



ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ  
ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ

---

Οργάνωση Υπολογιστών

**Ασκήσεις 5: Εντολές Συγκρίσεων και Μεταφοράς  
Ελέγχου**

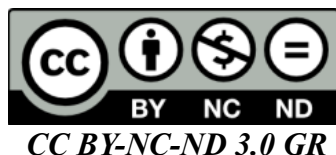
Μανόλης Γ.Η. Κατεβαίνης

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*)



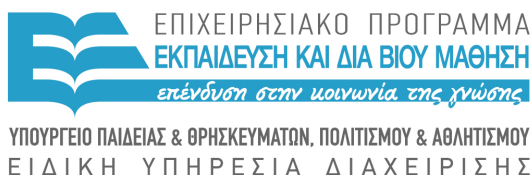
- Εξαιρείται από την ως άνω άδεια υλικό που περιλαμβάνεται στις διαφάνειες του μαθήματος, και υπόκειται σε άλλου τύπου άδεια χρήσης. Η άδεια χρήσης στην οποία υπόκειται το υλικό αυτό αναφέρεται ρητώς.

## 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 5:

## Εντολές Συγκρίσεων και Μεταφοράς Ελέγχου

**Βιβλίο:** Διαβάστε την §2.7, σελίδες 145-151, καθώς και τις σελίδες 171-175.

### 5.1 Περίληψη Συγκρίσεων, Διακλαδώσεων, και Αλμάτων στον MIPS

Οι εντολές μεταφοράς ελέγχου (CTI, control transfer instructions) καθορίζουν να εκτελεστεί σαν επόμενη εντολή --πάντοτε ή υπό ορισμένες συνθήκες μόνο-- μια έντολη άλλη από την "επόμενη από κάτω" τους εντολή. Όταν η μεταφορά ελέγχου γίνεται υπό συνθήκη, οι εντολές συνήθως ονομάζονται **διακλαδώσεις** (branch). Όταν η μεταφορά γίνεται πάντοτε, οι εντολές συνήθως λέγονται **άλματα** (jump). Επίσης υπάρχουν καλέσματα διαδικασιών, λειτουργικού συστήματος, και επιστροφές από αυτά.

Στη γλώσσα Assembly, η διεύθυνση προορισμού της διακλάδωσης δηλώνεται απλά με μια ετικέτα (label), και αναλαμβάνει ο Assembler να υπολογίσει και να βάλει τη σωστή δυαδική τιμή. Στη γλώσσα μηχανής, οι εντολές **διακλάδωσης** ακολουθούν το **I-format**, και η διεύθυνση προορισμού προκύπτει ως εξής:

$$PC\_new := (PC\_br + 4) + 4 * ImmOffset (signed)$$

όπου  $PC\_br$  είναι η διεύθυνση της ίδιας της εντολής διακλάδωσης,  $ImmOffset$  είναι η σταθερή ποσότητα των 16 bits του I-format θεωρούμενη ως προσημασμένος αριθμός σε συμπλήρωμα ως προς 2 (δηλαδή sign-extended), και  $PC\_new$  είναι η διεύθυνση της εντολής προορισμού σε περίπτωση επιτυχίας της διακλάδωσης. Η αύξηση ( $PC\_br+4$ ) γίνεται για λόγους ευκολίας του hardware (όλες οι εντολές αυξάνουν τον PC κατά 4). Ο πολλαπλασιασμός του  $ImmOffset$  επί 4 γίνεται για να εκμεταλλευτούμε το γεγονός ότι η διεύθυνση όλων των εντολών του MIPS είναι ακέραιο πολλαπλάσιο του 4, κι έτσι να τετραπλασιάσουμε το "βεληνεκές" των διακλαδώσεων --με άλλα λόγια, ο αριθμός  $ImmOffset$  μετράει πλήθος εντολών μπροστά (θετικός) ή πίσω (αρνητικός), αντί να μετρά πλήθος bytes μπροστά ή πίσω. (στην πραγματικότητα, ο MIPS έχει "καθυστερημένες (όψιμες) διακλαδώσεις" (delayed branches), για λόγους καλύτερης εκμετάλλευσης της ομοχειρίας (pipelining) του hardware, αλλά εμείς θα το αγνοήσουμε σε αυτό το μάθημα --το θέμα αυτό θίγεται στο HY-425).

