Οργάνωση και Σχεδίαση Υπολογιστών (HY232)
Χειμερινό Εξάμηνο 2019-2020

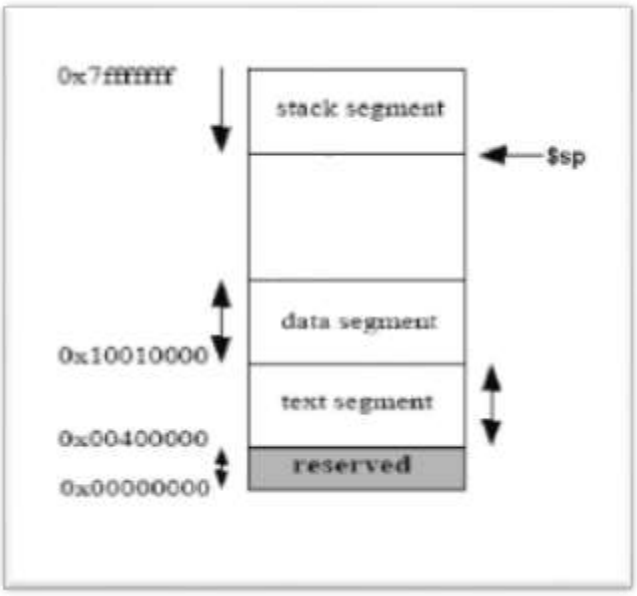
Εργαστήριο 2

Στόχοι του εργαστηρίου

- Χρήση στοίβας
- Συναρτήσεις
- Θεωρητικές Ασκήσεις

Στοίβα

Κατά την εκτέλεση ενός προγράμματος ένα μέρος της μνήμης δεσμεύεται και παραχωρείται για το συγκεκριμένο process. Το κομμάτι αυτό της μνήμης χωρίζεται σε ορισμένους τομείς όπου σε κάθε ένα αποθηκεύονται συγκεκριμένου τύπου δεδομένα. Όπως φαίνεται και στην εικόνα 1 οι τομείς όπου χωρίζεται η μνήμη είναι οι εξής:



Εικόνα 1 (memory segments)

Reserved: Δεσμευμένη από το λειτουργικό περιοχή της μνήμης όπου το πρόγραμμα σας δεν μπορεί να έχει πρόσβαση.

Text Segment: Περιοχή της μνήμης όπου αποθηκεύονται οι εντολές του προγράμματος. Πρόκειται για εντολές σε γλώσσα μηχανής. Όταν ο κώδικας που έχετε γράψει δεν αποτελείται από τις λεγόμενες ψευδοεντολές αλλά από πραγματικές εντολές τότε για κάθε

εντολή που έχετε συντάξει έχει αποθηκευτεί σε αυτό το κομμάτι της μνήμης η αντίστοιχη εντολή συστήματος σε binary μορφή.

Data Segment: Περιοχή της μνήμης όπου αποθηκεύονται τα δεδομένα του προγράμματος. Όλες οι μεταβλητές, οι πίνακες, και γενικά ότι έχετε δηλώσει στην περιοχή .data του κώδικα σας αποθηκεύεται σε αυτό το κομμάτι της μνήμης. Αν αναλύσουμε περισσότερο θα δούμε ότι χωρίζεται σε 2 κομμάτια. Το static data segment όπου αποθηκεύονται τα στατικά δεδομένα (αυτά που δηλώθηκαν στο .data) και το dynamic data segment όπου είναι ο επιπλέον χώρος που παραχωρεί δυναμικά το λειτουργικό.

Stack Segment: Περιοχή της μνήμης που χρησιμοποιείται από την στοίβα. Τα όρια αυτού του τομέα καθορίζονται από τον δείκτη $sp τον οποίο μπορείτε να αλλάζετε κατά την εκτέλεση του προγράμματος. Αντίθετα τα όρια του data segment καθορίζονται από το λειτουργικό και για να δεσμεύσετε επιπλέον μνήμη πρέπει να καλέσετε την αντίστοιχη λειτουργία του συστήματος (εντολή malloc στην C).

Κενό κομμάτι: Ένα κομμάτι μεταξύ του data segment και του stack segment το οποίο αν και δεν περιέχει δεδομένα δεσμεύεται και παραχωρείται από το λειτουργικό στο συγκεκριμένο process. Αν κατά την διάρκεια της εκτέλεσης χρειάζεται να αυξηθεί η περιοχή της στοίβας ή η περιοχή των δεδομένων τότε θα πάρουνε επιπλέον χώρο από αυτό το κομμάτι.

