Μικροεπεξεργαστές και Περιφερειακά

Εργαστήριο 1ο - Ομάδα 1

Ονοματεπώνυμο	ΑΕΜ	E-mail
Καλαντζής Γεώργιος	**8818**	**gkalantz@ece.auth.gr**
Κοσέογλου Σωκράτης	**8837**	**sokrkose@ece.auth.gr**

1. Σκοπός Εργασίας

Σκοπός του 1ου εργαστηρίου είναι η δημιουργία μιας ρουτίνας γραμμένη σε γλώσσα **Assembly ARM,** η οποία υπολογίζει το συνολικό hash ενός αλφαριθμητικού με βάση τον παρακάτω πίνακα.

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		

2. Υλοποίηση σε C

Αρχικά, για απλότητα υλοποιήσαμε τον επιθυμητό αλγόριθμο σε γλώσσα **C** στο περιβάλλον **VSCode.** Ο κώδικας αυτός φαίνεται στην παρακάτω φωτογραφία. Όπως φαίνεται, αρχικά, δημιουργούμαι έναν πίνακα **hash[26]** ο οποίος αποθηκεύει τις τιμές του παραπάνω πίνακα για κάθε κεφαλαίο γράμμα του λατινικού αλφαβήτου. Στην συνέχεια αρχικοποιούμε ένα τυχαίο αλφαριθμητικό **string** το οποίο θα ελέγξουμε. Έπειτα καλούμε την ρουτίνα η οποία υπολογίζει το συνολικό **hash** του **string**. Στην συνάρτηση αυτή, όπως φαίνεται, τρέχει μια **while loop** μέχρις ότου φθάσουμε στον τέλος του αλφαριθμητικού **string.** Εσωτερικά της λούπας, βρίσκουμε τον αριθμό **ASCII** του εκάστοτε γράμματος του αλφαριθμητικού και στην συνέχεια το συγκρίνουμε με τα διαστήματα [65, 90] και [48, 57] τα οποία αναπαριστούν στο **ASCII** τους κεφαλαία λατινικά γράμματα [A-Z] καθώς και τους αριθμούς [0-9]. Σε περίπτωση που το εκάστοτε γράμμα βρίσκεται στο 1ο διάστημα, τότε προσθέτουμε σε μια μεταβλητή **sum (**που κρατάει το άθροισμα hash του αλφαριθμητικού) την τιμή **hash[ascii - 65]** ώστε να πάρουμε την κατάλληλη τιμή του πίνακα, βάζοντας ένα offset 65. Ενώ, εάν η τιμή **ASCII** βρίσκεται στο δεύτερο διάστημα, αφαιρούμε από την μεταβλητή **sum** τον αριθμό **ascii – 48**. Τέλος επιστρέφουμε την τιμή **sum** και την κάνουμε plot στην κονσόλα μέσω της **printf**.

```c
#include <stdio.h>
#include <string.h>

int myHashSummarizer(char* string, int* hash){
    int sum = 0;
    int i = 0;
    while(string[i] != NULL){
        int ascii = (int ) string[i];
        if(ascii >= 65 && ascii <= 90){
            sum += hash[ascii - 65];
        }
        if(ascii >= 48 && ascii <= 57){
            sum -= (ascii - 48);
        }
        i++;
    }
    return sum;
}

int main(){
    int hash[26] = {18, 11, 10, 21, 7, 5, 9, 22, 17, 2, 12, 3, 19, 1, 14, 16, 20, 8, 23, 4, 26, 15, 6, 24, 13, 25};
    char* string = "AaB9C 5F!-EZ.";
    int sum = myHashSummarizer(string, hash);
    printf("sum = %d", sum);
    return 0;
}
```

3. Υλοποίηση σε Assembly ARM

Στην συνέχεια, υλοποιήσαμε την παραπάνω ρουτίνα σε **Assembly ARM** στο περιβάλλον **keil µVision 5**.

