

Σειρά Ασκήσεων 2

Εντολές μεταφοράς δεδομένων και βρόχοι στον MIPS.

Οι γλώσσες προγραμματισμού εκτός από απλές μεταβλητές που περιέχουν κάποια δεδομένα, έχουν όπως γνωρίζετε και περισσότερο σύνθετες δομές δεδομένων όπως π.χ. οι πίνακες. Αυτοί περιέχουν πολύ μεγαλύτερο αριθμό θέσεων δεδομένων, από τους registers που μπορεί να περιέχει μια μηχανή. Έτσι για να μπορέσει η μηχανή να προσπελάσει αυτά τα δεδομένα που βρίσκονται σε κάποιο χώρο στη μνήμη, χρειάζεται και ένα άλλο είδος εντολών που ονομάζονται εντολές μεταφοράς δεδομένων *data transfer instructions*. Την μνήμη μπορούμε να την αναπαραστήσουμε σαν έναν μονοδιάστατο πίνακα, όπου η διεύθυνση χρησιμοποιείται σαν δείκτης που μας δείχνει το δεδομένο. Έτσι αυτές οι εντολές χρειάζεται να τους παρέχουμε την διεύθυνση στη μνήμη, των δεδομένων που πρέπει να μεταφέρουν. Η εντολή που φέρνει δεδομένα στον επεξεργαστή από την μνήμη, παραδοσιακά ονομάζεται εντολή φόρτωσης (load). Η εντολή που μεταφέρει (αποθηκεύει) δεδομένα από τον επεξεργαστή στη μνήμη, ονομάζεται εντολή αποθήκευσης (store). Ο επεξεργαστής (MIPS), την εντολή Load την ονομάζει **lw** και την Store **sw**.

Αναπαράσταση εντολών στον υπολογιστή.

Όπως γνωρίζουμε οι αριθμοί αποθηκεύονται στη μνήμη του υπολογιστή H/Y σαν μια ακολουθία από high and low ηλεκτρικά σήματα. Για παράδειγμα, ο αριθμός 67 στο δεκαδικό, που οι άνθρωποι καλλίτερα αντιλαμβάνονται, στο δυαδικό αναπαριστάνεται σαν 1000011, που ο υπολογιστής ευκολότερα καταλαβαίνει. Όπως και οι αριθμοί, έτσι και οι εντολές στον H/Y μπορούν επίσης να αναπαρασταθούν σαν ακολουθία από bits. Και έτσι ακόμη και οι εντολές φαίνονται στον H/Y σαν αριθμοί.

[Στο εργαστήριο μιλήσαμε για τα πεδία των εντολών του MIPS.](#)

Εντολές Διακλάδωσης υπό Συνθήκη στον MIPS

Όπως λέγαμε και στην άσκηση 1, για να εκτελεστεί ένα πρόγραμμα, οι εντολές του γράφονται στην κεντρική μνήμη η μία "κάτω" από την άλλη, δηλαδή σε συνεχόμενες θέσεις (διευθύνσεις) μνήμης. Μετά την ανάγνωση και εκτέλεση μιας εντολής, ο επεξεργαστής αυξάνει τον PC κατά το μέγεθος της εντολής που εκτελέστηκε, οπότε αυτός (ο PC) δείχνει στην επόμενη (την "από κάτω") εντολή. Η σειριακή αυτή εκτέλεση εντολών διακόπτεται όταν εκτελείται μια εντολή **μεταφοράς ελέγχου** (CTI - control transfer instruction). Είδαμε ήδη μία τέτοια, την εντολή άλματος `j label` ("jump" to label), που κάνει ώστε η επόμενη εντολή που θα εκτελεστεί να είναι η εντολή στη διεύθυνση μνήμης `label`, αντί να είναι η "από κάτω" εντολή. Με άλλα λόγια, η εντολή `j label` φορτώνει τη διεύθυνση `label` στον καταχωρητή PC. Χρησιμοποιώντας αυτή την εντολή άλματος στην άσκηση 1 φτιάξαμε έναν "άπειρο βρόχο", δηλαδή κάναμε τον υπό προσομοίωση υπολογιστή να εκτελεί συνεχώς το ίδιο "μπλοκ" εντολών.