Χρήση της στοίβας

Καταρχάς όταν αναφερόμαστε σε μια στοίβα εννοούμε μια LIFO (Last In First Out) λίστα. Αυτό σημαίνει ότι μπορούμε κάθε φορά να ανακτούμε τα στοιχεία με την ανάποδη σειρά από αυτή που τα αποθηκεύσαμε (πρώτα ανακτούμε τα τελευταία στοιχεία που μπήκαν στην λίστα). Από την εικόνα 1 βλέπετε ότι για να αυξήσουμε τον χώρο που καταλαμβάνει η στοίβα πρέπει να μετακινήσουμε τον δείκτη $sp προς τα κάτω. Για να γίνει αυτό πρέπει να **μειώσουμε** την τιμή του τόσες θέσεις όσα και τα επιπλέον bytes που θέλουμε να αυξηθεί η στοίβα. Επειδή κατά κανόνα στην στοίβα σώζουμε τα περιεχόμενα καταχωρητών, οι οποίοι έχουν μέγεθος 4 bytes, ο επιπλέον χώρος στην στοίβα πρέπει να εξασφαλίσουμε ότι θα είναι πολλαπλάσιο του 4 χωρίς όμως αυτό να είναι δεσμευτικό.

Γιατί να χρησιμοποιήσω την στοίβα;

Κατά την εκτέλεση ενός προγράμματος οι καταχωρητές είναι όλοι τους ορατοί από οποιαδήποτε σημείο του κώδικα, ακόμα και από τις διάφορες συναρτήσεις. Αυτό όμως μπορεί να δημιουργήσει πρόβλημα κατά την εκτέλεση ενός προγράμματος. Όταν ένας συγκεκριμένος καταχωρητής χρησιμοποιείται για εγγραφή μέσα σε μια συνάρτηση (ας την πούμε Α) και παράλληλα περιέχει δεδομένα απαραίτητα για μια άλλη συνάρτηση(ας την πούμε Β) η οποία σε κάποιο σημείο της εκτέλεσης της κάλεσε την Α τότε τα δεδομένα αυτού του καταχωρητή χάνονται για την συνάρτηση Β.
Για να καταλάβετε καλύτερα το πρόβλημα που δημιουργείται ας δούμε το παρακάτω παράδειγμα:

```
Func1:

…
```

```
li $s0,50          # εντολή No1: Στον καταχωρητή $s0 αποθηκεύονται δεδομένα
                   # απαραίτητα για την Func1. Έστω η τιμή 50.
…

jal      Func2           # κλήση της Func2. Η εκτέλεση μεταπηδά στην Func2:

…

move $a0, $s0   # Πρόβλημα. Τα δεδομένα στον $s0 δεν είναι αυτά της
              # εντολής No1. Η συνάρτηση περίμενε να βρει 50.
…

Func2:             #συνάρτηση func2

…

addi $s0, $zero,100       #Στον καταχωρητή $s0 αποθηκεύεται η τιμή 100.
                        #Οτι δεδομένα υπήρχαν  πριν μέσα χάθηκαν.
…

jr $ra
```

Στο παραπάνω παράδειγμα βλέπουμε το πρόβλημα που δημιουργείται με τον καταχωρητή $s0. Τόσο η func1 όσο και η func2 χρησιμοποιούν αυτόν τον καταχωρητή. Το πρόβλημα δημιουργείται στην func1 η οποία μετά την κλήση της func2 έχασε τα περιεχόμενα που είχε στον $s0.
Θα μπορούσε να ισχυριστεί κανείς ότι σε τέτοιες περιπτώσεις προσέχοντας ποιους καταχωρητές θα χρησιμοποιήσουμε μπορούμε να αποφύγουμε τέτοιες δυσάρεστες παρενέργεια. Τι γίνεται όμως όταν καλείστε να υλοποιήσετε μια συνάρτηση την οποία αργότερα θα χρησιμοποιήσει κάποιος άλλος προγραμματιστής; Ή επίσης τι γίνεται αν χρειασθεί να υλοποιήσετε μεγάλους σε έκταση κώδικες όπου οι κλήσεις συναρτήσεων είναι πολύ συχνές;

Πως λύνεται το πρόβλημα με χρήση στοίβας.

