ΗΥ-225: Οργάνωση Υπολογιστών
Άνοιξη 2025

Τμ. Επ. Υπολογιστών
© Πανεπιστήμιο Κρήτης

Σειρά Ασκήσεων 5: Διακλαδώσεις - Άλματα, If-Then-Else, Βρόχοι, Switch, Κλήση, Συγκρίσεις

Από νωρίς την 4η για αργά την 5η εβδομάδα του Εξαμήνου
Υπενθύμιση: η Εξέταση Προόδου πλησιάζει

Βιβλίο: Διαβάστε τις §2.7, σελίδες 134-140, και §2.10, σελίδες 157-165.

5.1 Διακλαδώσεις υπό Συνθήκη στον RISC-V

Οι εντολές μεταφοράς ελέγχου (CTI, control transfer instructions) καθορίζουν να εκτελεστεί σαν επόμενη εντολή –πάντοτε ή υπό συνθήκη– μια έντολη άλλη από την "επόμενη από κάτω" τους εντολή. Όταν η μεταφορά γίνεται υπό συνθήκη, οι εντολές συνήθως ονομάζονται **διακλαδώσεις** (branch). Όταν η μεταφορά γίνεται πάντοτε, δηλαδή χωρίς συνθήκη, οι εντολές συνήθως λέγονται **άλματα** (jump). Μία παραλλαγή άλματος είναι το κάλεσμα διαδικασίας, και μιά άλλη η επιστροφή από διαδικασία.

Το πιό δημοφιλές *addressing mode*, δηλαδή τρόπος διευθυνσιοδότησης ή τρόπος προσδιορισμού της διεύθυνσης προορισμού των διακλαδώσεων, είναι το "**PC-Relative** addressing mode, δηλαδή ο προορισμός της διακλάδωσης δηλώνεται μέσα στην εντολή *σε σχέση με την εντολή διακλάδωσης αυτή καθεαυτή*, δηλαδή σαν σχετική απόσταση (μετακίνηση) από εδώ που βρισκόμαστε τώρα προς τα εκεί όπου θέλουμε να πάμε. Ο RISC-V χρησιμοποιεί και αυτός PC-Relative mode γιά τις εντολές CTI. Το PC-relative branch addressing mode mode προσφέρει δύο σημαντικά πλεονεκτήματα. Πρώτον, επειδή η απόσταση αυτή είναι συνήθως μικρή, αυτή μπορεί να χωρέσει σε λιγότερα bits μέσα στην εντολή· και είναι συνήθως μικρή επειδή οι διακλαδώσεις είναι εντός των if-then-else και εντός των βρόχων, που και τα δύο είναι εντός της ίδιας διαδικασίας (procedure), και όπου όλα αυτά είναι συνήθως σχετικά μικρά. Δεύτερον, όταν ο Linker συνενώνει αρχεία που έχουν γίνει compiled χωριστά γιά να φτιάξει ένα ενιαίο εκτελέσιμο (ή όταν κάνουμε dynamic linking at runtime), τότε η καθε διαδικασία καταλήγει να ξεκινά από μιά "καινούργια" διεύθυνση που δεν την ξέραμε νωρίτερα, άρα ο Linker έχει σχετικές διορθώσεις διευθύνσεων μέσα στον κώδικα που πρέπει να κάνει. Οι διορθώσεις αυτές είναι λιγότερες όπου χρησιμοποιείται PC-relative addressing mode, διότι όταν μετακινείται *ολόκληρη* μιά διαδικασία σε νέες διευθύνσεις, οι *σχετικές* αποστάσεις των εντολών *μέσα* στη διαδικασία *δεν* αλλάζουν.

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 του RISC-V, οι εντολές **διακλάδωσης** ακολουθούν το **B-format**, που όπως είπαμε είναι παραλλαγή του I-format και αφιερώνει, όπως κι εκείνο, 12 bits γιά μιά σταθερή ποσότητα, Imm12, που χρησιμοποιείται γιά να προσδιοριστεί η διεύθυνση προορισμού, ως εξής:

if (condition) then: `PC_new ← PC_br + (2*(signed)Imm12)` *else:* `PC_new ← PC_br + 4`

όπου `PC_br` είναι η διεύθυνση της ίδιας της εντολής διακλάδωσης, `Imm12` είναι η σταθερή ποσότητα των 12 bits μέσα στην εντολή, 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. Ο πολλαπλασιασμός του Imm12 επί 2 γίνεται γιά να εκμεταλλευτούμε το γεγονός ότι η διεύθυνση όλων των εντολών του RISC-V είναι ακέραιο πολλαπλάσιο του 2 (συχνά, που δεν έχουμε και μικρές (16-μπιτες) εντολές, η διευθύνσεις όλων των εντολών είναι και ακέραια πολλαπλάσια του 4, αλλά αυτό δεν είναι πάντα εγγυημένο, κι έτσι ο RISC-V δεν στηρίζεται σε αυτό). Με τον τρόπο αυτό, διπλασιάζουμε το "βεληνεκές" των διακλαδώσεων –με άλλα λόγια, ο αριθμός Imm12 μετράει πλήθος "μικρών" (16-μπιτων) εντολών μπροστά (θετικός) ή πίσω (αρνητικός), αντί να μετρά πλήθος Bytes μπροστά ή πίσω.

