2025

ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΕΠΕΞΕΡΓΑΣΤΩΝ ΚΑΙ ΠΕΡΙΦΕΡΕΙΑΚΩΝ

ΑΝΑΦΟΡΑ 1ΟΥ ΕΡΓΑΣΤΗΡΙΟΥ

ΠΑΝΑΓΙΩΤΗΣ ΠΑΠΑΔΟΠΟΥΛΟΣ | ΑΛΕΞΑΝΔΡΟΣ ΚΑΡΓΙΩΤΗΣ

10697 | 10662

Εισαγωγή

Για το πρώτο εργαστήριο ζητήθηκε:

1. Μία ρουτίνα σε assembly που θα υπολογίζει το hash του string, θα αποθηκεύει την τιμή του σε μια θέση μνήμης και θα την επιστρέφει στην main.

2. Μία ρουτίνα σε assembly που θα υπολογίζει το άθροισμα των ψηφίων από το hash και στη συνέχεια το mod7 του, ενώ στη συνέχεια θα αποθηκεύει την τιμή του σε μια θέση μνήμης και θα την επιστρέφει στην main.

3. Μία ρουτίνα σε assembly που θα υπολογίζει το αποτέλεσμα της συνάρτησης Fibonacci, ενώ στη συνέχεια θα αποθηκεύει την τιμή του σε μια θέση μνήμης και θα την επιστρέφει στην main.

4. Μια έξτρα ρουτίνα σε assembly, η οποία για το δοθέν αλφαριθμητικό θα υπολογίζει το bitwise XOR όλων των χαρακτήρων του string και θα το αποθηκεύει σε ξεχωριστή θέση μνήμης.

Επιπλέον χρησιμοποιήθηκε ο UART κώδικας που δόθηκε για την ανάγνωση των string.

Ανάλυση Κώδικα

Main.c

Ο κωδικας στη main καλεί τις συναρτήσεις και κάνει initialize δεδομένα και πίνακες όπου είναι απαραίτητο. Δόθηκε έμφαση στην χρήση μικρών σε μέγεθος μεταβλητών όπου είναι δυνατό. Εκτυπώνεται τόσο το string που επιλέγει αρχικά ο χρήστης όσο και η τιμή του hash μέσα από τα διαδοχικά στάδια. Το checksum εκτυπώνεται σε μορφή 16δικου αριθμού για ευκολία.

```
    ////////////////////////////////////////////////////////////////////////////// First part
    printf("//////////////////////////////////////////////////////////////////////////\n");
    printf("String input: %s\n", buff);
    // map of hash values of each number 0-9
    const uint8_t digit_values[10] = {5, 12, 7, 6, 4, 11, 6, 3, 10, 23};

    int init_result;
    hash_init(buff_index, buff, digit_values, &init_result);  // buff_index = len(buff) + ('\0')

    printf("Initial Hash Value:  %d\n", init_result );
    printf("Buffer size: %d\n", buff_index);
    ////////////////////////////////////////////////////////////////////////////// Second part
    uint8_t interm_result;
    hash_interm(init_result, &interm_result);

    printf("Intermediate Hash Value:  %d\n", interm_result );
    ////////////////////////////////////////////////////////////////////////////// Third part
    int hash_final = fibonacci(interm_result);
    printf("Final Hash Value: %d\n", hash_final);
    ////////////////////////////////////////////////////////////////////////////// Bonus part
    uint8_t checksum;
    xor_checksum(buff_index, buff, &checksum);

    printf("Xor checksum: %x\n", checksum);
  }
  return 0;
}
```

Εικόνα 1. Ο κώδικας που γράφτηκε στην main

Λόγω της διατύπωσης των ασκήσεων, επιλέχθηκε υλοποίηση για τις ρουτίνες που δεν επιστρέφει τιμές. Αντ' αυτού, οι τιμές γίνονται pass by reference στην εκάστοτε παράμετρο.

```
void hash_init(uint8_t word_size, const char *word, const uint8_t *digit_values, int* result);
void hash_interm(int num, uint8_t *result);
int fibonacci(int n);
void xor_checksum(uint8_t word_size, const char *word, uint8_t *checksum);
```

Εικόνα 2. Function declarations.