Η διαδικασία που πρέπει να ακολουθήσετε για να αποθηκεύσετε κάτι στην στοίβα είναι η εξής:

1. Μειώνεται των δείκτη $sp κατά τόσες θέσεις ανάλογα με το πλήθος των καταχωρητών που θέλετε να αποθηκεύσετε. Σε κάθε καταχωρητή αντιστοιχούν 4 bytes. Έστω ότι δεσμεύεται χώρο για N words. Μειώνετε κατά N*4 bytes.
2. Κάνετε store με την εντολή sw τα περιεχόμενα των καταχωρητών που θα χρησιμοποιήσετε στις θέσεις μνήμης από εκεί που δείχνει ο $sp μέχρι συν N*4 θέσεις.
3. Χρησιμοποιείτε κανονικά μέσα στην συνάρτηση τους καταχωρητές.
4. Κάνετε load με την εντολή lw τα αρχικά περιεχόμενα των καταχωρητών αντίστοιχα όπως τα κάνατε store στο βήμα 2
5. Αυξάνετε τον $sp κατά N*4 θέσεις ώστε να επανέλθει στη θέση που ήταν πριν το βήμα 1.
6. Η συνάρτηση επιστρέφει.

Παράδειγμα χρήσης της στοίβας

Παρακάτω ακολουθεί η υλοποίηση μιας συνάρτησης η οποία σώζει ότι είναι απαραίτητο στην στοίβα.

Παράδειγμα:

```
func1:
addi $sp,$sp,-12          #δεσμεύουμε θέσεις για τρεις λέξεις στην στοίβα.
sw $ra,0($sp)             #κάνουμε push στην στοίβα τα περιεχόμενα του $ra
sw $s1,4($sp)             #κάνουμε push στην στοίβα τα περιεχόμενα του $s1
sw $s2,8($sp)             #κάνουμε push στην στοίβα τα περιεχόμενα του $s2

…

addi $s1, $0,10  #αλλάζουμε χωρίς περιορισμούς τα περιεχόμενα των 2
move $s2, $s1    #καταχωρητών που κάναμε push

…

jal func2                 #μπορούμε να καλέσουμε χωρίς πρόβλημα και άλλες
            # συναρτήσεις
…

jal func1                 #μπορούμε να καλέσουμε ακόμα και την ίδια.
                          #συνάρτηση(αναδρομικά) χωρίς κανένα πρόβλημα
…

lw $ra, 0($sp)    #κάνουμε pop από την στοίβα τα περιεχόμενα του $ra
lw $s1,4($sp)     #κάνουμε pop από την στοίβα τα περιεχόμενα του $s1
lw $s2,8($sp)     #κάνουμε pop από την στοίβα τα περιεχόμενα του $s2
addi $sp, $sp,12#επαναφέρουμε τον stack pointer
jr $ra
```

Στον επεξεργαστή MIPS υπάρχει ένα μεγάλο πλήθος καταχωρητών οι περισσότεροι από τους οποίους έχουν και συγκεκριμένη χρήση. Οι καταχωρητές $s0-$s7 και $t0-$t9 ονομάζονται γενικής χρήσης και χρησιμοποιούνται συχνότερα κυρίως για πράξεις μεταξύ καταχωρητών. Μπορείτε να δείτε από τον πίνακα 1 ο οποίος υπάρχει και στο Instructions Set το πως πρέπει να μεταχειρίζεστε τον κάθε καταχωρητή ανάλογα με την ομάδα που ανήκει.

Register	Number	Usage
zero	0	Constant 0
at	1	Reserved for assembler
v0	2	Used for return values from function calls.
v1	3	
a0	4	Used to pass arguments to procedures and functions.
a1	5	
a2	6	
a3	7	
t0	8	Temporary (Caller-saved, need not be saved by called procedure)
t1	9	
t2	10	
t3	11	
t4	12	
t5	13	
t6	14	
t7	15	
s0	16	Saved temporary (Callee-saved, called procedure must save and restore)
s1	17	
s2	18	
s3	19	
s4	20	
s5	21	
s6	22	
s7	23	
t8	24	Temporary (Caller-saved, need not be saved by called procedure)
t9	25	
k0	26	Reserved for OS kernel
k1	27	
gp	28	Pointer to global area
sp	29	Stack pointer
fp	30	Frame pointer
ra	31	Return address for function calls.