Η συνθήκη διακλάδωσης αφορά πάντοτε τη σύγκριση των δύο καταχωρητών πηγής, **rs1** και **rs2**, που περιέχει το B-format. Οι εντολές διακλάδωσης που υπάρχουν στον βασικό RISC-V, το συντακτικό τους σε Assembly, και οι συγκρίσεις που κάνουν είναι οι εξής:

- **`beq rs1, rs2, label`** # διακλάδωση εάν: `rs1 == rs2`
- **`bne rs1, rs2, label`** # διακλάδωση εάν: `rs1 ≠ rs2`
- **`blt rs1, rs2, label`** # διακλάδωση εάν: `rs1 < rs2` –θεωρώντας τους rs1, rs2 σαν προσημασμένους (signed)
- **`bge rs1, rs2, label`** # διακλάδωση εάν: `rs1 ≥ rs2` –θεωρώντας τους rs1, rs2 σαν προσημασμένους (signed)
- **`bltu rs1, rs2, label`** # διακλάδωση εάν: `rs1 < rs2` –θεωρώντας τους rs1, rs2 σαν απρόσημους (unsigned)
- **`bgeu rs1, rs2, label`** # διακλάδωση εάν: `rs1 ≥ rs2` –θεωρώντας τους rs1, rs2 σαν απρόσημους (unsigned)

Οι συντομογραφίες των ονομάτων προκύπτουν από τα: eq = equal; ne = not equal; lt = less than; ge = greater of equal. Οι συγκρίσεις ισότητας και ανισότητας (≠) είναι προφανώς οι ίδιες γιά αριθμούς signed και unsigned, αφού η ισότητα απαιτεί όλα τα bits να είναι ίδια, ένα προς ένα, ανεξαρτήτως του τι παριστάνει ο κάθε συνδυασμός από bits, άρα περιττεύουν εντολές "bequ" ή "bneu" αφού αυτές θα ήταν εντελώς ισοδύναμες με τις beq και bne που υπάρχουν.

Υπάρχει ένα περίφημο κόλπο γιά να ελέγξει κανείς την διπλή ανίσωση **`0 ≤ i < N`**, που είναι ο έλεγχος ορίων γιά το index ενός πίνακα μεγέθους N, χρησιμοποιώντας μία μόνο εντολή διακλάδωσης. Έστω ότι το array index i είναι προσημασμένος (signed) ακέραιος και βρίσκεται στον καταχωρητή x20, και το μέγεθος N ειναι θετικός ακέραιος και βρίσκεται στον x11. Τότε η εντολή: **`bgeu x20, x11, indexOutOfBounds`** πραγματοποιεί και τις δύο συγκρίσεις ταυτοχρόνως, διότι ερμηνεύει και συγκρίνει τον (προσημασμένο) ακέραιο i (x20) σαν να ήταν unsigned: Όταν το κάνουμε αυτό, σε αναπαράσταση συμπληρώματος ως προς 2, τότε, ως γνωστό, όλοι οι αρνητικοί αριθμοί μοιάζουν σαν πολύ μεγάλοι θετικοί αριθμοί – μεγαλύτεροι από τον μεγαλύτερο προσημασμένο θετικό· έτσι, εάν το index i είναι <0, τότε στη *απρόσημη (unsigned)* σύγκριση το i θα μοιάζει σαν πολύ μεγάλος θετικός αριθμός, σαφώς μεγαλύτερος από το N, άρα θα εμφανίζεται να παραβιάζει τον περιορισμό μας ξανά από την "επάνω" πλευρά, όπως και όταν i≥N.

Άσκηση 5.2: Απουσία Διακλαδώσεων *ble, bgt, bequ, bneu*, και Καταχωρητή-Σταθεράς

(α) Ο RISC-V *δεν* έχει εντολή ble (branch if less or equal), γιά περιπτώσεις όπως π.χ. if (i≤j), ούτε εντολή bgt (branch if greater than), γιά περιπτώσεις όπως π.χ. if (i>j). Γιατί; Θεωρήστε π.χ. ότι η μεταβλητή i βρίσκεται στον καταχωρητή x22 και η μεταβλητή j στον x23, και γράψτε τις παραπάνω διακλαδώσεις μέσω εντολών που έχει ο RISC-V.

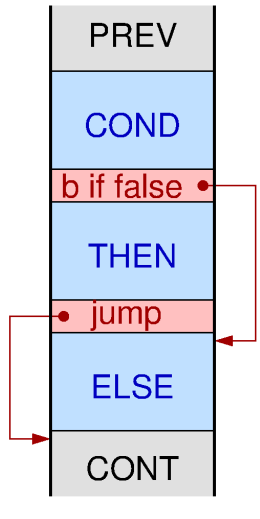
(β) Επίσης, ο RISC-V *δεν* έχει εντολές bequ (branch if equal unsigned), bneu (branch if not equal unsigned) που θα έκαναν τις αντίστοιχες συγκρίσεις αλλά θεωρώντας τους rs1, rs2 σαν απρόσημους ακεραίους (unsigned). Γιατί άραγε; Τι ακριβώς συγκρίνουν οι beq/bne στα bits των rs1, rs2, και τι ακριβώς θα συνέκτριναν οι bequ/bneu εάν υπήρχαν;

(γ) Επίσης, οι εντολές διακλάδωσης του RISC-V συγκρίνουν πάντα και μόνον δύο καταχωρητές μεταξύ τους –δεν υπάρχουν εντολές διακλάδωσης όπως π.χ. "`beqi rs1, Immediate, Label`" που να συγκρίνουν το περιεχόμενο ενός καταχωρητή με μία σταθερή ποσότητα. Γιατί άραγε;

(δ) Δεδομένων λοιπόν όλων των παραπάνω που δεν υπάρχουν, πώς θα μεταφράζατε σε Assembly του RISC-V τον ψευδοκώδικα "branch if i≤13" (ενδεχομένως με τη χρήση κάποιου ενδιάμεσου προσωρινού καταχωρητή);
Δώστε τις απαντήσεις σας, με τις εξηγήσεις τους, γραπτώς.

