) και ύστερα παίρνει τη θέση του αρχικού σημείου του πίνακα (γρ.160). Ακολούθως επιστρέφει στο Block 4 (γρ.162).

```
157    //************[bb9]*************
158    move_right:
159            mov     x4,x24
160            mov     x22,x4
161
162            b       return
```

BB10: Εδώ μεταφέρεται ο αριθμός του μέσου από το saved register στο result register (γρ.166) και ακολούθως παίρνει τη θέση του τελευταίου σημείου του πίνακα (γρ.167). Ακολούθως επιστρέφει στο Block 4 (γρ.169).

```
164    //************[bb10]*************
165    move_left:
166            mov     x4,x24
167            mov     x21,x4
168
169            b       return
```

BB11: Εδώ μεταφέρεται ο αριθμός του μέσου από το saved register στο result register (γρ.173) και ακολούθως παίρνει τη θέση του counter (γρ.174) για να υποδείξουμε ότι ο αριθμός βρέθηκε καθώς και το σημείο του στον πίνακα. Ακολούθως τυπώνεται έξω το κατάλληλο μήνυμα (γρ.177-178) και επιστρέφουμε στο Block 4 (γρ.180).

```
171    //************[bb11]*************
172    found:
173            mov     w4,w24
174            mov     w26,w4
175            mov     w1,w26
176
177            adr     x0,output_str
178            bl      printf
179
180            b       check
```

BB12: Αυτό το Block εκτελείται όταν βγούμε από το function. Γίνεται πρώτα ο έλεγχος του counter και τον συγκρίνουμε με το -1 (γρ.58), δηλαδή ελέγχουμε αν δεν βρέθηκε στον πίνακα ο αριθμός που ψάχναμε. Αν ισούται με -1, τότε μεταφερόμαστε στο Block 9 (γρ.59). Αν δεν ισούνται, τότε το πρόγραμμα τερματίζεται (γρ.61-62).

```
56     //************[bb12]*************
57
58             cmp     w26,-1
59             b.eq    error
60
61             ldp     x29,x30,[sp],32
62             ret
```

ΠΡΟΣΠΑΘΕΙΕΣ ΒΕΛΤΗΣΤΟΠΟΙΗΣΗΣ ΤΟΥ ΚΩΔΙΚΑ ΣΥΜΒΟΛΙΚΗΣ ΓΛΩΣΣΑΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ

Όταν χρησιμοποιήσαμε το bl clock για να μετρήσουμε το χρόνο στην πρώτη εκδοχή, ο χρόνος που υποδήχθηκε ήταν 38474 microseconds στην assembly. Αυτό προσπαθήσαμε να το βελτιστοποιήσουμε αφαιρώντας τις περιττές μεταβλητές, δηλαδή προσπαθήσαμε να κάνουμε τις πράξεις μας με result registers το x0, το x1 και το x2. Αυτό προστέθηκε στην δεύτερη εκδοχή του προγράμματός μας και ο χρόνος μειώθηκε κατά 3000 microseconds, δηλαδή ήταν κοντά στα 35325 microseconds. Οι αλλαγές αυτές φαίνονται πιο κάτω για κάθε Block ξεχωριστά:

Block 1 για την πρώτη εκδοχή:

```
//**************[bb1]*********************
binarySearch:
        stp     x29,x30,[sp,-32]!
        add     x29,sp,0

        mov     x0,x19
        mov     x2,x21
        mov     x3,x22
        mov     x6,x25
        mov     x5,2
        mov     x7,x26
        mov     x27,4

        subs    x4,x2,x3
        sdiv    x4,x4,x5
        add     x4,x4,x3
        mov     x24,x4


        mul     x4,x4,x27
        ldr     w1,[x0,x4]
        cmp     x1,x6
        b.lt    move_right
```