Πίνακας 1

Οι καταχωρητές $t0 έως $t9 ονομάζονται Temporary ή Caller-saved. Αυτό μας λέει ότι μια συνάρτηση δεν είναι υποχρεωμένη να κρατήσει το περιεχόμενο των καταχωρητών ίδιο όταν επιστρέψει. Αν ο Caller, που μπορεί να είναι μια άλλη συνάρτηση ή η main, θέλει να έχει το περιεχόμενο αυτών τον καταχωρητών και μετά την κλήση κάποιον συναρτήσεων οφείλει να τα σώσει ο ίδιος στην στοίβα.

Οι καταχωρητές $s0 $s7 ονομάζονται Saved Temporary ή Callee-Saved(ο καλούμενος σώζει). Αυτό σημαίνει ότι μια συνάρτηση οφείλει να κρατήσει σταθερό το περιεχόμενο αυτών των καταχωρητών με αυτό που είχαν πριν. Για να γίνει αυτό πρέπει να σώσει στην αρχή το περιεχόμενο τους στην στοίβα και να το επαναφέρει μόλις τελειώσει. Αυτό ακριβώς γίνεται και στο παράδειγμα.

Για το επόμενο εργαστήριο έχετε να υλοποιήσετε τις παρακάτω ασκήσεις. Την ώρα του εργαστηρίου θα εξετασθείτε προφορικά πάνω στους κώδικες που θα παραδώσετε.

Άσκηση 1– Θεωρητικές Ασκήσεις (5+5+5 = 15 μονάδες)

Οι παρακάτω ασκήσεις είναι θεωρητικές και πρέπει να παραδοθούν γραμμένες στο χαρτί κατά την διάρκεια της εξέτασης του εργαστηρίου. Κατά την διάρκεια της εξέτασης είναι πιθανόν να σας κάνουμε ερωτήσεις επιπλέον αλλά σχετικές με τις θεωρητικές ασκήσεις για να διαπιστώσουμε εάν καταλάβατε την λύση που δώσατε.

Ψευδοεντολές. Χρησιμοποιώντας αποκλειστικά και μόνο πραγματικές εντολές MIPS, να υλοποιήσετε τις παρακάτω ψευδοεντολές.

1) Την ψευδοεντολή *bner rs, rt, rd* έμμεσης διακλάδωσης μέσω καταχωρητή (Indirect Register Conditional Branch). Εάν *rs* και *rt* δεν είναι ίσια, πήγαινε στην διεύθυνση που είναι αποθηκευμένη στον *rd.*

2) Την ψευδοεντολή *interleave rd, rs, rt* η οποία συγχωνεύει τα περιεχόμενα δύο καταχωρητών *rs,* και *rt* έτσι ώστε:

   a. τα bits του καταχωρητή *rs* που βρίσκονται σε άρτιες θέσεις, πχ. bits 0, 2, 4, .., 30, αποθηκεύονται στις αντίστοιχες θέσεις bit του καταχωρητή *rd.*

   b. τα bits του καταχωρητή *rt* που βρίσκονται σε περιττές θέσεις, πχ. bits 1, 3, 5, .., 31, αποθηκεύονται στις αντίστοιχες θέσεις bit του καταχωρητή *rd.*

   Για παράδειγμα, εάν $s0 = 0xA3476678 και $s1 = 0xBB110011, τότε η εντολή *interleave $s2, $s0, $s1* παράγει το στο LSB του $s2 το αποτέλεσμα *$s2 = 0x0……50.*

Κωδικοποίηση Εντολών. Η επόμενη ερώτηση αναφέρεται στην κωδικοποίηση συνόλου εντολών.

3) Θεωρείστε την περίπτωση ενός επεξεργαστή με 32 καταχωρητές στον οποίο το μήκος όλων των εντολών είναι 12 bits. Μπορούμε να έχουμε ταυτόχρονα τις παρακάτω κωδικοποιήσεις εντολών;
   - 3 εντολές με διευθυνσιοδότηση 2 καταχωρητών, και
   - 31 εντολές με διευθυνσιοδότηση 1 καταχωρητή, και
   - 35 εντολές χωρίς διευθυνσιοδότηση καταχωρητών

   Μην απαντάτε με ένα απλό ναι ή όχι; εξηγήστε λεπτομερώς την απάντησή σας και δώστε τα διαφορετικά formats των εντολών καθώς και το μήκος του κάθε πεδίου για κάθε διαφορετικό format.