5.3 Μετάφραση των If-Then-Else

```
PREV;
if (COND) {
    THEN;
} else {
    ELSE;
}
CONT;
```



Στο σχήμα δίπλα φαίνεται η γενική δομή της μετάφρασης σε Assembly ενός if-then-else statement, με τη χρήση μιάς εντολής διακλάδωσης (branch if... - "b if") και μιάς εντολής άλματος (χωρίς συνθήκη - "jump"). Το μπλοκ "PREV" παριστά τον κώδικα που υπάρχει στο πρόγραμμα πριν το if-then-else, και το μπλοκ "CONT" (continue) τον κώδικα μετά το if-then-else. Το μπλοκ "COND" (condition) παριστά τον κώδικα (έκφραση - expression) συνθήκης που υπάρχει μέσα στην παρένθεση του "if", που μπορεί να είναι από πολύ μικρή εώς πολύ μεγάλη· όταν αυτή η συνθήκη είναι πολύ απλή (π.χ. "if (i==j)") τότε η μετάφρασή της μπορεί να γίνεται με μόνο την εντολή διακλάδωσης, ενώ όταν η συνθήκη είναι πιό πολύπλοκη τότε απαιτείται κώδικας "προετοιμασίας" πριν φτάσουμε στη διακλάδωση –τον οποίο κώδικα προετοιμασίας τον παριστούμε σαν "COND" στο σχήμα. Τα μπλοκ κώδικα THEN και ELSE συνήθως βολεύει να τα τοποθετήσει ο compiler στη μνήμη εντολών με την ίδια σειρά με την οποία τα διαβάζει από τον πηγαίο κώδικα, δηλαδή πρώτα το THEN και μετά το ELSE.

Η κάτω πλευρά του σχήματος παριστά τη μνήμη εντολών, με τις διευθύνσεις να αυξάνουν από πάνω προς τα κάτω. Αμέσως μετά την εκτέλεση των εντολών PREV, ο επεξεργαστής *πάντα* αρχίζει να υπολογίζει τη συνθήκη COND, της οποίας οι εντολές είναι τοποθετημένες "κολλητά" μετά τις εντολές PREV. Αμέσως μετά από αυτό τον υπολογισμό της συνθήκης, η εκτέλεση πρέπει να συνεχίσει είτε στο μπλοκ THEN είτε στο μπλοκ ELSE· αυτό επιτυγχάνεται με μιάν εντολή διακλάδωσης, η οποία πρέπει να οδηγήσει τον επεξεργαστή στο μπλοκ ELSE (που είναι σε απόσταση) εάν η συνθήκη είναι ψευδής, αλλιώς θα συνεχίσει στο μπλοκ THEN (που είναι τοποθετημένο "κολλητά

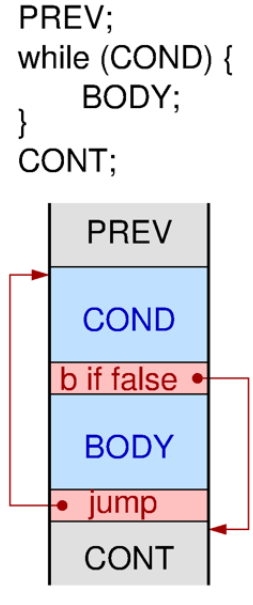
από κάτω"). Εάν εκτελεστεί το μπλοκ THEN, τότε μετά το τέλος του πρέπει να *παρακάμψει* ο επεξεργαστής το μπλοκ ELSE, άρα χρειαζόμαστε μιάν εντολή άλματος (χωρίς συνθήκη) που να μας οδηγήσει κατευθείαν στο μπλοκ CONT, δηλαδή αμέσως μετά την έξοδο από το if-then-else. Αντίθετα, εάν εκτελεστεί το μπλοκ ELSE, τότε μετά το τέλος του βγαίνουμε από το if-then-else και συνεχίζουμε κανονικά στο μπλοκ CONT, που είναι τοποθετημένο "κολλητά" από κάτω, χωρίς να απαιτείται προς τούτο καμιά διακλάδωση ή άλμα. Στο βιβλίο υπάρχει το εξής παράδειγμα (με μία προσθήκη, εδώ, κώδικα "CONT" (μηδενισμός της μεταβλητής h) γιά περισσότερη σαφήνεια):

```
if ( i==j ) { f = g+h; } else { f = g-h; } h=0;
```

όπου οι μεταβλητές f, g, h, i, j θεωρείται ότι βρίσκονται στους καταχωρητές x19, x20, x21, x22, και x23, αντίστοιχα. Τότε αυτό μεταφράζεται σε Assembly του RISC-V ως εξής (τα ονόματα των labels είναι εμπνευσμένα από την αρίθμηση της παραγράφου, §5.3) (η (ψεύδο-)εντολή "j" (jump) υλοποιεί το άλμα που χρειαζόμαστε, και θα μιλήσουμε γιά αυτήν παρακάτω, στην §5.7):

```
            ...{PREV}...            # instructions before entering into "if"
            bne  x22, x23, else53   # if ( i!=j ) goto ELSE
            add  x19, x20, x21      # f = g+h; {THEN}
            j    cont53             # skip over "ELSE"
else53:     sub  x19, x20, x21      # f = g-h; {ELSE}
cont53:     add  x21, x0, x0        # h=0;  {CONT}
```