Για να φτιάξουμε ένα κανονικό (όχι άπειρο) βρόχο χρειαζόμαστε μια εντολή **διακλάδωσης υπό συνθήκη** (conditional branch), δηλαδή μια εντολή που μερικές φορές προκαλεί διακλάδωση και μερικές φορές όχι, ανάλογα με το αν ισχύει ή δεν ισχύει κάποια κατάλληλη συνθήκη. Η βασική τέτοια εντολή είναι η **beq** (branch if equal): Η εντολή "beq \$16, \$17, label" διαβάζει τους καταχωρητές 16 και 17, και τους συγκρίνει. Εάν τους βρει ίσους (equal) διακλαδίζεται στη θέση label, δηλαδή κάνει τον επεξεργαστή να διαβάσει και εκτελέσει την εντολή από εκείνη τη διεύθυνση σαν επόμενη εντολή. Αλλιώς, δεν κάνει τίποτα, οπότε επόμενη εντολή θα διαβαστεί και εκτελεστεί η "από κάτω" εντολή. Η εντολή **bne** (branch if not equal) κάνει τα ανάποδα, δηλαδή διακλαδίζεται εάν βρει τους καταχωρητές άνισους (not equal), αλλιώς συνεχίζει "από κάτω".

Ασκ2.1.

Στον ακόλουθο κώδικα C οι x, y, z, i, και j, είναι μεταβλητές.

```
if ( i == j ) goto L1;
x = y + z ;
x = x - i ;
```

```
L1:  x=0;
```

Υποθέτοντας ότι οι μεταβλητές από το x έως το i αντιστοιχούν στους registers του MIPS από τον \$t2 έως τον \$t6 ποια είναι η έξοδος του C compiler για τον MIPS.

Κώδικας Βρόχου και Εισόδου / Εξόδου Κονσόλας

Για να επικοινωνούν τα προγράμματα που τρέχουμε στον SPIM με τον έξω κόσμο, ο SPIM προσομοιώνει μερικές υποτυπώδεις υπηρεσίες λειτουργικού συστήματος για είσοδο / έξοδο (I/O) στην "κονσόλα" (ένα απλό τερματικό ASCII). Δεν είναι ανάγκη προς στιγμήν να καταλάβετε όλες τις λεπτομέρειες του πώς γίνεται η κλήση αυτών των λειτουργιών (system call) --αρκεί να μιμηθείτε το παρακάτω παράδειγμα και να καταλάβετε τις εξηγήσεις που δίνονται κάτω από αυτό. **Μελετήστε και αντιγράψτε** σε ένα αρχείο (π.χ. "ask2.s") τον παρακάτω κώδικα --ή διάφορες παραλλαγές του που προτιμάτε-- και τρέξτε τον στον SPIM.

```
# compute s = 1+2+3+...+(n-1), for n>=2
# register $16: n
# register $17: s
# register $18: i

.data          # init. data memory with the strings needed:
str_n: .asciiz "n = "
str_s: .asciiz "      s = "
str_nl: .asciiz "\n"

.text         # program memory:
.globl main   # label "main" must be global;
```

```

                                # default trap.handler calls main.
                                # global symbols can be specified
                                # symbolically as breakpoints.
.globl loop

main:                                # (1) PRINT A PROMPT:
    addi    $2, $0, 4                # system call code for print_string
    la     $4, str_n                 # pseudo-instruction: address of string
    syscall                               # print the string from str_n
                                # (2) READ n (MUST be n>=2 --not checked!):
    addi    $2, $0, 5                # system call code for read_int
    syscall                               # read a line containing an integer
    add     $16, $2, $0              # copy returned int from $2 to n
                                # (3) INITIALIZE s and i:
    add     $17, $0, $0              # s=0;
    addi    $18, $0, 1               # i=1;
loop:                                # (4) LOOP starts here
    add     $17, $17, $18            # s=s+i;
    addi    $18, $18, 1              # i=i+1;
    bne    $18, $16, loop            # repeat while (i!=n)
                                # LOOP ENDS HERE
                                # (5) PRINT THE ANSWER:
    addi    $2, $0, 4                # system call code for print_string
    la     $4, str_s                 # pseudo-instruction: address of string
    syscall                               # print the string from str_s
    addi    $2, $0, 1                # system call code for print_int
    add     $4, $17, $0              # copy argument s to $4
    syscall                               # print the integer in $4 (s)
    addi    $2, $0, 4                # system call code for print_string
    la     $4, str_nl                # pseudo-instruction: address of string
    syscall                               # print a new-line
                                # (6) START ALL OVER AGAIN (infinite loop)
    j      main                       # unconditionally jump back to main

```

Ο κώδικας αυτός υπολογίζει το άθροισμα $s=1+2+3+\dots+(n-1)$, για n μεγαλύτερο ή ίσο του 2 --προσοχή: αν δοθεί n μικρότερο του 2, ο κώδικας θα μπει σε (σχεδόν) άπειρο βρόγχο! Η "καρδιά" του κώδικα είναι τα κομμάτια (3) --αρχικοποιήσεις-- και (4) --βρόγχος υπολογισμού. Προσέξτε τις παρακάτω εξηγήσεις:

- Το κομμάτι κάθε γραμμής μετά το # είναι σχόλια, όπως είπαμε και στην άσκηση 1.
- Οι γραμμές που αρχίζουν με τελεία (".") είναι **οδηγίες** (directives) προς τον Assembler, και όχι εντολές Assembly του MIPS. Ο πλήρης κατάλογος των οδηγιών που δέχεται ο SPIM βρίσκεται στις σελίδες A-51 έως A-53 του Παραρτήματος Α, σε περίπτωση που θέλετε να τον συμβουλευθείτε.
- Η οδηγία **.data** σημαίνει ότι ό,τι ακολουθεί είναι δεδομένα (και όχι εντολές), και πρέπει να τοποθετηθούν στο κομμάτι της μνήμης που προορίζεται για αυτά (data segments) (στον SPIM αυτό αρχίζει από τη διεύθυνση 10000000 δεκαεξαδικό).
- Η οδηγία **.asciiz** σημαίνει να αρχικοποιήσει ο Assembler τις επόμενες θέσεις (bytes) μνήμης με το ASCII string που ακολουθεί, τερματισμένο με ένα NULL byte όπως και στην C. Οι ετικέτες (labels) `str_n`, `str_s`, και `str_nl`, ακολουθούμενες από άνω-κάτω τελεία ":", ορίζουν την κάθε ετικέτα σαν την διεύθυνση μνήμης όπου ο Assembler βάζει το αντίστοιχο string (τη διεύθυνση μνήμης του πρώτου byte του string).
- Η οδηγία **.text** σημαίνει, όπως είπαμε και στην άσκηση 1, ότι ό,τι ακολουθεί είναι εντολές (και όχι δεδομένα), και πρέπει να τοποθετηθούν στο κομμάτι της μνήμης που προορίζεται για αυτές (text segments).

- Οι οδηγίες `.globl` λένε στον Assembler να βάλει τις ετικέτες (labels) `main` και `loop` στον πίνακα καθολικών (global) συμβόλων. Για την ετικέτα `main`, είπαμε στην άσκηση 1 γιατί χρειάζεται αυτό. Για την ετικέτα `loop` (που είναι η αρχή του βρόχου μας), με το να την κάνουμε `global`, μπορούμε να την δίνουμε και συμβολικά --όχι μόνο αριθμητικά-- σαν διεύθυνση breakpoint στον SPIM.
- Το κομμάτι **(1)** του κώδικα είναι ένα κάλεσμα του λειτουργικού συστήματος (system call) προκειμένου να τυπωθεί το string `str_n` στην κονσόλα (πρόκειται για το string "n = " που ορίστηκε παραπάνω). Για να καταλάβει το λειτουργικό σύστημα ποιο από όλα τα system calls ζητάμε, βάζουμε στον καταχωρητή `$2` σαν παράμετρο (argument) τον αριθμό 4, που σημαίνει ότι ζητάμε το system call υπ' αριθμό 4, που είναι το `print_string` (τα system calls που υλοποιεί ο SPIM περιγράφονται στις σελίδες A-48 και A-49 του Παραρτήματος Α). Επίσης, για να ξέρει το λειτουργικό σύστημα ποιο string θέλουμε να τυπώσει στην κονσόλα, βάζουμε στον καταχωρητή `$4` σαν παράμετρο (argument) τη διεύθυνση μνήμης αυτού του string (δηλ. έναν pointer σε αυτό το string), που στην περίπτωσή μας είναι η ετικέτα `str_n` που ορίσαμε παραπάνω (το "1a" είναι ψεύδοεντολή (pseudoinstruction) του Assembler του SPIM, και όχι κανονική εντολή του MIPS, και λει στον Assembler να γεννήσει μία ή δύο πραγματικές εντολές που τοποθετούν τη διεύθυνση της ετικέτας `str_n` στον καταχωρητή `$4`, ανάλογα αν η διεύθυνση αυτή χωρά ή όχι στα 16 bits μίας σταθεράς "immediate" όπως θα δούμε αργότερα).
- Το κομμάτι **(2)** του κώδικα είναι ένα ανάλογο κάλεσμα (το κάλεσμα υπ' αριθμό 5, δηλαδή `read_int`), που περιμένει να διαβάσει έναν ακέραιο από την κονσόλα: ο προσομοιωτής θα περιμένει εκεί μέχρι να πληκτρολογήσετε έναν ακέραιο και ένα RETURN στο παράθυρο "SPIM Console". Μέσω της επόμενης εντολής, `add`, ο ακέραιος που επιστρέφει το κάλεσμα (στον καταχωρητή `$2`) αρχικοποιεί τη μεταβλητή μας `n` (στον καταχωρητή `$16`).
- Το κομμάτι **(3)** του κώδικα είναι η αρχικοποίηση των μεταβλητών `s` (καταχωρητής `$17`) και `i` (καταχωρητής `$18`) πριν μπούμε στο βρόχο.
- Το κομμάτι **(4)** του κώδικα είναι ο κυρίως βρόχος υπολογισμού. Σε κάθε επανάληψή του αυξάνει το `s` κατά `i` και το `i` κατά 1, και στη συνέχεια συγκρίνει το `i` (καταχωρητής `$18`) με το `n` (καταχωρητής `$16`) και διακλαδίζεται (πηγαίνει) πίσω στην ετικέτα `loop`, δηλαδή στην αρχή του βρόχου, όσο αυτές οι δύο μεταβλητές δεν είναι ίσες μεταξύ τους, δηλαδή όσο το `i` δεν έφτασε ακόμα το `n`. Αλλιώς, μόλις το (ήδη αυξημένο) `i` γίνει ίσο με `n`, δεν διακλαδιζόμαστε πίσω, αλλά συνεχίζουμε με την επόμενη εντολή, δηλαδή το κομμάτι (5) του κώδικα.
- Το κομμάτι **(5)** του κώδικα είναι τρία καλέσματα συστήματος για να τυπωθούν το string `str_s`, η απάντηση `s`, και το string `str_nl`. Τέλος, η εντολή `jump` στο (6) μας επιστρέφει πάντα πίσω στο `main`, ώστε το πρόγραμμα να ξανατρέχει συνεχώς μέχρι να τερματίσετε τον SPIM.

