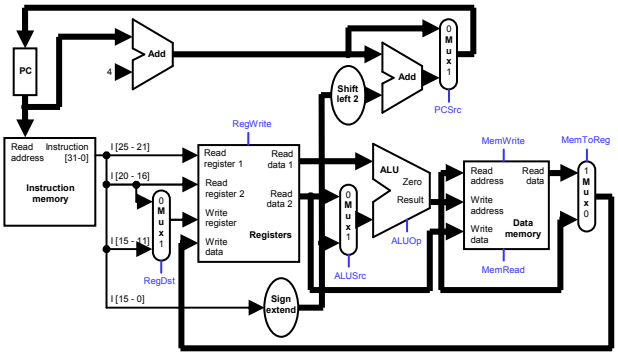
Πανεπιστήμιο Πειραιώς
Τμήμα Πληροφορικής

Σημειώσεις Εργαστηρίου

Μάθημα

ΑΡΧΙΤΕΚΤΟΝΙΚΗ ΥΠΟΛΟΓΙΣΤΩΝ

Εργαστήριο MIPS / QtSpim



Συγγραφή Σημειώσεων

Δάκης Παυλίδης, Ε.ΔΙ.Π.
Μιχάλης Ψαράκης, Επίκουρος Καθηγητής

Άνοιξη 2015

ΕΝΟΤΗΤΑ 1:

Γνωριμία με το περιβάλλον του προσομοιωτή QtSpim

Εισαγωγή

Σκοπός του εργαστηριακού μαθήματος της πρώτης ενότητας είναι η γνωριμία με το περιβάλλον προσομοίωσης και συμβολομετάφρασης του επεξεργαστή MIPS και η εξοικείωση στη χρήση βασικών εντολών συμβολικής γλώσσας για μεταφορά δεδομένων μεταξύ μνήμης και καταχωρητών. Θα γίνει χρήση βασικών οδηγιών προς τον προσομοιωτή, των κλήσεων συστήματος που υποστηρίζει ο προσομοιωτής. Επίσης, θα επιδειχθεί η δυνατότητα προσομοίωσης με βηματικό τρόπο (single step) και με σημεία διακοπής (breakpoints).

Ο προσομοιωτής **QtSPIM** είναι ήδη εγκατεστημένος στο εργαστήριο στα λειτουργικά συστήματα Windows και Linux, και μπορείτε να τον αντιγράψετε από την ιστοσελίδα του μαθήματος http://gunet2.cs.unipi.gr/courses/TMA106/. Υπάρχουν διαθέσιμες εκδόσεις για λειτουργικά συστήματα Windows, Debian Linux και Mac OS X. Ο προσομοιωτής διατίθεται από τον σχεδιαστή του James Larus στη σελίδα http://sourceforge.net/projects/spimsimulator/files/ και διανέμεται δωρεάν.

Οι ασκήσεις της πρώτης ενότητας χωρίζονται σε τρία μέρη: (1) Ασκήσεις που αφορούν την εκτέλεση προγραμμάτων assembly στον προσομοιωτή QtSpim, το τμήμα κώδικα και το τμήμα δεδομένων των προγραμμάτων και τη χρήση εντολών μεταφοράς δεδομένων, (2) ασκήσεις που αφορούν τις κλήσεις συστήματος που υποστηρίζει ο προσομοιωτής και (3) μία επί πλέον άσκηση.

Μέρος 1ο: Εκτέλεση, τμήμα κώδικα και δεδομένων, μεταφορά δεδομένων

Ασκηση 1.1

Ξεκινήστε τον προσομοιωτή QtSpim και εκτελέστε τις ακόλουθες ενέργειες:

- Φορτώστε (`File → Reinitialize and Load File`) το πρόγραμμα `lab1_1.s` που βρίσκεται στην ιστοσελίδα του μαθήματος (Κατάλογος "Εγγραφα/Εργαστήριο/Προγράμματα Assembly").
- Παρατηρήστε στον QtSpim στο παράθυρο του κώδικα (Text Segment) και των δεδομένων (Data Segment) τις διευθύνσεις μνήμης που έχουν καταχωρηθεί ο κώδικας και τα δεδομένα. Κάνοντας αντιπαραβολή στο παράθυρο του κώδικα μεταξύ των συμβολικών εντολών του προγράμματος και των εντολών του πηγαίου κώδικα (εμφανίζονται με italics μετά το σύμβολο ;) διαπιστώστε ποιες από τις εντολές του `lab1_1.s` είναι ψευδοεντολές (δηλαδή αναλύονται σε περισσότερες από μία εντολές του MIPS ή αντιστοιχούν σε διαφορετικό μνημονικό).
- Τρέξτε το πρόγραμμα με βηματικό τρόπο (`Simulator → Single` Step ή `F10`) και παρατηρήστε τις αλλαγές των σχετικών καταχωρητών σε κάθε βήμα.

Ψευδοεντολή (Pseudoinstruction): *Μια συνήθης παραλλαγή των εντολών της συμβολικής γλώσσας που συχνά αντιμετωπίζεται σαν να ήταν και αυτή εντολή. Το υλικό δε χρειάζεται να υλοποιεί αυτές τις εντολές· ωστόσο, η παρουσία τους στη συμβολική γλώσσα απλοποιεί τη μετάφραση και τον προγραμματισμό.*

Για παράδειγμα, ο συμβολομεταφραστής κάνει δεκτή την εντολή `move` παρά το γεγονός ότι δεν υπάρχει στην αρχιτεκτονική τού MIPS. Έτσι, ο συμβολομεταφραστής μετατρέπει την εντολή συμβολικής γλώσσας

```
move $t0,$t1
```

σε γλώσσα μηχανής ισοδύναμη με την παρακάτω εντολή:

```
add $t0,$zero,$t1
```

```
################################################
# lab1_1.s                                     #
# Pseudoinstruction examples - System calls    #
# t0 - holds each byte from string in turn     #
# t1 - contains count of characters            #
# t2 - points to the string                    #
################################################

################################################
#                                              #
#           text segment                       #
#                                              #
################################################

      .text
            .globl __start
__start:                # execution starts here
      la $t2,str        # t2 points to the string
      li $t1,0          # t1 holds the count
nextCh: lb $t0,($t2)    # get a byte from string
      beqz $t0,strEnd   # zero means end of string
      add $t1,$t1,1     # increment count
      add $t2,1         # move pointer one character
      j nextCh          # go round the loop again
strEnd: la $a0,ans      # system call to print
      li $v0,4          # out a message
      syscall
      move $a0,$t1      # system call to print
      li $v0,1          # out the length worked out
      syscall
      la $a0,endl       # system call to print
      li $v0,4          # out a newline
      syscall
      li $v0,10
      syscall           # au revoir...

################################################
#                                              #
#                 data segment                 #
#                                              #
################################################

.data
str:  .asciiz     "hello world"
ans:  .asciiz     "Length is "
endl: .asciiz     "\n"

################################################
#                                              #
# End of File                                  #
#                                              #
################################################
```

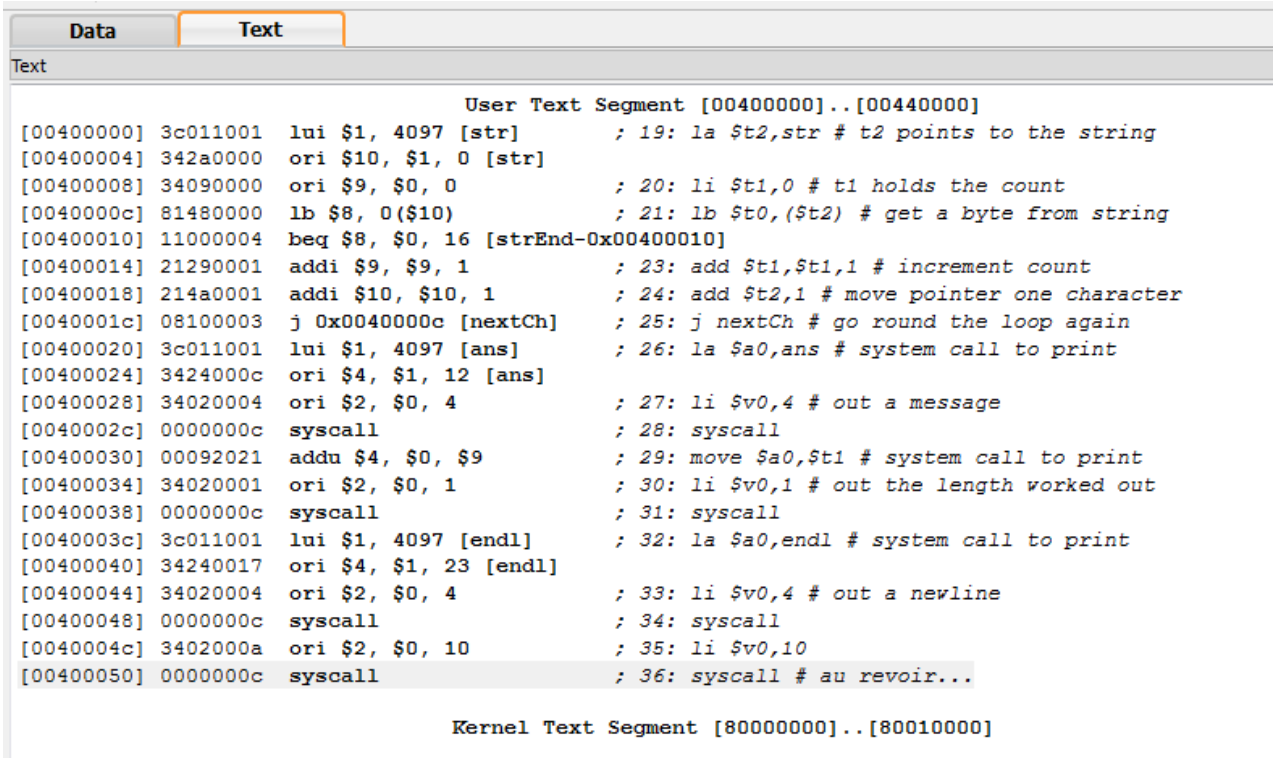
Το παράθυρο του κώδικα φαίνεται παρακάτω και δείχνει τη διεύθυνση κάθε εντολής (μέσα σε [ ]). Οι εντολές απέχουν μεταξύ τους 4 byte. Στη δεύτερη στήλη, δεξιά από τη στήλη που περιέχει τις διευθύνσεις, βρίσκεται η στήλη με το περιεχόμενο της κάθε λέξης εντολής (σε δεκαεξαδική μορφή), ακολουθεί η πραγματική εντολή μηχανής MIPS, και μετά το ελληνικό ερωτηματικό φαίνεται η αντίστοιχη εντολή από τον πηγαίο κώδικα του αρχείου assembly.

Παρατηρήστε ότι η ψευδοεντολή `la` (load address) αναλύεται σε δύο πραγματικές εντολές (τις `lui` και `ori`) οι οποίες μαζί έχουν σαν αποτέλεσμα να φορτωθεί στον καταχωρητή `$t2` η διεύθυνση της

συμβολοσειράς `str`. Η διεύθυνση αυτή είναι η δεκαεξαδική `0x10010000` στην οποία αρχίζουν τα δεδομένα του χρήστη (User Data).

Μία άλλη ψευδοεντολή είναι η `move` η οποία μετατρέπεται σε addu.

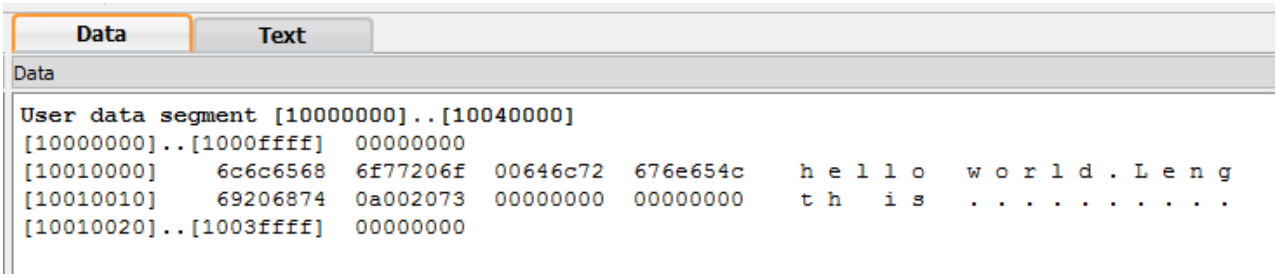
Επίσης παρατηρήστε την ψευδοεντολή `add $t2,1` η οποία μετατρέπεται από τον assembler στην πραγματική εντολή MIPS `addi $t2, $t2, 1` (αντί για `$t2` βλέπετε `$10` και το 10 είναι ο αριθμός του καταχωρητή `$t2`). Τόσο το μνημονικό όνομα της εντολής αλλάζει (αντί `addi` χρησιμοποιούμε στον πηγαίο κώδικα το `add`), όσο και το πλήθος των τελεστέων: ενώ οι πραγματικές εντολές απαιτούν τρεις, εδώ στον πηγαίο κώδικα χρησιμοποιούμε μόνο δύο).



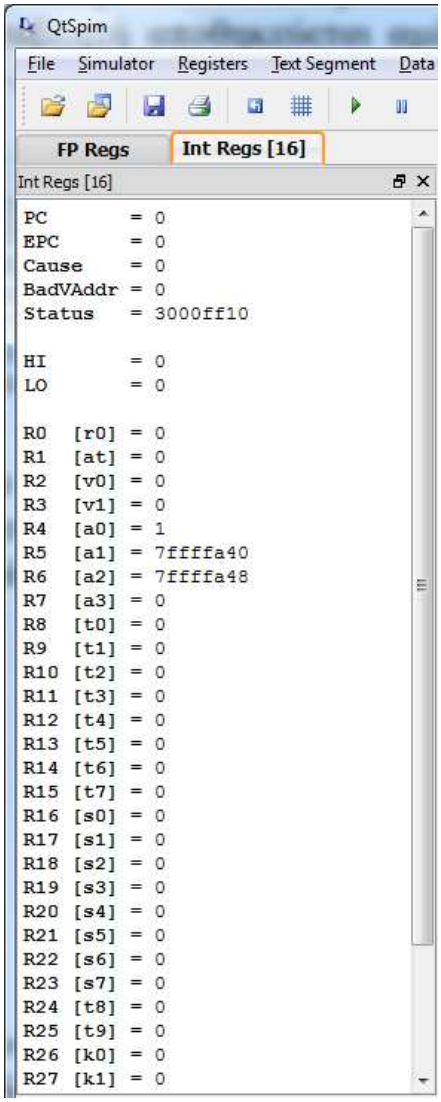
Το παράθυρο δεδομένων φαίνεται παρακάτω. Τα δεδομένα χρήστη αρχίζουν στη δεκαεξαδική διεύθυνση `0x10010000` και κάθε γραμμή που ακολουθεί μια διεύθυνση δείχνει το περιεχόμενο 16 διαδοχικών byte της μνήμης. Για κάθε byte δίνεται αρχικά το δεκαεξαδικό του περιεχόμενο (π.χ. το byte στη διεύθυνση `0x10010000` είναι ίσο με `0x6C` δηλαδή δυαδικά ίσο με `01101100`). Μετά τη δεκαεξαδική αναπαράσταση του περιεχομένου τού κάθε byte δίνεται ο ASCII χαρακτήρας που αυτό παριστάνει. Το `0x6C` είναι ίσο με 108 στο δεκαδικό και το 108 είναι ο ASCII κωδικός για το πεζό γράμμα της αγγλικής αλφαβήτου "`h`".

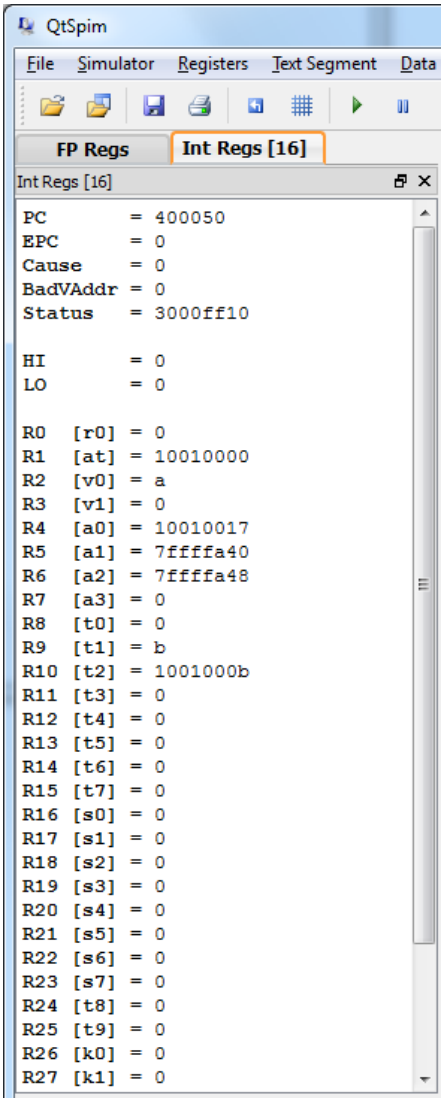
Βλέπετε ολόκληρη τη συμβολοσειρά "`hello world`" σε διαδοχικά byte της μνήμης. Όταν ένα byte αντιστοιχεί σε μη εκτυπώσιμο χαρακτήρα του ASCII κώδικα τότε βλέπουμε μια τελεία. Η τελεία που ακολουθεί το "`hello world`" είναι ο μηδενικός χαρακτήρας (null character) που στη C δηλώνει το τέλος μιας συμβολοσειράς. Μετά το "`hello world`" ακολουθεί η επόμενη συμβολοσειρά του προγράμματος "`Length is `".

Όταν μια μεταβλητή οποιουδήποτε τύπου δηλώνεται στο τμήμα δεδομένων του προγράμματος (Data Segment) μετά την οδηγία `.data`, τότε η μεταβλητή αποθηκεύεται αμέσως μετά την προηγούμενη μεταβλητή σε διαδοχικές θέσεις μνήμης. Καταλαμβάνει τόσα byte όσο είναι το μέγεθος του τύπου της.



Στην εκτέλεση με βηματικό τρόπο τα περιεχόμενα των καταχωρητών μπορείτε να τα βλέπετε στο σχετικό παράθυρο του QtSpim. Η εικόνα που ακολουθεί δείχνει το αρχικό περιεχόμενο των καταχωρητών πριν την εκτέλεση του προηγούμενου προγράμματος (αριστερά) και το τελικό περιεχόμενό τους μετά την εκτέλεσή του (δεξιά). Προσέξτε, για παράδειγμα, την τελική τιμή του καταχωρητή `$t1` ο οποίος μετρά το πλήθος των χαρακτήρων της συμβολοσειράς. Είναι ίση με το δεκαεξαδικό `b` δηλαδή 11 στο δεκαδικό σύστημα αρίθμησης που είναι και το σωστό αποτέλεσμα του προγράμματος αφού το μήκος της συμβολοσειράς είναι 11 χαρακτήρες.





Ασκηση 1.2

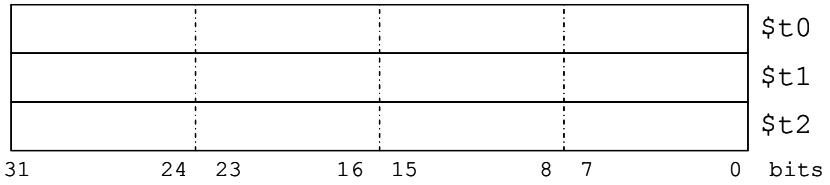
Στην άσκηση αυτή θα φανεί η διαφοροποίηση της ποικιλίας εντολών φόρτωσης και αποθήκευσης καταχωρητών από/προς τη μνήμη που διαθέτει ο QtSPIM.

(α) Φορτώστε και εκτελέστε το πρόγραμμα lab1_2a.s. Το πρόγραμμα δημιουργεί στο τμήμα δεδομένων (data segment) την ακόλουθη διάταξη (χάρτης μνήμης), χρησιμοποιώντας τις οδηγίες `.data`, `.byte` και `.word`. Συγκρίνετε το τμήμα δεδομένων του QtSpim με τον παρακάτω χάρτη μνήμης.