Απόδοση Συστήματος. Θεωρείστε δύο διαφορετικές μικρο-αρχιτεκτονικές υλοποιήσεις, P1 και P2, της νέας μας αρχιτεκτονικής MIPS_SMALL. Έστω ότι υπάρχουν πέντε διαφορετικές κλάσεις εντολών στην MIPS_SMALL (A, B, C, D, E). Ο παρακάτω πίνακας που θα χρησιμοποιηθεί στα ερωτήματα αυτής της άσκησης δείχνει την συχνότητα του ρολογιού και το CPI (clocks per instruction) για κάθε κλάση εντολών της MIPS_SMALL.

Επεξεργαστής	**Συχνότητα Ρολογιού**	**CPI A**	**CPI B**	**CPI C**	**CPI D**	**CPI E**
P1	1.0 GHz	1	1	2	3	2
P2	1.5 GHz	1	2	3	4	3

a) Θεωρείστε ότι η μέγιστη απόδοση ενός επεξεργαστή βρίσκεται όταν ο επεξεργαστής εκτελέσει την πιο γρήγορη αλληλουχία εντολών. Ποια είναι η μέγιστη απόδοση (σε εντολές ανά sec) για τους επεξεργαστές P1 και P2;
b) Κατά την διάρκεια εκτέλεσης ενός προγράμματος ό ίδιος αριθμός εντολών εκτελείται από κάθε μία από τις 5 κλάσεις εντολών, εκτός από τις εντολές A που εκτελούνται δύο φορές πιο συχνά από τις άλλες. Ποιος από τους 2 επεξεργαστές είναι πιο γρήγορος και κατά πόσο;
c) Κατά την διάρκεια εκτέλεσης ενός προγράμματος ό ίδιος αριθμός εντολών εκτελείται από κάθε μία από τις 5 κλάσεις εντολών, εκτός από τις εντολές E που εκτελούνται δύο φορές πιο συχνά από τις άλλες. Ποιος από τους 2 επεξεργαστές είναι πιο γρήγορος και κατά πόσο;

Άσκηση 2 - Data Compression (10 μονάδες)

Μία πολύ συνηθισμένη μέθοδος συμπίεσης (compression) ενός συνόλου δεδομένων είναι η μέθοδος *Run-length encoding (RLE).* Η μέθοδος *RLE* μετατρέπει
μία ακολουθία ακεραίων αριθμών που έχουν την ίδια τιμή C (*runs* of data), σε ένα ζεύγος τιμών που αποτελείται από το πλήθος των διαδοχικών εμφανίσεων του C, μαζί με την τιμή C. Μια ενδιαφέρουσα εκδοχή της συμπίεσης *RLE* είναι όταν ο αριθμός C είναι το μηδέν, και είναι ιδιαιτέρως χρήσιμη όταν ένας μικρός αριθμός μη μηδενικών δεδομένων βρίσκεται μέσα σε έναν μεγάλο όγκο από μηδενικά. Η μέθοδος αυτή χρησιμοποιείται ευρύτατα σε εφαρμογές όπως συμπίεση video (πχ MPEG4), γραφικά, κείμενο και animations, οι οποίες διακρίνονται για τον τεράστιο όγκο όμοιων δεδομένων τα οποία θα κατελάμβαναν πολύ χώρο εάν δεν κωδικοποιούνταν με την μέθοδο RLE.

Παράδειγμα:
Αρχική ακολουθία ακεραίων:
1, 0, -3, 0, 0, 0, 0, 34, 0, 0, 0, 0, 0, -1, 0, 0, 11, 0, 0x111B

Συμπιεσμένη ακολουθία:
(0, 1) (1, -3) (4, 34) (5, -1) (2, 11) (1, 0X111B)

Να υλοποιηθεί συνάρτηση που να συμπιέζει μία ακολουθία από ακεραίους χρησιμοποιώντας την μέθοδο *Run-length encoding (RLE).* Η αρχική ακολουθία ακεραίων τερματίζεται με τον χαρακτήρα 0X111B.
Η συνάρτηση θα παίρνει σαν παράμετρο την διεύθυνση της ακολουθίας που πρόκειται να συμπιεστεί και την διεύθυνση στην οποία θα μπει το αποτέλεσμα.
Τέλος, η συνάρτηση θα εκτυπώνει στην οθόνη το περιεχόμενο της ακολουθίας που παράχθηκε μετά από την συμπίεση.