Ασκήσεις Τρεξίματος στον SPIM

- Ξεκινήστε το `spim` με τον τρόπο που είπαμε στην άσκηση 1, και φορτώστε το αρχείο με το παραπάνω πρόγραμμα που γράψατε μέσω του κουμπιού "load".
- Μέσω του κουμπιού "step" ζητήστε single-stepping, δηλαδή να εκτελούνται μία-μία οι εντολές και να τις βλέπετε. Η εκτέλεση αρχίζει στη διεύθυνση `__start` (0x00400000) όπου υπάρχει κώδικας από το αρχείο `trap.handler`. Στη διεύθυνση 0x00400014 υπάρχει μία εντολή καλέσματος διαδικασίας (`jal --jump and link`) η οποία καλεί τον κώδικά σας στο `main`, και η εκτέλεση πηγαίνει στη διεύθυνση 0x00400020. Όταν φτάσετε στο δεύτερο κάλεσμα συστήματος (0x00400030), μην ξεχάσετε να πληκτρολογήσετε έναν ακέραιο μεγαλύτερο ή ίσο του 2.

- Αφού βαρεθείτε να βλέπετε τις εντολές να εκτελούνται μία-μία, χρησιμοποιήστε το κουμπί "**breakpoints**" για να ορίσετε ένα breakpoint στο πρόγραμμά σας, π.χ. τη διεύθυνση "loop. Μετά, πείτε στο πρόγραμμα να τρέξει, μέσω του κουμπιού "run", οπότε αυτό τρέχει "σιωπηλά" μέχρι να ξαναφθάσει στη διεύθυνση loop. Έτσι μπορείτε να επιταχύνετε την παρακολούθηση ενός προγράμματος, και να το κάνετε να σταματάει σε "ενδιαφέροντα" ή "ύποπτα" σημεία.
- Τέλος, αφαιρέστε όλα τα breakpoints, και τρέξτε το πρόγραμμα κανονικά, οπότε θα βλέπετε μόνο τις εισόδους και εξόδους στην κονσόλα.

Τρόπος παράδοσης:

Θα παραδώσετε όλες τις ασκήσεις ηλεκτρονικά. Σε αυτήν του Spim όπως στην προηγούμενη άσκηση θα παραδώσετε ένα στιγμιότυπο της οθόνης καθώς τρέχετε το πρόγραμμα spim, και όταν αυτό βρίσκεται σε ένα 'ενδιαφέρον' ενδιαμέσο σημείο της επιλογής σας. Το στιγμιότυπο θα το πάρετε και θα το παραδώσετε ως εξής:

Αφού βρίσκεστε στο ενδιαφέρον κομμάτι της προσομοίωσης πατήστε το κουμπί 'Print Screen'. Μετά μπορείτε να ανοίξετε το πρόγραμμα Paint των Windows και να κάνετε εκεί 'paste', μετά θα σώσετε αυτό το αρχείο με όνομα ask2.jpg.

Έπειτα θα δημιουργήσετε ένα φάκελο με όνομα "**tp2108_ασκ2_student_ip**", όπου **student** είναι το επίθετό σας, **ip** είναι ο αριθμός μητρώου σας στη σχολή. Μέσα σε αυτό το φάκελο θα τοποθετήσετε το ask2.pdf (που θα περιέχει την ask2.1.txt, και επίσης το αρχείο "ask2.jpg").

Την ώρα του επόμενου εργαστηρίου θα ανοίξει ένας ftp server με τον οποίο θα συνδεθείτε και θα μεταφέρετε εκεί το φάκελο με τη δουλειά σας.