Byte Address	Data Segment				Byte and Word Address
	⋮				
	0x87654321				0x0100100C
	0x12345678				0x01001008
0x010010007	0x84	0x83	0x82	0x81	0x01001004
0x010010003	0x04	0x03	0x02	0x01	0x01001000

Επίσης, το πρόγραμμα χρησιμοποιεί τις εντολές μη προσημασμένης φόρτωσης `lbu`, `lhu` και `lw` για να φορτώσει στους καταχωρητές `$t0`, `$t1` και `$t2` τα bytes, half words και words αντίστοιχα, που ξεκινούν από τη διεύθυνση `0x10010000` (αυτά που έχουν τιμές `0x01`, `0x0201` και `0x04030201`, αντίστοιχα). Διαπιστώστε ότι έχουν φορτωθεί αυτές οι τιμές στους καταχωρητές και συμπληρώστε τον παρακάτω πίνακα.

Καταχωρητές με μη προσημασμένη φόρτωση




```
################################################
# lab1_2a.s                                    #
# Registers - Memory Data Exchange             #
################################################
      .text
      .globl __start
__start:                # execution starts here
      la $a0,start_b    # load base address into $a0
      lbu $t0,0($a0)    # load first byte into $t0
      lhu $t1,0($a0)    # load first halfword into $t1
      lw $t2,0($a0)     # load first word into $t2

      li $v0,10
      syscall           # exit
      .data
start_b:    .byte 0x01, 0x02, 0x03, 0x04
            .byte 0x81, 0x82, 0x83, 0x84
            .word 0x12345678, 0x87654321
```

Οδηγίες για διαμόρφωση data segment

.data	*Τα αντικείμενα που ακολουθούν αποθηκεύονται στο τμήμα δεδομένων.*
.byte b1,..., bn	*Αποθήκευση των n τιμών σε διαδοχικά byte στη μνήμη.*
.word w1,...,wn	*Αποθήκευση των n ποσοτήτων των 32 bit σεδιαδοχικές λέξεις μνήμης.*

Εντολές φόρτωσης

Load byte — Φόρτωση byte

`lb rt, address`

Load unsigned byte — Απρόσημη φόρτωση byte

`lbu rt, address`

Φόρτωση του byte που βρίσκεται στη διεύθυνση address στον καταχωρητή rt. Το byte υφίσταται επέκταση προσήμου από την lb, αλλά όχι από την lbu.

Load halfword — Φόρτωση ημιλέξης

`lh rt, address`

Load unsigned halfword — Απρόσημη φόρτωση ημιλέξης

`lhu rt, address`

Φόρτωση της ποσότητας 16 bit (ημιλέξη — halfword) που βρίσκεται στη διεύθυνση address, στον καταχωρητή rt. Η ημιλέξη υφίσταται επέκταση προσήμου από την lh, αλλά όχι από την lhu.

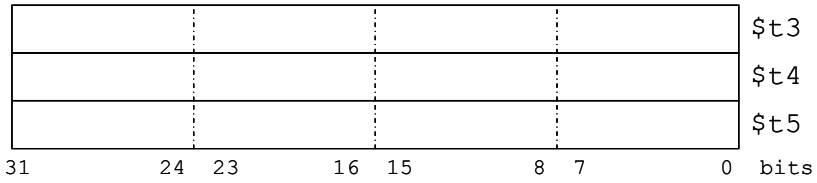
Load word — Φόρτωση λέξης

`lw rt, address`

Φόρτωση της ποσότητας 32 bit (λέξη) που βρίσκεται στη διεύθυνση address, στον καταχωρητή rt.

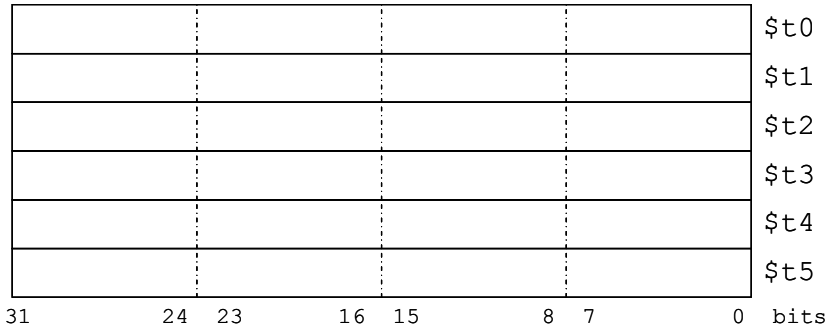
(β) Εμπλουτίστε το προηγούμενο πρόγραμμα έτσι ώστε να φορτωθούν με μη προσημασμένο τρόπο στους καταχωρητές `$t3`, `$t4` και `$t5` τα bytes, half words και words που ξεκινούν από τη διεύθυνση `0x01001004` (αυτά που έχουν τιμές `0x81`, `0x8281` και `0x84838281` , αντίστοιχα). Συμπληρώστε στον παρακάτω πίνακα τις τιμές των καταχωρητών μετά την εκτέλεση των νέων εντολών:

Καταχωρητές με μη προσημασμένη φόρτωση



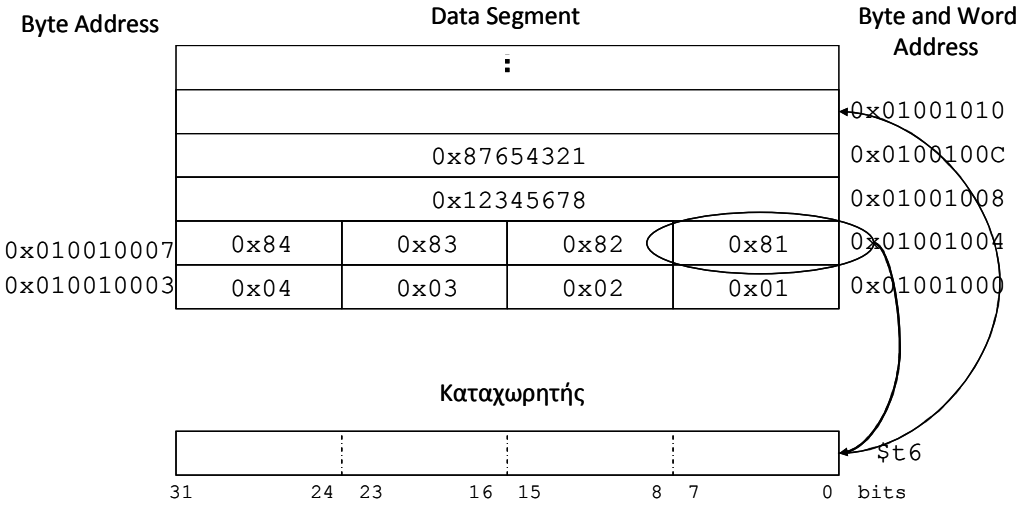
(γ) Αλλάξτε το πρόγραμμα που προέκυψε στο προηγούμενο βήμα επαναλαμβάνοντας τις προηγούμενες μεταφορές στους ίδιους καταχωρητές (`$t0`..`$t5`) χρησιμοποιώντας όμως αυτή τη φορά τις εντολές προσημασμένης φόρτωσης `lb`, `lh` και `lw`. Συμπληρώστε τον παρακάτω πίνακα και παρατηρήστε την επέκταση προσήμου που συμβαίνει μετά την εκτέλεση των εντολών προσημασμένης φόρτωσης.

Καταχωρητές με προσημασμένη φόρτωση



*Η λειτουργία μιας προσημασμένης φόρτωσης είναι η αντιγραφή του προσήμου επαναληπτικά για τη συμπλήρωση του υπόλοιπου καταχωρητή — ονομάζεται **επέκταση προσήμου (sign extension)** — αλλά ο σκοπός της είναι η τοποθέτηση μιας σωστής αναπαράστασης του αριθμού μέσα σε εκείνον τον καταχωρητή. Οι απρόσημες φορτώσεις απλώς συμπληρώνουν με 0 τις θέσεις στα αριστερά των δεδομένων εφόσον ο αριθμός που αναπαρίσταται με τη σειρά των bit είναι απρόσημος.*

(δ) Φορτώστε στον καταχωρητή `$t6` με μη προσημασμένο τρόπο το byte της διεύθυνσης `0x10010004` και κατόπιν αποθηκεύστε το χαμηλό του byte (με την εντολή `sb`) στην πρώτη διεύθυνση που είναι ελεύθερη (μετά τις πρώτες τέσσερις λέξεις που είναι ήδη αποθηκευμένες). Ελέγξτε αν σε αυτήν την περίπτωση έχουμε επέκταση προσήμου. Η εικόνα που ακολουθεί δείχνει την επιθυμητή λειτουργία. Συμπληρώστε το περιεχόμενο της λέξης που ξεκινά από τη διεύθυνση `0x10010010`.

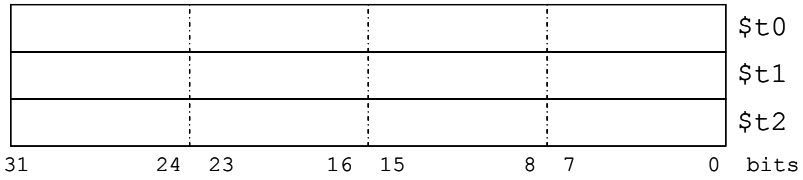


Ασκηση 1.3

Στην άσκηση αυτή θα διευκρινιστεί η έννοια της ευθυγράμμισης (alignment) στη μνήμη. Μία λέξη των 4 bytes είναι ευθυγραμμισμένη όταν αποθηκεύεται σε διεύθυνση που είναι πολλαπλάσιο του 4.

(α) Κρατήστε από το πρόγραμμα της Άσκησης 1.2 τη δομή του τμήματος δεδομένων (data segment) και φορτώστε διαδοχικά στους καταχωρητές `$t0`,`$t1` και `$t2` τα bytes (χωρίς επέκταση προσήμου), από τις διευθύνσεις `0x1001000A`, `0x1001000B` και `0x1001000C`, αντίστοιχα. Συμπληρώστε στον παρακάτω πίνακα τις τιμές των καταχωρητών μετά την εκτέλεση των εντολών:

Καταχωρητές με μη προσημασμένη φόρτωση από μη ευθυγραμμισμένη μνήμη



(β) Προσθέστε στο τμήμα δεδομένων ένα επί πλέον byte με τιμή `0x05` και μισή λέξη (οδηγία `.half`) με τιμή `0x6677` ανάμεσα στο byte με τιμή `0x04` και το byte με τιμή `0x81`. Φορτώστε το πρόγραμμα στον προσομοιωτή και συμπληρώστε τον πίνακα που ακολουθεί. Παρατηρήστε την αυτόματη ευθυγράμμιση στη διάταξη των δεδομένων στη μνήμη του προσομοιωτή.

Byte Address	Data Segment				Byte and Word Address
0x010010017					0x01001014
0x010010013					0x01001010
0x01001000F					0x0100100C
0x01001000B					0x01001008
0x010010007					0x01001004
0x010010003					0x01001000

Κατόπιν γράψτε τον εκτελέσιμο κώδικα έτσι ώστε στον καταχωρητή `$t0` να φορτωθεί η αποθηκευμένη λέξη `0x12345678` και στον `$t1` η μισή λέξη `0x6677` , εκτελέστε το πρόγραμμα και διαπιστώστε ότι πραγματοποιήθηκαν οι δύο φορτώσεις.

.half h1, ..., hn	*Αποθήκευση των n ποσοτήτων των 16 bit σε διαδοχικές ημιλέξεις μνήμης.*

(γ) Αλλάξτε το προηγούμενο πρόγραμμα έτσι ώστε τα δεδομένα να μην ευθυγραμμίζονται αυτόματα. Αυτό μπορεί να γίνει αν συμπεριληφθεί η οδηγία `.align 0` αμέσως μετά την `.data`. Προσπαθήστε να κάνετε ότι και πριν. Επειδή τώρα τα δεδομένα (εκτός από τα bytes) δεν είναι ευθυγραμμισμένα, εμφανίζεται μήνυμα λάθους "`Exception occurred at PC=0x00400008 Unaligned address in inst/data fetch: 0x1001000b`".

Byte Address	Data Segment				Byte and Word Address
0x010010017					0x01001014
0x010010013					0x01001010
0x01001000F					0x0100100C
0x01001000B					0x01001008
0x010010007					0x01001004
0x010010003					0x01001000

.align n	*Ευθυγράμμιση του επόμενου δεδομένου σε όριο 2n byte. Για παράδειγμα, η .align 2 ευθυγραμμίζει την επόμενη τιμή σε ένα όριο λέξης. Η .align 0 απενεργοποιεί την αυτόματη ευθυγράμμιση των οδηγιών .half, .word, .float, και .double μέχρι την επόμενη οδηγία .data ή .kdata.*

(δ) Στο σύνολο εντολών του MIPS υπάρχουν οι ψευδοεντολές `ulw` (unaligned load word) και `ulh` (unaligned half word) που επιτρέπουν την προσπέλαση σε μη ευθυγραμμισμένες λέξεις και μισές λέξεις αντίστοιχα. Χρησιμοποιήστε τις εντολές αυτές στη θέση των `lw` και `lh` που χρησιμοποιήσατε στο βήμα (γ) και δείτε πως αναλύονται σε πραγματικές εντολές MIPS.

Unaligned load halfword — Φόρτωση ημιλέξης χωρίς ευθυγράμμιση

`ulh rdest, address` *ψευδοεντολή*

Unaligned load halfword unsigned — Φόρτωση ημιλέξης χωρίς ευθυγράμμιση χωρίς πρόσημο

`ulhu rdest, address` *ψευδοεντολή*

Φόρτωση της ποσότητας 16 bit (ημιλέξη — halfword) που βρίσκεται στην πιθανόν μη ευθυγραμμισμένη διεύθυνση address, στον καταχωρητή rdest. Η ημιλέξη υφίσταται επέκταση προσήμου από την ulh, αλλά όχι από την ulhu.

Unaligned load word — Φόρτωση λέξης χωρίς ευθυγράμμιση

`ulw rdest, address` *ψευδοεντολή*

Φόρτωση της ποσότητας 32 bit (λέξης) που βρίσκεται στην πιθανόν μη ευθυγραμμισμένη διεύθυνση address στον καταχωρητή rdest.

Μέρος 2ο: Κλήσεις συστήματος

Ασκηση 1.4

Πρόγραμμα `lab1_4.s` διαβάζει μία συμβολοσειρά (string) από το παράθυρο εισόδου/εξόδου (Console ή κονσόλα) που του δίνει ο χρήστης, χρησιμοποιώντας την κλήση συστήματος `read_string`. Η συμβολοσειρά αποθηκεύεται στο τμήμα δεδομένων.

(α) Εκτελέστε το πρόγραμμα και παρατηρήστε το τμήμα δεδομένων στο οποίο έχει αποθηκευτεί η συμβολοσειρά που δώσατε.

```
################################################
# lab1_4.s                                     #
# read_string & print_string (to be completed)  #
################################################
      .text
      .globl __start
__start:                # execution starts here
      la $a0,string
      li $v0,8
      syscall           # read string in $a0 up to length $a1

      li $v0,10
      syscall           # exit
      .data
string:     .space 80
```

(β) Συμπληρώστε το πρόγραμμα έτσι ώστε μετά την ανάγνωση της συμβολοσειράς να εμφανίζει στην κονσόλα μόνο τους πέντε πρώτους χαρακτήρες από αυτήν, όπως δείχνει η εικόνα που ακολουθεί. Χρησιμοποιήστε την κλήση συστήματος print_string.

Ασκηση 1.5

Το πρόγραμμα `lab1_5a.s` προτρέπει το χρήστη να δώσει έναν ακέραιο από την κονσόλα χρησιμοποιώντας την κλήση συστήματος print_string και διαβάζει τον ακέραιο χρησιμοποιώντας την read_int. Συμπληρώστε το πρόγραμμα ώστε αφού τον διαβάσει να τον τυπώνει στην κονσόλα χρησιμοποιώντας την κλήση print_int. Πειραματιστείτε εισάγοντας μεταξύ άλλων και τους ακεραίους 2,147,483,648 (maxint+1) και -2,147,483,649 (-maxint-1). Τι αποτέλεσμα σας δίνει η εκτέλεση του προγράμματος στις περιπτώσεις αυτές;

```
################################################
# lab1_5a.s                                    #
# read_int & print_int  (to be completed) #
################################################
      .text
       .globl __start
__start:                # execution starts here
      la $a0,prompt
      li $v0,4
      syscall           # display prompt
      li $v0,5
      syscall           # read integer in $v0

      li $v0,10
      syscall           # exit
      .data
prompt:     .asciiz "Enter integer:"
```

Κλήσεις συστήματος

Υπηρεσία	*Κωδικός κλήσης συστήματος*	*Ορίσματα*	*Επιστροφή*
print_int	*1*	*$a0 = ακέραιος*	
print_string	*4*	*$a0 = διεύθυνση συμβολοσειράς*	
read_int	*5*		*$v0 = ακέραιος*
read_string	*8*	*$a0 = προσωρινή μνήμη* *$a1 = μήκος*	

Μέρος 3^{ο} : Επιπλέον Άσκηση

Ασκηση 1.6

Σε έναν πραγματικό επεξεργαστή MIPS μπορεί να επιλεχθεί κατά τη διάρκεια του hardware reset αν θα λειτουργεί βλέποντας τη μνήμη σαν big-endian ή little-endian. Ο προσομοιωτής QtSPIM όμως χρησιμοποιεί τον τρόπο που χρησιμοποιεί ο επεξεργαστής στον οποίο τρέχει η προσομοίωση. Στην περίπτωση επεξεργαστών Intel το μοντέλο είναι little-endian. Πώς θα αποθηκευόταν ο `$t6` στο βήμα (δ) της Άσκησης 1.2 στη μνήμη αν ο επεξεργαστής ακολουθούσε το μοντέλο big-endian; Συμπληρώστε τον πίνακα.

Οι επεξεργαστές μπορούν να αριθμήσουν τα byte στο εσωτερικό μιας λέξης έτσι, ώστε το byte με το μικρότερο αριθμό να είναι είτε το αριστερότερο είτε το δεξιότερο. Η σύμβαση που χρησιμοποιείται από μια μηχανή λέγεται σειρά byte (byte order). Οι επεξεργαστές MIPS μπορούν να λειτουργούν με σειρά byte είτε μεγάλου άκρου (big-endian) είτε μικρού άκρου (little-endian).

ΕΝΟΤΗΤΑ 2:

Αριθμητικές και Λογικές Πράξεις

Εισαγωγή

Σε αυτή την ενότητα θα μελετηθούν οι εντολές λογικών πράξεων, ολισθήσεων, περιστροφών στο πρώτο μέρος και οι εντολές αριθμητικών πράξεων μεταξύ ακεραίων στο δεύτερο (πρόσθεση) και τρίτο μέρος (πολλαπλασιασμός). Για τις αριθμητικές πράξεις πρόσθεσης-αφαίρεσης θα μελετηθεί επιπλέον η συμπεριφορά τους σε ακραίες περιπτώσεις, όπως για παράδειγμα όταν γίνεται υπερχείλιση. Επί πλέον, επειδή ο MIPS δε διαθέτει ειδικό καταχωρητή κατάστασης (status register) που να περιέχει ψηφία κρατουμένου και υπερχείλισης (C και V) για άμεση χρήση από εντολές, όπως άλλοι μικροεπεξεργαστές, μέρος των ασκήσεων περιλαμβάνει την παραγωγή των παραπάνω ψηφίων με προγραμματιστικό τρόπο. Στο τέταρτο μέρος (επιπλέον ασκήσεις) μπορείτε να βρείτε άλυτες ασκήσεις για τον υπολογισμό του κρατουμένου πρόσθεσης με λογικές πράξεις και για την εντολή διαίρεσης του MIPS.

Μέρος 1^{ο} : Λογικές Πράξεις και Ολισθήσεις

Άσκηση 2.1

Γράψτε ένα πρόγραμμα που θα μετατρέπει έναν δυαδικό αριθμό των 32-bit στον αντίστοιχό του σε κωδικοποίηση Gray.

