ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

Τμήμα Μηχανικών Η/Υ, Τηλεπικοινωνιών και Δικτύων

Τομέας Υλικού και Αρχιτεκτονικής Υπολογιστών

ΗΥ134 - Εισαγωγή τους Η/Υ

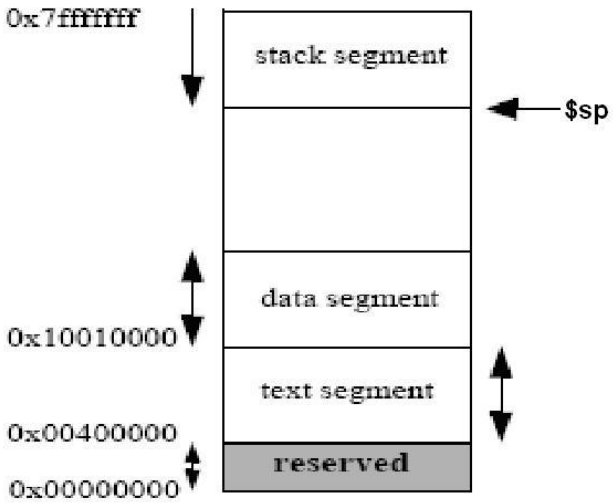
**Εργαστηριακή Άσκηση 4**

**Εαρινό εξάμηνο 2010**

Σκοπός της 4ης εργαστηριακής άσκησης είναι να εξοικειωθείτε με την χρήση της στοίβας. Αυτό θα σας επιτρέψει να υλοποιείτε καλύτερα συναρτήσεις οι οποίες θα μπορούν να καλέσουν με την σειρά τους άλλες συναρτήσεις ή ακόμα και τον εαυτό τους οπότε και λέγονται αναδρομικές. Η στοίβα έχει και άλλες λειτουργικότητες όπως οταν πρέπει να περάσετε πολλές παραμέτρους σε μια συνάρτηση.
Στο πρώτο κομμάτι θα δούμε τι ακριβώς είναι η στοίβα και στο δεύτερο που και πως πρέπει να χρησιμοποιείται.