Στον MIPS, η συνθήκη διακλάδωσης μπορεί να έχει περιορισμένες μόνο μορφές, για λόγους ταχύτητας. Οι μόνες συνθήκες διακλαδώσεων που υπάρχουν αφορούν συγκρίσεις ενός καταχωρητή με το μηδέν (δεν θα ασχοληθούμε με αυτές στο "δικό μας" υποσύνολο του MIPS), και συγκρίσεις δύο καταχωρητών *μόνο για ισότητα ή ανισότητα* και όχι για άλλων μορφών σχέσεις (μικρότερος, μεγαλύτερος, κλπ). Οι εντολές αυτές είναι:

- **beq \$rs, \$rt, label** # διακλάδωση εάν:  $\$rs == \$rt$
- **bne \$rs, \$rt, label** # διακλάδωση εάν:  $\$rs != \$rt$

Οι υπόλοιπες μορφές συγκρίσεων, που δεν γίνονται μέσα στις εντολές διακλάδωσης, υλοποιούνται με ειδικές, ξεχωριστές, αριθμητικές εντολές σύγκρισης. Πρόκειται για εντολές ανάλογες προς την πρόσθεση ή την αφαίρεση, μόνο που το αποτέλεσμά τους είναι τύπου Boolean αντί τύπου ακέραιος. Τέτοια αποτελέσματα τύπου Boolean έχουν δύο μόνο δυνατές τιμές: 0 για ψευδές, και 1 για αληθές. Τα αποτελέσματα αυτά γράφονται στους γνωστούς μας, κανονικούς (32μπιτους) καταχωρητές, σαν οι ακέραιοι 0 ή 1, δηλαδή στο LS bit του καταχωρητή, με όλα τα υπόλοιπα bits του καταχωρητή μηδενικά. Οι εντολές σύγκρισης που εμείς θα έχουμε στο δικό μας υποσύνολο του MIPS είναι οι παρακάτω δύο (αυτές κάνουν σύγκριση προσημασμένων - ο πραγματικός MIPS έχει κι άλλες, που κάνουν σύγκριση μη προσημασμένων, αλλά εμείς δεν θα ασχοληθούμε με εκείνες):

- **slt \$rd, \$rs, \$rt** # set less than --αριθμητική σύγκριση των καταχωρητών:  $\$rs < \$rt$ . Το αποτέλεσμα, τύπου Boolean, γράφεται στον καταχωρητή \$rd.
- **slti \$rd, \$rs, imm** # set less than immediate --αριθμητική σύγκριση καταχωρητή και προσημασμένης (sign-extended) σταθερής ποσότητας imm:  $\$rs < imm$ . Το αποτέλεσμα (Boolean) γράφεται στον καταχωρητή \$rd.

Εκτός από τις εντολές διακλάδωσης υπό συνθήκη, το υποσύνολο εντολών του MIPS που χρησιμοποιούμε στο μάθημα περιλαμβάνει και τις παρακάτω άλλες εντολές μεταφοράς ελέγχου:

### **j target**

Αλμα (jump) χωρίς συνθήκη: επόμενη προς εκτέλεση εντολή είναι η εντολή στη διεύθυνση target. Χρησιμοποιεί το J-format, το οποίο φαίνεται στη σελίδα 171 του βιβλίου, επάνω. Η τελική διεύθυνση προορισμού (32 bits) προκύπτει από τα 4 παλαιά MS bits του PC, τα 26 bits του πεδίου προορισμού της εντολής, και από 2 μηδενικά LS bits, όπως εξηγείται στη σελίδα 172 του βιβλίου (κάτω). Στην σελίδα 174 φαίνεται και η χρήση της εντολής jump, μαζί με μια branch, για την έμμεση σύνθεση διακλάδωσης υπό συνθήκη σε απόσταση μεγαλύτερη από το βεληγκεές των απλών διακλαδώσεων.