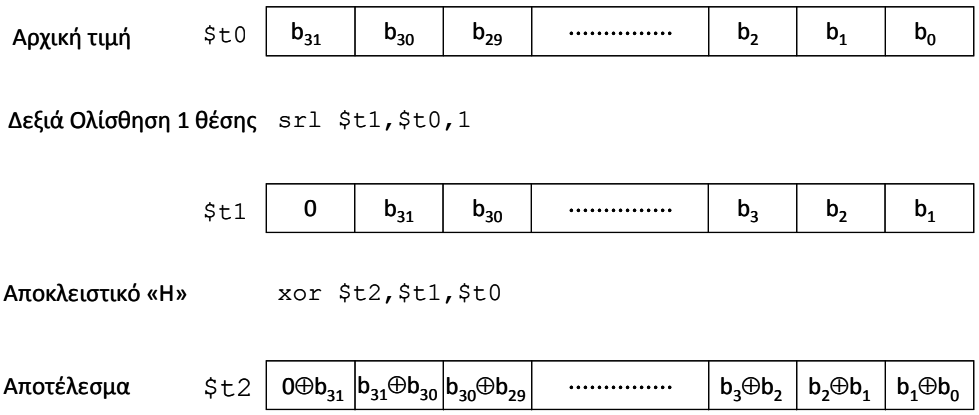
Ο αλγόριθμος μετατροπής ενός δυαδικού αριθμού n bits σε κωδικοποίηση Gray δίνεται παρακάτω, όπου b_i είναι το bit i-στής τάξης του δυαδικού αριθμού και g_i το bit i-στής τάξης του αντίστοιχου Gray αριθμού. Στη συγκεκριμένη περίπτωση για τον MIPS είναι n=32.

$g_n = b_n$

$g_i = b_{i+1}$ XOR b_i , για $i=0 \ldots n-1$

Binary	Hex	Gray
0000	0	0000
0001	1	0001
0010	2	0011
0011	3	0010
0100	4	0110
0101	5	0111
0110	6	0101
0111	7	0100
1000	8	1100
1001	9	1101
1010	A	1111
1011	B	1110
1100	C	1010
1101	D	1011
1110	E	1001
1111	F	1000

Χρησιμοποιήστε μόνο λογικές πράξεις και ολισθήσεις για να πραγματοποιήσετε τον παραπάνω αλγόριθμο όπως προτείνεται στο παρακάτω σχήμα. Ο αριθμός θα εισάγεται απ' ευθείας στον καταχωρητή `$t0`. Αυτό γίνεται με δεξί κλικ στο όνομα του καταχωρητή στο παράθυρο των καταχωρητών και την επιλογή Change Register Contents από την λίστα επιλογών που εμφανίζονται. Φροντίστε οι καταχωρητές να εμφανίζονται σε δεκαεξαδική μορφή ώστε να είναι εύκολη η ανάγνωσή τους σε δυαδικό σύστημα (επιλογή Registers → Hex). Πειραματιστείτε με μικρούς αριθμούς και συμβουλευτείτε τον παραπάνω πίνακα αντιστοίχισης για τον κώδικα Gray των 4 bit για να επιβεβαιώσετε το αποτέλεσμα, παρατηρώντας τον Gray κωδικοποιημένο αριθμό που θα βρίσκεται στο καταχωρητή `$t2`.



Λογικές Εντολές

AND - Λογικό «ΚΑΙ»

`and rd, rs, rt`

Τοποθέτηση του λογικού AND μεταξύ των καταχωρητών `rs` *και* `rt` *στον καταχωρητή* `rd`*.*

NOR – Λογικό «ΟΧΙ Η»

`nor rd, rs, rt`

Τοποθέτηση του λογικού NOR των καταχωρητών `rs` *και* `rt` *στον καταχωρητή* `rd`*.*

NOT – Λογικό «ΟΧΙ»

`not rdest, rsrc` *(ψευδοεντολή)*

Τοποθέτηση της κατά bit λογικής άρνησης του καταχωρητή `rsrc` *στον καταχωρητή* `rdest`*.*

OR – Λογικό «Η»

`or rd, rs, rt`

Τοποθέτηση του λογικού OR των καταχωρητών `rs` *και* `rt` *στον καταχωρητή* `rd`*.*

Exclusive OR — «Αποκλειστικό Η»

`xor rd, rs, rt`

Τοποθέτηση του λογικού XOR μεταξύ των καταχωρητών `rs` *και* `rt` *στον καταχωρητή* `rd`*.*

Εντολές Ολίσθησης

Shift left logical — Αριστερή λογική ολίσθηση

`sll rd, rt, shamt`

Shift right arithmetic — Αριθμητική δεξιά ολίσθηση

`sra rd, rt, shamt`

Shift right logical — Δεξιά λογική ολίσθηση

`srl rd, rt, shamt`

Ολίσθηση του καταχωρητή `rt` *αριστερά (δεξιά) κατά την απόσταση που δείχνει το άμεσο* `shamt` *ή ο καταχωρητής* `rs` *και τοποθέτηση του αποτελέσματος στον καταχωρητή* `rd`*.*

Εντολές Περιστροφής

Rotate left — Αριστερή περιστροφή

`rol rdest, rsrc1, rsrc2` *(ψευδοεντολή)*

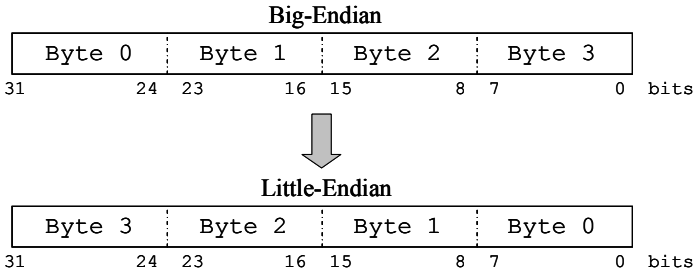
Rotate right — Δεξιά περιστροφή

`ror rdest, rsrc1, rsrc2` *(ψευδοεντολή)*

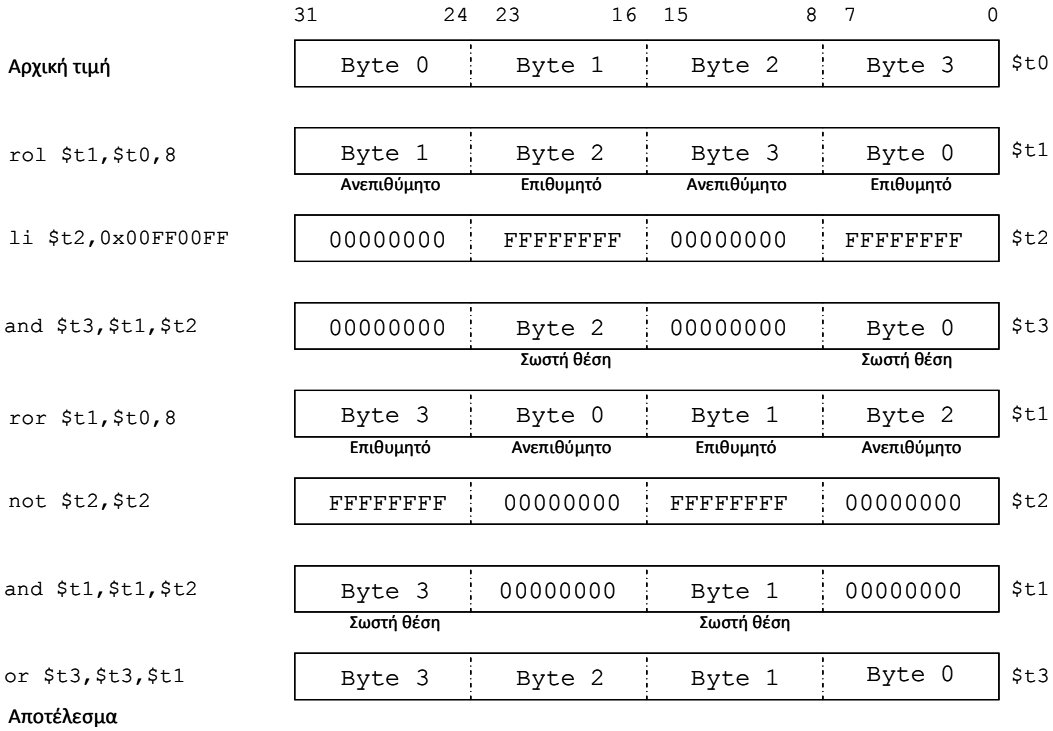
Περιστροφή του καταχωρητή `rsrc1` *αριστερά (δεξιά) κατά την απόσταση που ορίζει ο* `rsrc2` *και τοποθέτηση του αποτελέσματος στον καταχωρητή* `rdest`*.*

Άσκηση 2.2

Υποθέστε ότι ένας επεξεργαστής MIPS λειτουργεί χρησιμοποιώντας το μοντέλο μνημης little-endian (μικρού άκρου) για την οργάνωση των bytes (όπως ο QtSpim σε PC) και επικοινωνεί με διάφορες άλλες μηχανές κάποιες από τις οποίες χρησιμοποιούν το μοντέλο μνήμης big-endian (μεγάλου άκρου). Τα δεδομένα που προέρχονται από τις άλλες μηχανές είναι φυσικά λέξεις των 32 bit και εισάγονται σε έναν καταχωρητή όχι με την μορφή που τα χρειάζεται ο MIPS.
Θέλουμε να μετατρέψουμε τα δεδομένα από τη «λανθασμένη» μορφή big-endian με την οποία λαμβάνονται στη μορφή little-endian σε έναν άλλο καταχωρητή, όπως στο παρακάτω σχήμα. Να σημειωθεί ότι μέσα σε κάθε byte η σειρά των bit είναι σωστή (δηλ. το byte 0 αποθηκεύει τα bit 7-0, το byte 1 αποθηκεύει τα bit 15-8, κ.ο.κ)



Το παρακάτω σχήμα δείχνει μία από τις πολλές λύσεις που μπορούν να βρεθούν. Είναι μία ακολουθία εντολών μαζί με τα ενδιάμεσα αποτελέσματα που προκύπτουν στους διάφορους καταχωρητές που χρησιμοποιούνται.



Γράψτε το σχετικό πρόγραμμα μετατροπής, χωρίς να χρησιμοποιήσετε καθόλου προσπέλαση στη μνήμη για τη μετατροπή, παρά μόνο λογικές πράξεις και περιστροφές και αυτό για λόγους ταχύτητας επεξεργασίας. Τα δεδομένα με τη μορφή big-endian να δίνονται σαν αρχική τιμή στον καταχωρητή `$t0` χρησιμοποιώντας την εντολή `li` και το τελικό αποτέλεσμα της μορφής little-endian να υπολογίζεται στον καταχωρητή `$t3`.
Συμπληρώστε το αρχικό πρότυπο `lab2_2a.s` με την ακολουθία εντολών που φαίνονται στο παραπάνω σχήμα. Τρέξτε το πρόγραμμα μία-μία εντολή και κάθε φορά παρατηρήστε τις αλλαγές που συμβαίνουν στους καταχωρητές που εμπλέκονται κάθε φορά. Προσέξτε ότι κάποιες εντολές είναι ψευδοεντολές και απαιτούν περισσότερα από ένα βήματα για να ολοκληρωθούν.

```
################################################
# lab2 2a.s                                    #
# Big-Endian to Little-Endian (to be completed) #
################################################
      .text
      .globl __start
__start:                # execution starts here
      li $t0,0x12345678 # data to be converted supposed to be in $t0

      li $v0,10
      syscall                 # exit
```

Άσκηση 2.3

Το παρακάτω πρόγραμμα (`lab2_3a.s`) φορτώνει από τη μνήμη μία λέξη στον καταχωρητή `$t1` και δίνει σαν έξοδο μία τιμή στον καταχωρητή `$t2`. Αλλάξτε το πρόγραμμα έτσι ώστε να λειτουργεί ακριβώς όπως το αρχικό και να μην χρησιμοποιεί την εντολή `and` αλλά εντολές ολίσθησης και περιστροφής. Η ισοδυναμία των δύο προγραμμάτων θα πρέπει να ισχύει για οποιαδήποτε λέξη φορτώνεται στον καταχωρητή `$t1` από τη μνήμη (στο παράδειγμα η λέξη έχει τεθεί στην τιμή `0x12345678`).

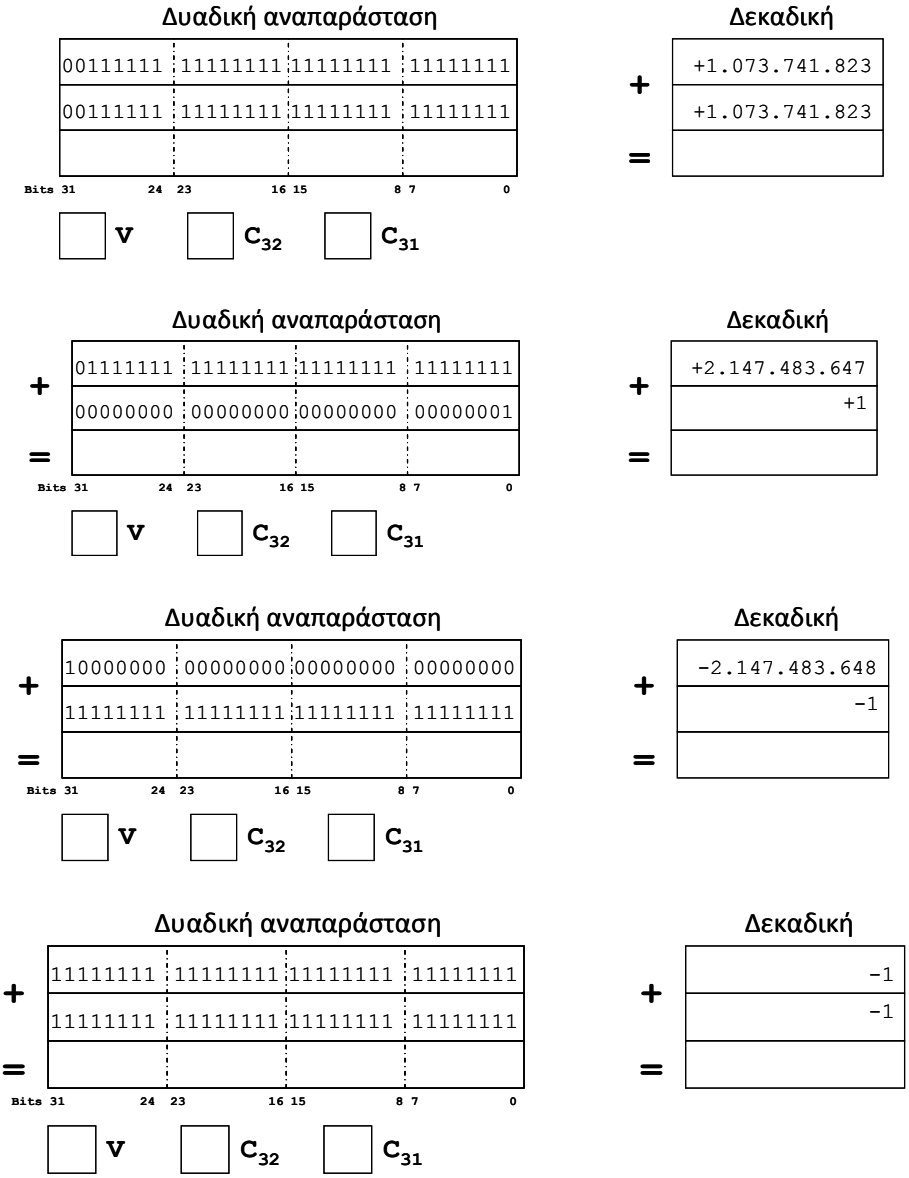
```
################################################
# lab2_3a.s                                    #
################################################
      .text
      .globl __start
__start:                    # execution starts here
      la $a0,input          # address of word to transform
      lw $t1,0($a0)         # load word to $t1
      li $t0,0xFFF00FFF     # mask definition
      and $t2,$t1,$t0       # masked word
      li $v0,10
      syscall
      .data
input:          .word 0x12345678
```

Μέρος 2ο: Πρόσθεση - Αφαίρεση - Υπερχείλιση - Κρατούμενο

Ασκηση 2.4

(α) Θεωρείστε τα παρακάτω ζευγάρια προσημασμένων δυαδικών αριθμών των 32-bit σε αναπαράσταση συμπληρώματος ως προς 2. Κάντε τις προσθέσεις που φαίνονται σε δυαδικό σύστημα και συμπληρώστε το αποτέλεσμα. Επίσης συμπληρώστε το σχετικό bit V της υπερχείλισης (1 αν έχει συμβεί και 0 αν δεν έχει συμβεί υπερχείλιση). Επί πλέον συμπληρώστε τα κρατούμενα που δίνει η πρόσθεση στα bit τελευταίας και προτελευταίας τάξης, δηλαδή τα C_{32} και C_{31} , αντίστοιχα. (Προσοχή, τα bit C_{32}, C_{31} και V δεν υπάρχουν στον MIPS αλλά είναι εδώ απλά συμβολισμός της άσκησης).

(β) Κάντε την αντίστοιχη πράξη σε δεκαδικό σύστημα και ελέγξτε κατά πόσον το αποτέλεσμα είναι ίδιο με αυτό της δυαδικής αναπαράστασης.



(γ) Εκτελέστε τις ίδιες πράξεις στο περιβάλλον προσομοίωσης QtSpim χρησιμοποιώντας τις εντολές του MIPS `add` (add signed) και `addu` (add unsigned). Συμπληρώστε το πρόγραμμα `lab2_4a.s` βάζοντας εναλλάξ μία από τις δύο εντολές πρόσθεσης στο κατάλληλο σημείο. Τους ακεραίους να τους αλλάζετε κάθε φορά στο data segment (δηλ. μετά την οδηγία `.data`) και το αποτέλεσμα της άθροισης να το παρατηρείτε ως δεκαεξαδικό αριθμό στα περιεχόμενα των καταχωρητών, αλλά και σαν δεκαδικό αριθμό στο παράθυρο console (κλήση συστήματος `print_int`). Ποιες διαφορές έχουν οι δύο αυτές εντολές; Πως φαίνεται αν έχει γίνει υπερχείλιση;

```
################################################
# lab2_4.s                                     #
# Signed vs Unsigned addition (to be completed) #
################################################
     .text
     .globl   start
__start:                    # execution starts here
      la $a1,start_int      # address of first integer
      lw $t1,0($a1)         # load first integer in $t0
      lw $t2,4($a1)         # load second integer in $t1

      li $v0,1
      syscall               # print sum ($a0)
      li $v0,10
      syscall               # exit

      .data
start_int:  .word 1073741823  # first integer
            .word 1073741823  # second integer
```

Εντολές Πρόσθεσης

Addition (with overflow) — Πρόσθεση (με υπερχείλιση)

`add rd, rs, rt`

Addition (without overflow) — Πρόσθεση (χωρίς υπερχείλιση)

`addu rd, rs,rt`

Τοποθέτηση του αθροίσματος των καταχωρητών `rs` *και* `rt` *στον καταχωρητή* `rd`

Ασκηση 2.5

Ο MIPS δεν προσφέρει κάποιο bit ανίχνευσης υπερχείλισης, όπως άλλοι μικροεπεξεργαστές. Η ανίχνευση της υπερχείλισης επιτυγχάνεται με τη δημιουργία εσωτερικής διακοπής ή εξαίρεσης (exception). Παρακάτω υπάρχει ένα παράδειγμα (από το βιβλίο) ανίχνευσης υπερχείλισης αποφεύγοντας τη δημιουργία εσωτερικής διακοπής.

