

# Οργάνωση και Σχεδίαση Υπολογιστών (ECE 219)

## Χειμερινό Εξάμηνο 2020-2021

### Εργαστήριο 2

8% του συνολικού εργαστηριακού βαθμού

#### Στόχοι του εργαστηρίου

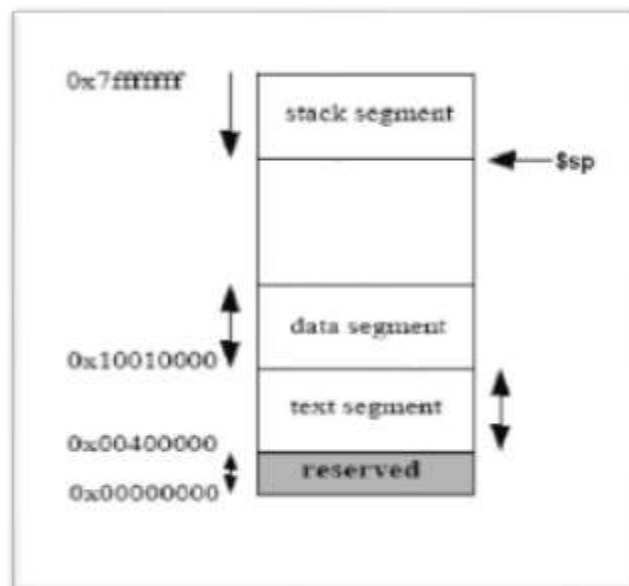
---

- 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 πάρουν 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 A) και παράλληλα περιέχει δεδομένα απαραίτητα για μια άλλη συνάρτηση (ας την πούμε B) η οποία σε κάποιο σημείο της εκτέλεσης της κάλεσε την A τότε τα δεδομένα αυτού του καταχωρητή χάνονται για την συνάρτηση B.

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 παρενέργειες. 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 των καταχωρητών και μετά την κλήση κάποιων 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 (Merge Function : 3 μονάδες)

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 ή εναλλακτικά να μπαίνουν σαν σταθεροί πίνακες στο *.data segment* του προγράμματος. 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* πριν 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 εισόδου είναι ήδη ταξινομημένες δεν θα γίνεται δεκτή.

## Άσκηση 2– Substring (4 μονάδες)

---

Να υλοποιηθεί η συνάρτηση

```
int substring(const char *str, const char *substr)
```

η οποία θα παίρνει ως ορίσματα το string *str*, και το string *substr*. Η συνάρτηση θα ψάχνει το μέγιστο τμήμα του *substr* (ξεκινώντας πάντα από την αρχή του *substring*) που βρίσκεται στο string *str* και θα επιστρέφει το μέγιστο μήκος του *substr* που βρέθηκε στο *str*.

### Παράδειγμα:

str: “abriaaklrivcariver”

substr: “river”

Η συνάρτηση θα επιστρέψει 5.

Μέρος του *substr* που υπάρχει στο *str*, είναι το *ri*, το *riv*, και το *river*. Όμως, Το μέγιστο κομμάτι του *substr* που υπάρχει στο *str*, είναι το *river*, γι’ αυτό η συνάρτηση επιστρέφει ως 5.

Ο χρήστης θα δίνει από την κονσόλα το αρχικό string, και το *substring* που ψάχνουμε, και ως έξοδο θα βγάζει το μέγιστο σε μήκος *substring* που βρέθηκε.

Παραδείγματα εκτέλεσης:

Please give string: aafgbba

Please give substring: fab

The max substring has length : 1

Please give string: kalmacc

Please give substring: vbd

The max substring has length : 0

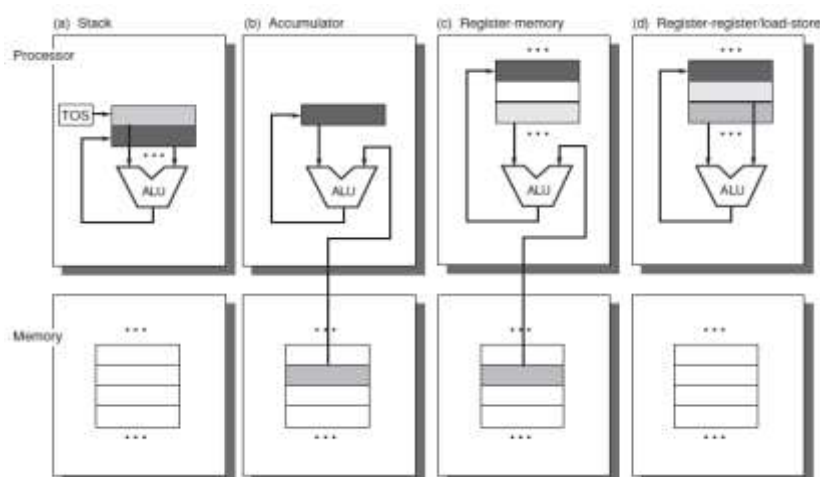
Please give string: alloavellof

Please give substring: llof

The max substring has length : 4

### Άσκηση 3– Εναλλακτικές Αρχιτεκτονικές (3 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.



Γράφημα 1. Οι τέσσερις βασικές αρχιτεκτονικές της άσκησης ανάλογα με την τοποθεσία των ορισμάτων (operands) εισόδου και εξόδου. Τα κουτιά με ανοιχτό γκρι χρώμα είναι είσοδοι στην ALU, ενώ τα κουτιά με σκούρο γκρι χρώμα είναι εξόδοι της ALU. Στην αρχιτεκτονική stack (a) η ALU παίρνει εισόδους αποκλειστικά από την στοίβα (stack) και συγκεκριμένα τα δύο ορίσματα που είναι στην κορυφή της στοίβας, το ένα κάτω από το άλλο. Ο pointer TOS (Top Of Stack) δείχνει κάθε χρονική στιγμή την κορυφή της στοίβας. Η ALU παράγει το αποτέλεσμα της πράξης το οποίο και τοποθετείται στην θέση του δεύτερου μετά την κορυφή ορίσματος ενώ το πρώτο όρισμα βγαίνει από την στοίβα και διαγράφεται. Στην αρχιτεκτονική accumulator (b) οι πράξεις γίνονται μεταξύ του accumulator και της μνήμης, και το αποτέλεσμα μεταβαίνει στον accumulator. Στην αρχιτεκτονική register-memory (c) (πχ Intel's x86), μπορούμε να εκτελέσουμε πράξεις απευθείας μεταξύ καταχωρητών και μνήμης, ενώ το αποτέλεσμα μπαίνει στον καταχωρητή. Η τελευταία αρχιτεκτονική register-register (d), είναι όμοια με την αρχιτεκτονική του MIPS.

Stack	Accumulator	Register-Memory	Register-Register
Push A # load from Mem[A] to stack	Load A	Load R1, A	Load R1, A
Push B # load from Mem[B] to stack	Add B	Add R2, R1, B	Load R2, B
Add	Store C	Store R2, C	Add R3, R1, R2
Pop C #write from stack to Mem[B]			Store R3, C

Γράφημα 2. Ο κώδικας  $C=A+B$  για κάθε μία από τις τέσσερις αρχιτεκτονικές του Γραφήματος 1. Προσέξτε ότι η εντολή Add στην αρχιτεκτονική stack δεν περιέχει εισόδους, επειδή οι είσοδοι υπονοείται ότι είναι οι δύο μεταβλητές στην κορυφή της λίστας. Οι μεταβλητές A, B και C βρίσκονται στην μνήμη και οι μεταβλητές A και B πρέπει να διατηρηθούν στην μνήμη (δηλ. να μην διαγραφούν) και μετά την εκτέλεση της εντολής  $C=A+B$ .

Θέλουμε να αξιολογήσουμε τις τέσσερις αρχιτεκτονικές που περιγράφονται στα γραφήματα 1 και 2 όσον αφορά την αποδοτικότητά τους να προσπελαίνουν την κύρια μνήμη. Για να μετρήσουμε την αποδοτικότητα αυτή κάνουμε τις εξής παραδοχές όσον αφορά τα τέσσερα σύνολα εντολών:

- Όλες οι εντολές έχουν μήκος πολλαπλάσιο του ενός byte
- Το opcode των εντολών έχει μήκος 1 byte
- Η προσπέλαση της μνήμης για δεδομένα γίνεται αποκλειστικά χρησιμοποιώντας absolute addressing (βλ. γράφημα 2).
- Οι μεταβλητές A, B, C και D είναι αρχικά στην μνήμη.

- a) Χρησιμοποιώντας εντολές assembly σαν αυτές που φαίνονται στο γράφημα 2, να αναπτύξετε τον παρακάτω κώδικα για κάθε μία από τις τέσσερις αρχιτεκτονικές:

$A = B + C;$   
 $B = A + C;$   
 $D = A - B;$

- b) Θεωρείστε τώρα ότι η παραπάνω ακολουθία των τριών εντολών εκτελείται σε επεξεργαστές με 16-bit διεύθυνση μνήμης και ότι οι μεταβλητές A, B, C και D είναι όλες επίσης 16 bits. Στην περίπτωση που χρειάζονται, υπάρχουν 16 καταχωρητές. Για κάθε μία αρχιτεκτονική ξεχωριστά και βασιζόμενοι στον κώδικα που αναπτύξατε στο ερώτημα (a), να απαντήσετε στα παρακάτω ερωτήματα:
- i. Πόσα bytes εντολών διαβάζονται από την μνήμη συνολικά;
  - ii. Πόσα bytes δεδομένων διαβάζονται από και γράφονται στην μνήμη συνολικά;
  - iii. Ποια αρχιτεκτονική είναι η πιο αποδοτική όσον αφορά την συνολική επικοινωνία με την μνήμη;