### **jr \$rs**

Αλμα σε προορισμό που καθορίζεται από καταχωρητή (jump register): επόμενη προς εκτέλεση εντολή είναι η εντολή στη διεύθυνση που περιέχεται στον καταχωρητή rs (με άλλα λόγια, ο \$rs περιέχει τη διεύθυνση προορισμού, δηλαδή έναν pointer στην επόμενη προς εκτέλεση εντολή). Η εντολή αυτή μας επιτρέπει να μεταφέρουμε τον έλεγχο (την εκτέλεση του προγράμματος) σε αυθαίρετη θέση μνήμης, η οποία μπορεί και να ποικίλει κατά την εκτέλεση του προγράμματος (run-time variable) και πιθανόν να εξαρτάται και από τα δεδομένα (data dependent). Χρησιμοποιείται: (α) για επιστροφή από διαδικασία, όπως περιγράφεται αμέσως παρακάτω, (β) για μεταφορά του ελέγχου οσοδήποτε μακριά (σελίδα 172 κάτω του βιβλίου), και (γ) για μετάφραση του **switch** statement.

Για το **switch** statement, όπως περιγράφεται στη σελίδα 150 του βιβλίου, η βασική ιδέα είναι η εξής: ο κώδικας της κάθε περίπτωσης (case) γράφεται σε κάποια περιοχή μνήμης, και ο compiler θυμάται σε ποιά διεύθυνση αρχίζει αυτός ο κώδικας, για την καθεμία περίπτωση. Στη συνέχεια, ο compiler κατασκευάζει έναν πίνακα (array), τον "jump table", ο οποίος περιέχει σε κάθε θέση του αυτές τις διευθύνσεις όπου αρχίζει ο κώδικας της κάθε περίπτωσης, με τη σειρά, για όλες τις (αριθμητικές) τιμές της κάθε περίπτωσης. Όταν είναι να εκτελεστεί το **switch** statement, παίρνουμε την τιμή της μεταβλητής του switch (αυτήν της οποίας την κάθε περίπτωση τιμών εξετάζουμε), και χρησιμοποιούμε αυτή την τιμή σαν **index** στον jump table, για να επιλέξουμε ένα από τα περιεχόμενα του jump table --αυτό που αντιστοιχεί στην περίπτωση της τωρινής τιμής της μεταβλητής του switch. Το στοιχείο του πίνακα που διαλέξαμε είναι η διεύθυνση του κώδικα που πρέπει να εκτελεστεί για την περίπτωση της μεταβλητής που τώρα μας ήλθε. Αυτό ακριβώς το στοιχείο του jump table το φέρνουμε σε έναν καταχωρητή \$r, και στη συνέχεια εκτελούμε την εντολή **jr \$r**, η οποία κάνει ώστε η επόμενη εντολή που θα εκτελεστεί να είναι εκεί που δείχνει ο \$r, δηλαδή εκεί που μας είπε ο jump table να πάμε για την περίπτωση της τιμής που τώρα είχε η μεταβλητή.

### **jal target**

Αλμα και Σύνδεση (jump and link) -- κάλεσμα διαδικασίας: έχει το ίδιο format με την εντολή jump, και κάνει τα ίδια με εκείνην (ίδια διεύθυνση προορισμού), συν, επιπλέον, αποθηκεύει τη διεύθυνση της επόμενης της εντολής (παλαιό PC συν 4) στον καταχωρητή 31 (\$ra, ή \$31). Χρησιμοποιείται για κάλεσμα διαδικασίας, όπως περιγράφεται στις σελίδες 152-153 του βιβλίου. Η διεύθυνση που αποθηκεύεται μπορεί στη συνέχεια να χρησιμοποιηθεί για επιστροφή από τη διαδικασία, μέσω της εντολής **jr \$ra**. Η διεύθυνση επιστροφής

αποθηκεύεται πάντα στον \$ra (\$31) --το "31" δεν φαίνεται πουθενά μέσα στην εντολή `jal`, απλώς το βάζει το hardware αυτόματα.

