Εργασία 1 Μικροεπεξεργαστές

Βογιατζής Χαρίσιος
AEM:9192

April 9, 2025

Github Source Code

1 Εισαγωγή

Ο στόχος της εργασίας είναι να δημιουργήσουμε μία ρουτίνα σε ARM assembly η οποία θα δέχεται ένα αλφαριθμητικό και θα υπολογίζει ένα custom hash σύμφωνα με τις προδιαγραφές που δόθηκαν.

2 hash.s

Η συνάρτηση αυτή υλοποιεί το πρώτο κομμάτι της hashing διαδικασίας.
Αρχικά υπολογίζει υπολογίζει το μήκος του αλφαριθμητικού στον πρώτο βρόχο και θέτει έτσι την τιμή του hash.
Στο δεύτερο βρόχο προσθέτει μία κατάλληλη τιμή ανάλογα με τις ακόλουθες περιπτώσεις κεφαλαίο, μικρό, ψηφίο, άλλο.
Η συνάρτηση χρησιμοποιεί left shift για πολλαπλασιασμό με το 2, ένα lookup table και multiply and accumulate για τον υπολογισμό τετραγώνου αριθμού.
Επίσης δημιουργεί την global μεταβλητή hash_result στην οποία αποθηκεύει την υπολογιζόμενη τιμή.
Οι δύο βρόχοι θα μπορούσαν να ενωθούν σε ένα χωρίς βλάβη του αλγορίθμου και με επιτάχυνση του, αλλά επιλέχθηκε να παραμείνουν χωριστοί για καλύτερη ανάγνωση και αντιστοιχία με τα ερωτήματα της εργασίας.
Ο αλγόριθμος αυτός είναι order-invariant, συνεπώς αλφαριθμητικά που αποτελούνται από τους ίδιους χαρακτήρες σε διαφορετικές θέσεις παράγουν το ίδιο hash.
Επίσης οι άλλοι χαρακτήρες δεν συνεισφέρουν στο hash (πέρα από την αλλαγή στο μήκος του αλφαριθμητικού).

3 modhash.s

Η συνάρτηση αυτή υλοποιεί το δεύτερο κομμάτι της hashing διαδικασίας.
Δουλεύει με δεδομένο το υπολογισμένο hash του προηγούμενου βήματος.
Εαν το hash είναι διψήφιο (¿10) τότε προσθέτει τα ψηφία του hash και στη συνέχεια υπολογίζει το modulo 7.
Για τον υπολογισμού του αθροίσματος των ψηφίων ακολουθούμε την παρακάτω διαδικασία.
Αρχικά διαιρούμε με το 10 για να αποκόψουμε το τελευταίο ψηφίο.
Στη συνέχεια το πολλαπλασιάζουμε με το 10 και αφαιρούμε το αποτέλεσμα από τον αρχικό αριθμό με αποτέλεσμα να έχουμε απομονώσει μόνο το τελευταίο ψηφίο το οποίο προσθέτουμε στο άθροισμα.
Η διαδικασία αυτή εκτελείται επαναλληπτικά έως να επεξεργασθούν όλα τα ψηφία.
Στη συνέχεια με παρόμοιο τρόπο υπολογίζουμε το mod7, διαιρώντας δηλαδή αρχικά με το 7 και στην συνέχεια αφαιρώντας το αποτέλεσμα επί 7 από τον αρχικό αριθμό.
Τέλος, αποθηκεύει το αποτέλεσμα στη global hash_result.

4 fibonacci.s

Η συνάρτηση αυτή υλοποιεί το τρίτο κομμάτι της hashing διαδικασίας.
Αποτελεί μια αναδρομική υλοποίηση του fibonacci.
Λόγω της αναδρομικής φύσης, στη συνάρτηση αυτή δόθηκε ιδιαίτερη έμφαση στη διαχείριση του stack και περιορίστηκε ή χρήση των non-scratch καταχωρητών στους R4 και R5 τους οποίους σώζουμε και επαναφέρουμε μεταξύ των αναδρομικών κλήσεων.
Δημιουργεί μία unitiliazed μεταβλητή fib_result στο stack χρησιμοποιώντας το section .bss, δεσμεύοντας έτσι 4 bytes μόνο μία φορά και αποθηκεύει εκεί τα αποτελέσματα των υπολογισμών.

Επεξήγηση: Πριν από την ADD, το αποτέλεσμα της fib(n-1) αποθηκεύεται στον R5.
Η δεύτερη κλήση υπολογίζει το fib(n-2) και το αποτέλεσμα καταλήγει στον R0.
Όταν και τα 2 αποτελέσματα είναι έτοιμα (fib(n-1) στον R5 και fib(n-2) στον R0) η ADD τους προσθέτει και υπολογίζει το fib(n).

