Εργαστηριο 1 - Μικροεπεξεργαστες & Περιφερειακα

Σακελλαρίου Βασίλειος - AEM:9400
Φίλης Χάρης - AEM:9449

April 2021

## 0.1 Abstract

Στόχος της δοθείσας εργασίας είναι να εξοικειωθούμε με το εργαλείο της *Keil - uVision5/MDK - Version 5* και η συγγραφή μιας ρουτίνας ***get_hash(char *a, char *hashTable, int* dst)*** σε *assembly* που:

- Για κάθε Κεφαλαίο λατινικό γράμμα θα προσθέτει στο *hash* τον αριθμό που φαίνεται στον παρακάτω πίνακα.
- για κάθε αριθμητικό ψηφίο θα αφαιρεί την τιμή του αριθμητικού ψηφίο από το *hash* του αλφαριθμητικού.
- και το *hash* του αλφαριθμητικού δεν επηρεάζεται από οποιοδήποτε στοιχείο του αλφαριθμητικού που δεν είναι κεφαλαίο λατινικό γράμμα ή αριθμητικό ψηφίο .

| | | | | | |
|---|---|---|---|---|---|
| A | 18 | J | 2 | S | 23 |
| B | 11 | K | 12 | T | 4 |
| C | 10 | L | 3 | U | 26 |
| D | 21 | M | 19 | V | 15 |
| E | 7 | N | 1 | W | 6 |
| F | 5 | O | 14 | X | 24 |
| G | 9 | P | 16 | Y | 13 |
| H | 22 | Q | 20 | Z | 25 |
| I | 17 | R | 8 | | |

Figure 1: Hash_Table

## 0.2 Implementation

Αρχικά υλοποιήσαμε την ρουτίνα που μας ζητήθηκε σε κώδικα C καθώς αυτό μας ήταν πιο οικείο. Ο κώδικας δίνεται για τυπικά για τον αναγνώστη.

```
int get_hash(char* a, int* hashTable){
    int str_index=0;
    int hashKey_index=0;//Initialising with this value, and
   we map 'A' in ASCII to hashTable_indexing
    char tempChar;
    int hash = 0;
    while(a[str_index]!='\0'){
        tempChar=a[str_index];
        //Case of a capital letter
        if(tempChar>64 && tempChar<91){
```

```
10              hashKey_index=(int)tempChar-65;//With the
    typecasting I get the ASCII counterpart of the character
11              hash=hash+hashTable[hashKey_index];
12          }
13          //Case of a number
14          else if(tempChar>47 && tempChar<58){
15              hash=hash-(int)tempChar+48;
16          }
17          str_index++;
18      }
19      return hash;
20  }
```