Παράδειγμα : *Ο MIPS μπορεί να παγιδεύσει την υπερχείλιση αλλά, σε αντίθεση με πολλούς άλλους υπολογιστές, δεν υπάρχει διακλάδωση υπό συνθήκη για τον έλεγχο της υπερχείλισης. Μια ακολουθία εντολών του MIPS μπορεί να εντοπίσει την υπερχείλιση. Για την προσημασμένη πρόσθεση, η ακολουθία είναι η παρακάτω*

```
addu $t0, $t1,$t2                 # $t0 =sum, αλλά όχι παγίδα
xor  $t3, $t1,$t2                 # Έλεγχος αν διαφέρουν τα πρόσημα
slt  $t3, $t3, $zero              # $t3 = 1 αν διαφέρουν τα πρόσημα
bne  $t3, $zero, No_overflow      # πρόσημα $t1,$t2 διάφορα,  άρα όχι υπερχείλιση
xor  $t3, $t0, $t1                # πρόσημα ίσα˙ πρόσημο αθροίσματος
                                  # ταιριάζει επίσης $t3 αρνητικός αν
                                  # πρόσημο αθροίσματος διαφέρει
slt  $t3, $t3, $zero              # $t3 = 1 αν πρόσημο αθροίσματος  διαφορετικό
bne  $t3, $zero, Overflow         # Και τα τρία πρόσημα διάφορα˙μετάβαση σε υπερχείλιση
```

(α) Κάντε ένα ανάλογο πρόγραμμα που μπορεί να ανιχνεύει υπερχειλίσεις δύο προσημασμένων αριθμών που πρόκειται να προστεθούν χωρίς όμως να χρησιμοποιήσετε υπό συνθήκη άλματα (conditional branches) όπως στο παράδειγμα του βιβλίου, παρά μόνο λογικές πράξεις μεταξύ καταχωρητών. Το bit υπερχείλισης να αποθηκεύεται σε ένα bit κάποιου καταχωρητή.
Χρησιμοποιήστε τον ορισμό της υπερχείλισης σύμφωνα με τον οποίο αν προστεθούν δύο προσημασμένοι αριθμοί A και B δίνοντας άθροισμα S=A+B, τότε υπάρχει υπερχείλιση αν ο A είναι ομόσημος με τον B και ταυτόχρονα το άθροισμα είναι ετερόσημο με τον A (ή ισοδύναμα τον B, αφού A και B είναι ομόσημοι).
Δηλαδή το bit υπερχείλισης V θα δίνεται από τη λογική συνάρτηση V= [not (A_{31} xor B_{31})] and (A_{31} xor S_{31}), όπου A_{31}, B_{31} και S_{31} είναι τα πιο σημαντικά bits του A, B και S, αντίστοιχα και δηλώνουν το πρόσημο (0 για θετικό ακέραιο και 1 για αρνητικό).
Συμπληρώστε το πρόγραμμα `lab2_5a.s` το οποίο φορτώνει από τη μνήμη τους δύο ακεραίους A και B που πρόκειται να προστεθούν στους καταχωρητές `$t1` και `$t2`, αντίστοιχα και υπολογίζει το άθροισμα S στον καταχωρητή `$a0` προκειμένου να εμφανιστεί στην κονσόλα από την κλήση `print_int`, έτσι ώστε σε κάποιο bit ενός καταχωρητή που θα επιλέξετε να υπολογίζεται το bit υπερχείλισης. Γιατί χρησιμοποιείται η εντολή `addu` αντί της `add` αφού πρόκειται για προσημασμένους αριθμούς;

```
################################################
# lab2_5a.s                                    #
# Overflow generation (to be completed)        #
################################################
      .text
      .globl   __start
__start:                    # execution starts here
      la $a1,start_int      # address of first integer
      lw $t1,0($a1)         # load first integer in $t0
      lw $t2,4($a1)         # load second integer in $t1
      addu $a0,$t1,$t2      # sum in $a0

      li $v0,1
      syscall               # print sum ($a0)
      li $v0,10
      syscall               # exit

      .data
start_int:.word 2147483000  # first integer
          .word 1000        # second integer
```

εξαίρεση (exception) *Ονομάζεται και διακοπή (interrupt). Ένα μη προγραμματισμένο συμβάν που διακόπτει τον κανονικό ρυθμό της εκτέλεσης του προγράμματος· χρησιμοποιείται γιατην ανίχνευση της υπερχείλισης*

διακοπή (interrupt) *Εξαίρεση που προέρχεται από το εξωτερικό του επεξεργαστή. (Μερικές αρχιτεκτονικές χρησιμοποιούν τον όρο διακοπή για όλες τις εξαιρέσεις.)*

(β) Σε άλλους μικροεπεξεργαστές η σημαία υπερχείλισης (overflow flag) παράγεται από τα κρατούμενα τελευταίας και προτελευταίας τάξης που παράγει ο αθροιστής τους. Παρατηρήστε τα αποτελέσματα που πήρατε στην προηγούμενη άσκηση και βρείτε ποια λογική συνάρτηση υπολογίζει την υπερχείλιση.

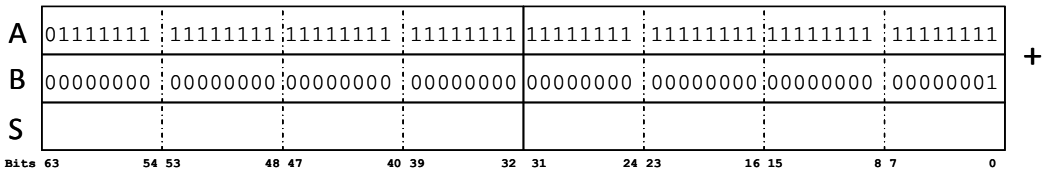
Ασκηση 2.6

Στην άσκηση αυτή θα δημιουργήσετε ένα πρόγραμμα που θα προσθέτει δύο προσημασμένους ακεραίους αριθμούς A και B των 64 bit. Αρχικά οι αριθμοί θα βρίσκονται στο data segment και η διάταξή τους θα ακολουθεί το μοντέλο little–endian, δηλαδή η λέξη με τα λιγότερο σημαντικά bit θα βρίσκεται σε χαμηλότερη διεύθυνση, όπως φαίνεται και στο παρακάτω σχήμα. Το αποτέλεσμα της άθροισης S θα αποθηκεύεται και αυτό στη μνήμη, όπως φαίνεται στο παρακάτω σχήμα. Η άθροιση των 64 bit θα γίνεται με εκμετάλλευση της εντολής άθροισης ακεραίων 32 bit που προσφέρει ο MIPS.

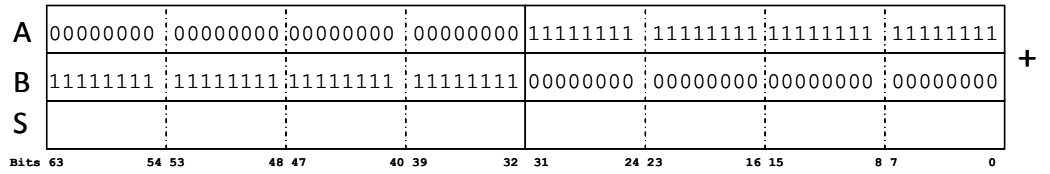
Data Segment	Byte and Word Address
⋮	
S (high bits 63..32)	0x01001014
S (low bits 31..0)	0x01001010
B (high bits 63..32)	0x0100100C
B (low bits 31..0)	0x01001008
A (high bits 63..32)	0x01001004
A (low bits 31..0)	0x01001000

(α) Αρχικά υπολογίστε τα δύο αθροίσματα των ζευγών ακεραίων των 64 bit στους δύο επόμενους πίνακες προκειμένου να διαπιστώσετε την ανάγκη για υπολογισμό του κρατούμενου στην άθροιση των χαμηλών 32 bits.

Δυαδική αναπαράσταση



Δυαδική αναπαράσταση



(β) Συμπληρώστε το πρόγραμμα `lab2_6a.s` έτσι ώστε να εκτελεί την πρόσθεση των δύο ακεραίων 64 bit που φορτώνονται στους καταχωρητές `$t0`,`$t1`,`$t2`,`$t3` (σε τμήματα των 32 bit) από τη μνήμη. Χρησιμοποιήστε εντολές πρόσθεσης των 32 bit ώστε τα χαμηλά bit του αθροίσματος να υπολογίζονται στον καταχωρητή `$a0`, ενώ τα υψηλά στον `$a1`. Προκειμένου να υπολογίσετε το απαιτούμενο κρατούμενο που απαιτείται από την πρόσθεση των χαμηλών bit μπορείτε να εκμεταλλευτείτε την εντολή `sltu` σε συνδυασμό με το γεγονός ότι σε μία πρόσθεση απρόσημων ακεραίων δημιουργείται τελικό κρατούμενο αν το άθροισμα είναι μικρότερο από τον προσθέτη και τον προσθετέο.

Εντολή Απρόσημης Σύγκρισης

Set less than unsigned — Θέσε όταν μικρότερος

`sltu rd, rs, rt`

Εξίσωση του καταχωρητή `rd` με 1 αν ο καταχωρητής `rs` είναι μικρότερος από

τον `rt`, και με 0 διαφορετικά.

Αφού συμπληρώσετε το πρόγραμμα δοκιμάστε τα ζεύγη αριθμών που δόθηκαν στους πίνακες παραπάνω και διαπιστώστε την ορθή λειτουργία του. (Οι τιμές εισόδου αφορούν το πρώτο από τα δύο παραδείγματα).

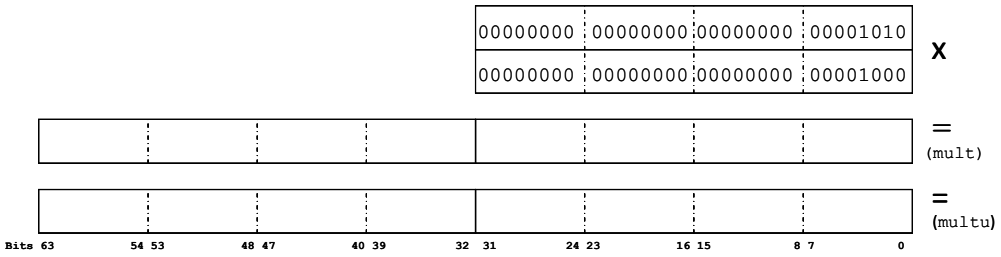
```
################################################
# lab2_6a.s                                    #
# 64 bits addition (to be completed)           #
################################################
      .text
     .globl   start
__start:                            # execution starts here
      la $a3,start_int              # address of first integer
      lw $t0,0($a3)                 # load low  bits of first integer in $t0
      lw $t1,4($a3)                 # load high bits of first integer in $t1
      lw $t2,8($a3)                 # load low  bits of second integer in $t2
      lw $t3,12($a3)                # load high bits of first integer in $t3

      sw $a0,16($a3)                # store low bits of sum in memory
      sw $a1,20($a3)                # store high bits of sum in memory
      li $v0,10
      syscall                       # exit

      .data
start_int: .word 0xFFFFFFFF         # low  bits of first integer
           .word 0x7FFFFFFF         # high bits of first integer
           .word 0x00000001         # low  bits of second integer
           .word 0x00000000         # high bits of second integer
```

Μέρος 3ο : Πολλαπλασιασμός

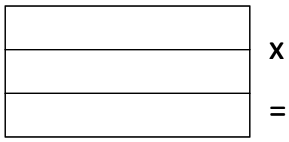
Ασκηση 2.7

Σκοπός αυτής της άσκησης είναι να δείξουμε τη διαφορά μεταξύ των δύο πραγματικών εντολών πολλαπλασιασμού ακεραίων `mult` (multiply with sign) και `multu` (multiply unsigned) του MIPS, δηλαδή ο διαφορετικός τρόπος με τον οποίο βλέπουν τους τελεστέους αλλά και η διαφορά στο αποτέλεσμα που παράγουν.

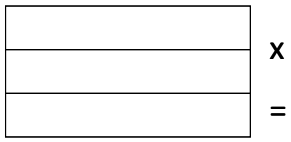
(α) Εκτελέστε τους δύο πολλαπλασιασμούς στους δύο παρακάτω πίνακες σε δυαδικό σύστημα και δεκαδικό σύστημα. Για το κάθε ζεύγος αριθμών εκτελέστε δύο πολλαπλασιασμούς, τον έναν για την περίπτωση που οι ακέραιοι θεωρούνται απρόσημοι και τον άλλο για την περίπτωση που θεωρούνται προσημασμένοι.



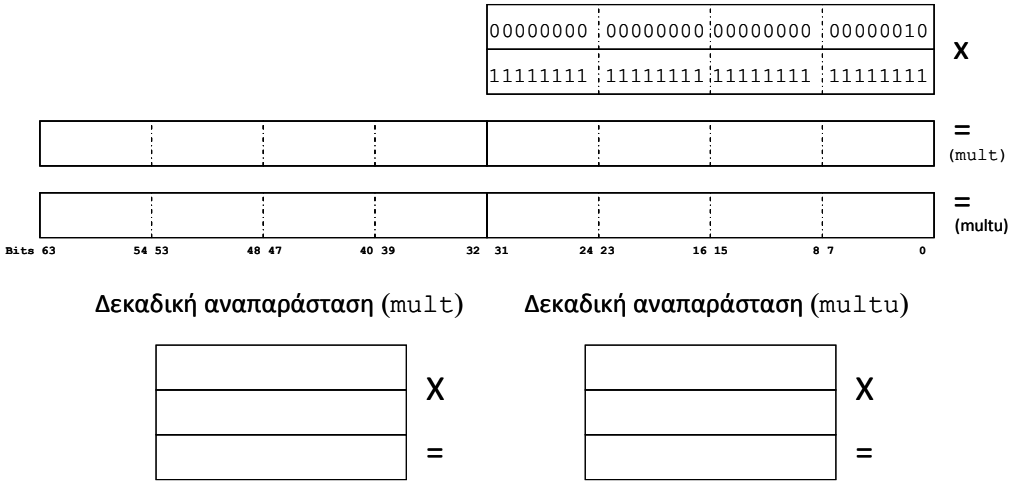
Δεκαδική αναπαράσταση (`mult`)



Δεκαδική αναπαράσταση (`multu`)



Δυαδική αναπαράσταση



Δεκαδική αναπαράσταση (`mult`) Δεκαδική αναπαράσταση (`multu`)

(β) Συμπληρώστε το πρόγραμμα lab2_7a.s έτσι ώστε να εκτελεστούν οι ίδιοι πολλαπλασιασμοί με απρόσημο τρόπο (χρησιμοποιώντας την εντολή multu) και με προσημασμένο τρόπο. Μεταφέρετε τα χαμηλά και υψηλά bits του γινομένου στους καταχωρητές $t2 και $t3, χρησιμοποιώντας τις εντολές `mflo` (move from lo) και `mfhi` (move from hi), αντίστοιχα. Συγκρίνετε τα αποτελέσματα που πήρατε με τους υπολογισμούς που κάνατε στο προηγούμενο βήμα.

```
################################################
# lab2_7a.s                                    #
# 32 x 32 bits multiplication (to be completed) #
################################################
      .text
      .globl __start
__start:                    # execution starts here
      la $a1,start_int      # address of first integer
      lw $t0,0($a1)         # load first integer in $t0
      lw $t1,4($a1)         # load second integer in $t1

      li $v0,10
      syscall               # exit
      .data
start_int: .word 0x0000000A   # first integer
           .word 0x00000008   # second integer
```

Εντολές Πολλαπλασιασμού

Multiply — Πολλαπλασιασμός

mult rs, rt

Unsigned multiply — Απρόσημος πολλαπλασιασμός

`multu rs, rt`

Πολλαπλασιασμός των καταχωρητών `rs` και `rt`. Η λέξη χαμηλής τάξης του γινομένου μένει στον καταχωρητή `lo` και η λέξη υψηλής τάξης στον καταχωρητή `hi`.

Εντολές Μετακίνησης από τους Καταχωρητές `HI` και `LO`

Move from hi — Μεταφορά από τον καταχωρητή `hi`

`mfhi rd`

Move from lo — Μεταφορά από τον καταχωρητή `lo`

`mflo rd`

Η μονάδα πολλαπλασιασμού και διαίρεσης παράγει το αποτέλεσμά της σε δύο επιπλέον καταχωρητές, τους `hi` και `lo`. Αυτές οι εντολές μεταφέρουν τιμές προς και από αυτούς τους καταχωρητές. Οι ψευδοεντολές πολλαπλασιασμού, διαίρεσης, και υπολοίπου, οι οποίες κάνουν τη μονάδα αυτή να φαίνεται ότι λειτουργεί σε γενικούς καταχωρητές, μεταφέρουν το αποτέλεσμα μετά την ολοκλήρωση του υπολογισμού.

Μεταφορά τού καταχωρητή `hi` (`lo`) στον καταχωρητή `rd`.

Μέρος 4^{ο} : Επί πλέον ασκήσεις

Ασκηση 2.8

Το ζητούμενο αυτής της άσκησης είναι ένας τρόπος παραγωγής του κρατουμένου της πρόσθεσης ακεραίων των 32 bit στον MIPS, επειδή αυτό δεν προσφέρεται από τον συγκεκριμένο μικροεπεξεργαστή. Η παραγωγή του κρατουμένου θα γίνεται αποκλειστικά με λογικές πράξεις (χωρίς την εντολή `sltu` που χρησιμοποιήθηκε στην Άσκηση 2.6). Υποθέτουμε ότι ο αθροιστής της Αριθμητικής Λογικής Μονάδας του MIPS αποτελείται από 32 πλήρεις αθροιστές του ενός bit συνδεδεμένους σε δομή κρατούμενου ριπής (ripple carry adder). Ο πίνακας αληθείας ενός πλήρους αθροιστή για το bit i δίνεται παρακάτω.

ΕΙΣΟΔΟΙ			**ΕΞΟΔΟΙ**	
C_i	A_i	B_i	C_{i+1}	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

όπου A_i και B_i είναι τα δύο bit i-οστης τάξης που πρόκειται να προστεθούν, C_i είναι το κρατούμενο που δίνει ο αθροιστής της προηγούμενης τάξης. S_i είναι το άθροισμα και C_{i+1} το κρατούμενο που παράγει ο αθροιστής που εξετάζουμε.

(α) Βρείτε τον πίνακα αλήθειας του C_{i+1} όταν είναι γνωστά τα A_i , B_i και S_i (αντί για το C_i). Ποια είναι η προϋπόθεση για να υπάρχει τέτοιος πίνακας ;

ΕΙΣΟΔΟΙ			**ΕΞΟΔΟΣ**
Si	A_i	B_i	C_{i+1}
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

(β) Κατόπιν γράψτε ένα πρόγραμμα στο QtSpim που να υπολογίζει το τελικό κρατούμενο, δηλαδή το 33^{ο} bit, μίας πρόσθεσης (C_{32}) μεταξύ δύο καταχωρητών και να το αποθηκεύει σε ένα bit κάποιου άλλου καταχωρητή. Οι τελεστέοι της πρόσθεσης να εισάγονται μέσω του data segment.

Άσκηση 2.9

Χρησιμοποιήστε το πρόγραμμα `lab2_9.s` που ακολουθεί στο τέλος της άσκησης για να συμπληρώσετε τον παρακάτω πίνακα διαιρέσων. Το πρόγραμμα κάθε φορά που εκτελείται δέχεται από το παράθυρο console δύο ακέραιους προσημασμένους αριθμούς (κλήση `read_int`) και τους διαιρεί με την εντολή προσημασμένης διαίρεσης `div`. Τα αποτελέσματα (πηλίκο και υπόλοιπο) εμφανίζονται στο ίδιο παράθυρο (κλήση συστήματος `print_int`).
Πειραματιστείτε με τους παρακάτω δεκαδικούς αριθμούς, ελέγξτε την ορθότητα των αποτελεσμάτων και διαπιστώστε πότε λαμβάνετε κάποιο μήνυμα σφάλματος από τον προσομοιωτή ή δημιουργείται κάποια εσωτερική διακοπή.

ΔΙΑΙΡΕΤΕΟΣ	ΔΙΑΙΡΕΤΗΣ	ΠΗΛΙΚΟ	ΥΠΟΛΟΙΠΟ
13	3		
13	-3		
-13	3		
-13	-3		
13	0		
-2147483648	-1		
-2147483648	1000		

Εντολή Προσημασμένης Διαίρεσης

Divide (with overflow) — Διαίρεση (με υπερχείλιση)

`div rs, rt`

Διαίρεση του καταχωρητή `rs` με τον καταχωρητή `rt`. Το πηλίκο μένει στον καταχωρητή `lo` και το υπόλοιπο στον καταχωρητή `hi`. Σημειώστε ότι, αν ένας τελεστέος είναι αρνητικός, το υπόλοιπο δεν προσδιορίζεται από την αρχιτεκτονική του MIPS και εξαρτάται από τη σύμβαση της μηχανής στην οποία εκτελείται ο SPIM.

```
################################################
# lab2_9.s                                     #
# 32 / 32 bits  division                       #
################################################
      .text
      .globl __start
__start:                    # execution starts here
      la $a0,dividend
      li $v0,4
      syscall                # prompt for dividend
      li $v0,5
      syscall                # read dividend
      move $t0,$v0           # dividend in $t0
      la $a0,endl
      li $v0,4
      syscall                # newline
      la $a0,divisor
      li $v0,4
      syscall                # prompt for divisor
      li $v0,5
      syscall                # read divisor
      move $t1,$v0           # divisor in $t1
      div $t0,$t1            # make the division $t0/$t1
      mflo $t4               # move quotient
      mfhi $t5               # move remainder
      la $a0,endl
      li $v0,4
      syscall                # newline
      la $a0,quotient
      li $v0,4
      syscall                # display "quotient is :"
      move $a0,$t4
      li $v0,1
      syscall                # display quotient
      la $a0,endl
      li $v0,4
      syscall                # newline
      la $a0,remainder
      li $v0,4
      syscall                # display "remainder is :"
      move $a0,$t5
      li $v0,1
      syscall                # display remainder
      la $a0,endl
      li $v0,4
      syscall                # newline
      li $v0,10
      syscall                # exit
      .data
dividend:    .asciiz "Enter dividend:"
divisor:     .asciiz "Enter divisor:"
endl:        .asciiz "\n"
quotient:    .asciiz "quotient is :"
remainder:   .asciiz "remainder is :"
```

ΕΝΟΤΗΤΑ 3:

Έλεγχος Ροής Προγράμματος

Εισαγωγή

Αντικείμενο αυτής της ενότητας είναι η χρήση των εντολών διακλάδωσης υπό συνθήκη (conditional branch) και απλής διακλάδωσης (unconditional branch) που διαθέτει ο MIPS, προκειμένου να επιτευχθούν γνωστές δομές ελέγχου ροής, που συναντάμε σε γλώσσες υψηλού επιπέδου όπως διακλαδώσεις if – then – else, case, βρόχοι for, while, until κ.τ.λ. Το πρώτο μέρος αφορά τη δομή if-then-else, ενώ το δεύτερο μέρος αφορά βρόχους επανάληψης.

