Εργαστήριο 1

Γιώργος Σελιβάνωφ, Λαμπρινός Χατζηιωάννου

April 18, 2024

1 Main C Function

- Θέλουμε να την κάνουμε να δέχεται input
- Να δοκιμάζει το input βάσει των συναρτήσεων μας
- Να δείχνει output

Η τωρινή υλοποίηση λειτουργεί ως εξής:

1. Λαμβάνουμε ένα αλφαριθμητικό (μέχρι 128 χαρακτήρες) ως είσοδο από τον χρήστη με την χρήση UART
2. Εμφανίζουμε το αλφαριθμητικό μέσω printf
3. Παράγουμε το hash του αλφαριθμητικού,
4. Παράγουμε την ψηφιακή ρίζα του hash
5. Παράγουμε το Sum of Natural Numbers της ψηφιακής ρίζας
6. Εμφανίζουμε όλα τα αποτελέσματα στον χρήστη μέσω UART και printf

2 Hashing Function

Παρόλο που φαινόταν δύσκολη κατα την εκφώνηση η συγκεκριμένη λειτουργία αποδείχτηκε αρκετά εύκολη για την υλοποίηση της σε assembly. Ουσιαστικά, υλοποιήθηκε ο αλγόριθμος που εμφανίζεται στο παρακάτω απόσπασμα κώδικα C.

```
int hashfunc(char inputString[]) {
  int values[] = {10, 42, 12, 21, 7, 5, 67, 48, 69, 2, 36, 3, 19,
                  1, 14, 51, 71, 8, 26, 54, 75, 15, 6, 59, 13, 25};
  int hash = 0;

  for (int ind = 0; inputString[ind]!='\0'; ind++)
    {
```

```
      // ASCII: Caps: (64-91), Lower: (96-123), Digits: (47-58)
      if (inputString[ind] > 64 && inputString[ind] < 91) // Meaning caps
        hash += values[inputString[ind] - 65];
      else if (inputString[ind] > 96 && inputString[ind] < 123) // Meaning lowercase
        hash -= values[inputString[ind] - 97];
      else if (inputString[ind] > 47 && inputString[ind] < 58) // Meaning integer
        hash += inputString[ind] - 48;
    }
  return hash;
}
```

3 Digital Root

Για την υλοποίηση της ρουτίνας σε Assembly που θα υπολογίζει την ψηφιακή ρίζα ενός αριθμού, αποφασίσαμε να αντικαταστήσουμε τον αναδρομικό υπολογισμό του αθροίσματος ($68 \rightarrow 6+8 = 14 \rightarrow 1+4 = 5$) με μια εναλλακτική μέθοδο: τον υπολογισμό του υπολοίπου της ευκλείδειας διαίρεσης του αριθμού αυτού με το 9. Οι δύο διαδικασίες είναι ισοδύναμες ($68 \mod 9 = 5$), με μοναδική εξαίρεση την περίπτωση που η ψηφιακή ρίζα του αριθμού είναι 9 και άρα το υπόλοιπό του 0. Μία απλή αντικατάσταση του 0 με το 9 καθιστά τις 2 αυτές διαδικασίες πλήρως ταυτόσημες ως προς το αποτέλεσμά τους. Με την χρήση αυτού του αλγορίθμου εξοικονομούμε αρκετούς κύκλους ρολογιού, καθώς αποφεύγουμε τα branches του αναδρομικού αλγορίθμου.

Όσον αφορά την υλοποίηση της διαίρεσης με το 9 επιλέχθηκε η χρήση του πολλαπλασιασμού με αντίστροφο, παρότι ο **Cortex-M4** διαθέτει έτοιμες εντολές διαίρεσης. Η επιλογή αυτή έγινε διότι οι εντολές DIV είναι ιδιαίτερα ακριβές σε κύκλους ρολογιού (2-12) σε σύγκριση με την UMULL (που απαιτεί μόνο 1). Αφού ο διαιρέτης είναι σταθερός, ο αντίστροφος μπορεί απλώς να οριστεί ως μια σταθερά της ρουτίνας.

Περισσότερες πληροφορίες σχετικά με τον τρόπο υπολογισμού του αντίστροφου και την ορθότητα της μεθόδου μπορούν να βρεθούν στην δημοσίευση με τίτλο "Division by Invariant Integers using Multiplication" των Torbjorn Granlund και Peter L. Montgomery[1]

Η χρήση των `.N` suffix στις εντολές που προηγούνται των εντολών `IT` γίνεται ώστε να κωδικοποιηθούν από τον compiler με 16-bit, γεγονός που σύμφωνα με το documentation της ARM για τον **Cortex-M4** σημαίνει πως η εκτέλεση της εντολής `IT` θα ξοδέψει 0 κύκλους ρολογιού, καθώς θα ενσωματωθεί στην προηγούμενη[2]

4 Sum of Natural Numbers

Για την υλοποίηση της ρουτίνας Sum of Natural Numbers υλοποιήσαμε ένα απλό loop:

Όσο ο αριθμός εισόδου είναι μεγαλύτερος του 0, η ρουτίνα τον προσθέτει στο άθροισμα, τον μειώνει κατά 1 και κάνει branch στην αρχή του loop. Όταν η συνθήκη παύει να ισχύει, η ρουτίνα επιστρέφει το τρέχον άθροισμα.

5 Bibliography

References

[1] T. Granlund and P. L. Montgomery, "Division by Invariant Integers using Multiplication,"

[2] "Documentation – Arm Developer." https://developer.arm.com/documentation/ddi0439/b/CHDDIGAC.