### **Άσκηση 5.2: Διακλάδωση με Σύγκριση Καταχωρητή-Σταθεράς**

Είπαμε ότι, για λόγους ταχύτητας, οι μόνες συνθήκες διακλάδωσης του MIPS που αφορούν δύο αυθαίρετους αριθμούς (όχι έναν αριθμό και το μηδέν) είναι συγκρίσεις ισότητας/ανισότητας και όχι συγκρίσεις μεγαλύτερος/μικρότερος. Όμως, επίσης, οι εντολές αυτές (`beq`, `bne`) υπάρχουν μόνο στη μορφή που δέχεται δύο καταχωρητές σαν τελεστέους, και δεν μπορούν να συγκρίνουν καταχωρητή με σταθερή ποσότητα (`immediate`). Ο λόγος δεν μπορεί να έχει να κάνει με ταχύτητα, αφού η σύγκριση ισότητας/ανισότητας καταχωρητή-σταθεράς είναι τουλάχιστο το ίδιο γρήγορη με την αντίστοιχη σύγκριση δύο καταχωρητών. Γιατί λοιπόν πιστεύετε ότι δεν υπάρχουν τέτοιες εντολές "`beqi`" και "`bnei`" στον MIPS; Δώστε την απάντησή σας γραπτά και εξηγήστε.

### **Άσκηση 5.3: Εντολές Σύγκρισης και Διακλάδωσης**

Έστω ότι η μεταβλητή `i` βρίσκεται στον καταχωρητή \$16, η μεταβλητή `j` στον καταχωρητή \$17, και ότι `CONST` σημαίνει μια αυθαίρετη προσημασμένη σταθερή ποσότητα μέχρι και 16 bits. Συνθέστε τις παρακάτω περιπτώσεις κώδικα χρησιμοποιώντας αποκλειστικά και μόνο εντολές μεταξύ των:

**`beq, bne, slt, slti, addi`**

και καμια άλλη. Όπου χρειάζεστε προσωρινό καταχωρητή, χρησιμοποιήστε τον \$at (Assembler temporary) ο οποίος είναι ο \$1 (αυτόν χρησιμοποιεί ο Assembler για να συνθέτει τις ψευδοεντολές του). Δώστε τις απαντήσεις σας γραπτά.

- i. `if ( i == j ) goto L1;` (ίσο)
- ii. `if ( i != j ) goto L1;` (διάφορο)
- iii. `if ( i < j ) goto L1;` (μικρότερο)
- iv. `if ( i <= j ) goto L1;` (μικρότερο ή ίσο)
- v. `if ( i > j ) goto L1;` (μεγαλύτερο)
- vi. `if ( i >= j ) goto L1;` (μεγαλύτερο ή ίσο)
- vii. `if ( i == CONST ) goto L1;` (ίσο)
- viii. `if ( i != CONST ) goto L1;` (διάφορο)
- ix. `if ( i < CONST ) goto L1;` (μικρότερο)
- x. `if ( i <= CONST ) goto L1;` (μικρότερο ή ίσο)
- xi. `if ( i > CONST ) goto L1;` (μεγαλύτερο)
- xii. `if ( i >= CONST ) goto L1;` (μεγαλύτερο ή ίσο)

**Υπόδειξη:** παίξτε με τη σειρά που βάζετε τους 2 τελεστέους πηγής στην εντολή σύγκρισης, με τον καταχωρητή \$0 που περιέχει πάντα μηδέν (ή "ψευδές"), με το είδος της διακλάδωσης (ίσο/άνισο ή ψευδές/αληθές), με τις σταθερές `CONST`, `CONST+1`, `CONST-1`, `-CONST`, και με τους (πολλούς) συνδυασμούς όλων αυτών. Θα εκτιμήστε το πόσα πολλά μπορεί να κάνει το software, εκμεταλλευόμενο λίγους, προσεκτικά επιλεγμένους δομικούς λίθους hardware, χωρίς απώλεια ταχύτητας!