Block 1 για τη δεύτερη εκδοχή:

```
//**************[bb1]*********************
binarySearch:
        stp     x29,x30,[sp,-32]!
        add     x29,sp,0

        mov     x1,x21
        mov     x0,x22
```

```
        mov     x2,2

        subs    x1,x1,x0
        sdiv    x1,x1,x2
        add     x1,x1,x0
        mov     x24,x1

        mov     x2,4
        mul     x1,x1,x2
        mov     x0,x19

        ldr     w0,[x0,x1]
        mov     x1,x25

        cmp     x0,x1
        b.lt    move_right
```

Στο Block 1 για να καταφέρουμε καλύτερο optimization του κώδικα μας αρχικά μειώσαμε τους καταχωρητές που χρησιμοποιούσαμε. Στην πρώτη εκδοχή χρησιμοποιήσαμε οκτώ διαφορετικούς result registers για πράξεις ενώ στη δεύτερη μειώθηκαν σε τρείς, τους x0, x1, x2. Με τη μείωση των καταχωρητών που χρησιμοποιήσαμε πετύχαμε να μειώσουμε και τις εντολές mov και να γίνει πιο γρήγορα load η μεσαία θέση του πίνακα στον καταχωρητή w0.

Block 4 για την πρώτη εκδοχή:

```
//**************[bb4]********************
return:
        mov     x2,x21
        mov     x3,x22

        cmp     w2,w6
        b.lt    false_return
```

Block 4 για την δεύτερη εκδοχή:

```
//**************[bb4]********************
return:
        mov     x0,x21
        mov     x1,x25

        cmp     w0,w1
        b.lt    false_return
```

Στο συγκεκριμένο block πετύχαμε να έχουμε και πάλι καλύτερο χρόνο χρησιμοποιώντας και πάλι μόνο τους καταχωρητές x0 και x1.

Block 5 για την πρώτη εκδοχή:

```
//**************[bb5]********************

        cmp     w3,w6
        b.ge    false_return
```

Block 5 για την δεύτερη εκδοχή:

```
//**************[bb5]********************

        mov     x0,x22

        cmp     w0,w1
        b.ge    false_return
```

Στο Block 5 χρησιμοποιούμε και πάλι έξυπνα τους καταχωρητές x0 και x1 για τη σύγκριση μας.

Block 6 για την πρώτη εκδοχή:

```
//**************[bb6]********************

check:
        mov     w7,w26

        cmp     w7,-2
        b.eq    binarySearch

        ldp     x29,x30,[sp],32
        ret
```

Block 6 για την δεύτερη εκδοχή:

```
//**************[bb6]********************

check:
        mov     w0,w26

        cmp     w0,-2
        b.eq    binarySearch

        ldp     x29,x30,[sp],32
        ret
```

Σε αυτό το block για καλύτερο optimization χρησιμοποιήσαμε για τον έλεγχο μας μόνο τον x0, όταν πριν χρησιμοποιούσαμε τον καταχωρητή w7.

Block 9 για την πρώτη εκδοχή:

```
//**************[bb9]********************

move_right:
        mov     x4,x24
        mov     x22,x4

        b       return
```

Block 9 για την δεύτερη εκδοχή:

```
//**************[bb9]********************
```

```
move_right:
        mov     x0,x24
        mov     x22,x0

        b       return
```

Στο παραπάνω block προσπαθήσαμε και αλλάξαμε τον καταχωρητή x4 με τον καταχωρητή x0 και πετύχαμε καλύτερο optimization στις δύο εντολές mov που κάνουμε.

Block 10 για την πρώτη εκδοχή:

```
//**************[bb10]**********************

move_left:
        mov     x4,x24
        mov     x21,x4

        b       return
```

Block 10 για την δεύτερη εκδοχή:

```
//**************[bb10]**********************

move_left:
        mov     x0,x24
        mov     x21,x0

        b       return
```

Στο αυτό το block προσπαθήσαμε και αλλάξαμε τον καταχωρητή x4 με τον καταχωρητή x0 και πετύχαμε καλύτερο optimization στις δύο εντολές mov που κάνουμε.