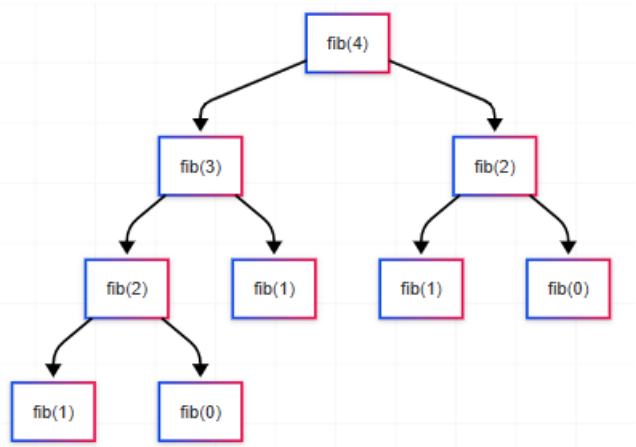
Ο R4 αποθηκεύει το αρχικό n μεταξύ αναδρομικών κλήσεων.
Ο R5 αποθηκεύει προσωρινά το fib(n-1) όσο υπολογίζουμε το fib(n-2).

Κάθε κλήση επιστρέφει το αποτέλεσμα στον R0, συνεπώς μετά την ADD ο R0 είναι έτοιμος για τον caller.

Με αυτό τον τρόπο κάθε αναδρομικό επίπεδο περιμένει να ολοκληρωθούν τα 2 παιδιά του και τα προσθέτει μία φορά, επιστρέφοντας τον αποτέλεσμα στο πάνω επίπεδο.

Έτσι χρειαζόμαστε/χρησιμοποιούμε μία ADD.

Παρακάτω φαίνεται το Call Stack Diagram για fib(4).



Η βηματική εκτέλεση του αλγορίθμου.

Algorithm 1: Step-by-step execution flow of `fib(4)`

Input: Input: `R0 = 4`
Output: Output: `R0 = 3`

1. `fib(4)`: `R4 = 4` (save `n`) Call `fib(3)` → Returns `R0 = 2` `R5 = 2` (save `fib(n-1)`) Call `fib(2)` → Returns `R0 = 1` `R0 = R5 + R0 = 2 + 1 = 3` (final result);
2. `fib(3)`: `R4 = 3` Call `fib(2)` → Returns `R0 = 1` `R5 = 1` Call `fib(1)` → Returns `R0 = 1` `R0 = 1 + 1 = 2`;
3. `fib(2)`: `R4 = 2` Call `fib(1)` → Returns `R0 = 1` `R5 = 1` Call `fib(0)` → Returns `R0 = 0` `R0 = 1 + 0 = 1`;
4. Base Cases: `fib(1)` → Returns `R0 = 1` `fib(0)` → Returns `R0 = 0`;

Πίνακας με τα Register States

Call	R0 (n)	R4 (saved n)	R5 (fib(n-1))	R0 (return val)	Note
fib(4)	4	4	-	-	Enter fib(4)
fib(3)	3	3	-	-	Call fib(3)
fib(2)	2	2	-	-	Call fib(2)
fib(1)	1	-	-	1	Base case: return 1
fib(0)	0	-	-	0	Base case: return 0
fib(2)	-	2	1	1	fib(2) = 1 + 0
fib(1)	1	-	-	1	Base case: return 1
fib(3)	-	3	1	2	fib(3) = 1 + 1
fib(2)	2	2	-	-	Call fib(2)
fib(1)	1	-	-	1	Base case: return 1
fib(0)	0	-	-	0	Base case: return 0
fib(2)	-	2	1	1	fib(2) = 1 + 0
fib(4)	-	4	2	3	fib(4) = 2 + 1

5 crc.s

Το προαιρετικό αυτό κομμάτι δημιουργεί ένα CRC-like checksum υπολογίζοντας το bitwise XOR όλων των χαρακτήρων του αλφαριθμητικού.
Αυτό το checksum είναι order sensitive και δεν αγνοεί τους άλλους χαρακτήρες όπως η προηγούμενη hash, το οποίο επιβεβαιώσαμε και στο testing.

6 Testing

Για τον έλεγχο της ορθής λειτουργίας των παραπάνω συναρτήσεων έχουν δημιουργηθεί οι αντίστοιχες reference συναρτήσεις σε C καθώς και οι testing συναρτήσεις που συγκρίνουν τα αποτελέσματα των 2 υλοποιήσεων.
Επιπλέον έχουν επιλεγεί κάποια χαρακτηριστικά αλφαριθμητικά για την επίδειξη των ιδιοτήτων του κάθε αλγορίθμου.

7 UART

Έχει υλοποιηθεί μία συνάρτηση ώστε να λαμβάνεται το αλφαριθμητικό μέσω UART (με χρήση της uart_rx) αλλά δεν έχει δοκιμαστεί.

8 Debugging

Ένα πρόβλημα που αντιμετώπισαμε ήταν στην αναδρομική fibonacci την οποία κατά λάθος καλούσαμε με το hash από το πρώτο step και όχι μετά το modhash με αποτέλεσμα να προσπαθεί να υπολογίσει το fibonacci(164) το οποίο οδηγούσε σε memory error καθώς γέμμιζε το stack.
Για τον εντοπισμό και την επίλυση των προβλημάτων χρησιμοποιήθηκε ο debugger του Keil με break-points, step-by-step analysis και κοιτάζοντας τις τιμές των καταχωρητών.

9 Source Code

Github Source Code