Στη συνέχεια κάναμε αρκετές προσπάθειες για την δημιουργία κώδικα *assembly* και τελικά καταλήξαμε στον παραδοτέο(τα σχόλια είναι υπέρ αναλυτικά για την κατανόηση του κώδικα).
Ωστόσο για να εξηγήσουμε ο κώδικας χωρίζεται σε τρία μέρη - *tags*: **loop**, **condNum**, **exit**.
Οι καταχωρητές r0 και r1 είναι οι δείκτες στα ορίσματα της συνάρτησης δηλαδή στόν πίνακα της εισαγόμενης συμβολοσειράς και στο *hash_table* αντίστοιχα. Επίσης, ο r0 έχει και στο τέλος την επιστρεφόμενη τιμή (hash). Επιπλέον, σαν όρισμα έχουμε και την διεύθυνση στην οποία αποθηκεύεται στην μνήμη το αποτέλεσμα (*dst / *hash*) και χρησιμοποιείται για αυτό σύμφωνα με το πρότυπο ο καταχωρητής r2. Δεσμεύουμε στατική μνήμη με την εντολή *PUSH* για άλλους 4 καταχωρητές των οποίων η λειτουργία εξηγείται αναλυτικά στα σχόλια.Αυτοί αρχικοποιούνται με 0 εκτός από τον r6.
Ο r6 χρησιμοποιείται σαν πολλαπλασιαστής στον δείκτη σε bytes ώστε να διατρέχουμε τον *hash_table* σωστά εφόσον είναι πίνακας ακεραίων (4 bytes).
Το ***loop*** αντιπροσωπεύει το βασικό *while loop* του κώδικα c 0.2 που τερματίζει όταν ο r0 φτάνει στο τερματικό χαρακτήρα που είναι το NULL = "\0". Σε αυτόν γίνονται οι έλεγχοι για το πρώτο if όπου ελέγχουμε αν ο χαρακτήρας είναι κεφαλαίο γράμμα. Εφόσον είναι αληθείς οι συνθήκες, αντιστοιχίζεται ο χαρακτήρας στο αντίστοιχο *hash_key* (αναλυτικά φαίνεται στο κώδικα σε σχόλια). Αν η αναπαράσταση του χαρακτήρα σε ASCII είναι μικρότερη η ίση του 64 πηγαίνουμε στο label ***condNum***. Όταν δεν πληρούνται οι έλεγχοι σε καμία περίπτωση πηγαίνουμε στον επόμενο χαρακτήρα του αλφαριθμητικού.
**Σημαντικό** σημείο που πρέπει να τονίσουμε είναι οτι για να διατρέξουμε το hashTable που είναι πίνακας από *Integers* πρέπει το offset με το οποίο ζητάμε στοιχεία από τον r1 πρέπει να είναι 4 bytes κάτι που μας ήταν γνωστό και από την MIPS.
Το ***condNum*** αντιπροσωπεύει το 2 if condition του κώδικα 0.2. Ελέγχουμε αν ο χαρακτήρας είναι αριθμός μεταξύ 0-9. Εφόσον είναι αληθείς οι συνθήκες προκύπτει μια τιμή του hash (αναλυτικά φαίνεται στο κώδικα σε μορφή

σχολίων). Από την άλλη αν δεν πληρούνται οι έλεγχοι πάμε στο ***loop*** tag. Τέλος το label ***exit*** είναι εκεί που τελειώνει η λούπα, Φορτώνεται ο r0 με το αποτέλεσμα που εχει αποθηκευτεί στον sp από το r4 σε κάθε tag. Στην συνέχεια αποθηκεύουμε την τιμή του hash του καταχωρητή r4 στην μνήμη δηλαδή στον r2 καταχωρητή(όπως ζητά η εκφώνηση της εργασίας).Τέλος κάνουμε *branch to link register* για να επιστρέψουμε στην *main* αφού έχουμε αποδεσμεύσει τους καταχωρητές που δεσμεύσαμε στο stack.

## 0.3 Problems that we faced

Διάφορα προβλήματα πάνω στον κώδικα διορθώθηκαν πριν το εργαστήριο.
Στο εργαστήριο είχαμε ξεχάσει να κάνουμε αποθήκευση στην μνήμη του αποτελέσματος του hash (καταχωρητής r4) το οποίο επισημάνθηκε. Στη συνέχεια υλοποιήθηκε αυτό όπως όριζε η εκφώνηση της άσκησης.

## 0.4 Testing

Όσον αφορά το testing του κώδικα, σαν πρώτο βήμα κάναμε αναλυτικό debugging βλέποντας την τιμή του κάθε καταχωρητή ***Registers Tab*** και εκεί βρήκαμε και πολλά λάθη της υλοποίησης που διορθώθηκαν.
Στην συνέχεια βάλαμε πλήθος διαφορετικών συμβολοσειρών σαν *input* και με διάφορους ελέγχους ή main επιστρέφει διαφορετικό *return value* πχ 1.

### 0.4.1 Remark

Γενικά η τελική μορφή του κώδικα δουλεύει για όλες τις συμβολοσειρές αν μπουν μεμονωμένα σαν είσοδος δηλαδή κληθεί μια φορά η συνάρτηση *__asm int get_hash(...,...)*. Ωστόσο, σε πολλαπλές κλήσεις (για διαδοχικά αλφαριθμητικά) η επιστρεφόμενη τιμή από την πρώτη κλήση είναι λάθος. Το αποδίδουμε αυτό στο board και στην διαχείριση του stack του. Δεν μπόρεσε να λυθεί στο εργαστήριο το πρόβλημα. Βέβαια αυτό δεν ζητούνταν από την εκφώνηση, απλά προέκυψε κατά την διαδικασία του testing.

**ΤΕΛΟΣ ΑΝΑΦΟΡΑΣ**