Μέρος 1^{ο} : Διακλαδώσεις Υπό Συνθήκη

Ασκηση 3.1

Σκοπός αυτής της άσκησης είναι η κατασκευή ενός προγράμματος σε συμβολική γλώσσα MIPS που θα υλοποιεί μία δομή τύπου if-then-else μίας γλώσσας υψηλού επιπέδου. Το πρόγραμμα θα πρέπει να ελέγχει τη διάταξη δύο ακεραίων αριθμών που αρχικά βρίσκονται στη μνήμη μεταφορτώνοντάς τους σε δύο καταχωρητές. Συγκεκριμένα, θα πρέπει να εξετάζει *αν ο δεύτερος αριθμός είναι μεγαλύτερος του πρώτου* (if – then) και να τυπώνει στην κονσόλα σχετικό μήνυμα, *αλλιώς* (else) να τυπώνει ένα διαφορετικό μήνυμα.
Ένα αντίστοιχο τμήμα προγράμματος σε γλώσσα C θα ήταν το παρακάτω (`x1` είναι η μεταβλητή που περιέχει τον πρώτο αριθμό και `x2` η μεταβλητή που περιέχει το δεύτερο):

```
if (x2>x1) then {
                printf("Second number greater than first\n");
}
else {
                printf("Second number NOT greater than first\");
}
```

Κατασκευάστε το ζητούμενο χρησιμοποιώντας το παρακάτω πρότυπο προγράμματος σε assembly που θεωρεί ότι οι δύο αριθμοί είναι αποθηκευμένοι στο data segment και τυπώνει τα δύο μηνύματα. Συμπληρώστε το με τις εντολές `bgt` και `b` και χρησιμοποιώντας όπου χρειάζεται ετικέτες στο text segment. Το πρότυπο θα το βρείτε και στην ιστοσελίδα του μαθήματος με το όνομα αρχείου `lab3_1a.s`. Παρατηρήστε πως μεταφράζονται οι ψευδοεντολές `bgt` και `b` . Εκτελέστε το πρόγραμμα δοκιμάζοντας την ορθή λειτουργία του εξετάζοντας τις δύο διαφορετικές περιπτώσεις διάταξης των αριθμών.

Εντολές Διακλάδωσης

Branch on greater than — Διακλάδωση αν μεγαλύτερο

`bgt rsrc1, src2, label` (ψευδοεντολή)

Branch instruction — Εντολή διακλάδωσης

`b label` (ψευδοεντολή)

Διακλάδωση χωρίς συνθήκη στην εντολή που βρίσκεται στην ετικέτα `label`.

```
################################################
# lab3_1a.s                                    #
# if-then-else    (to be completed)            #
################################################
      .text
      .globl   start
__start:                    # execution starts here
      la $a0,num1           # load address of first integer
      lw $t0,($a0)          # load first integer into $t0
      la $a0,num2           # load address of second integer
      lw $t1,($a0)          # load second integer into $t1

      la $a0,mesg1          # Print
      li $v0,4              # "Second number NOT greater than first"
      syscall               #

n2Gn1: la $a0,mesg2         # Print
      li $v0,4              # "Second number greater than first"
      syscall               #

      li $v0,10
      syscall               # exit
      .data
num1:       .word 15
num2:       .word 25
mesg1:            .asciiz "Second number NOT greater than first\n"
mesg2:            .asciiz "Second number greater than first\n"
```

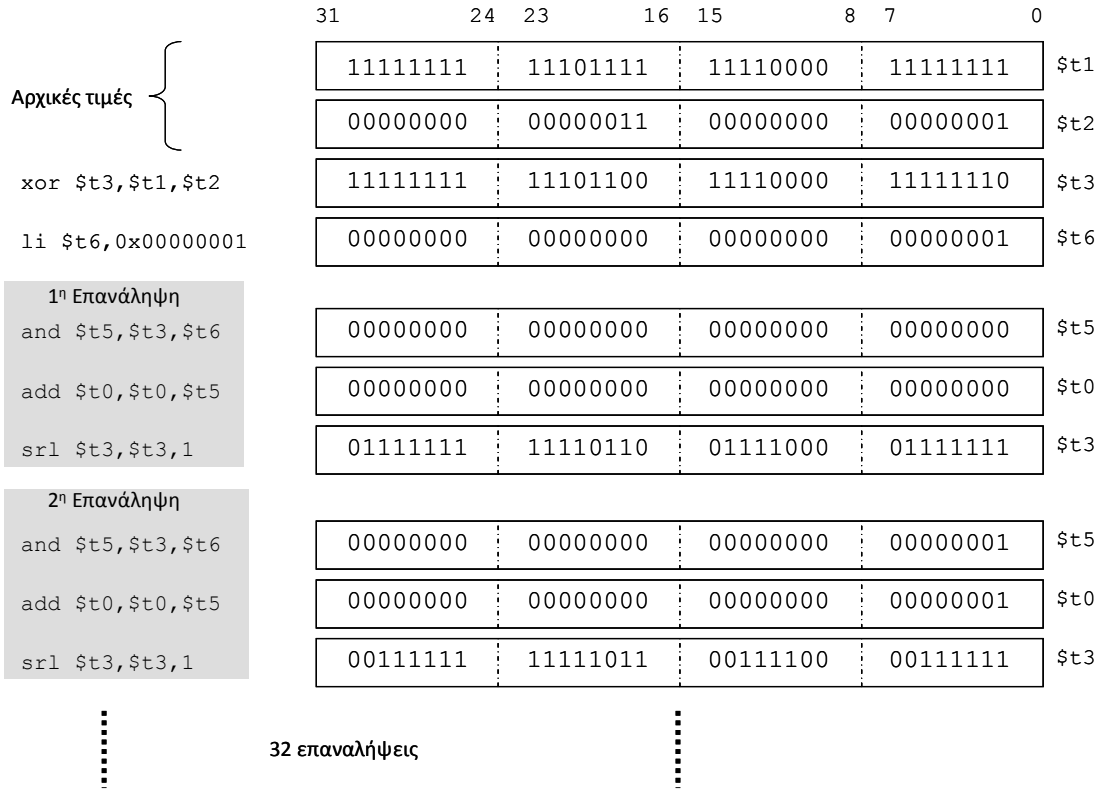
Μέρος 2^{ο} : Βρόχοι

Ασκηση 3.2

Η απόσταση Hamming μεταξύ δύο δυαδικών λέξεων είναι ο συνολικός αριθμός των θέσεων που διαφέρουν τα bit τους. Για παράδειγμα, η απόσταση Hamming των λέξεων (των 8 bit) 11001010 και 11010010 είναι 2 γιατί οι δύο λέξεις έχουν διαφορετικά bit στις θέσεις 4 και 5 (ξεκινώντας την αρίθμηση των bit με 0 από τα δεξιά).

Γράψτε ένα πρόγραμμα που θα υπολογίζει την απόσταση Hamming μεταξύ δύο λέξεων των 32 bit που είναι αποθηκευμένες στη μνήμη. Ένας τρόπος υπολογισμού της απόστασης Hamming περιγράφεται στο επόμενο σχήμα. Αν υποτεθεί ότι οι δύο λέξεις φορτώνονται αρχικά στους καταχωρητές `$t1` και `$t2`, τότε ένα αποκλειστικό «Η» μεταξύ τους στον καταχωρητή `$t3` θα έχει ως αποτέλεσμα ο αριθμός των άσσων στον `$t3` να δίνει την απόσταση Hamming. Επομένως χρειάζεται να μετρηθεί ο αριθμός των άσσων του `$t3`. Αυτό μπορεί να γίνει με ένα βρόχο επανάληψης 32 φορών όπου κάθε φορά θα λαμβάνεται υπόψη μόνο το bit0 του `$t3` (το LSB) μέσω της χρήσης μάσκας με τιμή 0x00000001. Το αποτέλεσμα θα προστίθεται σε έναν καταχωρητή (π.χ. τον `$t0`) που έχει αρχικά τιμή 0 και τέλος θα γίνεται μία δεξιά ολίσθηση του `$t3` κατά μία θέση, ώστε να έρθει στην θέση του bit0 του `$t3` το επόμενο bit προς έλεγχο. Έτσι, με 32 επαναλήψεις της παραπάνω διαδικασίας, θα έχουν προστεθεί στον `$t0` όλοι οι πιθανοί άσσοι του `$t3`, και τελικά αυτός θα περιέχει τη ζητούμενη απόσταση Hamming.

Χρησιμοποιήστε σαν σημείο εκκίνησης το πρόγραμμα `lab2_3a.s` το οποίο φορτώνει τους καταχωρητές `$t1` και `$t2` με τις δύο λέξεις που βρίσκονται αρχικά στη μνήμη, κάνει κάποιες από τις απαιτούμενες αρχικοποιήσεις (`xor`, ορισμός μάσκας, μηδενισμός `$t0`) και εμφανίζει το περιεχόμενο του `$t0` στην κονσόλα υποθέτωντας ότι σε αυτόν τον καταχωρητή θα έχει υπολογιστεί η απόσταση Hamming.




```
################################################
# lab3_2a.s                                    #
# Hamming Distance loop (to be completed)      #
################################################
      .text
      .globl __start
__start:                    # execution starts here
      la $a1,word1
      lw $t1,0($a1)         # load word 1 into $t1
      la $a1,word2
      lw $t2,0($a1)         # load word 2 into $t2
      xor $t3,$t1,$t2       # number of 1s in $t2 is the Hamming distance
      li $t0,0              # Hamming distance initially is 0
      li $t6,0x00000001     # mask used to retain only LSB of $t3

      la $a0,answer
      li $v0,4
      syscall               # print "Hamming distance is"
      move $a0,$t0
      li $v0,1
      syscall               # print the Hamming distance contained in $t0
      la $a0,endl
      li $v0,4
      syscall               # print "\n"
EXIT: li $v0,10
      syscall               # exit
      .data
word1:            .word 0xFFEFF0FF  # word 1
word2:            .word 0x00030001  # word 2
answer:           .asciiz "Hamming distance is : "
endl:             .asciiz "\n"
```

Εντολές Διακλάδωσης

Branch on not equal — Διακλάδωση αν διάφορο

`bne rs, rt, label`

Διακλάδωση υπό συνθήκη κατά τον αριθμό των εντολών που καθορίζονται από τη σχετική απόσταση (offset) αν ο καταχωρητής rs δεν είναι ίσος με τον `rt`.

Ασκηση 3.3

Δίνεται το παρακάτω πρόγραμμα με όνομα `lab3_3a.s`. Η λειτουργία του είναι να διαβάζει το string που είναι αποθηκευμένο στη μνήμη (οδηγία `.asciiz`) και να το αντιγράφει σε μία άλλη θέση μνήμης που έχει δεσμευτεί (οδηγία `.space 80`) από την αρχή του μέχρι την πρώτη εμφάνιση ενός χαρακτήρα που φαίνεται στη γραμμή `li $s0,'t'`. Ουσιαστικά πρόκειται για ένα βρόχο τύπου while με συνθήκη *«ο τρέχων χαρακτήρας που διαβάζεται δεν είναι ο 't'»*.

```
################################################
# lab3_3a.s                                    #
# while loop      char!=ASCII 0                #
################################################
      .text
      .globl __start
__start:                          # execution starts here

      li $t1,0                    # counter for string
      li $s0,'t'                  # chararacter to end copy
while: lbu $t0,string($t1)        # load a character
      beq $t0,$s0,end             # if character to end copy then exit loop
      sb $t0,copy($t1)            # copy character
      addi $t1,$t1,1              # increment counter
      b while                     # repeat while loop
end:  li $t2,0
      sb $t2,copy($t1)            # append end character to copied string
      la $a0,copy                 # display copy
      li $v0,4
      syscall
      li $v0,10                   # exit
      syscall
      .data
string:     .asciiz "Mary had a little lamb"
copy:       .space 80
```

Το ζητούμενο είναι να τροποποιήσετε το πρόγραμμα έτσι ώστε ο βρόγχος while να μετατραπεί σε : *while «ο τρέχων χαρακτήρας που διαβάζεται δεν είναι ο 't' **ΚΑΙ** δεν έχουν διαβαστεί πάνω από n χαρακτήρες»*, όπου το n είναι μία σταθερά που είναι αποθηκευμένη σε έναν καταχωρητή. Εκτελέστε το πρόγραμμα για διάφορες τιμές του n (π.χ. n=6 και n=17) ώστε να διαπιστώσετε την ορθή λειτουργία του.
Ένα αντίστοιχο τμήμα προγράμματος σε γλώσσα C θα ήταν το παρακάτω (`s` είναι ο πίνακας που περιέχει το string που πρόκεται να αντιγραφεί και `d` ο πίνακας στον οποίο θα αντιγραφεί το string):

```
i=0;
while (s[i]!='t') && (i<=20)) {
           d[i]=c[i];
           i++;
}
```

Υπόδειξη: Μπορείτε να χρησιμοποιήσετε ψευδοεντολές για τα υπό συνθήκη άλματα που θα χρειαστείτε, όπως για παράδειγμα `beq`, `bne`, `bge`, `bgt`, `ble`, `blt` κ.τ.λ.

Εντολές Διακλάδωσης

Branch on equal — Διακλάδωση αν ίσο

`beq rs, rt, label`

Διακλάδωση υπό συνθήκη κατά τον αριθμό των εντολών που καθορίζονται από τη σχετική απόσταση (offset) αν ο καταχωρητής `rs` *είναι ίσος με τον* `rt`*.*

Branch on greater than equal — Διακλάδωση αν μεγαλύτερο ή ίσο

`bge rsrc1, rsrc2, label` *(ψευδοεντολή)*

Διακλάδωση υπό συνθήκη στην εντολή που βρίσκεται στην ετικέτα `label` *αν ο καταχωρητής* `rsrc1` *είναι μεγαλύτερος ή ίσος με τον* `rsrc2`*.*

Branch on greater than — Διακλάδωση αν μεγαλύτερο

`bgt rsrc1, src2, label` *(ψευδοεντολή)*

Διακλάδωση υπό συνθήκη στην εντολή που βρίσκεται στην ετικέτα `label` *αν ο καταχωρητής* `rsrc1` *είναι μεγαλύτερος από* `src2`*.*

Branch on less than equal — Διακλάδωση αν μικρότερο ή ίσο

`ble rsrc1, src2, label` *(ψευδοεντολή)*

Διακλάδωση υπό συνθήκη στην εντολή που βρίσκεται στην ετικέτα `label` *αν ο καταχωρητής* `rsrc1` *είναι μικρότερος ή ίσος με* `src2`*.*

Branch on less than — Διακλάδωση αν μικρότερο

`blt rsrc1, rsrc2, label` *(ψευδοεντολή)*

Διακλάδωση υπό συνθήκη στην εντολή που βρίσκεται στην ετικέτα `label` *αν ο καταχωρητής* `rsrc1` *είναι μικρότερος από* `rsrc2`*.*

Οδηγία Δέσμευσης Χώρου στη Μνήμη

`.space n` Κατανομή χώρου `n` byte στο τρέχον τμήμα (που πρέπει να είναι το τμήμα δεδομένων στον SPIM).

Ασκηση 3.4

Γράψτε ένα πρόγραμμα σε συμβολική γλώσσα MIPS που θα δέχεται σαν είσοδο από το χρήστη μία ακολουθία χαρακτήρων και θα τυπώνει στην έξοδο την ίδια ακολουθία με κεφαλαία γράμματα (όπου υπήρχαν χαρακτήρες ‘a’-‘z’ θα αντικαθίστανται με τους ‘A’-‘Z’). Οι κωδικές τιμές ASCII των χαρακτήρων ‘a’-‘z’ είναι 97-122 σε δεκαδικό σύστημα ενώ των χαρακτήρων ‘A’-‘Z’ είναι 65-90.
Βασιστείτε στο πρόγραμμα `lab3_4a.s` το οποίο απλώς αντιγράφει το εισαγόμενο `string` που αποθηκεύεται στη θέση μνήμης με ετικέτα `string1` σε μία άλλη με ετικέτα `string2` και ακολούθως εμφανίζει το αντιγραμμένο `string2`. Κάνετε τις απαραίτητες επεμβάσεις στο πρόγραμμα `lab3_4a.s` έτσι ώστε να κάνει την απαιτούμενη μετατροπή κατά την αντιγραφή από το `string1` στο `string2`.

```
################################################
# lab3_4a.s                                    #
# while loop with nested if (to be completed)  #
################################################
      .text
      .globl __start
__start:              # execution starts here
      la $a0,prompt
      li $v0,4
```

```
      syscall                  # display prompt
      la $a0,string1           # address to store string1
      li $a1,80                # max length of string1
      li $v0,8
      syscall                  # read string
      li $t1,0                 # counter for string1 and string2
while: lbu $t0,string1($t1)    # load a character of string1
      beq $t0,$0,end           # if character to end copy then exit loop
      sb $t0,string2($t1)      # copy character to string2
      addi $t1,$t1,1           # increment counter
      j while                  # repeat while
end:
      la $a0,string2
      li $v0,4
      syscall                  # display copied string2
      li $v0,10                # exit
      syscall
      .data
prompt:           .asciiz "Enter string to transform : "
string1:    .space 80
string2:    .space 80
```

Ασκηση 3.5

Το παρακάτω πρόγραμμα ζητάει συνεχώς από το χρήστη να εισάγει ακεραίους και βρίσκει το άθροισμά τους μέχρι να του δοθεί ο ακέραιος 0. Ο βρόχος επανάληψης του προγράμματος δημιουργείται με δύο εντολές διακλάδωσης, την υπό συνθήκη `beq` και την `b`.
Αλλάξτε το πρόγραμμα έτσι ώστε να επιτελεί τον ίδιο σκοπό αλλά να χρησιμοποιεί μόνο μία εντολή διακλάδωσης υπό συνθήκη.

```
################################################
# lab3_5a.s                                    #
################################################

      .text
      .globl __start
__start:                       # execution starts here
      li $t1,0                 # sum


loop: la $a0,str2
      li $v0,4
      syscall                  # display "Give integer :"
      li $v0,5
      syscall                  # read integer in $v0
      move $t2,$v0             # move integer in $t2
      la $a0,endl
      li $v0,4
      syscall                  # display end of line

      add $t1,$t1,$t2
      beq $t2,$0,end           # exit loop if $t2=0
      b loop                   # repeat
end:  la $a0,str1
      li $v0,4
      syscall                  # display "Sum is :"
      move $a0,$t1
      li $v0,1
      syscall                  # display the sum
      la $a0,endl
      li $v0,4
```

```
      syscall                       # display end of line
      li $v0,10
      syscall                       # exit
      .data
str1:       .asciiz "Sum is : "
str2:       .asciiz "Give integer :"
endl:       .asciiz "\n"
```

ΕΝΟΤΗΤΑ 4:

Διαδικασίες και στοίβα

Εισαγωγή

Σε αυτήν την ενότητα θα μελετηθεί η κλήση διαδικασιών (procedure call), η χρήση της στοίβας (stack) και οι συμβάσεις για τη χρήση καταχωρητών. Το πρώτο μέρος αφορά απλή κλήση διαδικασίας ενώ το δεύτερο μέρος (επί πλέον ασκήσεις) αναδρομικές κλήσεις διαδικασίας (recursive calls).

