Μικροεπεξεργαστές και Περιφερειακά

Αναφορά Εργαστηρίου 1

Ονοματεπώνυμο	ΑΕΜ	E-mail
Κωνσταντίνος Πράττης	10056	kprattis@ece.auth.gr
Βασιλική Ρίζου	10021	vasilikirn@ece.auth.gr

Εισαγωγή

Η παρούσα αναφορά αποσκοπεί στην παρουσίαση και περιγραφή των 2 ξεχωριστών συναρτήσεων σε assembly, που αναπτύχθηκαν στα πλαίσια του πρώτου εργαστηρίου του μαθήματος του εαρινού εξαμήνου 'Μικροεπεξεργαστές και Περιφερειακά', καθώς και της μεθόδου αξιολόγησης και ελέγχου αυτών. Στόχος αποτέλεσε ο προγραμματισμός σε assembly ενός μικροελεγκτή ARM. Συγκεκριμένα, ζητήθηκε η συγγραφή κώδικα σε assembly για την αντιστοίχιση ενός αλφαριθμητικού-string σε έναν αριθμό-hash.

Για τον σκοπό αυτό υλοποιήθηκαν 2 ρουτίνες. Η πρώτη, η **hash**, υπολογίζει μία τιμή *n* αθροίζοντας τις επιμέρους τιμές των χαρακτήρων ενός *string (S)* βάση ενός *hashing table (A)*, το οποίο αναθέτει σε κάθε πεζό λατινικό χαρακτήρα μία τιμή. Η δεύτερη ρουτίνα, η **drootfactorial**, υπολογίζει την ψηφιακή ρίζα *(dr[n])* του αποτελέσματος της hash και επιστρέφει την τιμή του παραγοντικού αυτής *(dr[n]!)*.

hash.s

Input:	***R0***	Η διεύθυνση του πρώτου χαρακτήρα του string S
	R1	Η διεύθυνση του πρώτου στοιχείου του πίνακα A
Output:	***R0***	Ο ακέραιος hash που αντιστοιχεί στο string S

Η υλοποίησή μας για τη συνάρτηση hash, εκμεταλλεύεται τη διάταξη στον κώδικα *ASCII* των χαρακτήρων ενδιαφέροντος, δηλαδή αυτών που καθορίζουν τα όρια των χαρακτήρων-ψηφίων ('0' [=*48*] – '9' [=*57*]) και αυτών που καθορίζουν τα όρια των χαρακτήρων-πεζών λατινικών γραμμάτων ('a' [=*97*] – 'z' [=*122*]). Έτσι, συγκρίνουμε κάθε χαρακτήρα c του string S με τα '0', '9', 'a' και 'z', με τη σειρά αυτή.

```
while((c = S[i++]) != '\0'){
  c -= 48;  //'0' = 48
  if(c <= 0){
    continue;
  }
  else if(c <= 9){
    res -= c;
    continue;
  }

  c -= 49 ; //('a' - 48) = 49
  if(c < 0){
    continue;
  }
  else if(c < 26)
    res += A[c];
}
```

Βέβαια, θα πρέπει σε περίπτωση που ο χαρακτήρας είναι μεταξύ '0' και '9', να αντιστοιχηθεί στο διάστημα [0 - 9]. Παρόμοια, αν ο χαρακτήρας είναι στο διάστημα ['a' – 'z'], θα πρέπει να αντιστοιχηθεί στο διάστημα [0-25], ώστε να γίνει η προσπέλαση του σωστού στοιχείου του πίνακα A. Ο τρόπος που αντιμετωπίζουμε το πρόβλημα αυτό είναι με την αντικατάσταση των εντολών CMP με το '0' και με το 'a', με εντολές SUBS, δηλαδή ταυτόχρονη σύγκριση και αποθήκευση της διαφοράς. Προσέχουμε όμως να αλλάξουμε κατάλληλα και τις τιμές με τις οποίες γίνονται και οι επόμενες συγκρίσεις (π.χ αντί για c < 'z' [=122] ελέγχουμε αν c < 26). Στο διπλανό σχήμα φαίνεται το αντίστοιχο κομμάτι κώδικα σε C.

drootfactorial.s

Input:	***R0***	Ο ακέραιος hash [n] που αντιστοιχεί στο string S
Output:	***R0***	Ο ακέραιος που αντιστοιχεί στο παραγοντικό της ψηφιακής ρίζας του ακεραίου hash [dr(n)]!