Block 11 για την πρώτη εκδοχή:

```
//**************[bb11]**********************

found:
        mov     w4,w24
        mov     w26,w4
        mov     w1,w26

        adr     x0,output_str
        bl      printf

        b       check
```

Block 11 για την δεύτερη εκδοχή:

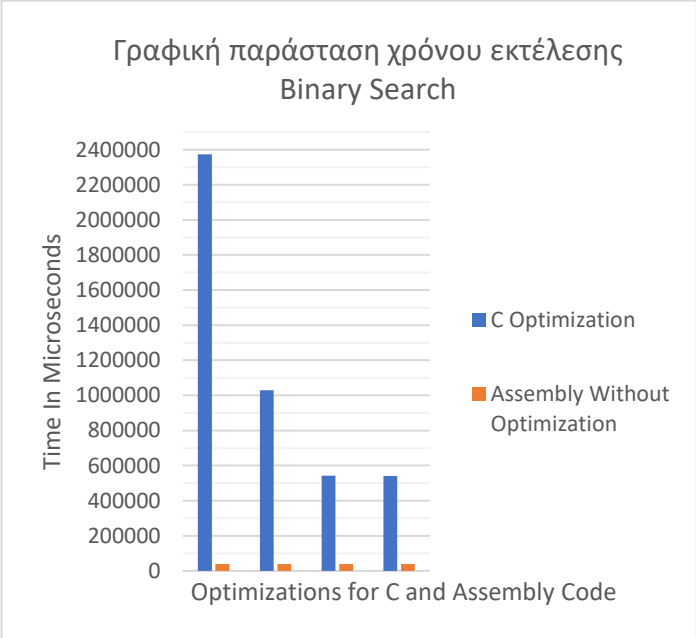
```
//**************[bb11]**********************

found:

        mov     w0,w24
        mov     w26,w0
        mov     w1,w26

        adr     x0,output_str
        bl      printf

        b       check
```

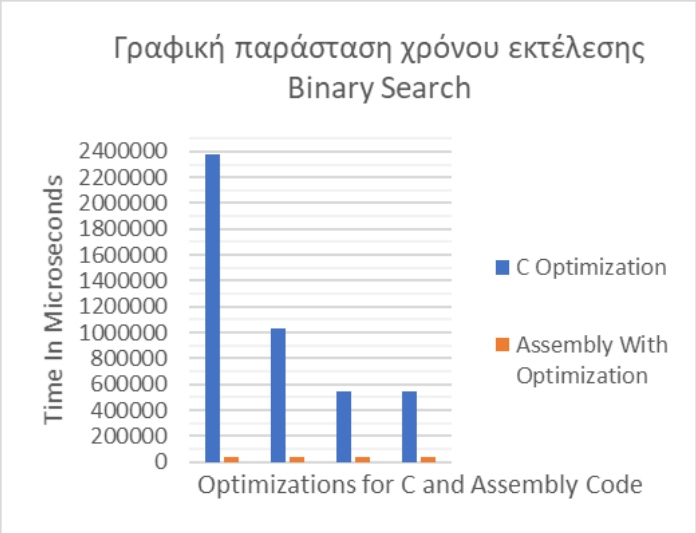
Στο συγκεκριμένο block δεν χρησιμοποιούμε στη δεύτερη μας εκδοχή τον καταχωρητή w4 αλλά μόνο τους καταχωρητές w0 κα w1 για τις εντολές mov μας και έτσι ο κώδικας γίνεται πιο optimized.

Αυτά ήταν τα Blocks τα οποία τροποποιήθηκαν κατά τη διάρκεια της βελτιστοποίησης. Τα υπόλοιπα διατηρήθηκαν τα ίδια.

