Σειρά Ασκήσεων 5: Εντολές Συγκρίσεων και Διακλαδώσεων

Προθεσμία έως Τετάρτη 10 Μαρτίου 2010, 23:59 (βδομάδα 5.2)(από βδομάδα 3.3)

[**Up** - Table of Contents] [printer version - **PDF**]
[**Prev** - 4. Instruction Formats] [6. Procedure Call - **Next**]

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. Στον MIPS, η συνθήκη διακλάδωσης μπορεί να έχει περιορισμένες μόνο μορφές, γιά 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 γιά ισότητα ή ανισότητα* και όχι γιά 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`**

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`

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 γιά λόγους ευκολίας του hardware (όλες οι εντολές αυξάνουν τον PC κατά 4). Ο πολλαπλασιασμός του ImmOffset επί 4 γίνεται γιά 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 διακλαδώσεις" (delayed branches), γιά λόγους καλύτερης εκμετάλλευσης της ομοχειρίας (pipelining) του hardware, αλλά εμείς θα το αγνοήσουμε σε αυτό το μάθημα –το θέμα αυτό συζητείται εν εκτάσει στο HY-425). Στην υλοποίηση με καθυστερημένες διακλαδώσεις ο επεξεργαστής εκτελεί πάντα και χωρίς συνθήκη την εντολή που ακολουθεί το branch, για αυτό ο μετρητής προγράμματος αυξάνεται πάντα κατά 4 bytes μετά το branch πριν υπολογιστεί η απόσταση από την εντολή προορισμού. Εδώ πρέπει να σημειώσουμε ότι εξ'ορισμού ο προσομοιωτής SPIM δεν προσομοιώνει τις καθυστερημένες διακλαδώσεις και κατά συνέπεια η απόσταση από την εντολή προορισμού υπολογίζεται σαν **`PC_new := PC_br+4*ImmOffset`**.

Οι υπόλοιπες μορφές συγκρίσεων, που δεν γίνονται μέσα στις εντολές διακλάδωσης, υλοποιούνται με ειδικές, ξεχωριστές, αριθμητικές εντολές σύγκρισης. Πρόκειται γιά 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 γιά ψευδές, και 1 γιά 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:

- **`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 148 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 150 του βιβλίου (κάτω). Στην ίδια σελίδα φαίνεται και η χρήση της εντολής jump, μαζί με μια branch, γιά την έμμεση σύνθεση διακλάδωσης υπό συνθήκη σε απόσταση μεγαλύτερη από το βεληνεκές 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 γιά μετάφραση του **switch** statement, όπως περιγράφεται στη σελίδα 94 του βιβλίου (Α' τόμος Ελληνικής έκδοσης), γιά μεταφορά του ελέγχου οσοδήποτε μακριά (σελίδα 117 κάτω του βιβλίου), και γιά επιστροφή από διαδικασία όπως περιγράφεται αμέσως παρακάνω.

jal target

Αλμα και Σύνδεση (jump and link) –κλήση διαδικασίας: έχει το ίδιο format με την εντολή jump, και κάνει τα ίδια με εκείνην (ίδια διεύθυνση προορισμού), συν, επιπλέον, αποθηκεύει τη διεύθυνση της επόμενής της εντολής (παλαιό PC συν 4) στον καταχωρητή 31 ($ra, ή $31). Χρησιμοποιείται γιά κλήση διαδικασίας, όπως περιγράφεται στις σελίδες 97-98 του βιβλίου (Α' τόμος Ελληνικής έκδοσης). Η διεύθυνση που αποθηκεύεται μπορεί στη συνέχεια να χρησιμοποιηθεί γιά 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: Υλοποίηση if-then-else

Έστω ότι θέλουμε να υλοποιήσουμε μία απλή δομή **if-then-else** της γλώσσας προγραμματισμού C, όπως στον παρακάτω κώδικα, ο οποίος αναθέτει σε μία μεταβλητή το μέγιστο μεταξύ δύο στοιχείων.

```
if (i < j)
  a = j;
else /* i >= j */
  a = i;
```

Για την υλοποίηση αυτή πρέπει να αξιοποιήσουμε τις εντολές διακλάδωσης σε συνδυασμό με τις εντολές σύγκρισης. Αν υποθέσουμε ότι οι μεταβλητές **i, j, a** βρίσκονται στους καταχωρητές **$8, $9, $16** (δηλαδή τους **$t0, $t1, $s0** αν χρησιμοποιούμε συμβολικά ονόματα καταχωρητών), τότε η σύγκριση **if (i < j)** μπορεί να μεταφραστεί στην εντολή:

```
                slt $10, $8,$9          #if (i < j), $10 = 1
                bne $10, $0, thenpath   #jump to then path
                add $16, $8, $0         # this is the else path, a = i;
                j   exit                # we don't want to execute the then path!
then:           add $16, $9, $0 #this is the then path, a = j;
exit:
```

Παρατηρήστε ότι σε αντίθεση με τη γλώσσα προγραμματισμού C, στην assembly αν η συνθήκη είναι αληθής εκτελείται το άλμα στο "μονοπάτι" **then** και όχι η εντολή που βρίσκεται κάτω από τον έλεγχο της συνθήκης. Αντίθετα η συγκεκριμένη εντολή υλοποιεί το "μονοπάτι" **else**. Ωστόσο, δεδομένου ότι ο κώδικας πρέπει να εκτελέσει είτε το μονοπάτι **then**, είτε το μονοπάτι **else** αλλά όχι και τα δύο μονοπάτια, χρησιμοποιούμε μία εντολή **j** για να αποφύγουμε το μονοπάτι **then** αν έχουμε εκτελέσει το μονοπάτι **else**. Ας υποθέσουμε ότι θέλουμε να "διορθώσουμε" αυτή την ασυμμετρία μεταξύ C και assembly έτσι ώστε στον κώδικά assembly να εμφανίζεται πρώτα το **then** μονοπάτι και μετά το **else** μονοπάτι. Δώστε τον ισοδύναμο κώδικα σε assembly μετά από αυτή την αλλαγή. Εάν σας εξυπηρετεί, δοκιμάστε τον κώδικα στον SPIM για να επαληθεύσετε ότι η λύση σας είναι σωστή.