## *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 βλέπεται 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. Οταν 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.
.                      #Ότι 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 οταν 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 1:**

```
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 λεει ότι μια συνάρτηση δεν είναι υποχρεωμένη να κρατήσει το περιεχόμενο των καταχωρητών ίδιο οταν 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 1.

**1η Ασκηση (30%)**

Η συνάρτηση *int dec2bin2oct(int X)* υλοποιεί έναν αλγόριθμο μετατροπής ενός αριθμού από το δεκαδικό σύστημα αρίθμησης στο δυαδικό και στη συνέχεια στο οκταδικό σύστημα αρίθμησης. Η συνάρτηση θα δέχεται έναν θετικό ακέραιο αριθμό, μικρότερο του 1024 και στη συνέχεια θα τον μετατρέπει στον αντίστοιχο δυαδικό αριθμό. Από τον δυαδικό αριθμό, χωρίζοντάς τον σε τριάδες θα υπολογίζεται ο αντίστοιχος οκταδικός αριθμός, ο οποίος και θα επιστρέφεται. Για παράδειγμα, έστω ο αριθμός $(973)_{10}$. Η συνάρτηση *int dec2bin2oct* τον δέχεται σαν όρισμα, και τον μετατρέπει στο string «1111001101» που αντιστοιχεί στον δυαδικό $(1111001101)_2$ .
Η μετατροπή του τελευταίου σε οκταδικό γίνεται ως εξής: χωρίζουμε τον αριθμό σε τριάδες και για κάθε μια υπολογίζουμε τον αντίστοιχο δεκαδικό. Επομένως για το συγκεκριμένο παράδειγμα θα έχουμε 1->1 111->7 001->1 101->5, αρά το τελικό αποτέλεσμα (δηλ. η τιμή που θα επιστρέψει η συνάρτηση) θα είναι ο πίνακας χαρακτήρων «1715» που αντιστοιχεί στον οκταδικό $(1715)_8$.
Η άσκηση αυτή σας ζητάει να γράψετε ένα πρόγραμμα MIPS assembly που να υλοποιεί την ρουτίνα *int dec2bin2oct*. Ο κώδικας σας θα διαβάζει το X από την κονσόλα, θα καλεί την *int dec2bin2oct* από την *main*, θα σχηματίζει τους δύο πίνακες στο .data section και θα τους εκτυπώνει στην κονσόλα.

**Υποδείξεις :**
Μελετήστε του ASCII χαρακτήρες σαν αριθμούς.
Ισχύει ότι:

- ‘0’ = $(48)_{10}$
- ‘1’ = $(49)_{10}$

**2η Ασκηση (Binary Search) (35%)**

Η συνάρτηση *int binary_search(int *A, const int N, const int X)* υλοποιεί έναν γρήγορο αλγόριθμο αναζήτησης του ακεραίου αριθμού X στον **ταξινομημένο** πίνακα A[N]. Η συνάρτηση επιστρέφει την θέση του X στον πίνακα, ή -1 εάν ο X δεν υπάρχει στον πίνακα A. Για παράδειγμα, *binary_search( {-8,2,5,10,20,100,123}, 7 ,20) = 4*. Επίσης *binary_search({-8,2,5,10,20,100,123}, 7, 0) = -1*. Η περιγραφή του αλγόριθμου του binary search είναι η εξής:
Ο *binary search* ελέγχει το διάμεσο (median) στοιχείο του ταξινομημένου πίνακα (για παράδειγμα, το 10 στην παραπάνω λίστα). Εάν ο X είναι μικρότερος από τον median, συνεχίζουμε την αναζήτηση στο κάτω μισό της λίστας, ενώ εάν είναι μεγαλύτερος από το median, συνεχίζουμε την αναζήτηση στο πάνω μισό της λίστας. Εάν είναι ίσος, τότε έχουμε βρεί την θέση του X στην λίστα και επιστρέφουμε. Με αυτό τον τρόπο διαιρούμε το μέγεθος της λίστας που ψάχνουμε κάθε φορά κατά 2. Εάν το μέγεθος της λίστας που έχει απομείνει είναι <=0, τότε επιστρέφουμε με -1 (το X δεν είναι στην λίστα).
Η άσκηση αυτή σας ζητάει να γράψετε ένα πρόγραμμα MIPS assembly που να υλοποιεί την ρουτίνα binary search, δεδομένου ενός πίνακα A μήκους N. Ο κώδικας σας θα διαβάζει το N, στην συνέχεια θα διαβάζει τον ταξινομημένο κατά αύξουσα

σειρά πίνακα Α (θεωρείστε ότι ο χρήστης τον εισάγει πάντα με τον σωστό τρόπο) και θα τον αποθηκεύει στο .data segment του προγράμματος. Μετά ο χρήστης θα διαβάζει το Χ. Στην συνέχεια θα καλεί την *binary search* από την *main*, και θα εκτυπώνει το αποτέλεσμα στην κονσόλα. Είναι **προφανές** ότι η λύση σας θα πρέπει να καλύπτει οποιεσδήποτε ακέραιες λίστες και όχι μόνο την λίστα του παραπάνω παραδείγματος**. Λύση η οποία κάνει μία απλή γραμμική αναζήτηση για να βρεί το Χ δεν θα γίνεται δεκτή.**
Είναι καλή ιδέα να γράψετε πρώτα το πρόγραμμα αυτό σε C (ή Java) πριν αρχίσετε με την assembly.

**3η Άσκηση (Merge two sorted lists) (35%)**
Η συνάρτηση *void merge(int *out, const int *in1, int N, const int *in2, int M)* συγχωνεύει τους δύο ταξινομημένους πίνακες ακεραίων *in1[N]* και *in2[M]* σε ένα νέο, επίσης ταξινομημένο, πίνακα *out*. Το μέγεθος του out είναι ίσο με το άθροισμα των μεγεθών των *in1* και *in2*. Για παράδειγμα, εάν ο *in1[4]* = {-5, 0, 2, 11} και *in2[5]* = {-2, 0, 1, 13, 15}, ο πίνακας *out[9]*={-5, -2, 0, 0, 1, 2, 11, 13, 15}.
Η άσκηση αυτή σας ζητάει να γράψετε ένα **γρήγορο** πρόγραμμα MIPS assembly που να υλοποιεί την συνάρτηση *merge* και να παράγει τον ταξινομημένο πίνακα out. Το μήκος των πινάκων καθώς και οι ταξινομημένοι πίνακες *in1* και *in2* θα πρέπει να διαβάζονται από την κονσόλα με *syscall* και να αποθηκεύονται στην μνήμη. Ο πίνακας *out[]* θα πρέπει να δημιουργείται στην μνήμη και να τυπώνεται στην κονσόλα.
Θα πρέπει να δείξετε την κλήση της *merge* από την *main*. Η λύση σας θα πρέπει να καλύπτει οποιεσδήποτε ακέραιες λίστες οποιουδήποτε μήκους και όχι μόνο τις λίστες του παραπάνω παραδείγματος. Επίσης θα πρέπει ο κώδικας σας να καλύπτει και ακραίες περιπτώσεις, όπως να είναι NULL ο ένας ή και οι δύο από τους πίνακες in1 και in2. Προσέξτε ότι υπάρχουν 5 παράμετροι εισόδου, και άρα θα πρέπει ένας από αυτούς να τοποθετηθεί στην στοίβα από την main πρίν την κλήση της συνάρτησης merge. Υποθέστε ότι ο pointer στον πίνακα str_out τοποθετείται στην στοίβα από την main και μετά διαβάζεται μέσα στην merge. Λύση η οποία δεν χρησιμοποιεί το γεγονός ότι οι δύο λίστες είναι ήδη ταξινομημένες δεν θα γίνεται δεκτή.