### **Άσκηση 5.4: Μετάφραση Βρόχου "While"**

Στην παράγραφο 2.7 του βιβλίου, σελ. 147, υπάρχει ένας βρόχος μεταφρασμένος σε Assembly. Θεωρήστε την εξής μικρή παραλλαγή του βρόχου αυτού που ψάχνει σε έναν πίνακα (ακεραίων) `table[]` μέχρι να βρεί μία επιθυμητή τιμή `value` (που ξέρουμε ότι υπάρχει στον πίνακα):

```
i=0; while (table[i] != value) { i = i+1; }
```

Ας υποθέσουμε ότι η μεταβλητή  $i$  βρίσκεται στον καταχωρητή  $\$s3$ , η τιμή  $value$ , βρίσκεται στον καταχωρητή  $\$s5$ , και ότι ο καταχωρητής  $\$s6$  περιέχει τη διεύθυνση του στοιχείου  $table[0]$ , δηλαδή τη διεύθυνση βάσης του πίνακα. Τότε, ο βρόχος αυτός, όπως υπάρχει στο βιβλίο, είναι ως εξής, όπου η εντολή `sll` είναι η "shift left logical", δηλαδή αριστερή ολίσθηση, εδώ κατά 2 θέσεις (bits), δηλαδή πολλαπλασιασμός επί 4:

```
    add  $s3, $0, $0      # i=0;
Loop: sll  $t1, $s3, 2    # t1 = 4 * i
    add  $t1, $t1, $s6   # t1 = διεύθυνση του table[i]
    lw   $t0, 0($t1)    # t0 = table[i]
    beq  $t0, $s5, Exit  # if table[i] == value goto Exit
    addi $s3, $s3, 1    # i = i+1
    j    Loop           # επανάληψη: goto Loop
Exit: ....             # υπόλοιπο πρόγραμμα
```

Παρατηρήστε τώρα ότι σε κάθε ανακύκλωση αυτού του βρόχου εκτελούνται τόσο μια διακλάδωση υπό συνθήκη (branch), όσο και ένα άλμα χωρίς συνθήκη (jump). Όμως, μόνον οι εξαιρετικά απλοϊκοί μεταφραστές θα έφτιαχναν τέτοιο κώδικα --οι συνηθισμένοι μεταφραστές παράγουν κώδικα που τρέχει πιο γρήγορα στη *συνηθισμένη περίπτωση*. Η συνηθισμένη περίπτωση είναι ο βρόχος να επαναλαμβάνεται αρκετές φορές πριν βγούμε από αυτόν --γύρω στις 10 φορές κατά μέσον όρο, λένε οι στατιστικές.

Ξαναγράψτε το βρόχο σε Assembly, ούτως ώστε **μόνο ένα** branch ή jump να εκτελείται σε κάθε ανακύκλωση. Κατά την είσοδο ή την έξοδο από το βρόχο επιτρέπεται να εκτελούνται δύο εντολές μεταφοράς ελέγχου --αυτό που μας ενδιαφέρει είναι να γλυτώνουμε τη μια από αυτές κατά τις υπόλοιπες επαναλήψεις του βρόχου, που αποτελούν και την πλειοψηφία των φορών που αυτός εκτελείται. Έστω ότι ο βρόχος εκτελείται 10 φορές. Πόσες εντολές συνολικά εκτελούνταν με τον παλιό κώδικα, και πόσες συνολικά με τον νέο;