ΣΥΓΚΡΙΣΗ ΤΟΥ ΚΩΔΙΚΑ ΣΥΜΒΟΛΙΚΗΣ ΓΛΩΣΣΑΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ ΜΕ ΤΟΝ ΑΝΤΙΣΤΟΙΧΟ ΚΩΔΙΚΑ ΠΟΥ ΠΑΡΑΓΕΙ Ο GCC

Συγκρίναμε τις δύο εκδοχές μας με τα αποτελέσματα του clock στην C για O0 ,O1 ,O2 και O3 και είχαμε τα ακόλουθα αποτελέσματα:





Για να εξάγουμε τις πιο πάνω γραφικές τρέξαμε τον αλγόριθμο μας για ένα εκατομμύριο φορές τόσο στην Assembly όσο και στη C.

Στην πρώτη γραφική παράσταση βλέπουμε τη σύγκριση του κώδικα της assembly με τον αντίστοιχο στη C για O0, O1, O2, O3. Παρατηρούμε ότι πάντα ο χρόνος εκτέλεσης του αλγορίθμου μας σε assembly ήταν μικρότερος από τον αντίστοιχο της C.

Στη δεύτερη μας γραφική παράσταση βλέπουμε τη σύγκριση του optimized κώδικα της assembly με τον αντίστοιχο στη C για O0, O1, O2, O3. Πάλι παρατηρούμε ότι πάντα ο χρόνος εκτέλεσης του αλγορίθμου μας σε assembly ήταν μικρότερος από τον αντίστοιχο της C, αλλά σε αυτή την περίπτωση ο κώδικας μας σε assembly ήταν κατά 3000 microseconds πιο γρήγορος από τον προηγούμενο κώδικα σε assembly.

ΣΥΜΠΕΡΑΣΜΑΤΑ

Μέσα από το συγκεκριμένο project πέραν του ότι είχαμε να υλοποιήσουμε τον αλγόριθμο Binary Search προσπαθήσαμε να κατανοήσουμε το πως γίνεται το σωστό optimization σε μία γλώσσα χαμηλού επιπέδου. Με το σωστό optimization πετύχαμε μέσα από τα κατάλληλα πειράματα να μειώσουμε το χρόνο που χρειάζεται ο αλγόριθμος μας για να τρέξει και να γίνει πιο αποδοτικός.

ΑΝΑΦΟΡΕΣ

[1] https://en.wikipedia.org/wiki/Binary_search_algorithm
[2] http://www.geeksforgeeks.org/binary-search/

ΠΑΡΑΡΤΗΜΑ «Α»

ΚΩΔΙΚΑΣ ΣΕ ASSEMBLY ΠΟΥ ΠΑΡΟΥΣΙΑΖΕΙ ΤΗ MAIN ΤΟΥ ΚΩΔΙΚΑ ΜΑΣ

```
[1]         .data
[2]  message_str:       .string "Give me the number you want to find: "
[3]  scanf_str:         .string "%llu"
[4]  FileOpenMode:      .string     "r"
[5]  FileName:          .string     "numbers_1mil.txt"
[6]  fscanf_str:        .string     "%llu"
[7]  output_str:        .string "The number is found in place %llu of the table!\n"
[8]  error_str:         .string "The number is not found in the table!\n"
[9]  //*************************************************************
         .text
[10]        .global     main
[11]
[12] main:
[13]        stp         x29, x30, [sp, -32]! // Move the stack pointer 32 bytes down and store x29 and x30
[14]
[15]        add         x29, sp, 0          // Copy the value of stack pointer to Frame Pointer
[16]
[17]        ldr         x23,=4000000         // Load number 4 mil. in reg x23
[18]        ldr         x24,=1000000        // Load number 1 mil. in reg x24
[19]        mov         x0, x23             // Move x23 in register x0
[20]        bl          malloc              // Do malloc
[21]
[22]        str         x0, [x29, 24]       // Store the x0 into stack
[23]        ldr         x0, [x29, 24]       // Move address of allocated space
[24]        mov         x22,x24             // Move x24 in register x22
[25]        bl          initTable           // Call function initTable
[26]        ldr         x0, [x29, 24]       // Move address of allocated space
[27]        mov         x22,x24             // Move x24 in register x22
[28]        ldr         x0,[x29,24]         // Move address of allocated space
[29]        mov         x19,x0              // Move x in register x19
[30]
[31] Question:
[32]        adr         x0,message_str      // Load the address of message_str
[33]        bl          printf              // Call printf
[34]
[35]        add         x1,x29,28           // Set the location to save the state
[36]        adr         x0,scanf_str        // Load the address of scanf_str
[37]        bl          scanf               // Call scanf
[38]
[39]        ldr         x25,[x29,28]        // Load the read value in register x25
[40]        ldr         x21,=1000000        // Load number 1 mil in reg x21
[41]        mov         x22,0               // Put zero in register x22
[42]        mov         x26,-2              // Put -2 in register x26
[43]
[44]       bl          binarySearch         // Call function binarySearch
[45]
[46] //*************************[bb12]*************************
[47]
```

```
[48]    cmp     w26,-1          // Compare w26 with -1
[49]    b.eq    error           // if equal jump in Error
[50]
[51]    ldp     x29,x30,[sp],32 // Release the stack space
[52]    ret                     // Return to loading function
```