5.4 Μετάφραση των Βρόχων While



Κατ' αναλογία προς το προηγούμενο, στο εδώ σχήμα φαίνεται η μετάφραση σε Assembly ενός βρόχου while. Και πάλι, τα μπλοκ PREV και CONT είναι ό,τι υπάρχει πριν και μετά το βρόχο. Το μπλοκ COND είναι η συνθήκη γιά να εκτελέσουμε το σώμα, BODY, και γιά την πρώτη και γιά όλες τις (τυχόν) υπόλοιπες φορές. Η τοποθέτηση του κώδικα συνθήκης, COND, πρώτα και ύστερα του σώματος, BODY, είναι αυτή που βολεύει τον compiler, δηλαδή η ίδια με τη σειρά που αυτός τα διαβάζει από τον πηγαίο κώδικα –δείτε όμως και την αμέσως επόμενη άσκηση 5.5. Η τοποθέτηση των εντολών στη μνήμη, όπως στο κάτω μέρος του σχήματος, ακολουθεί τη λογική του βρόχου: αμέσως μετά την εκτέλεση των εντολών PREV, ο επεξεργαστής συναντά "κολλητά" τον υπολογισμό της συνθήκης COND, και αρχίζει *πάντα* με αυτόν. Εάν η συνθήκη είναι ψευδής –είτε την πρώτη είτε οιαδήποτε επόμενη φορά– τότε βγαίνουμε από το while μέσω της εντολής διακλάδωσης, και συνεχίζουμε με το CONT· αλλιώς, δηλαδή εάν η διακλάδωση αποτύχει, άρα όταν η συνθήκη είναι αληθής, συνεχίζουμε ακριβώς από κάτω, δηλαδή μπαίνουμε στο σώμα του βρόχου και εκτελούμε τον κώδικα BODY. Μετά την εκτέλεση του σώματος BODY, πρέπει *πάντα* να ρωτήσουμε "να το ξανακάνω;", δηλαδή πρέπει πάντα να επανυπολογίσουμε τη συνθήκη COND γιά να δούμε εάν θα επαναλάβουμε το βρόχο ή θα βγούμε από αυτό. Γι' αυτό, στο τέλος του BODY τοποθετούμε μιάν εντολή άλματος που μας γυρίζει πάντα πίσω στο COND· προφανώς, η (μόνη) έξοδος από αυτό το βρόχο είναι όποτε πετύχει η διακλάδωση, δηλαδή όταν η συνθήκη COND γίνει ψευδής.

Στο βιβλίο υπάρχει το παρακάτω παράδειγμα, ελαφρώς παραλλαγμένο εδώ. Ψάχνουμε σε έναν πίνακα ακεραίων double: `long long int table[SIZE]` μέχρι να

βρούμε εκεί μίαν επιθυμητή τιμή `key`, η οποία υποτίθεται ότι ξέρουμε ότι υπάρχει στον πίνακα. Ας υποθέσουμε ότι η μεταβλητή i βρίσκεται στον καταχωρητή "t1" (που είναι ο x6 όπως θα πούμε στην §6.1), η τιμή key που αναζητούμε βρίσκεται στον καταχωρητή t2 (δηλαδή τον x7), και η διεύθυνση βάσης του πίνακα (που ως συνήθως *δεν* χωρά στη 12-μπιτη σταθερά Imm12 της εντολής ld) βρίσκεται στον καταχωρητή s0 (δηλ. x8). Αφού τα στοιχεία του πίνακα είναι double words, μεγέθους 8 Bytes καθένα, το στοιχείο `table[i]` θα βρίσκεται στη διεύθυνση: $s0 + 8 \cdot i$.

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

Θα χρησιμοποιήσουμε τον καταχωρητή t0 (δηλ. x5) γιά τα ενδιάμεσα, προσωρινά αποτελέσματά μας. Η εντολή `slli` (shift left logical immediate) προκαλεί αριστερή ολίσθηση των bits του πρώτου τελεστέου πηγής (εδώ του t1) κατά το πλήθος θέσεων (bits) που ορίζει ο δεύτερος (σταθερός - immediate) τελεστέος πηγής (εδώ: 3 θέσεις), γεμίζοντας τις κενούμενες θέσεις με μηδενικά ("logical"), και γράφει το αποτέλεσμα στον καταχωρητή προορισμού (εδώ: t0)· όπως ξέρουμε (§6.1 Ψηφιακής Σχεδίασης), μιά τέτοια αριστερή ολίσθηση ισοδυναμεί με πολλαπλασιασμό επί 2 είς την δύναμη του πλήθους των θέσεων ολίσθησης –άρα εδώ πολλαπλασιασμός επί $2^3 = 8$. Έτσι, ο παραπάνω κώδικας μεταφράζεται σε:

```
        add  t1, x0, x0      # i=0;
loop54: slli t0, t1, 3       # tmp = 8 * i (start COND evaluation)
        add  t0, s0, t0      # tmp = διεύθυνση του table[i]
        ld   t0, 0(t0)       # tmp = table[i]
        beq  t0, t2, cont54  # if table[i] == key goto cont54 (loop exit)
        addi t1, t1, 1       # i = i+1 (loop BODY)
        j    loop54          # repeat the loop
cont54: ....                 # contine with the rest of the program
```

Άσκηση 5.5: Βελτιστοποίηση Βρόχου "While"

Στο σχήμα και στο παράδειγμα που δόθηκε αμέσως παραπάνω, στην §5.4, παρατηρήστε ότι σε κάθε ανακύκλωση του βρόχου while εκτελούνται τόσο μια διακλάδωση υπό συνθήκη (branch), όσο και ένα άλμα χωρίς συνθήκη (jump). Όμως, μόνον οι απλοί μεταφραστές θα έφτιαχναν τέτοιο κώδικα, δεδομένου ότι αυτός μπορεί να βελτιωθεί –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. Μας ενδιαφέρει μιά τεχνική με γενικότερη εφαρμογή: σκεφτήτε την πάνω στο σχήμα της προηγουμένης παραγράφου, 5.4. (Στις διαφάνειες των διαλέξεων δόθηκε και μία άλλη, ορθογώνια βελτιστοποίηση του παραπάνω βρόχου, με σάρωση του βρόχου μέσω pointer αντί μέσω array-index· κάντε αυτή την άσκηση πάνω στον αρχικό κώδικα, αυτόν της §5.4 όχι στον βελτιστοποιημένο με pointers).

