



ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ  
ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ

---

Οργάνωση Υπολογιστών

**Ασκήσεις 2: Βρόχοι και Επικοινωνία Κονσόλας στον SPIM**

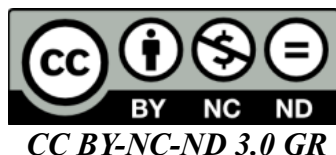
Μανόλης Γ.Η. Κατεβαίνης

Τμήμα Επιστήμης Υπολογιστών

---

## Άδειες Χρήσης

- Το παρόν εκπαιδευτικό υλικό υπόκειται στην άδεια χρήσης **Creative Commons** και ειδικότερα **Αναφορά – Μη εμπορική Χρήση – Όχι Παράγωγο Έργο 3.0 Ελλάδα** (*Attribution – Non Commercial – Non-derivatives 3.0 Greece*)



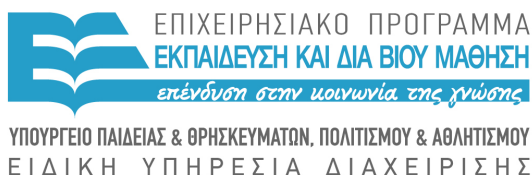
- Εξαιρείται από την ως άνω άδεια υλικό που περιλαμβάνεται στις διαφάνειες του μαθήματος, και υπόκειται σε άλλου τύπου άδεια χρήσης. Η άδεια χρήσης στην οποία υπόκειται το υλικό αυτό αναφέρεται ρητώς.

## Χρηματοδότηση

- Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στα πλαίσια του εκπαιδευτικού έργου του διδάσκοντα.
- Το έργο «**Ανοικτά Ακαδημαϊκά Μαθήματα στο Πανεπιστήμιο Κρήτης**» έχει χρηματοδοτήσει μόνο τη αναδιαμόρφωση του εκπαιδευτικού υλικού.
- Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «**Εκπαίδευση και Δια Βίου Μάθηση**» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.



Ευρωπαϊκή Ένωση  
Ευρωπαϊκό Κοινωνικό Ταμείο



Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης



## Σειρά Ασκήσεων 2: Βρόχοι και Επικοινωνία Κονσόλας στον SPIM

### 2.1 Εντολές Διακλάδωσης υπό Συνθήκη στον MIPS

Όπως λέγαμε και στην §1.3, 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 (κατά 4 στον MIPS, αφού όλες οι εντολές του MIPS έχουν μέγεθος 4 Bytes), οπότε αυτός (ο PC) δείχνει στην επόμενη (την "από κάτω") εντολή. Η σειριακή αυτή εκτέλεση εντολών διακόπτεται όταν εκτελείται μία 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 "μπλόκ" εντολών.

Γιά 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 το ξεχωριστό, 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".

### 2.2 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. `ex02.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.
.globl loop                # global symbols can be specified
                           # symbolically as breakpoints.

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 κωδικά 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 A, 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 A). 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 ψευδοεντολή (pseudoinstruction) του Assembler του SPIM, και όχι κανονική εντολή του MIPS, και λέει 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_n1`. Τέλος, η εντολή `jump` στο (6) μας επιστρέφει πάντα πίσω στο `main`, ώστε το πρόγραμμα να ξανατρέχει συνεχώς μέχρι να τερματίσετε τον SPIM.

### **Άσκηση 2.3: Τρέξιμο στον SPIM**

- Ξεκινήστε το **QtSpim** με τον τρόπο που είπαμε στην §1.4, και φορτώστε το αρχείο με το παραπάνω πρόγραμμα που γράψατε μέσω του κουμπιού "load file".
- Μέσω του κουμπιού "**Single step**" ζητήστε single-stepping, δηλαδή να εκτελούνται μια-μια 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 μια εντολή καλέσματος διαδικασίας (`jal --jump and link`) η οποία καλεί τον κώδικά σας στο `main`, και η εκτέλεση πηγαίνει στη διεύθυνση 0x00400024. Όταν φτάσετε στο δεύτερο κάλεσμα συστήματος (0x00400034), μην ξεχάσετε να πληκτρολογήσετε έναν ακέραιο μεγαλύτερο ή ίσο του 2 (πρέπει να τον πληκτρολογήσετε *αφού* ο SPIM φτάσει εκεί --παλαιότερες πληκτρολογήσεις συνήθως χάνονται).
- Αφού βαρεθείτε να βλέπετε τις ενοχές να εκτελούνται μία-μία, ορίστε "**breakpoint(s)**" στο πρόγραμμά σας: κάντε δεξιά κλικ στην επιθυμητή γραμμή --π.χ. τη διεύθυνση "loop"-- και επιλέξτε "Set Breakpoint" από το εμφανιζόμενο μενού. Μετά, πείτε στο πρόγραμμα να τρέξει, μέσω του κουμπιού "run", οπότε αυτό τρέχει "σιωπηλά" μέχρι να ξαναφτάσει 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 --κάνοντας δεξί κλικ ξανά στις γραμμές με τα breakpoints και επιλέγοντας "Clear Breakpoint"-- και τρέξτε το πρόγραμμα κανονικά, οπότε θα βλέπετε μόνο τις εισόδους και εξόδους στην κονσόλα.