ΠΑΡΑΡΤΗΜΑ «Β»

ΚΩΔΙΚΑΣ ΣΕ ASSEMBLY ΠΟΥ ΠΑΡΟΥΣΙΑΖΕΙ ΤΗΝ ΑΡΧΙΚΟΠΟΙΗΣΗ ΚΑΙ ΤΟ ΓΕΜΙΣΜΑ ΤΟΥ ΠΙΝΑΚΑ ΜΑΣ

```
[53] initTable:
[54]    stp     X29, X30, [sp, -32]!  // Move the stack pointer 32 bytes down and store x29 and x30
[55]    add     x29, sp, 0            // Add sp in register x29
[56]
[57]    mov     x19,x0                // Move x0 in register x19
[58]
[59]    // Open the File For reading
[60]    adr     x1, FileOpenMode      // FileOpenMode go in register x1
[61]    adr     x0, FileName          // FileName go in register x0
[62]    bl      fopen                 // Call fopen
[63]
[64]    mov     x20, x0               // Move x0 in register x20
[65]
[66] loop_init:
[67]    add     x2,x19, 0             // location the value will be stored
[68]    add     x19,x19,4             // x19 = x19 + 4
[69]    adr     x1, fscanf_str        // Put in x1 the scanf value
[70]    mov     x0, x20               // The File pointer
[71]    bl      fscanf                // Call fscanf
[72]
[73]    add     x22,x22,-1            // x22 = x22 - 1
[74]    cbnz    x22, loop_init        // If x22 != 0 then call loop_init
[75]
[76]    ldp     x29,x30,[sp],32       // Release the stack space
[77]    ret                           // Return to loading function
```

ΠΑΡΑΡΤΗΜΑ «Γ»