Hash_init.s

Σε αυτήν τη ρουτίνα ζητούνται τα ακόλουθα:

a) **Η αρχική τιμή του hash να είναι ίση με το μήκος του string.**
b) Για κάθε **κεφαλαίο** λατινικό γράμμα, προσθέστε στο hash τον **ASCII κωδικό του γράμματος πολλαπλασιασμένο με 2**.
c) Για κάθε **μικρό** λατινικό γράμμα, προσθέστε στο hash **το τετράγωνο της διαφοράς του ASCII κωδικού του γράμματος από το 'a' (97)**.
d) Για κάθε **αριθμητικό ψηφίο (0-9)**, προσθέστε στο hash μία τιμή που αντιστοιχεί σε κάθε ψηφίο **σύμφωνα με τον παρακάτω πίνακα**:

Ψηφίο	Τιμή
0	5
1	12
2	7
3	6
4	4
5	11
6	6
7	3
8	10
9	23

Η αντιστοίχιση να υλοποιηθεί μέσω πίνακα που θα αποθηκεύσετε στη μνήμη και θα προσπελαύνετε από την assembly ρουτίνα σας.

e) Για οποιοδήποτε άλλο χαρακτήρα, **αγνοείται**.

Εικόνα 3. Λειτουργία hash_init.s

Το r6, που περιέχει την τιμή του κάθε χαρακτήρα word[i] ελέγχεται διαδοχικά. Η τιμή του συγκρίνεται μέσω του CMP r6,#(ASCII_VAL) με τις ASCII τιμές των πρώτων και τελευταίων χαρακτήρων των αριθμών, κεφαλαίων γραμμάτων και πεζών γραμμάτων, και αντίστοιχα με BGT κ BLT ελέγχεται τα flags για να γίνει το κατάλληλο branching αν χρειαστεί, ώστε να προστεθεί με στο hash_value register r0 η κατάλληλη τιμή . Αν η τιμή του χαρακτήρα δεν ανήκει σε "σωστό" χαρακτήρα, μέσω του B LOOP επιστρέφει στην αρχή του LOOP για την σύγκριση του επόμενου χαρακτήρα. Αν το i ξεπεράσει το word_size, ή ρουτίνα κάνει branch στο label EXIT, όπου μέσω του STR r0,[r3], αποθηκεύεται by reference η τιμή του r0 στην παράμετρο *result.

Ιδιαίτερα για την περίπτωση όπου ο χαρακτήρας είναι αριθμός, τα παρακάτω instructions βρίσκουν την θέση του αριθμού στον πίνακα digit_values[] και εξάγουν μέσω του LDRB την τιμή που πρέπει να προστεθεί στο hash_value. Αυτό που κάνει το SUB, είναι να αφαιρεί από το ASCII value του χαρακτήρα(αριθμού) το ASCII value του χαρακτήρα '0' και να αποθηκεύει το αποτέλεσμα σε έναν register(index) r7, που θα λειτουργεί σαν offset.

```
//CREATE INDEX
SUB r7,r6,#48          // Offset for digit_values
//LOAD digital_values[]
LDRB r7,[r2,r7]        // r7 = *(&digit_values + offset)
//ADD TO HASH
ADD r0,r0,r7           // hash += digit_values[word[i]] (word[i]--> the actual number, not ASCII value)
```

Hash_interm.s

Σε αυτήν την ρουτίνα ζητούνται τα ακόλουθα:

Στη συνέχεια, αν το hash είναι μεγαλύτερο από 9,:

f) Να **αθροίσετε τα νούμερα του hash**.
g) Να **διαιρέσετε το διαδοχικά με το 7 και πάρτε το υπόλοιπο (mod 7)** μέχρι να καταλήξετε σε μονοψήφιο αριθμό.

Η άθροιση έγινε βρίσκοντας το mod10 του αρχικού hash και διαιρώντας με το 10, επαναλαμβάνοντας μέχρι η (ακεραια) διαίρεση να είναι ίση με το 0. Αθροίζοντας τα διαδοχικά mod7 παίρνουμε το ζητούμενο του f).

Η υλοποίηση του mod, ή οποία χρησιμοποιήθηκε και για το g έχει ως εξής:

- Ακέραια διαίρεση του αριθμου με το mod που θελουμε να βρουμε με την χρηση UDIV.
- Πολλαπλασιασμος του αποτελέσματος με το mod που θελουμε να βρουμε
- Αφαίρεση του τελικού αριθμού από τον αρχικό.

<u>Παραδειγμα</u>:

Hash_initial = 198, ψαχνω να βρω το mod10.