Έστω ότι το σώμα (BODY) αυτού του βρόχου επαναλαμβάνεται 10 φορές. Τότε, πόσες εντολές συνολικά εκτελούνταν με τον παλαιό κώδικα, και πόσες συνολικά με τον νέο;

5.6 Σύνθετες Συνθήκες με υπολογισμό Short-Circuit

Ας θεωρήσουμε τώρα μιά λίγο πιό σύνθετη συνθήκη διακλάδωσης, με λογικό ΚΑΙ μέσα της. Θεωρήστε ένα παράδειγμα ανάλογο με το προηγούμενο, όπου ψάχνουμε να βρούμε κάποιο "key", αλλά τώρα το ψάχνουμε σε linked list αντί σε array· επιπλέον, τώρα ενδέχεται και να μην υπάρχει το στοιχείο αυτό στη λίστα, άρα πρέπει να προβλέψουμε και την περίπτωση που η λίστα τελειώνει πριν το βρούμε. Στον παρακάτω κώδικα, έστω ότι η τιμή key που αναζητούμε βρίσκεται και πάλι στον καταχωρητή t2, και ότι ο pointer **p** βρίσκεται στον καταχωρητή t1. Ας κάνουμε αυτή τη φορά το παράδειγμα σε **32-μπιτο** RISC-V, αντί του 64-μπιτου πριν. Έτσι, ο p δείχνει σε structures με δύο στοιχεία καθένα, μεγέθους 4 Bytes καθένα (32-bits): το πρώτο στοιχείο είναι μία τιμή "value" σε offset 0 σε σχέση με την αρχή του structure, και το δεύτερο στοιχείο είναι ένας pointer "next" στο επόμενο structure της λίστας, σε offset 4 σε σχέση με την αρχή του structure.

```
while ( p!=NULL && p->value != key)  { p = p->next; }
```

Εδώ, ως γνωστόν, τα semantics της C μας βοηθάνε να *μην* προσπαθήσει ο επεξεργαστής να διαβάσει το p->value όταν ο pointer p τύχει να είναι NULL: η C προδιαγράφει *short circuit evaluation* του λογικού ΚΑΙ, δηλαδή εάν είναι ψευδές το πρώτο σκέλος, p!=NULL, τότε παρακάμπτεται και *δεν* υπολογίζεται το δεύτερο σκέλος, p->value != key. Γιά να υλοποιηθεί αυτό, απαιτούνται *δύο* διακλαδώσεις υπό συνθήκη σε αυτό το βρόχο:

```
loop56: beq  t1, x0, cont56 # if p==NULL then exit the loop
        lw   t0, 0(t1)      # tmp = p->value
        beq  t0, t2, cont56 # if p->value == key then exit the loop
        lw   t1, 4(t1)      # p = p->next (loop BODY)
        j    loop56         # repeat the loop
cont56: ....                # contine with the rest of the program
```

5.7 Κλήση Διαδικασιών (και Άλματα χωρίς συνθήκη): *jal*

Όπως είπαμε, οι εντολές μεταφοράς ελέγχου (CTI, control transfer instructions) κάνουν ώστε να εκτελεστεί σαν επόμενη εντολή μια έντολη άλλη από την "από κάτω" τους εντολή. Όταν αυτή η μεταφορά ελέγχου γίνεται υπό συνθήκη τότε αυτές λέγονται διακλαδώσεις (branches), ενώ όταν γίνεται χωρίς συνθήκη, δηλαδή πάντα, τότε αυτές λέγονται **άλματα** (jump). Η απλή εντολή άλματος χρειάζεται να περιέχει μέσα της μόνον έναν τελεστέο πέραν από τον opcode –τη διεύθυνση προορισμού– άρα το instruction format θα μπορούσε να αφιερώσει πολλά bits γιά τον προσδιορισμό αυτής της διεύθυνσης. Ο RISC-V χρησιμοποιεί και γιά τα άλματα το PC-Relative addressing mode, γιά τους ίδιους λόγους (πλεονεκτήματα) που το χρησιμοποιεί και γιά τις διακλαδώσεις.

Δεδομένου πάντως ότι τα απλά άλματα (δηλ. όχι τα καλέσματα διαδικασίας) χρησιμοποιούνται μόνον στα if-then-else και βρόχους, δηλαδή *μέσα* στις διαδικασίες, και μαζί με τη χρήση PC-relative mode, δεν υπάρχει σοβαρή ανάγκη γιά πολλά bits στη σχετική απόσταση που έχει να κωδικοποιήσει η σταθερή ποσότητα μέσα στην εντολή άλματος. Έτσι, ο RISC-V καταλήγει να μην χρειάζεται (και *δεν* έχει) ειδική εντολή γιά

το απλό άλμα (jump), αλλά υλοποιεί τέτοια απλά άλματα σαν ειδικές περιπτώσεις (ψευδοεντολή) της γενικότερης εντολής jump-and-link (jal) που προορίζεται γιά την κλήση διαδικασιών:

Κλήση Διαδικασίας (Procedure Call): αυτή μοιάζει με άλμα, αλλά επιπλέον πρέπει να κρατήσουμε πληροφορία γιά το πού να επιστρέψει η καλούμενη διαδικασία, αφού αυτή μπορεί να καλείται από πολλαπλά, διαφορετικά σημεία και πρέπει να ξέρει σε ποιό από αυτά να επιστρέψει. Στις αρχιτεκτονικές CISC, η εντολή καλέσματος, που συνήθως λέγεται "call", αποθηκεύει αυτή τη *διεύθυνση επιστροφής* στη στοίβα (στη μνήμη)· σε μερικές από αυτές, η ίδια αυτή εντολή call μπορούσε επιπλέον και να αποθηκεύει και καταχωρητές στη στοίβα. Στις αρχιτεκτονικές RISC, η εντολή καλέσματος αποθηκεύει τη διεύθυνση επιστροφής σε *καταχωρητή*, τόσον επειδή μόνον οι εντολές store γράφουν στη μνήμη, όσο και επειδή έτσι μπορούμε να γλυτώσουμε αυτή την εγγραφή και ανάγνωση από τη στοίβα γιά όλες τις κλήσεις σε διαδικασίες-φύλλα (§6.2). Προκειμένου να τονιστεί η διαφορά από την παραδοσιακή εντολή call, αυτές οι εντολές καλέσματος με την απλούστερη συμπεριφορά ονομάστηκαν **Jump and Link** (άλμα και συνένωση). Στον RISC-V, η εντολή αυτή γράφεται **`jal`**, ακολουθεί το J-format (§4.3), και κάνει τα εξής:

`jal rd, Imm20` ⇒ `rd ← PC+4; PC ← PC + 2×Imm20`*(signed)*

Με άλλα λόγια, η jump-and-link στον RISC-V, *(α)* γράφει στον καταχωρητή προορισμού τη διεύθυνση της "από κάτω" της εντολής (PC+4), δηλαδή τη διεύθυνση όπου πρέπει να επιστρέψει η καλούμενη διαδικασία, και *(β)* αλλάζει τον PC (εκτελεί άλμα) ούτως ώστε η επόμενη εντολή που θα εκτελεστεί να είναι η πρώτη εντολή της καλούμενης διαδικασίας. Ο καταχωρητής προορισμού, όπου γράφεται η διεύθυνση επιστροφής, είναι συνήθως ο **`ra`** (δηλαδή ο x1), όπως θα πούμε και στην §6.2, οπότε ο Assembler του RISC-V δέχεται και την ψευδοεντολή "jal label" και την μεταφράζει σε "jal x1, label". Μπορεί όμως ο rd να είναι και οιοσδήποτε άλλος καταχωρητής: σε μερικά περιβάλλοντα χρησιμοποιείται και ο t0 (δηλ. ο x5) ως "alternate link register" (π.χ. στο κάλεσμα "millicode routines", όπως γιά σώσιμο και επαναφορά καταχωρητών σε compressed code). Επίσης, ο καταχωρητής προορισμού μπορεί να είναι ο x0 στην πολύ συνηθισμένη ψευδοεντολή **jump** που λέγαμε παραπάνω, αφού ο x0 δεν αλλάζει ποτέ τιμή, και μένει πάντα 0, όσο και αν "γράφουμε" σε αυτόν:

Ψευδοεντολή Jump: **`j label ≡ jal x0, label`** ⇒ `PC ← PC + 2×Imm20`*(signed)*

Διεύθυνση Προορισμού: Η jump-and-link χρησιμοποιεί το J-format (§4.3), άρα προσδιορίζει τη διεύθυνση προορισμού της μέσω της 20-μπιτης σταθερής ποσότητας `Imm20` –και προφανώς το ίδιο ισχύει και γιά την (ψευδοεντολή) του απλού jump που υλοποιείται μέσω αυτής. Ο τρόπος που χρησιμοποιείται αυτό το πεδίο γιά να αλλάξει τον PC είναι πανομοιότυπος με εκείνον γιά τις διακλαδώσεις –απλώς εδώ το Immediate έχει 20 bits αντί 12 εκεί. Όπως και με τις διακλαδώσεις, η σταθερή ποσότητα `Imm20` θεωρείται προσημασμένος αριθμός σε συμπλήρωμα ως προς 2 (δηλαδή sign-extended), και ο πολλαπλασιασμός επί 2 αξιοποιεί το γεγονός ότι ο PC έχει τιμή πάντοτε ακέραιο πολλαπλάσιο του 2, κι έτσι διπλασιάζει το βεληνεκές της κλήσης ή του άλματος. Με άλλα λόγια, η σχετική απόσταση (offset) `Imm20` μετρά σε μονάδες half-words αντί Bytes, άρα διπλασιάζουμε το βεληνεκές σε $\pm 2^{19}$ = 512 K half-words (μικρές, 16-μπιτες εντολές) = ±1 MByte.

5.8 Άλματα και Κλήσεις σε Μεταβλητές Διευθύνσεις: *jalr* – Επιστροφή από Διαδικασία, Switch Statement

Indirect (indexed) Jump – `jr rs1` (ψευδοεντολή): Εκτός από άλματα σε σταθερές, πάντα τις ίδιες διευθύνσεις, οι υπολογιστές χρειάζονται και άλματα σε **μεταβλητούς** (runtime variable) προορισμούς, γιά μία σειρά από σκοπούς, όπως αναλύουμε εδώ. Στον RISC-V, τον ρόλο αυτό τον παίζει η ψευδοεντολή **`jr` (jump-register)**· όπως και με την απλή jump, η jump-register συντίθεται σαν ειδική περίπτωση της γενικότερης jump-and-link-register. Η `jr rs1` κάνει την εξής απλή αλλά εξαιρετικά "ισχυρή" λειτουργία:

`PC_new ← rs1`

Με άλλα λόγια, η jump-register κάνει ώστε η επόμενη εντολή να είναι η εντολή στην *οιαδήποτε αυθαίρετη* διεύθυνση μνήμης είχαμε υπολογίσει προηγουμένως και είχαμε τοποθετήσει στον καταχωρητή rs1. 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:

- Επιστροφή από διαδικασία, αφού επιστρέφουμε στον καλούντα ο οποίος (μπορεί να) είναι διαφορετικός την κάθε φορά που φτάνουμε στο τέλος της διαδικασίας και πρέπει να επιστρέψουμε.
- Μεταφορά του ελέγχου οσοδήποτε μακριά, αφού μπορούμε με προηγούμενες εντολές να φορτώσουμε την οποιαδήποτε διεύθυνση θέλουμε στον rs1.
- Γιά μετάφραση του **`Switch`** Statement, όπως περιγράφουμε επιγραμματικά εδώ:

`Switch` Statement: όπως περιγράφεται στο τέλος της §2.7 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, γιά 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, γιά 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, γιά 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 γιά 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 γιά την περίπτωση της τιμής που τώρα είχε η μεταβλητή.

Η jump-register υλοποιείται σαν ειδική περίπτωση (ψευδοεντολή) της "άλλης" εντολής καλέσματος, της **Jump-and-Link-Register (`jalr`)**, δηλαδή της έμμεσης (indirect / register-indexed) jump-and-link. Αυτή χρησιμοποιείται γιά κάλεσμα διαδικασίας σε αυθαίρετες διευθύνσεις, είτε μακρύτερα απ' όσο μπορεί να πάει η jal με το 20-μπιτο offset της, είτε γιά κλήση σε διαδικασίες που αλλάζουν σε runtime, όπως π.χ. κάνουν οι αντικειμενοστραφείς γλώσσες προγραμματισμού, όπου η ακριβής διαδικασία που θα κληθεί εξαρτάται από τον εκάστοτε τύπο των ορισμάτων με τα οποία αυτή καλείται. Η jalr ακολουθεί το I-format (§4.3), και η διευθυνσιοδότησή της είναι η ίδια όπως των εντολών load (και store), δηλαδή το Offset είναι 12-μπιτο (πάντοτε signed) και *δεν*

πολλαπλασιάζεται επί 2, παρ' ότι εδώ μιλάμε γιά τον PC:

`jalr rd, Offset(rs1)` ⇒ $\texttt{rd}_{\texttt{new}} \leftarrow \texttt{PC+4; PC} \leftarrow \texttt{rs1}_{\texttt{old}} + \texttt{Offset}$*(signed)*

Όπως και η απλή jump-and-link, έτσι και αυτή γράφει τη διεύθυνση της επόμενής της εντολής, PC+4, στον καταχωρητή προορισμού –που στην περίπτωση καλέσματος θα είναι συνήθως ο ra (= x1). Όμως η διαδικασία που καλείται (όταν προκειται πραγματικά γιά κάλεσμα) *δεν* είναι πάντα η ίδια, στατικά καθορισμένη όπως ήταν γιά την jal όπου η διεύθυνση άλματος προέκυπτε σαν μιά σταθερή απόσταση σε σχέση με την τρέχουσα εντολή (PC). Εδώ, αντιθέτως, η διεύθυνση προορισμού προκύπτει μέσω του (αυθαίρετου) καταχωρητή rs1, ο οποίος μπορεί να περιέχει έναν οιοδήποτε αυθαίρετο pointer, που μπορεί κάλιστα να μεταβάλεται από κάλεσμα σε κάλεσμα: πρόκειται γιά **κάλεσμα μέσω pointer**. (Επιτρέπεται, αν και δεν ξέρω εάν χρησιμεύει σε κάτι, ο καταχωρητής rd να είναι ο ίδιος με τον rs1, στην οποία περίπτωση η παλαιά τιμή του καταχωρητή χρησιμοποιείται γιά τον υπολογισμό της νέας τιμής του PC, ενώ αντίστροφα η παλαιά τιμή του PC, συν 4, γράφεται σαν νεά τιμή του καταχωρητή).

Το Offset (12 bits) μπορεί πολλές φορές να είναι άχρηστο, οπότε το βάζουμε 0, αλλά μερικές φορές μπορεί και να χρησιμοποιείται, π.χ., εάν η αμέσως προηγούμενη εντολή φορτώνει τα 20 bits της σταθεράς της Imm20 στο αριστερό μέρος του rs1, τότε το 12-μπιτο Offset προστιθέμενο σε αυτά δημιουργεί έναν αυθαίρετο 32-μπιτο αριθμό από τις δύο σταθερές, κι έτσι μας δίνει έναν τρόπο άλματος στην οιαδήποτε αυθαίρετη διεύθυνση μνήμης. Το 12-μπιτο Offset *δεν* διπλασιάζεται εδώ, παρ' ότι χρησιμοποιείται σε πράξη που το αποτέλεσμά της προορίζεται γιά τον PC, επειδή η εντολή αυτή έχει το ίδιο format (και χρησιμοποιεί τα ίδια κυκλώματα) με τις εντολές load (και παρόμοια με τις store), οι οποίες δεν διπλασιάζουν το Offset τους (και δεν χρειάζεται διπλασιασμός όταν τα 12 bits του Offset προστίθενται στα 20 bits αριστερά από ένα Imm20 γιά να δημιουργήσουν μιά αυθαίρετη 32-μπιτη σταθερά (διεύθυνση)). Παρ' όλα αυτά, στο τελικό αποτέλεσμα της πρόσθεσης rs1+Offset, το hardware του RISC-V μηδενίζει το (ένα) δεξιότερο (LS) bit του αποτελέσματος πριν το γράψει στον PC (ή αλλιώς: ο PC έχει στην πραγματικότητα ένα λιγότερο flip-flop δεξιά, και υποτίθεται ότι συμπληρώνεται πάντα με ένα μηδενικό δεξιά όταν είναι να πάει στη μνήμη (η οπία μνήμη, όταν είναι 16-μπιτη ή φαρδύτερη, αγνοεί ούτως ή άλλως το δεξιότερο (τουλάχιστο) bit)).