ΚΩΔΙΚΑΣ ΣΕ ASSEMBLY ΠΟΥ ΠΑΡΟΥΣΙΑΖΕΙ ΤΗΝ OPTIMIZED VERSION ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ ΜΑΣ.

```
[78] //*************************[bb1]*************************
[79] binarySearch:
[80]    stp     x29,x30,[sp,-32]!     // Move the stack pointer 32 bytes down and store x29 and x30
[81]    add     x29,sp,0              // Copy the value of stack pointer to Frame Pointer
[82]
[83]    mov     x1,x21                // Move x21 in register x1
[84]    mov     x0,x22                // Move x22 in register x0
[85]    mov     x2,2                  // Move number 2 in register x2
[86]
[87]    subs    x1,x1,x0              // x1 = x1 - x0
[88]    sdiv    x1,x1,x2              // x1 = x1/x2
[89]    add     x1,x1,x0              // x1 = x1 + x0
[90]    mov     x24,x1                // Move x1 in register x24
[91]
[92]    //comparison with arr[mid]
[93]    mov     x2,4                  // Move number 4 in register x2
[94]    mul     x1,x1,x2              // x1 = x1*x2
[95]    mov     x0,x19                // Move x19 in register x0
[96]    ldr     w0,[x0,x1]            // Load the value in place x1 of the table
[97]    mov     x1,x25                // Move x25 in register x1
[98]    cmp     x0,x1                 // Compares x0 with x1
[99]    b.lt    move_right            // x0 < x1 go to move_right
[100]
[101]//*************************[bb2]*************************
[102]   b.gt    move_left             // x0 > x1 go to move_left
[103]
[104]//*************************[bb3]*************************
[105]   b.eq    found                 // x0 = x1 go to found
[106]
[107]//*************************[bb4]*************************
[108]return:
[109]   mov     x0,x21                // Move x21 in register x0
[110]   mov     x1,x25                // Move x25 in register x1
[111]
[112]   cmp     w0,w1                 // Compares w0 with w1
[113]   b.lt    false_return          // w0 < w1 go to false_return
[114]//*************************[bb5]*************************
[115]   mov     x0,x22                // Move x22 in register x0
[116]   cmp     w0,w1                 // Compare w0 with w1
[117]   b.ge    false_return          // w0 >= w1 go to false_return
[118]
[119]//*************************[bb6]*************************
[120]check:
[121]   mov     w0,w26                // Move w26 in register w0
[122]   cmp     w0,-2                 // Move number -2 in register w0
[123]   b.eq    binarySearch          // w0 = -2 go to binarySearch
[124]
[125]   ldp     x29,x30,[sp],32       // Release the stack space
[126]   ret                           // Return to loading function
[127]
[128]
[129]//*************************[bb7]*************************
[130]false_return:
[131]   mov     w26,-1                // Move -1 in register w26
[132]
[133]   b       check                 // Go back to check
[134]
[135]//*************************[bb8]*************************
[136]error:
[137]   adr     x0,error_str          // Load the address of error_str
[138]   bl      printf                // Call printf
[139]
[140]   ldp     x29,x30,[sp],32       // Release the stack space
```

```
[141]       ret                         // Return to loading function
[142]
[143]//*************************[bb9]*************************
[144]move_right:
[145]       mov         x0,x24          // Move x24 in register x0
[146]
[147]       mov         x22,x0          // Move x0 in register x22
[148]
[149]       b           return          // Go back to return
[150]
[151]//*************************[bb10]************************
[152]move_left:
[153]       mov         x0,x24          // Move x24 in register x0
[154]
[155]       mov         x21,x0          // Move x0 in register x21
[156]
[157]       b           return          // Go back to return
[158]//*************************[bb11]************************
[159]found:
[160]
[161]       mov         w0,w24          // Move w24 in register w0
[162]
[163]       mov         w26,w0          // Move w0 in w26
[164]       mov         w1,w26          // Move w26 in register w1
[165]       adr         x0,output_str   // Load the address of output_str
[166]       bl          printf          // Call printf
[167]
[168]       b           check           // Go back to check
```

ΠΑΡΑΡΤΗΜΑ «Δ»

ΠΑΡΑΔΕΙΓΜΑΤΑ ΕΚΤΕΛΕΣΗΣ ΤΟΥ ΠΡΟΓΡΑΜΜΑΤΟΣ ΜΑΣ.

Give me the number you want to find: 5
The number is found in place 4 of the table!

Give me the number you want to find: 70
The number is found in place 69 of the table!

Give me the number you want to find: 567
The number is found in place 566 of the table!

Give me the number you want to find: 667
The number is found in place 666 of the table!

Give me the number you want to find: 601979
The number is found in place 601978 of the table!

Give me the number you want to find: 1000000
The number is found in place 999999 of the table!

Give me the number you want to find: 1
The number is found in place 0 of the table!