198/10 = 19 (ακεραιος)

19*10 = 190

198 – 190 = 8

Συνεπώς 198%10=8

Παρακάτω φαινεται ο κώδικας για την αθροιση των αριθμών του hash και του τελικού mod7:

```
LOOP:
CMP r0,#0
BLE EXIT                    // if num<=0, stop looping

MOV r6,#10
UDIV r2,r0,r6               // r2 = r0/10
MUL r3,r2,r6                // r3 = r2*10, this way r3=num-(num%10)
SUB r5,r0,r3                // r5 = num%10
ADD r4,r4,r5                // sum = sum + num%10
UDIV r0,r0,r6               // num = num / 10
B LOOP



EXIT:
MOV r6,#7
UDIV r5,r4,r6               //(int)r5=sum/10
MUL r5,r5,r6                // r5=r5*10, this way r5=sum-(sum%7)
SUB r4,r4,r5                // r4 = sum%7
```

Hash_interm.s

Βασικός προβληματισμός ήταν ο τρόπος που θα υλοποιούσαμε το recursion. Αυτό επετεύχθη με την χρήση του BL για το branching στο κώδικα Fibonacci κρατώντας το return address register.

```
    fibonacci:
// r0 = n
    PUSH {lr}       // Push Link register to stack

    B0:
    CMP r0,#0
    BNE B1          // If r0 = n != 0, goto B1
    MOV r0,#0       // If n = 0 ---> r0=0, will be returned on function end
    POP {pc}

    B1:
    CMP r0,#1
    BNE B2          // Else if n != 1, go to B2
    MOV r0,#1       // If n = 1 ---> r0=1, will be return on function end
    POP {pc}

    B2:
    PUSH {r4,r5}
    MOV r4,r0       // Save current n before recursion, to use it in fibonacci(n-2)
    SUB r0,r0,#1    // r0 <--- n = n - 1
    BL fibonacci    // fibonacci(n-1)
    MOV r5,r0       // move result of fibonacci(n-1) return value to r5
    MOV r0,r4       // Get current n from stack, store it in r0
    SUB r0,r0,#2    // ro <--- n = n - 2
    BL fibonacci    // fibonacci (n-2)
    ADD r0,r0,r5    // r0 <--- fibonacci(n-1)+fibonacci(n-2)
    POP{r4,r5,pc}
.end
```

Xor_checksum.s

Η όλη υλοποίηση βασίστηκε στο instruction EOR, όπου ακολουθιακά καναμε load το byte κάθε χαρακτήρα του string, εφαρμόζαμε το EOR και το προσθέταμε στον ίδιο register.

```
    xor_checksum:
// r0 = word_size
// r1 = &word[0]
// r2 = &checksum
    PUSH{r4,r5,lr}
    MOV r3,#0           // r3 <--- value of checksum

    MOV r4,#0           // r4 <--- i

    LOOP:
    CMP r4,r0
    BGE EXIT            // If i>= word_size goto EXIT

    LDRB r5,[r1,r4]     // r5 <--- word[i] = *(&word + i)
    ADD r4,r4,#1        // i = i + 1
    EOR r3,r3,r5        // checksum = checksum^word[i]
    B LOOP

    EXIT:
    STRB r3,[r2]
    POP {r4,r5,pc}
.end
```

Προβλήματα

- Αρχικά προσπαθήσαμε να γράψουμε την assembly με inline assembly syntax, στο αρχείο main.c. Δυστυχώς το version 6 του arm compiler κατήργησε αυτήν την δυνατότητα και το documentation της ARM δεν ήταν πολύ ξεκάθαρο.
- Ήταν η πρώτη φορά που γράφαμε assembly έτσι δεν γνωρίζαμε τι είναι τα assembly directives και πως να τα χρησιμοποιήσουμε.
- Υπήρξε προβληματισμός με την διατύπωση ” θα αποθηκεύει την τιμή του σε μια θεση μνήμης και θα την επιστρέφει στην main”. Αποφασίσαμε πως αυτό σημαίνει πως η συνάρτηση θα είναι void, και το αποτέλεσμα που ψάχνουμε θα δίνεται by reference.

Testing

Παρακάτω παραθέτονται μερικά από τα strings που ελέγχθηκαν:

String	Buffer size (String size + 1)	Hash_init result	Hash_interm result	Fibonacci result	XOR Checksum Result
“A9b3”	5	164	4	3	0x29
“”(Empty)	1	0	0	0	0x0
“zzzzzzz….” (128 characters)	128	79502	2	1	0x7a
“A9b3 “	6	165	5	5	0x9
“~”	2	1	1	1	0x7e

Έγινε έλεγχος του παραδείγματος που δίνεται όπως και ακραίων περιπτώσεων, για να διαπιστωθεί η ορθή λειτουργία κάθε ρουτίνας.