Ο υπολογισμός μονοψηφίου ακεραίου από hash (***π.χ.*** $66 := 6 + 6 = 12 : 1 + 2 = 3$) αποδεικνύεται ότι ισοδυναμεί με την εύρεση της ψηφιακής ρίζας του αριθμού.[1] Εκείνη υπολογίζεται μέσω της σχέσης :

Digital root of an integer n: $$dr[n] = 1 + (n - 1) \bmod 9 = \begin{cases} 0, if\ n = 0 \\ n \bmod 9, if\ 9 \nmid n \\ 9, if\ 9 \mid n \end{cases}$$

Στο προαναφερθέν παράδειγμα θα ήταν: $\mathrm{dr}[66] = 1 + (66 - 1) \mod 9 = 1 + 65 \bmod 9 = 1 + 2 = 3$, όπως ακριβώς και πριν. Εκμεταλλευόμενοι αυτή την ιδιότητα, παρατηρήσαμε το εξής:

Έστω n ο αρχικός ακέραιος, η ταυτότητα της ευκλείδειας διαίρεσης αυτού με το 8 είναι: $n = 8 * q + r$. Όπου q : quotient και r: remainder. Άρα θα ισχύει ότι: $\mathbf{n = (9 - 1) * q + r = 9 * q + r - q \Rightarrow n \equiv 9 * q + r - q \equiv (r - q) mod 9}$. Συνεπώς,

καταλήγουμε να έχουμε τρείς περιπτώσεις:

```
if (n == 0)
  result = 0;
else{
  q = n / 8;
  r = n % 8;
  while ((r - q) < 0){
    n = r - q;
    q = n / 8;
    r = n % 8;
    i++;
  }
  if ((r - q) == 0)
    result = 9;
  else{
    i = i % 2;
    if(i == 0)
      result = 9 - (r - q);
    else
      result = r - q;
  }
}
```

Case	*Resolve*
$r - q > 0$	$r - q \in [1, 7]$, και θα ισχύει: $n\ mod\ 9 = \begin{cases} 9 - (r - q), & i = 0 \\ r - q, & i = 1 \end{cases}$
$r - q = 0$	$n\ mod\ 9 = 0$
$r - q < 0$	θέτουμε $n = \lvert r - q \rvert$ και επαναλαμβάνω την διαδικασία

Η μεταβλητή i, λαμβάνει την τιμή 1 αν κατά την εκτέλεση του αλγορίθμου βρισκόμαστε σε περιττή επανάληψη, διαφορετικά λαμβάνει την τιμή 0. Η παραπάνω λογική, αποτυπώνεται σε κώδικα C, στην διπλανή εικόνα.

Ο λόγος που ακολουθήθηκε η ανωτέρω περιγραφόμενη μέθοδος, είναι για να εξασφαλίσουμε την λειτουργία του κώδικα assembly και σε επεξεργαστές τύπου ARM-M0, οι οποίοι δεν διαθέτουν εντολή DIV. Κατ' αυτόν τον τρόπο εκμεταλλευόμαστε την διαίρεση με το $\mathbf{8 = 2^3}$, η οποία εύκολα μπορεί να πραγματοποιηθεί με LSR R0, #3. Έτσι, παρατηρούνται το πολύ $\boldsymbol{log_8(2^{32} - 1) \approx 11}$ επαναλήψεις της κύριας λούπας.