Μέρος 1ο: Απλή κλήση και συμβάσεις καταχωρητών

Ασκηση 4.1

Σκοπός αυτής της άσκησης είναι η δημιουργία μία απλής διαδικασίας και η κλήση της από το κυρίως πρόγραμμα. Η διαδικασία θα δέχεται σαν ορίσματα τέσσερις ακεραίους α, β, γ και δ και θα επιστρέφει έναν ακέραιο ε που θα υπολογίζεται από τον τύπο ε=3α+7β+17γ+31δ. Για το πέρασμα των τεσσάρων ορισμάτων στη διαδικασία θα χρησιμοποιούνται οι καταχωρητές `$a0`, `$a1`, `$a2` και `$a3` και η επιστροφή του αποτελέσματος θα γίνεται μέσω του καταχωρητή `$v0`, ακολουθώντας τη σύμβαση καταχωρητών του MIPS. Χρησιμοποιήστε για τυχόν ενδιάμεσα αποτελέσματα μόνο τους καταχωρητές `$s` (σημείωση: ας θεωρήσουμε ότι οι καταχωρητές `$t` χρησιμοποιούνται για άλλους σκοπούς από την διαδικασία και δεν είναι διαθέσιμοι) καθώς και τη σύμβαση προγραμματισμού του MIPS που θέλει τους καταχωρητές `$s` να αποθηκεύονται στη στοίβα σε περίπτωση που γίνεται χρήση τους από τη διαδικασία. Για τους πολλαπλασιασμούς με σταθερά χρησιμοποιήστε συνδυασμό αριστερών ολισθήσεων (πολλαπλασιασμούς με δύναμη του 2) και προσθαφαιρέσεων (π.χ. 3α=2α+α).

Στο κυρίως πρόγραμμα αρχικοποιήστε τους καταχωρητές `$a0` … `$a3` με τέσσερις αριθμούς και επίσης δώστε στους καταχωρητές `$s` που θα χρησιμοποιήσετε στη διαδικασία κάποιες τυχαίες τιμές έτσι ώστε να ελέγξετε ότι όντως διατηρούνται τα αρχικά περιεχόμενά τους και μετά την επιστροφή από την κλήση της διαδικασίας.

Βασιστείτε στο παρακάτω πρόγραμμα με ονομασία `lab4_1a.s`. Αυτό είναι το κυρίως πρόγραμμα (main program) που αρχικοποιεί τις μεταβλητές οι οποίες χρησιμοποιούνται για να περάσουμε παραμέτρους στη διαδικασία και εμφανίζει στην οθόνη τα αποτελέσματά της. Συμπληρώστε τις απαραίτητες γραμμές κώδικα για το σώμα της διαδικασίας και για την κλήση της.

Τρέξτε το πρόγραμμα με βηματικό τρόπο ώστε να παρακολουθήσετε τη στοίβα να μεγαλώνει και να μικραίνει (δηλαδή να προστίθενται σε αυτή και να αφαιρούνται από αυτή στοιχεία). Παρακολουθήστε επίσης τα περιεχόμενα του καταχωρητή στοίβας `$sp` .

```
################################################
# lab4_1a.s                                    #
# simple procedure call (to be completed)      #
################################################
      .text
      .globl __start
__start:              # execution starts here

# start of main program
      li $a0,1        # Initialize variable a
      li $a1,2        # Initialize variable b
      li $a2,3        # Initialize variable c
      li $a3,4        # Initialize variable d
      li $s0,1234     # random value in $s0
      li $s1,5678     # random value in $s1

      move $t0,$v0
```

```
      la $a0,answer
      li $v0,4
      syscall           # display "Answer is :"
      move $a0,$t0
      li $v0,1
      syscall           # display result of procedure
      la $a0,endl
      li $v0,4
      syscall           # display end of line
      li $v0,10
      syscall           # exit
# end of main programm

# start of procedure

# end of procedure

      .data
answer:     .asciiz "Answer is : "
endl:       .asciiz "\n"
```

Κλήσεις Διαδικασιών

Τα παρακάτω βήματα περιγράφουν τη σύμβαση κλήσεων που χρησιμοποιείται στις περισσότερες μηχανές MIPS. Αυτή η σύμβαση έρχεται στο προσκήνιο σε τρία σημεία κατά τη διάρκεια της κλήσης μιας διαδικασίας: αμέσως πριν ο καλών ξεκινήσει τον καλούμενο, ακριβώς μόλις ο καλούμενος αρχίζει να εκτελείται, και αμέσως πριν ο καλούμενος επιστρέψει στον καλούντα.

Στο πρώτο μέρος, ο καλών τοποθετεί τα ορίσματα κλήσης της διαδικασίας σε τυπικές θέσεις και ζητάει από τον καλούμενο να κάνει τα εξής:

1. Μεταβίβαση ορισμάτων. Κατά σύμβαση, τα πρώτα τέσσερα ορίσματα μεταβιβάζονται στους καταχωρητές `$a0–$a3`. Τυχόν άλλα ορίσματα τοποθετούνται στη στοίβα και εμφανίζονται στην αρχή του πλαισίου στοίβας της καλούμενης διαδικασίας.

2. Αποθήκευση των καταχωρητών που αποθηκεύονται από τον καλούντα. Η καλούμενη διαδικασία μπορεί να χρησιμοποιήσει αυτούς τους καταχωρητές (`$a0–$a3` και `$t0–$t9`) χωρίς να αποθηκεύσει πρώτα την τιμή τους. Αν ο καλών αναμένει ότι θα χρησιμοποιήσει έναν από αυτούς τους καταχωρητές μετά από μια κλήση, πρέπει να αποθηκεύσει την τιμή του πριν από την κλήση.

3. Εκτέλεση μιας εντολής `jal` (δείτε την Ενότητα 2.8), η οποία πραγματοποιεί άλμα στην πρώτη εντολή του καλούμενου και αποθηκεύει τη διεύθυνση επιστροφής στον καταχωρητή `$ra`.

Πριν αρχίσει να εκτελείται μια ρουτίνα, πρέπει να ακολουθήσει τα παρακάτω βήματα για να διευθετήσει το πλαίσιο στοίβας της:

1. Κατανομή μνήμης για το πλαίσιο με αφαίρεση του μεγέθους του πλαισίου από το δείκτη στοίβας.

2. Αποθήκευση των καταχωρητών που αποθηκεύονται από τον καλούμενο στο πλαίσιο. Ένας καλούμενος πρέπει να αποθηκεύσει τις τιμές στους καταχωρητές (`$s0–$s7`, `$fp`, και `$ra`) πριν τις τροποποιήσει, αφού ο καλών αναμένει ότι θα βρει αυτούς τους καταχωρητές αμετάβλητους μετά την κλήση. Ο καταχωρητής `$fp` αποθηκεύεται από κάθε διαδικασία που κατανέμει ένα νέο πλαίσιο στοίβας. Ο καταχωρητής `$ra` όμως χρειάζεται να αποθηκευτεί μόνον αν ο ίδιος ο καλούμενος κάνει μια κλήση. Οι άλλοι αποθηκευόμενοι από τον καλούμενο καταχωρητές που χρησιμοποιούνται πρέπει επίσης να αποθηκεύονται.

3. Ρύθμιση του δείκτη πλαισίου με την πρόσθεση του μεγέθους πλαισίου στοίβας πλην 4 στον `$sp` και αποθήκευση του αθροίσματος στον καταχωρητή `$fp`.

Τέλος, ο καλούμενος επιστρέφει στον καλούντα με τα παρακάτω βήματα:

1. Αν ο καλούμενος είναι μια συνάρτηση που επιστρέφει τιμή, τοποθέτηση της επιστρεφόμενης τιμής στον καταχωρητή `$v0`.

2. Επαναφορά όλων των αποθηκευόμενων από τον καλούμενο καταχωρητών που αποθηκεύτηκαν κατά την είσοδο στη διαδικασία.

3. Εξαγωγή (pop) από το πλαίσιο στοίβας με την πρόσθεση του μεγέθους πλαισίου στον `$sp`.

4. Επιστροφή με άλμα στη διεύθυνση που περιέχεται στον καταχωρητή `$ra`.

Εντολές κλήσεις και επιστροφής διαδικασίας

Jump and link — Άλμα και σύνδεση

```
jal target
```

Άλμα χωρίς συνθήκη στην εντολή που βρίσκεται στην ετικέτα . Αποθήκευση της διεύθυνσης της επόμενης εντολής στον καταχωρητή .

Jump register — Άλμα σε καταχωρητή

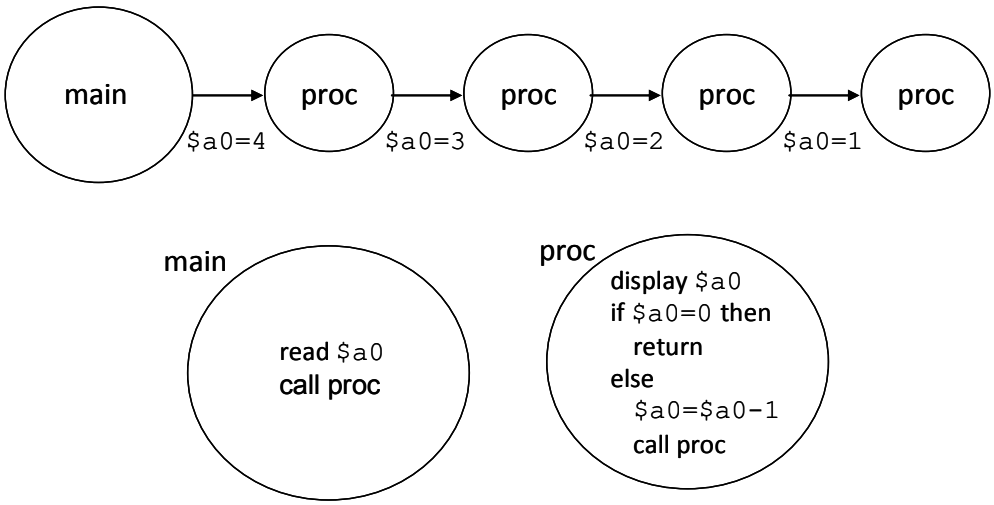
```
jr rs
```

Άλμα χωρίς συνθήκη στην εντολή της οποίας η διεύθυνση βρίσκεται στον καταχωρητή `rs`.

Μέρος 2ο :Επί πλέον Ασκήσεις - Αναδρομικές Κλήσεις

Ασκηση 4.2

Γράψτε ένα πρόγραμμα σε συμβολική γλώσσα MIPS που θα αποτελείται από δύο τμήματα, το κυρίως πρόγραμμα και μία διαδικασία που θα καλεί αναδρομικά τον εαυτό της. Το κυρίως πρόγραμμα θα ζητάει από το χρήστη έναν ακέραιο αριθμό και κατόπιν θα καλεί μία διαδικασία χρησιμοποιώντας ως όρισμα αυτό τον αριθμό. Η διαδικασία θα εξετάζει την τιμή του ορίσματος και αν είναι μηδέν θα επιστρέφει. Αν δεν είναι μηδέν, θα μειώνει τον ακέραιο αριθμό κατά ένα και θα καλεί πάλι τον εαυτό της περνώντας της σαν όρισμα τον μειωμένο ακέραιο μέσω του καταχωρητή `$a0`. Σε κάθε περίπτωση (μηδέν ή όχι) θα εμφανίζει στην οθόνη την τιμή του αριθμού που έχει παραλάβει. Το παρακάτω σχήμα δείχνει τη διαδοχή των κλήσεων και τις ενέργειες σε ψευδοκώδικα που πρέπει να κάνει η το κύριο πρόγραμμα και η διαδικασία.

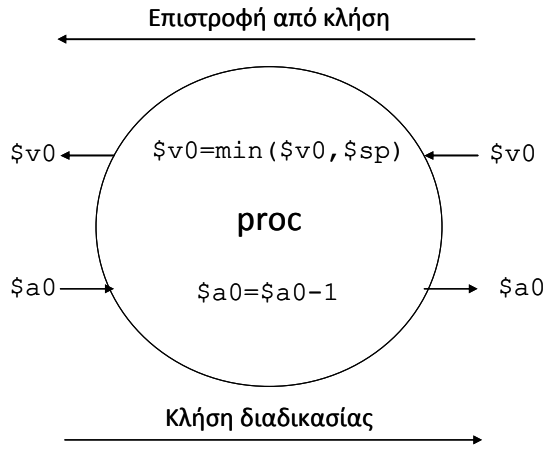


Βασιστείτε στο παρακάτω πρόγραμμα με ονομασία `lab4_2a.s` το οποίο ζητάει από το χρήστη έναν ακέραιο και συμπληρώστε τις απαραίτητες γραμμές κώδικα για τον ορισμό της διαδικασίας και την κλήση της από το κύριο πρόγραμμα.

```
################################################
# lab4_2a.s                                    #
# Recursive procedure call (to be completed)   #
################################################
      .text
      .globl   start
__start:                    # execution starts here

# start of main program
      la $a0,prompt
      li $v0,4
      syscall               # display "Enter integer  :"
      li $v0,5
      syscall               # read integer
      move $t0,$v0
      la $a0,endl
      li $v0,4
      syscall               # display end of line
      move $a0,$t0

      li $v0,10
      syscall               # exit
# end of main program

# start of procedure

# end of procedure

      .data
prompt:     .asciiz "Enter integer :"
endl:       .asciiz "\n"
```

Ασκηση 4.3

Η προηγούμενη άσκηση περιείχε μία διαδικασία που δεχόταν μία παράμετρο (`$a0`), χωρίς όμως να επιστρέφει κανένα αποτέλεσμα. Τώρα θα εμπλουτίσετε το πρόγραμμά σας έτσι ώστε η διαδικασία να επιστρέφει έναν ακέραιο αριθμό που θα δηλώνει την ελάχιστη διεύθυνση του δείκτη στοίβας που έχει χρησιμοποιήσει η διαδικασία ή κάποια άλλη που κλήθηκε από αυτήν. Χρησιμοποιήστε τη σύμβαση που έχει ο MIPS για τη χρήση καταχωρητών επιστροφής αποτελεσμάτων, π.χ. χρησιμοποιήστε τον `$v0`. Σχηματικά η διεργασία θα κάνει τη λειτουργία που φαίνεται στο σχήμα.
Το κυρίως πρόγραμμα θα διαβάζει το αποτέλεσμα από τις διαδοχικές κλήσεις, δηλαδή πόσο «χαμηλά» έφτασε η στοίβα και θα εμφανίζει το μέγεθος τη μνήμης που χρησιμοποιήθηκε για τη στοίβα, γνωρίζοντας την αρχική τιμή του $sp πριν να γίνει οποιαδήποτε κλήση διαδικασίας.



Ασκηση 4.4

Το επόμενο βήμα είναι ο εμπλουτισμός των προηγούμενων ασκήσεων έτσι ώστε να γίνεται υπολογισμός του *n*-οστού όρου της ακολουθίας Fibonacci. Η ακολουθία Fibonacci είναι αναδρομική, κάτι που κάνει τον υπολογισμό της να ταιριάζει με τη χρήση αναδρομικών κλήσεων διαδικασιών. Η ακολουθία Fibonacci ορίζεται από τον ακόλουθο τύπο :

$$f(n)=\begin{cases} f(n-1)+f(n-2) & , \quad n>1 \\ 1 & , \quad n=1 \\ 0 & , \quad n=0 \end{cases}$$

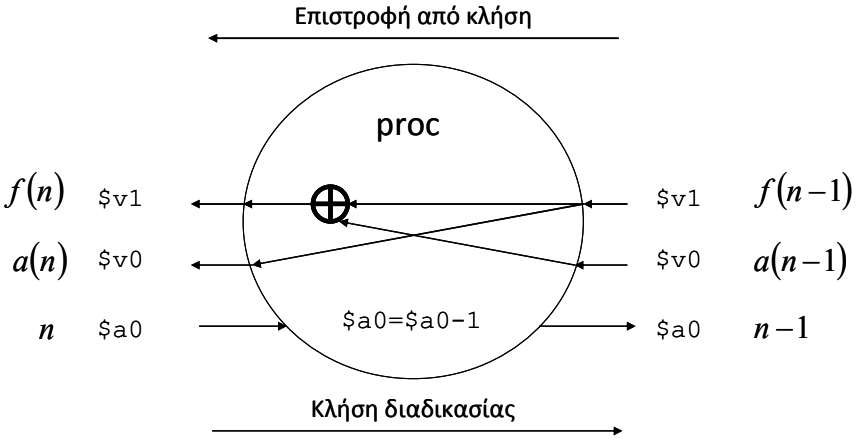
Αν ο παραπάνω τύπος υλοποιηθεί με τη χρήση μίας αναδρομικής διαδικασίας, μέσα στη διαδικασία θα πρέπει να υπάρχουν δύο κλήσεις προς τον εαυτό της, μία για $n-1$ και άλλη μία για $n-2$, κάτι που δημιουργεί ένα δενδρικό σχήμα στα διάφορα στιγμιότυπα των διαδικασιών. Για να διατηρήσουμε το γραμμικό σχήμα που ήδη έχουμε ξεκινήσει από την άσκηση 4.2 και για λόγους καλύτερης αποδοτικότητας θα χρησιμοποιήσουμε τους ισοδύναμους τύπους για τον υπολογισμό της ακολουθίας Fibonacci που δίνονται παρακάτω :

$$f(n)=\begin{cases} f(n-1)+a(n-1) & , \quad n>0 \\ 0 & , \quad n=0 \end{cases} \qquad \text{και} \qquad a(n)=\begin{cases} f(n-1) & , \quad n>0 \\ 1 & , \quad n=0 \end{cases}$$

Έτσι τώρα η αναδρομική διαδικασία καλεί μόνο μία φορά τον εαυτό της (για $n-1$). Οι παράμετροι εισόδου, τα αποτελέσματα και οι λειτουργίες της φαίνονται στο παρακάτω σχήμα (με την εξαίρεση οριακών συνθηκών για $n=0$).

Χρησιμοποιήστε την παρακάτω μορφή διαδικασίας για να υπολογίσετε τον n-οστό όρο της ακολουθίας και κάντε εμφάνιση του αποτελέσματος από το κυρίως πρόγραμμα.

Στη σύμβαση προγραμματισμού που χρησιμοποιείται από το QtSpim για την έξοδο αποτελεσμάτων από μία διαδικασία παρέχονται μόνο δύο καταχωρητές (οι `$v0` και `$v1`). Χρησιμοποιήστε τους για να την έξοδο των $f(n)$ και $a(n)$.



ΕΝΟΤΗΤΑ 5

Αριθμητική Κινητής Υποδιαστολής

Εισαγωγή

Αυτό το εργαστήριο χωρίζεται σε δύο μέρη. Στο πρώτο μέρος θα μελετηθούν η διαδικασία εισόδου–εξόδου και οι εντολές μετακίνησης αριθμών κινητής υποδιαστολής αριθμών απλής ακρίβειας, καθώς και ο τρόπος αναπαράστασης των αριθμών και οι περιορισμοί που υπάρχουν ως προς την ακρίβειά τους. Επίσης, θα πειραματιστούμε με πράξεις μεταξύ αριθμών κινητής υποδιαστολής για ειδικές περιπτώσεις, και θα φανεί η ανάγκη για μετατροπή ακεραίων σε αριθμούς κινητής υποδιαστολής. Στο δεύτερο μέρος θα αναπτυχθεί ένα πιο σύνθετο πρόβλημα υπολογισμού της συνάρτησης του εκθετικού e^x.

Μέρος 1ο: Αναπαράσταση - Ακρίβεια – Πράξεις – Μετατροπές

Ασκηση 5.1

Γράψτε ένα πρόγραμμα σε συμβολική γλώσσα MIPS που θα διαβάζει από το παράθυρο Console έναν αριθμό κινητής υποδιαστολής απλής ακρίβειας με τη χρήση της κλήσης συστήματος `read_float`. Κατόπιν χρησιμοποιήστε την κλήση συστήματος `print_float` για να εμφανίσετε τον αριθμό στην οθόνη. Θα χρειαστείτε την εντολή μεταφοράς αριθμών κινητής υποδιαστολής `mov.s` γιατί η κάθε κλήση συστήματος χρησιμοποιεί διαφορετικό καταχωρητή για να γράψει και να διαβάσει από την Console. Χρησιμοποιήστε σαν υπόδειγμα το παρακάτω πρόγραμμα `lab5_1a.s`.