Στις γλώσσες προγραμματισμού υψηλού επιπέδου χρησιμοποιούνται συχνά δομές **switch...case** οι

οποίες επιλέγουν μεταξύ τυπικά πολλών εναλλακτικών μονοπατιών κώδικα που ακολουθούνται με βάση την τιμή μίας μεταβλητής ελέγχου, όπως για παράδειγμα ο κώδικας:

```
switch (n) {
        case 0: a = b + c; break;
        case 1: a = d + e; break;
        case 2: a = b - c; break;
        case 3: a = d - e; break;
        default: break;
}
```

Αν και οι δομές αυτές θα μπορούσαν να υλοποιηθούν με ακολουθίες από δομές **if-then-else**, αυτό θα περιέπλεκε αρκετά τον κώδικα και θα αύξανε σημαντικά το μέγεθός του. Για να παράγουμε πιο σύντομο και αποδοτικό κώδικα σε αυτές τις περιπτώσεις χρησιμοποιούμε πίνακες διευθύνσεων αλμάτων (jump address tables) στη μνήμη. Συγκεκριμένα, για κάθε πιθανό μονοπάτι του **case** αναθέτουμε μία ετικέτα (που δείχνει στην αρχή του κώδικα του μονοπατιού), της οποίας τη διεύθυνση αποθηκεύουμε στη μνήμη, σε έναν πίνακα. Ο πίνακας αυτός περιέχει τόσες διευθύνσεις όσα τα εναλλακτικά μονοπάτια της δομής **switch...case** και αρχικοποιείται με τις ετικέτες, κάθε μία από τις οποίες ορίζει ένα μονοπάτι. Ο κώδικας που υλοποιεί το **switch...case** ελέγχει την τιμή της μεταβλητής ελέγχου, και τη χρησιμοποιεί σαν δείκτη στον πίνακα με τις ετικέτες. Για παράδειγμα, αν χρησιμοποιείται μία μεταβλητή ελέγχου **n** όπως παραπάνω, ο κώδικας πολλαπλασιάζει το **n** με 4 για να βρει την αντίστοιχη ετικέτα, φορτώνει τη διεύθυνση της ετικέτας σε έναν καταχωρητή (με εντολές **la, lw**) και κατόπιν χρησιμοποιεί την εντολή **jr** με την τιμή του καταχωρητή για να κάνει το άλμα στο σωστό μονοπάτι. Η δομή **switch...case** είναι μία απο τις σημαντικές περιπτώσεις χρήσης της εντολής **jr** Στο μάθημα θα υλοποιήσουμε ένα παράδειγμα δομής **switch...case**

Άσκηση 5.3: Διακλάδωση με Σύγκριση Καταχωρητή-Σταθεράς

Είπαμε ότι, γιά 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. Ομως, 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 και εξηγήστε.

Άσκηση 5.4: 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 γιά να συνθέτει τις ψευδοεντολές του).

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 5.5: Μετάφραση Βρόχου "While"

Μεταφράστε τον παρακάτω κώδικα C που υλοποιεί ένα βρόχο **`while`** σε Assembly. Θεωρήστε ότι ο καταχωρητής **`$16`** περιέχει την αρχική διεύθυνση του πίνακα **`a[]`** στη μνήμη και ότι ο καταχωρητής **`$9`** περιέχει τον δείκτη του βρόχου **`i`**. Ο κώδικάς σας πρέπει: i) να υπολογίζει σωστά τις θέσεις μνήμης των στοιχείων του πίνακα **`a[]`** χρησιμοποιώντας μόνο εντολές **`add, addi`**; ii) να εκτελεί **μόνο ένα branch ή jump** 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).

```
while (a[i] >= 4) {
 i = i+1;
}
```

Έστω ότι ο βρόχος εκτελείται 225 φορές. Πόσες εντολές συνολικά εκτελούνται με τον κώδικά σας;

Τρόπος Παράδοσης: Για να παραδώσετε την άσκηση χρησιμοποιήστε την ακόλουθη διαδικασία. Συνδεθείτε σε ένα μηχάνημα Linux του τμήματος με το login και password που χρησιμοποιείτε στο Τμήμα. Η λίστα μηχανημάτων Linux του Τμήματος είναι διαθέσιμη **εδώ**. Ετοιμάστε ένα directory με το αρχείο που σας ζητάει η άσκηση (ask5.txt). Ας υποθέσουμε ότι το όνομα του directory είναι <somepath>/mydir. Μετακινηθείτε στο directory <somepath> και εκτελέστε την εντολή:

```
/usr/local/bin/submit exercise5@hy225 mydir
```

Η διαδικασία submit θα σας ζητήσει να επιβεβαιώσετε την αποστολή των αρχείων.

Περισσότερες πληροφορίες και αναλυτικές οδηγίες για τη νέα διαδικασία submit είναι διαθέσιμες εκτελώντας:

```
man submit
```

σε κάποιο από τα μηχανήματα Linux του Τμήματος. Σημειώστε ότι η απομακρυσμένη πρόσβαση στα μηχανήματα του Τμήματος γίνεται από 1.2.2010 με νέο ασφαλή τρόπο μέσω gateways και με χρήση tunnel.