- main():

Στην **main()** όπως και προηγουμένως, αρχικοποιούμε τον πίνακα που κρατάει τα **hash** για κάθε κεφαλαίο λατινικό γράμμα, **hash[]**. Επίσης, αρχικοποιούμε την μεταβλητή **string = "AaB9C 5F!-EZ."** την οποία θα ελέγξουμε, αλλά και την μεταβλητή **actualSum = 62** η οποία κρατάει το πραγματικό hash το οποίο υπολογίσαμε «στο χέρι», έτσι ώστε να το συγκρίνουμε στην συνέχεια με αυτό που υπολόγισε η ρουτίνα μας και να ελέγξουμε την ορθότητα της. Στην συνέχεια καλούμε την assembly ρουτίνα __asm int myASM_HashSummarizer(string, hash), δίνοντας ως ορίσματα το αλφαριθμητικό **string** και τον πίνακα **hash[]**.

- __asm int myASM_HashSummarizer(string, hash):

Στην Assembly ρουτίνα, αρχικά, εκτελούμε την εντολή **PUSH {r4, r5, lr}** έτσι ώστε να βάλουμε τους καταχωρητές r4, r5 και link register (ώστε να κρατήσουμε την return address) στο activation record της ρουτίνας μας. Έπειτα, κάνουμε **MOV r2, r0** και **MOV r4, r1** έτσι ώστε να αντιγράψουμε το περιεχόμενο του καταχωρητή r0 (που είναι το 1ο όρισμα, δηλαδή η μεταβλητή **string**) στον καταχωρητή r2 καθώς και να αντιγράψουμε το περιεχόμενο του καταχωρητή r1 (που είναι το 2ο όρισμα, δηλαδή η πίνακας **hash**) στον καταχωρητή r4. Στην συνέχεια αρχικοποιήσαμε τους καταχωρητές r1 και r3 ως 0 με τις εντολές **MOVS r1, #0** και **MOVS r3, #0**, όπου ο καταχωρητής r1 αναπαριστά την μεταβλητή **sum** και ο καταχωρητής r3 την μεταβλητή **i** που δείχνει σε ποιο γράμμα του αλφαριθμητικού βρισκόμαστε. Έπειτα κάνουμε **branch** στο block goto1 μέσω της εντολής **B goto1**.

- goto1:

Στο block goto1 εκτελούμε την εντολή **LDRB r0, [r2, r1]** με την οποία φορτώνουμε στον καταχωρητή r0 την διεύθυνση του καταχωρητή r2 με offset την τιμή του καταχωρητή r1, δηλαδή φορτώνουμε την τιμή ***(string + i)**. Στην συνέχεια συγκρίνουμε την τιμή αυτή με το 0 με την εντολή **CMP r0, #0**, έτσι ώστε να ελέγξουμε εάν βρισκόμαστε στο τέλος του αλφαριθμητικού, καθώς το 0 σε ASCII είναι το **NULL**. Εάν όχι, τότε κάνουμε **branch** στο block **whileLoop** μέσω της εντολής **BNE whileLoop**.

- whileLoop:

Στο block whileLoop εκτελούμε αρχικά την εντολή **CMP r0, #65** ώστε να συγκρίνουμε την τιμή ***(string + i)** με τον αριθμό 65. Εάν το περιεχόμενο του καταχωρητή r0 είναι μικρότερο από 65 (**BLT goto2**) πήγαινε στο block goto2. Στην συνέχεια, συγκρίνουμε την τιμή ***(string + i)** με την τιμή 90 μέσω της εντολής **CMP r0, #90** και εάν είναι μεγαλύτερη τότε πήγαινε στο block goto2 (**BGT goto2**). Εάν δεν έχουμε κάνει branch, συνεχίζουμε στο block whileLoop που σημαίνει ότι το περιεχόμενο του καταχωρητή r0 βρίσκεται στο διάστημα [65, 90]. Στην συνέχεια, εκτελούμε την εντολή **SUB r5, r0, #65**, η οποία αφαιρεί από την τιμή ***(string + i)** το **65** και το αποθηκεύει στον καταχωρητή r5. Έπειτα εκτελείτε η εντολή **LDR r5, [r4, r5, LSL #2]** η οποία φορτώνει στον καταχωρητή r5 την διεύθυνση r4+r5, απλά χρησιμοποιώντας την εντολή **LSL #2** κάνουμε shift left κατά 2, δηλαδή πολλαπλασιασμό με το 4 ώστε να προχωρήσουμε 4 bytes δεδομένου ότι ο καταχωρητής r5 έχει αποθηκευμένη μια μεταβλητή τύπου **integer**. Τέλος, εκτελούμε την **ADD r3, r3, r5** όπου προσθέτουμε την τιμή που βρήκαμε προηγουμένως και αποθηκεύσαμε στον καταχωρητή r5 με την τιμή του καταχωρητή r3 (που είναι η μεταβλητή **sum**) και την αποθηκεύουμε στον r3.

- goto2:

Στο block goto2 εκτελούμε την σειρά εντολών **CMP r0, #48 -> BLT goto3 -> CMP r0, #57 -> BGT goto3** έτσι ώστε να ελέγξουμε εάν η τιμή του καταχωρητή r0 (δηλαδή η τιμή ***(string + i)**) είναι στο διάστημα [48, 57]. Αν όχι, τότε κάνουμε branch στο block goto3. Αν ναι, συνεχίζουμε εκτελώντας την εντολή **SUB r5, r0, #48** όπου αφαιρούμαι από τον καταχωρητή r0 την τιμή 48 και αποθηκεύουμε το αποτέλεσμα στον καταχωρητή r5 και στην συνέχεια εκτελούμε την **SUBS r3, r3, r5** όπου αφαιρούμε από τον καταχωρητή r3 την τιμή του καταχωρητή r5 και το αποθηκεύουμε στον καταχωρητή r3 (**sum**).

- goto3:

Στο block goto3 εκτελούμε την εντολή **ADDS r1, r1, #1** (δηλαδή i++) έτσι ώστε να πάμε στο επόμενο γράμμα του αλφαριθμητικού και τέλος εκτελείτε η ψευδο-εντολή **NOP** η οποία δεν κάνει τίποτα παρά μόνο να δείξει ότι

βρισκόμαστε στο τέλος της **while**.

Τέλος, εάν στο block goto1 η εντολή **BNE whileLoop** δεν κάνει **branch** βγαίνουμε από την while και μεταφέρουμε το περιεχόμενο του καταχωρητή r3 στον καταχωρητή r0 (**MOV r0, r3**), έτσι ώστε η return value της ρουτίνας να είναι σωστά αποθηκευμένη στον καταχωρητή r0 και τέλος κάνουμε **POP {r4, r5, pc}** ώστε να κάνουμε deallocate του καταχωρητές r4 και r5 από το activation record και να δώσουμε την τιμή του **link register** που κρατάει την return address στον **Program Counter** (PC) και να επιστρέψουμε στην **main()**.

4. <u>Debug & Testing</u>

Όσον αφορά το debug, έγινε χρήση του **default compiler version 5** καθώς και του **Simulation Mode** όπως φαίνεται στις φωτογραφίες παρακάτω. Ενώ όσον αφορά το testing, όπως είπαμε δημιουργήσαμε μια μεταβλητή **actualSum** η οποία έχει την τιμή που υπολογίσαμε «στο χέρι» και έπειτα στην main() την συγκρίνουμε με αυτή που επέστρεψε η ρουτίνα μας. Εάν η ρουτίνα δουλεύει σωστά, η μεταβλητή **validation** παίρνει την τιμή 1, όπως φαίνεται στην φωτογραφία παρακάτω.