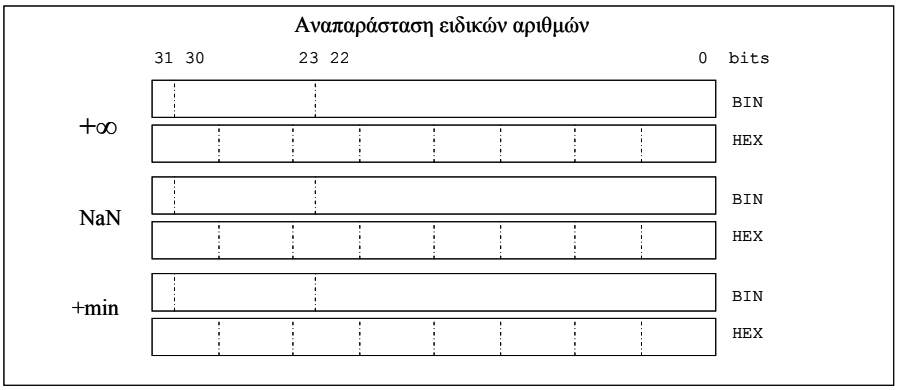
```
################################################
# lab5_1a.s                                    #
# single floating test  (to be completed)      #
################################################
      .text
      .globl __start
__start:                    # execution starts here
      la $a0,prompt
      li $v0,4
      syscall               # display prompt
      la $v0,6
      syscall               # read float in $f0
      la $a0,endl
      li $v0,4
      syscall               # display new line

      li $v0,10             # exit
      syscall
      .data
prompt:     .asciiz "Enter float number : "
endl:       .asciiz "\n"
```

(α) Πειραματιστείτε εισάγοντας στο πρόγραμμα αριθμούς που είναι δυνάμεις ή αθροίσματα δυνάμεων του 2, π.χ. -0.75, 1.0, 1.5, 0.0000152587890625 (=$1/2^{16}$) κ.τ.λ.

(β) Πειραματιστείτε με τυχαίους αριθμούς, π.χ. 0.775, 0.1, 1.23456789 κ.τ.λ.

(γ) Αλλάξτε το πρόγραμμα, έτσι ώστε να εισάγετε απ’ ευθείας σε έναν καταχωρητή το περιεχόμενό του (δεξί κλικ στο όνομα του καταχωρητή στο παράθυρο των καταχωρητών και επιλογή Change Register

Contents). Βρείτε την αναπαράσταση των παρακάτω αριθμών σε μορφή κινητής υποδιαστολής και εισάγετέ τους ως δεκαεξαδικούς αριθμούς. Για την περίπτωση NaN, επιλέξτε έναν οποιοδήποτε συνδυασμό δίνει έναν μη έγκυρο αριθμό. Η περίπτωση +min είναι ο μικρότερος θετικός μη κανονικοποιημένος αριθμός.



Μετακίνηση μεταξύ καταχωρητών απλής ακρίβειας

Move floating-point single — Μετακίνηση κινητής υποδιαστολής απλής ακρίβειας

`mov.s fd, fs`

Μετακινεί τον αριθμό κινητής υποδιαστολής διπλής (απλής) ακρίβειας από τον καταχωρητή `fs` στον καταχωρητή `fd`.

Κλήσεις συστήματος

Υπηρεσία	*Κωδικός κλήσης συστήματος*	*Ορίσματα*	*Επιστροφή*
print_float	*2*	*$f12 = float*	
read_float	*6*		*$f0 = float*

Ασκηση 5.2

Το παρακάτω πρόγραμμα ορίζει στο τμήμα δεδομένων έξι αριθμούς κινητής υποδιαστολής απλής ακρίβειας: 0.0, +∞, -∞, +NaN, x=1.0 και y=-6.0. Επί πλέον εκτελεί τον πολλαπλασιασμό x·y και εμφανίζει το αποτέλεσμα. Συμπληρώστε το πρόγραμμα ώστε κάθε φορά να εκτελεί τις πράξεις που φαίνονται στον παρακάτω πίνακα και να εμφανίζει τα αποτελέσματά τους. Συμπληρώστε τον παρακάτω πίνακα σύμφωνα με τα αποτελέσματα που θα πάρετε.

ΠΡΑΞΗ	**ΑΠΟΤΕΛΕΣΜΑ**
x · y	
x · (−∞)	
y / 0	
0 / 0	
0 · (+∞)	
(+∞) / (−∞)	
(+∞) + (−∞)	
x + NaN	

```
################################################
# lab5_2a.s                                    #
# floating point operations (to be completed)  #
################################################
      .text
      .globl   start
__start:                    # execution starts here
      la $t0,x
      la $t1,y
      lwc1 $f0,0($t0)
      lwc1 $f1,0($t1)
      mul.s $f12,$f0,$f1    # x*y
      li $v0,2
      syscall               # print float
      la $a0,endl
      li $v0,4
      syscall               # endl


      li $v0,10             # exit
      syscall
      .data
zero:       .float 0.0
plusInf:    .word 0x7F800000
minusInf:   .word 0xFF800000
plusNaN:    .word 0x7F800001
x:          .float +1.0
y:          .float -6.0
endl:       .asciiz "\n"
```

Φόρτωση σε Καταχωρητή Απλής Ακρίβειας

Load word coprocessor 1 — Φόρτωση λέξης συνεπεξεργαστή 1

`lwcl ft, address`

Φόρτωση της λέξης που βρίσκεται στη διεύθυνση address στον καταχωρητή ft της μονάδας κινητής υποδιαστολής.

Πρόσθεση, Πολ/σμός και Διαίρεση Απλής ακρίβειας

Floating-point addition single — Πρόσθεση κινητής υποδιαστολής απλής

ακρίβειας

`add.s fd, fs, ft`

Υπολογίζει το άθροισμα αριθμών κινητής υποδιαστολής απλής ακρίβειας που περιέχονται στους καταχωρητές `fs` και `ft` και το τοποθετεί στον καταχωρητή `fd`.

Floating-point multiply single — Πολλαπλασιασμός κινητής υποδιαστολής απλής ακρίβειας

`mul.s fd, fs, ft`

Υπολογίζει το γινόμενο των αριθμών κινητής υποδιαστολής απλής ακρίβειας που περιέχονται στους καταχωρητές `fs` και `ft` και το τοποθετεί στον καταχωρητή `fd`.

Floating-point divide single — Διαίρεση κινητής υποδιαστολής απλής ακρίβειας

`div.s fd, fs, ft`

Υπολογίζει το πηλίκο των αριθμών κινητής υποδιαστολής απλής ακρίβειας που περιέχονται στους καταχωρητές `fs` και `ft` και το τοποθετεί στον καταχωρητή `fd`.

Ασκηση 5.3

Πολλές φορές χρειάζεται να γίνουν πράξεις μεταξύ ακεραίων και πραγματικών αριθμών ή για κάποιο λόγο οι ακέραιοι να αναπαρίστανται σαν πραγματικοί. Ένα παράδειγμα είναι ο υπολογισμός της συνάρτησης του παραγοντικού. Αν γίνει ο υπολογισμός με πράξεις ακεραίων πολύ γρήγορα φτάνουμε στα όρια της αναπαράστασης ενός ακεραίου 32 bit, π.χ. το 13! = 6.227.020.800 δεν μπορεί να αναπαρασταθεί σε 32 bit.

(α) Το πρόγραμμα `lab5_3a.s` που δίνεται παρακάτω υπολογίζει και εμφανίζει όλα τα παραγοντικά μέχρι το n! με χρήση ακεραίων. Εκτελέστε το και παρατηρήστε τα αποτελέσματα για n>=13.

```
################################################
# lab5_3a.s                                    #
# Integer factorial                            #
################################################
      .text
      .globl __start
__start:                    # execution starts here
      la $a0,n
      lw $t0,0($a0)         # $t0 = n
      li $t2,1              # $t2 index i=1..n
      li $t1,1              # $t1 contains i!
loop: mul $t1,$t1,$t2       # factorial=factorial*i
      move $a0,$t2
      li $v0,1
      syscall               # display i
      la $a0,msg1
      li $v0,4
      syscall               # display "! is :"
      move $a0,$t1
      li $v0,1
      syscall               # display i!
      la $a0,endl
      li $v0,4
      syscall               # print end of line
      addi $t2,$t2,1        # i=i+1
      ble $t2,$t0,loop      # repeat if i<=n
      li $v0,10             # exit
      syscall
            .data
n:          .word 25
msg1:       .asciiz "! is :"
endl:       .asciiz "\n"
```

(β) Αλλάξτε το προηγούμενο πρόγραμμα έτσι ώστε ο υπολογισμός να γίνεται με αριθμητική κινητής υποδιαστολής απλής ακρίβειας. Θα χρειαστείτε πιθανόν τις εξής εντολές: `mtc1` για μεταφορά περιεχομένου από καταχωρητή γενικής χρήσης σε καταχωρητή κινητής υποδιαστολής, `cvt.s.w` για μετατροπή από αναπαράσταση ακεραίου σε αναπαράσταση κινητής υποδιαστολής και `mul.s` για πολλαπλασιασμό αριθμών κινητής υποδιαστολής.
Εκτελέστε το πρόγραμμα για n>20 και παρατηρήστε τα αποτελέσματα. Συγκρίνετε τα αποτελέσματα για μεγάλες τιμές του n με αυτά που δίνει η αριθμομηχανή (Calculator) των Windows.

Εντολές Μετακίνησης και Μετατροπής Απλής Ακρίβειας

Move to coprocessor 1 — Μεταφορά στο συνεπεξεργαστή 1

`mtc1 rd, fs`

Μεταφορά τού καταχωρητή της CPU `rt` *στον καταχωρητή* `rd` *που βρίσκεται σε ένα συνεπεξεργαστή (τον καταχωρητή* `fs` *της μονάδας κινητής υποδιαστολής).*

Convert integer to single — Μετατροπή από ακέραιο σε απλής ακρίβειας

`cvt.s w fd, fs`

Μετατρέπει τον αριθμό κινητής υποδιαστολής διπλής ακρίβειας ή ακέραιο αριθμό που περιέχεται στον καταχωρητή `fs` *σε έναν αριθμό απλής ακρίβειας και τον τοποθετεί στον καταχωρητή* `fd`*.*

Μέρος 2ο: Επί πλέον Ασκήσεις - Σύνθετοι Υπολογισμοί

Ασκηση 5.4

Ο υπολογισμός της εκθετικής συνάρτησης μπορεί να γίνει με προσέγγιση χρησιμοποιώντας τη σειρά Taylor :

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} \quad , \quad \forall x \in \Re \, .$$

Γράψτε ένα πρόγραμμα υπολογισμού της εκθετικής συνάρτησης προσεγγίζοντάς την με χρήση των k πρώτων όρων της σειράς. Ο αριθμός k θα είναι μία σταθερά μέσα στο πρόγραμμα (π.χ. k=8) και ο x θα δίνεται από τον χρήστη μέσα στο παράθυρο αλληλεπίδρασης.
Χρησιμοποιήστε αριθμητική κινητής υποδιαστολής απλής ακρίβειας, όπως και στην Άσκηση 5.3. Θα χρειαστείτε επί πλέον τις εντολές `mov.s` για μετακίνηση περιεχομένων μεταξύ καταχωρητών κινητής υποδιαστολής και `add.s` και `div.s` για πρόσθεση και διαίρεση, αντίστοιχα.

ΠΑΡΑΡΤΗΜΑ

Προσομοιωτής QtSpim

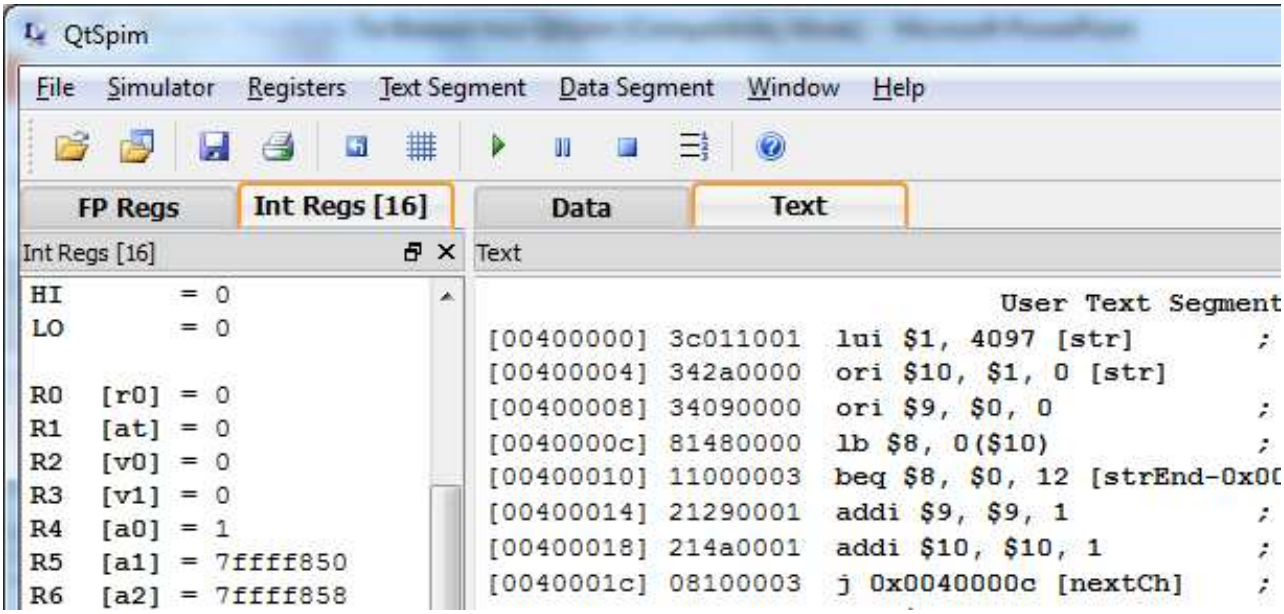
Εισαγωγή

Ο προσομοιωτής QtSpim είναι ένα περιβάλλον που μπορεί να εκτελέσει προγράμματα που είναι γραμμένα στην γλώσσα *Assembly* του μικροεπεξεργαστή MIPS και να εμφανίζει στην οθόνη τα περιεχόμενα της μνήμης και των καταχωρητών του MIPS. Διατίθεται δωρεάν και είναι το πιο πρόσφατο περιβάλλον προσομοίωσης και *συμβολομετάφρασης* (assembler) για τον μικροεπεξεργαστή MIPS. Βασίζεται στον Spim του James Larus που είχε πρωτοεμφανιστεί το 1990. Κυκλοφορεί σε εκδόσεις που τρέχουν κάτω από λειτουργικά συστήματα Microsoft Windows, Linux και Mac OS X.

Ακολουθεί μία σύντομη περιγραφή του QtSpim.

Γενική Περιγραφή

Ο QtSpim είναι ένα παραθυρικό περιβάλλον, που έχει δύο βασικά παράθυρα : Το κεντρικό (QtSpim) και το παράθυρο εισόδου/εξόδου δεδομένων (Console – κονσόλα). Το κεντρικό παράθυρο δείχνει τα περιεχόμενα των καταχωρητών του MIPS και τα περιεχόμενα της μνήμης του MIPS.
Οι καταχωρητές που εμφανίζονται σε δύο ξεχωριστά υποπαράθυρα (tabs) είναι οι καταχωρητές ακεραίων γενικού και ειδικού σκοπού και οι καταχωρητές κινητής υποδιαστολής.

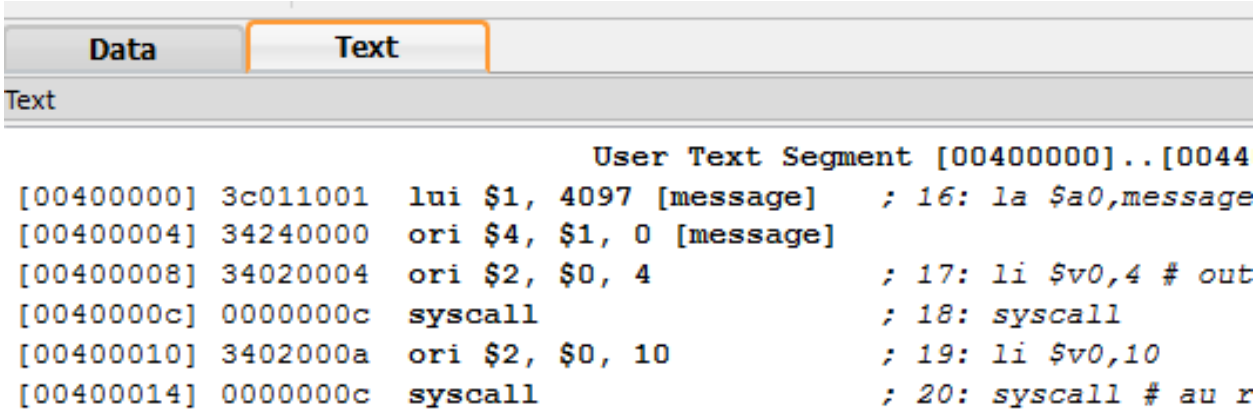


Η μνήμη εμφανίζεται και αυτή σε δύο ξεχωριστά υποπαράθυρα που δείχνουν το *τμήμα κώδικα* ή αλλιώς *τμήμα «κειμένου»* (text segment) και το *τμήμα δεδομένων* που χωρίζεται στο *τμήμα δεδομένων χρήστη* (User Data Segment), στο *τμήμα στοίβας* (User Stack Segment) και *τμήμα δεδομένων πυρήνα* (Kernel Data Segment).

Text Segment

Το υποπαράθυρο του Text Segment δείχνει σε τέσσερις στήλες τις εξής πληροφορίες για κάθε γραμμή εντολής που έχει εισαχθεί στο QtSpim, όπως φαίνεται και στο παρακάτω σχήμα :

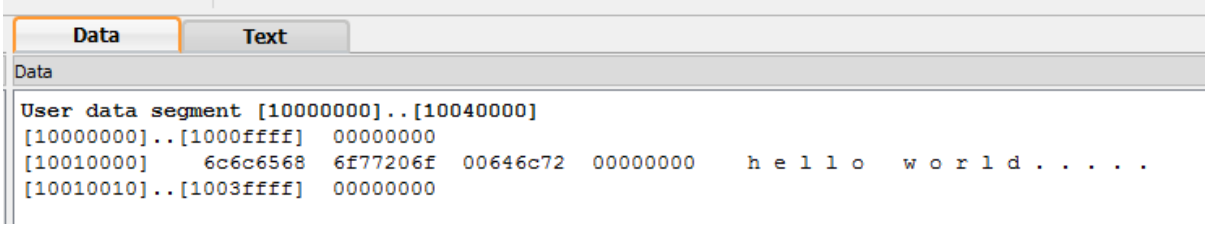
- Τη διεύθυνση σε bytes της εντολής (μέσα σε αγκύλες `[]`) σε δεκαεξαδικό σύστημα αρίθμησης (8 δεκαεξαδικά ψηφία που αντιστοιχούν σε 8Χ4=32 bit). Η κάθε επόμενη εντολή απέχει 4 bytes από την προηγούμενή της, δηλαδή μία λέξη των 32 bit.
- Το περιεχόμενο της διεύθυνσης της εντολής σε δεκαδικό σύστημα επίσης, δηλαδή τον κωδικό της εντολής σε γλώσσα μηχανής. Πρόκειται επίσης για μία λέξη των 8Χ4=32bit.
- Τις πραγματικές εντολές Assembly του MIPS (αυτές δηλαδή που έχουν προκύψει από το πρόγραμμα που έχουμε εισάγει στον QtSpim).
- Τον *πηγαίο κώδικα* (source code) που έχουμε εισάγει, ο οποίος μπορεί να περιέχει και *ψευδοεντολές*.