Σε δεύτερη φάση, εφόσον πλέον έχει υπολογιστεί η $\boldsymbol{dr[n]}$, μέσω της εντολής $\boldsymbol{B\ factorial}$, μεταβαίνουμε στο τελικό κομμάτι, που μέσω της ρουτίνας $\boldsymbol{factorial}$, υπολογίζει το παραγοντικό του μονοψηφίου $\boldsymbol{dr[n]}$. Αρχικά συγκρίνουμε το dr[n] με το μηδέν και αν ισχύει η ισότητα, η μονάδα (#1) φορτώνεται στον R0 και η συνάρτηση επιστρέφει. Σε αντίθετη περίπτωση, αρχικοποιείται ένας καταχωρητής (R4 στη προκειμένη) στην μονάδα, έχοντας τον ρόλο του 'counter' και ένας που διατηρεί το τρέχον αποτέλεσμα (R5). Έως ότου, ο R4 διατηρεί τιμή μικρότερη του dr[n], πολλαπλασιάζεται το τρέχον αποτέλεσμα με τον R4 ($\boldsymbol{MUL\ R5, R5, R4}$), και

[1] Στο σημείο αυτό αξίζει να αναφερθεί, πως έχουμε επιλέξει σε περίπτωση που ο προκύπτων από την hash.s αριθμός n, είναι αρνητικός, να λαμβάνεται ο αντίθετός του μέσω της εντολή $RSBLT\ R0, R0, \#0$ και να συνεχίζει κανονικά η εκτέλεση της ρουτίνας. Στην περίπτωση αυτή, υπολογίζεται επί της ουσίας το: $dr[-n]$!

αυξάνει η τιμή του counter (***ADD R4, R4, #1***). Μόλις ολοκληρωθεί ο υπολογισμός του παραγοντικού, η συνάρτηση επιστρέφει.

Testing method

Για τον έλεγχο της ορθότητας της υλοποίησης, χρησιμοποιήθηκε το *Keil* σε *debug mode simulator*, και με την βοήθεια της *printf*, δοκιμάσαμε διαφορετικές εισόδους στις δύο συναρτήσεις μας, για τις οποίες υπολογίσαμε το αναμενόμενο αποτέλεσμα με το χέρι.

Έτσι, δημιουργήσαμε δύο tests, ένα για τη συνάρτηση hash και ένα για τη συνάρτηση drootfactorial και τα δύο αποτελούμενα από 5 απλές εισόδους, αλλά με την ιδιότητα να εκφράζουν αντιπροσωπευτικό δείγμα του χώρου των εισόδων των συναρτήσεων, περιλαμβανομένων και των *edge cases*. Αναλυτικότερα, τα tests που επιλέξαμε είναι:

Test inputs for hash.s	*Expected Result*
CAPITAL?	0
16/04/23	-16
abcdfghijklmnopqrstuvwxyz	759
// This is Test #04 !!	321
fINAl TEST Aem : 10056, 10021	18

Test inputs for digitalrootfactorial.s	*Expected Result*
0	0! = 1
-16	7! = 5040
759	3! = 6
321	6! = 720
18	9! = 362880

Να σημειωθεί ότι στην *main.c* χρησιμοποιούνται *preprocessor directives* (π.χ. #define DEBUG) ώστε με ένα απλό comment out αυτών, ο κώδικας που εν τέλει θα γίνει load στον μικροεπεξεργαστή να μην περιέχει τα εκτενή κομμάτια κώδικα για το debugging παρά μόνο την απαραίτητη λειτουργικότητα, η οποία απλά δίνει την δυνατότητα εισαγωγής ενός αυθαίρετου string έως και 100 χαρακτήρων μέσω του παραθύρου που παρέχει το Keil. Ακόμα, με την χρήση ξεχωριστών tests, πετυχαίνουμε τον ξεχωριστό έλεγχο των ρουτίνων που υλοποιήσαμε, για τον οποίο αξιοποιήσαμε και την δυνατότητα παρακολούθησης μέσω του simulator του περιεχομένου των καταχωρητών στην τρέχουσα εντολή assembly.

Τέλος, αναφέρουμε ότι στον τελικό κώδικα που γίνεται load στον μικροεπεξεργαστή, αντί των *printf* χρησιμοποιούνται οι *uart_print*, κάτι που ήταν και ζητούμενο της άσκησης.