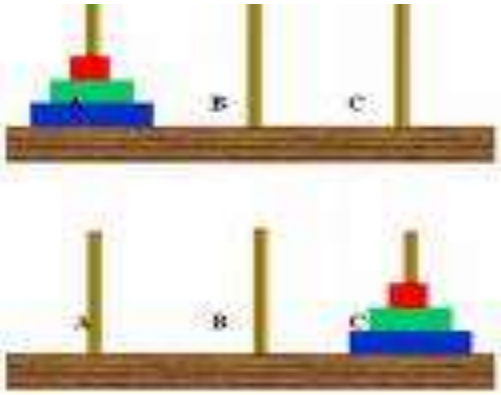
Να χρησιμοποιήσετε την ακολουθία του παραδείγματος των 64 ακεραίων στην υλοποίηση του προγράμματός σας (δεν χρειάζεται δηλ. να διαβάζετε από την κονσόλα):

```
.data
  uncompressed: .word
    12, -23, 0, 3, 0, 0, 0, 2, 0, 0, 0, 0, 0, -22, 0, 0,
     0, 0, 1, 0, 0, 0, -10, 11, -100, 0, 0, 0, 0, 34, 0, 5,
     -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -2,
     0, 0, 0, 0, 0, 0, 0, 0, 9, 0, 0, 0, 0, 0, 0, 0, 0X111B
```

Η παραπάνω ακολουθία είναι απλώς ένα παράδειγμα. Το πρόγραμμα σας θα πρέπει να δουλεύει σωστά για **οποιονδήποτε** πίνακα ακεραίων.

Άσκηση 3 – Αναδρομή (15 μονάδες)

Υλοποιήστε σε assembly κώδικα ένα πρόγραμμα το οποίο θα υλοποιεί τον αλγόριθμο «Ο πύργος της Βαβέλ».



Εικόνα 1. Ο πύργος της Βαβέλ

Περιγραφή Αλγορίθμου:
Υπάρχουν *n* δακτύλιοι και τρεις στύλοι. Αρχικά οι δακτύλιοι βρίσκονται στον πρώτο στύλο, τοποθετημένοι από τον μεγαλύτερο στον μικρότερο. (βλ. Εικόνα 1)
Να μεταφερθούν οι *n* δακτύλιοι από τον πρώτο στον τρίτο στύλο, χρησιμοποιώντας το δεύτερο ως βοηθητικό χώρο.

Κανόνες:
Επιτρέπεται η μετακίνηση ενός δακτυλίου τη φορά.
Κάθε μετακίνηση αποτελείται από την αφαίρεση ενός δακτυλίου που βρίσκεται στην κορυφή ενός στύλου και τοποθέτησή του στην κορυφή ενός άλλου δακτυλίου.
Δεν μπορεί να τοποθετηθεί μεγαλύτερος δακτύλιος πάνω από μικρότερο.

Περιγραφή της λύσης:
Μετακίνηση N-1 δακτυλίων από τον στύλο A στον στύλο B. (Χρησιμοποιώντας ως βοηθητικό τον στύλο C).
Μετακίνηση του μεγαλύτερου στύλου από τον δακτύλιο A στον δακτύλιο C.
Μετακίνηση N-1 δακτυλίων από τον στύλο B στον στύλο C. (Χρησιμοποιώντας ως βοηθητικό τον στύλο A).

Η έξοδος του προγράμματός σας για N=3 θα πρέπει να είναι ως εξής:
Enter number of disks: 3
Move disk 1 from peg 1 to peg 3
Move disk 2 from peg 1 to peg 2
Move disk 1 from peg 3 to peg 2
Move disk 3 from peg 1 to peg 3
Move disk 1 from peg 2 to peg 1
Move disk 2 from peg 2 to peg 3
Move disk 1 from peg 1 to peg 3

Σημείωση: Στο παράδειγμα της εξόδου έχουν αντικατασατθεί τα γράμματα A, B, C με τους αριθμούς 1, 2, 3, αντίστοιχα.

Η υλοποίηση σας πρέπει να γίνει με χρήση αναδρομικής συνάρτησης.

Πριν ξεκινήσετε την λύση του προβλήματος σε Assembly, θα ήταν χρήσιμο να την έχετ υλοποιήση πρώτα σε μορφή ψευτοκώδικα ή σε C. Ο,τι από τα δυο χρησιμοποιήσετε να το έχετε μαζί σας (είτε σε ηλεκτρονική μορφή είτε γραπτώς) κατα την εξέταση της άσκησης.