Data Segment

Το υποπαράθυρο του Data Segment και για τις τρεις περιπτώσεις (User Data Segment, User Stack Segment και Kernel Data Segment) παρουσιάζει τις εξής στήλες :

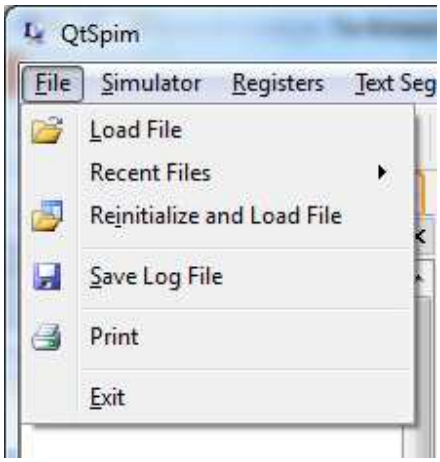
- Διεύθυνση σε bytes (μέσα σε αγκύλες `[]`) σε δεκαεξαδικό σύστημα αρίθμησης (8 δεκαεξαδικά ψηφία που αντιστοιχούν σε 8Χ4=32 bit).
- Τέσσερις στήλες που έχουν τα περιεχόμενα τεσσάρων διαδοχικών θέσεων μνήμης, ξεκινώντας από τη διεύθυνση μνήμης που αναφέρει η πρώτη στήλη. Τα περιεχόμενα παρουσιάζονται σε δεκαεξαδική μορφή, δηλαδή σε λέξεις των 8Χ4=32bit.
- Στα δεξιότερα παρουσιάζονται τα ίδια περιεχόμενα σε μορφή χαρακτήρων ASCII αν ο κωδικός ASCII είναι εκτυπώσιμος ή με μορφή τελείας (.) αν ο χαρακτήρας δεν είναι εκτυπώσιμος (π.χ. control χαρακτήρες). Επειδή κάθε σειρά απεικονίζει τέσσερις λέξεις των 32 bit και ο κάθε χαρακτήρας αντιστοιχεί σε ένα byte, δηλαδή 8 bit, προκύπτει ότι σε κάθε σειρά θα εμφανίζονται συνολικά 16 χαρακτήρες.



Επιλογή File

Η επιλογή File προσφέρει τις εξής λειτουργίες:

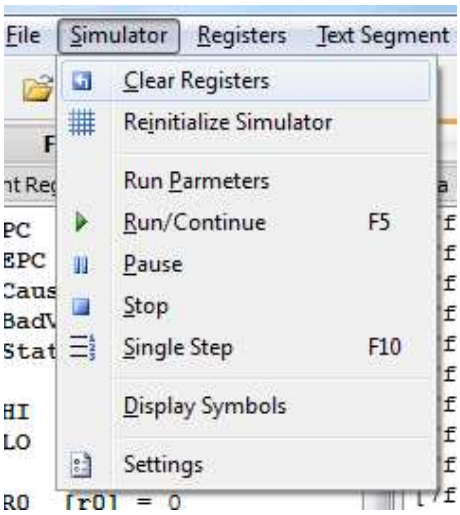
- `Load File` (Φόρτωση Αρχείου) : Φορτώνει στο περιβάλλον του QtSpim ένα αρχείου κειμένου που περιέχει ένα πρόγραμμα Assembly για τον MIPS. Το αρχείο αυτό πρέπει να έχει κατάληξη `.s` , `.asm` ή `.txt`
- `Recent File` (Πρόσφατα Αρχεία) : Φορτώνει κάποιο από τα πιο πρόσφατα αρχεία Assembly που είχαν προσπελαστεί στο παρελθόν.
- `Reinitialize and Load File` (Αρχικοποίηση προσομοιωτή και φόρτωση αρχείου)
- `Save Log File` (Αποθήκευση Αρχείου Καταγραφής) : Αποθήκευση σε αρχείο του περιεχομένου των καταχωρητών, του Data Segment, του Text Segment, της κονσόλας ή συνδυασμού των παραπάνω.
- `Print` (Εκτύπωση) : Εκτύπωση του περιεχομένου των καταχωρητών, του Data Segment, του Text Segment, της κονσόλας ή συνδυασμού των παραπάνω.
- `Exit` (Έξοδος) : Κλείσιμο του QtSpim



Επιλογή Simulator

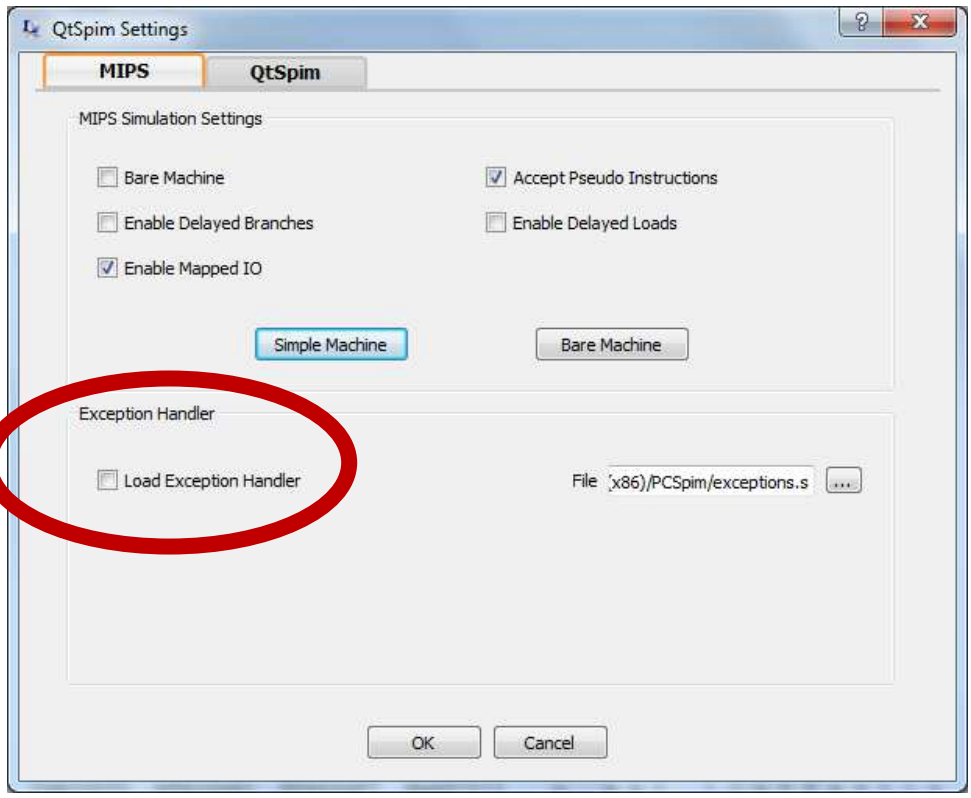
Η επιλογή Simulator προσφέρει τις εξής λειτουργίες:

- `Clear Register` (Καθαρισμός Καταχωρητών) : Μηδενίζει όλους τους καταχωρητές
- `Reinitialize Simulator` (Αρχικοποίηση Προσομοιωτή)
- `Run Parameters`
- `Run/Continue – F5` (Εκτέλεση/Συνέχιση) : Ξεκινάει την εκτέλεση του προγράμματος που έχει φορτωθεί ή συνεχίζει την προηγούμενη εκτέλεση από το σημείο που είχε διακοπεί προσωρινά
- `Pause` (Παύση) : Σταματάει προσωρινά την εκτέλεση του προγράμματος. Το πρόγραμμα μπορεί να συνεχίσει να εκτελείται από το ίδιο σημείο ή να συνεχίσει να εκτελείται βηματικά
- `Stop` (Διακοπή) : Μόνιμη διακοπή του προγράμματος
- `Single Step – F10`(Βηματική Εκτέλεση) : Εκτελεί την επόμενη εντολή του προγράμματος σε σχέση με το που έχει σταματήσει από την προηγούμενη φορά
- `Display Symbols` (Εμφάνιση Ετικετών)
- `Settings` (Ρυθμίσεις) : Ρυθμίσεις παραμετροποίησης για τον Spim και για θέματα εμφάνισης (χρώματα, γραμματοσειρές).

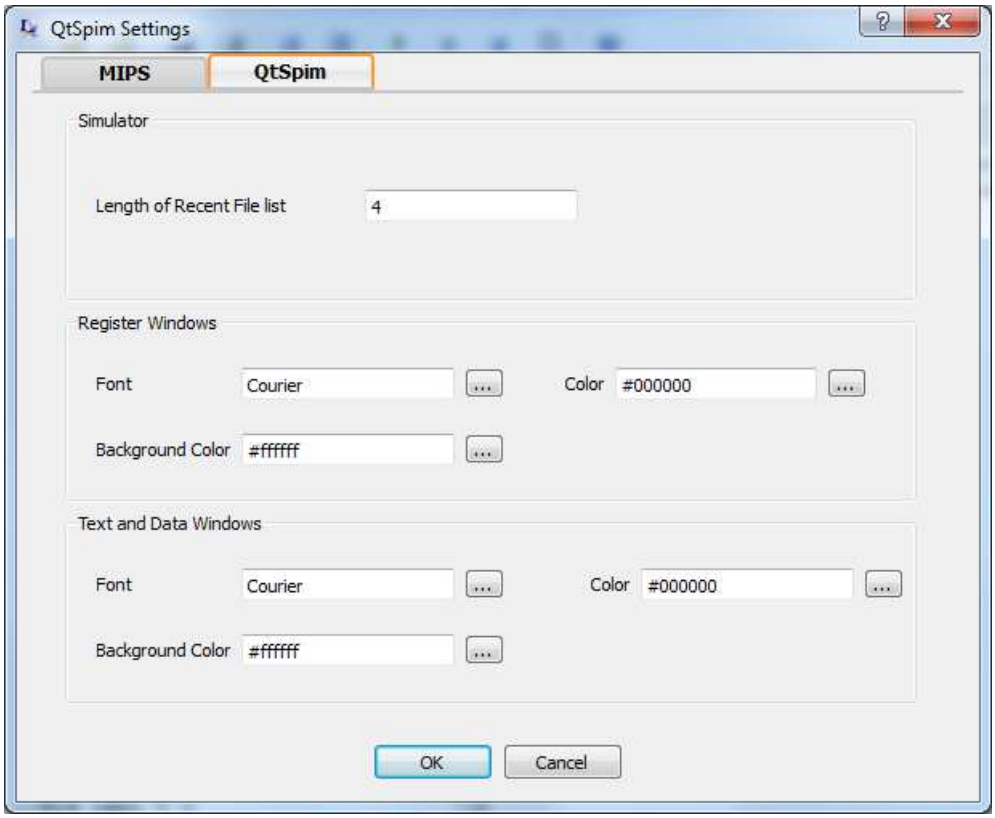


Επιλογή Simulator -> Settings

Στις ρυθμίσεις πρέπει να έχει επιλεγεί το κουμπί `Simple Machine` που έχει ως αποτέλεσμα τη χρήση ψευδοεντολών (`Accept Pseudoinstructions`), την επιλογή `Enable Mapped IO` στα πέντε πρώτα κουτάκια.
Επίσης πρέπει και να έχει αποεπιλεχθεί οπωσδήποτε η επιλογή `Load Exception Handler`.
Η παρακάτω εικόνα δείχνει την επιθυμητή διαρρύθμιση στο tab `MIPS` της κάρτας `QtSpim Settings`.



Οι υπόλοιπες ρυθμίσεις που είναι στο tab `QtSpim` μπορούν να είναι της αρεσκείας του χρήστη και η σχετική κάρτα φαίνεται στην παρακάτω εικόνα :



Επιλογή Registers

Με αυτήν την επιλογή ρυθμίζουμε σε ποια βάση θα εμφανίζονται τα περιεχόμενα των καταχωρητών, δηλαδή σε δυαδική μορφή (`Bin`), δεκαεξαδική μορφή (`Hex`) ή σε δεκαδική μορφή (`Decimal`).

Επιλογή Text Segment

Επιλέγουμε τι ακριβώς θέλουμε να εμφανίζεται στο παράθυρο `Text Segment`. Οι τέσσερις πιθανές επιλογές που μπορούν να συνδυαστούν μεταξύ τους με όποιο τρόπο θέλουμε είναι:

- `User Text` : Να εμφανίζεται το πρόγραμμα του χρήστη.
- `Kernel Text` : Να εμφανίζεται το πρόγραμμα του πυρήνα.
- `Comments` : Να εμφανίζονται πιθανά σχόλια που έχουμε βάλει στο πρόγραμμά μας.
- `Instruction Value` : Να εμφανίζεται ο κώδικας μηχανής της κάθε εντολής.

Οποιαδήποτε αλλαγή στον τρόπο εμφάνισης γίνεται ενεργή σε φόρτωση επόμενου προγράμματος.

Επιλογή Data Segment

Επιλέγουμε τι ακριβώς θέλουμε να εμφανίζεται στο παράθυρο Data Segment και με ποια μορφή (δυαδική, δεκαεξαδική ή δεκαδική). Μπορούν να εμφανιστούν τα τρία είδη δεδομένων μνήμης δεδομένων που είναι:

- `User Data` : Δεδομένα χρήστη
- `User Stack` : Στοίβα
- `Kernel Data` : Δεδομένα πυρήνα (Λειτουργικού συστήματος)

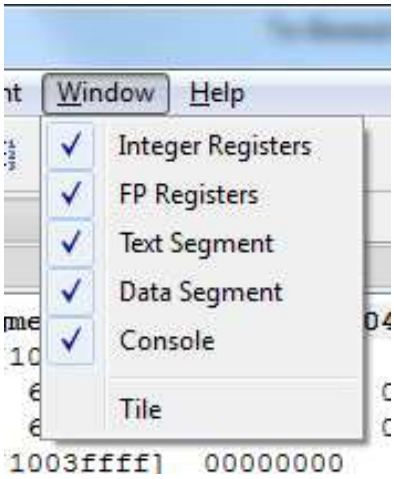
Τα δεδομένα του χρήστη (`User Data`) αρχίζουν από τη δεκαεξαδική διεύθυνση `0x10000000` και προχωρούν προς μεγαλύτερες διευθύνσεις.

Ο κώδικας (πρόγραμμα) του χρήστη (`Text Segment`) αρχίζει από τη δεκαεξαδική διεύθυνση `0x00400000` και συνεχίζει και αυτός προς μεγαλύτερες διευθύνσεις.

Επιλογή Window

Επιλογή των παραθύρων που θα είναι ενεργά. Οι δυνατές επιλογές (σε οποιοδήποτε συνδυασμό μεταξύ τους) είναι: `Integer Registers` για το tab των καταχωρητών γενικού και ειδικού σκοπού, `FP Registers` για το tab των καταχωρητών κινητής υποδιαστολής, `Text Segment`, `Data Segment` και `Console`.

Η επιλογή `Tile` επιτρέπει την σχετική μετακίνηση του κάθε υποπαραθύρου αντί να είναι «κλειδωμένο» σε σταθερή θέση.



Επιλογή Help

Με την υποεπιλογή `View Help` εμφανίζεται ο οδηγός χρήστη (user manual) του QtSpim που εκτός από τη λειτουργία του προγράμματος, περιέχει αναλυτική περιγραφή για όλες τις εντολές του MIPS και τη λειτουργία τους, των οδηγιών προς τον συμβολομεταφραστή (assembler directives), περιγραφή των κλήσεων συστήματος (system calls) και γενικά είναι ένας πολύ αναλυτικός οδηγός.

Περισσότερες λεπτομέρειες για τη συμβολική γλώσσα του MIPS υπάρχουν στο Παράρτημα Β του βιβλίου *Οργάνωση και Σχεδίαση Υπολογιστών* των D.A.Patterson, J.L.Hennessy, εκδ. Κλειδάριθμος.

Η υποεπιλογή `About QtSpim` μας πληροφορεί για το ποια έκδοση είναι αυτή που χρησιμοποιούμε.

Συγγραφή Κώδικα – Εκτέλεση Προγράμματος

Επειδή το QtSpim δε διαθέτει ενσωματωμένο παράθυρο για τη συγγραφή του κώδικα όπως άλλα ολοκληρωμένα περιβάλλοντα προγραμματισμού, είναι απαραίτητη η χρήση κάποιου «γυμνού» διορθωτή κειμένου (text editor), όπως το Notepad στα Windows ή ο Pico στο Linux, όπου θα γραφεί το πρόγραμμα. Ο διορθωτής αυτός δεν πρέπει να κάνει καμία μορφοποίηση στο κείμενο, έστω και κρυφή, έτσι ώστε να

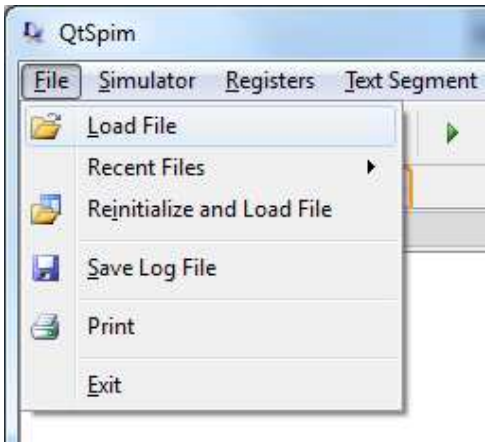
μην υπάρχουν καθόλου μη εκτυπώσιμοι/ειδικοί χαρακτήρες. Έτσι, το Microsoft Word δεν είναι αποδεκτό για τη συγγραφή ενός προγράμματος για MIPS.

Το πρόγραμμα πρέπει να αποθηκευτεί με κατάληξη `.s` ή `.asm` ή έστω `.txt` για να μπορέσει μετά να φορτωθεί στον QtSpim.

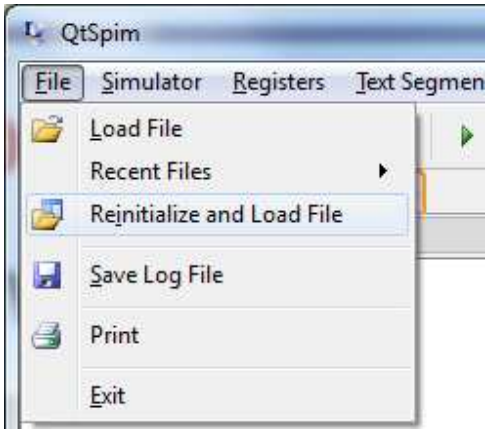
Ένα πρόγραμμα πρέπει να περιέχει τουλάχιστον το τμήμα κώδικα (που είναι οι εντολές) και δηλώνεται με την οδηγία προς τον assembler `.text` . Ότι ακολουθεί αυτή την οδηγία θεωρείται ότι είναι εντολές. Σε περίπτωση που υπάρχουν δεδομένα που είτε είναι αρχικά με τιμή μηδέν είτε έχουν κάποια άλλη τιμή, αυτά γράφονται σε άλλο τμήμα του κειμένου, ύστερα από τη οδηγία `.data` . Η Άσκηση 1.1 του παρόντος φυλλαδίου δείχνει ένα παράδειγμα.

Σε περίπτωση που επιθυμούμε να βάλουμε σχόλια για δική μας χρήση στο πρόγραμμα τα οποία θα πρέπει να αγνοήσει ο assembler, τότε ακριβώς πριν από τα σχόλια πρέπει να μπαίνει ο χαρακτήρας της δίεσης (#).

Αφού ολοκληρωθεί η συγγραφή του κώδικα και γίνει η αποθήκευσή του, μπορεί να γίνει η φόρτωση από το QtSpim με την επιλογή `File -> Load`.



Αν χρειαστεί να ξαναφορτώσουμε ένα πρόγραμμα που διορθώθηκε, τότε πρέπει να γίνει αρχικοποίηση στον προσομοιωτή. Αυτό μπορεί να γίνει είτε με την επιλογή `File -> Reinitialize and Load File` είτε με τη διαδοχή των επιλογών `Simulator -> Reinitialize Simulator` και ύστερα `File -> Load File`.



Τώρα μπορεί να εκτελεστεί το πρόγραμμα χωρίς διακοπή μέχρι το τέλος επιλέγοντας `Simulator -> Run/Continue` (η το πλήκτρο `F5`) ή να εκτελεστεί με βηματικό τρόπο, δηλαδή μία-μία εντολή με την επιλογή `Simulator -> Single Step` (η το πλήκτρο `F10`).

Υπάρχει η δυνατότητα, πολλές φορές για λόγους εκσφαλμάτωσης (debugging), να ορίσουμε στο πρόγραμμα σημεία διακοπής (breakpoints), δηλαδή εντολές στις οποίες θα γίνει προσωρινή διακοπή της εκτέλεσης από τον QtSpim. Όταν γίνει η διακοπή μπορούν να παρατηρηθούν τα περιεχόμενα της μνήμης και των καταχωρητών. Κατόπιν, μπορεί να δοθεί συνέχεια στην εκτέλεση με την επιλογή `Simulator -> Run/Continue` μέχρι το επόμενο σημείο διακοπής (αν έχει οριστεί) ή μέχρι το τέλος του προγράμματος.

Για να ορίσουμε ένα σημείο διακοπής, επιλέγουμε στο παράθυρο του κώδικα την εντολή που θέλουμε και με δεξί κλικ του ποντικιού επιλέγουμε `Set Breakpoint`.