Στον RISC-V, όπως η (ψεύδο)εντολή jump είναι στην πραγματικότητα ειδική περίπτωση της εντολής jump-and-link, έτσι και η (ψεύδο)εντολή **Jump-Register** που είπαμε παραπάνω συντίθεται σαν ειδική περίπτωση της jump-and-link-register, ως εξής (και η επιστροφή από διαδικασία είναι συνήθως η ακόμα πιό ειδική περίπτωση "jr ra", δηλαδή "jalr x0, 0(x1)"):

Ψευδοεντολή Jump-Register: **`jr rs1 ≡ jalr x0, 0(rs1)`** ⇒ `PC ← rs1`

5.9 Dynamic Linking, Relocatable code, Εντολή *auipc*

Ο RISC-V προσφέρει υποστήριξη γιά **Relocatable Code**, δηλαδή κώδικα Assermbly/Object τον οποίο μπορεί εύκολα ο Linker να αλλάξει (ή ακόμα και να μην χρειάζεται καμία αλλαγή) προκειμένου να τον συνενώνει (link) με άλλον κώδικα, τοποθετώντας (φορτώνοντάς) τον σε *αυθαίρετη* θέση στη μνήμη (δηλαδή να τον κάνει relocate), όπως π.χ. χρειάζεται γιά τη δυναμική συνένωση διαδικασιών βιβλιοθήκης (dynamically linked libraries - §2.12, pp. 130-132 Αγγλικού βιβλίου). Το βασικό addressing

mode (τρόπος δημιουργίας διευθύνσεων μνήμης) γιά σκοπούς εύκολου relocation είναι το PC-relative: όταν μιά διεύθυνση μνήμης εκφράζεται σαν *σχετική απόσταση* από την τιμή του PC της εντολής που την γεννά, τότε αυτή η απόσταση δεν αλλάζει κατά το relocation. Γιά τις διακλαδώσεις υπό συνθήκη, η διεύθυνση προορισμού δίδεται πάντα σαν PC-relative όπως έχουμε δεί, με βεληνεκές ±4 KBytes. Το ίδιο ισχύει και γιά το κάλεσμα διαδικασίας (jal), καθώς και γιά το απλό άλμα (jump) του συντίθεται ως ψευδοεντολή μέσω αυτού, με βεληνεκές ±1 MByte. Ο RISC-V προσφέρει *μία ακόμα εντολή*, την **`auipc`** (*add upper immediate to PC*), προκειμένου (α) να επεκτείνει το κάλεσμα/άλμα σε οιαδήποτε αυθαίρετη 32-μπιτη απόσταση, και (β) να προσφέρει επίσης PC-relative addressing, και μάλιστα με αυθαίρετη 32-μπιτη απόσταση, γιά εντολές *load και store δεδομένων*, ως εξής.

Η εντολή **`auipc rd, Imm20`** (*add upper immediate to PC*), με format "**U**" όπως και η `lui`, πρώτα κατασκευάζει την ίδια σταθερή ποσότητα όπως και η `lui`, και στη συνέχεια προσθέτει αυτή τη σταθερή ποσότητα στην τιμή του PC της και γράφει το αποτέλεσμα της πρόσθεσης στον καταχωρητή rd· με άλλα λόγια, είναι σαν να κάνει: `lui rd, Imm20` και αμέσως μετά να κάνει: `rd ← PC+rd`. Ή αλλιώς μπορούμε να πούμε ότι η `auipc rd, Imm20` κάνει: $rd \leftarrow PC + (sign\text{-}extended)Imm20 \times 2^{12}$. Όπως και η `lui`, η auipc μπορεί να χρησιμοποιηθεί σαν η πρώτη εντολή σε ένα ζευγάρι εντολών με δεύτερη εντολή είτε ένα κάλεσμα/άλμα είτε μία load/store, γιά να προκαλέσει κάλεσμα/άλμα ή προσπέλαση δεδομένων PC-relative στη διεύθυνση `PC+Const32`, όπου `Const32` είναι μιά αυθαίρετη 32-μπιτη σταθερά αποτελούμενη από δύο κομάτια, `HI` τα 20 αριστερά bits, και `LO` τα 12 δεξιά bits, ως εξής. Η πρώτη εντολή του ζεύγους κατασκευάζει την τιμή $PC+HI\times2^{12}$ και την τοποθετεί σ' έναν προσωρινό καταχωρητή, π.χ. τον t0: **`auipc t0, HI`** (ή `auipc t0, HI+1` εάν το κομάτι `LO` αρχίζει με 1, όπως σχολιάζαμε στην §4.4 και γιά την `lui`). Η δεύτερη εντολή του ζεύγους είναι είτε jalr γιά κάλεσμα/άλμα είτε load/store, όπου και στις δύο περιπτώσεις χρησιμοποιείται σαν pointer ο προηγούμενος καταχωρητής t0, και σαν Offset τα δεξιά 12 bits της Const32, το LO, άρα τελικά η διεύθυνση είναι η $PC+HI\times2^{12}+LO = PC+Const32$:

- Κάλεσμα PC-relative σε αυθαίρετη απόστ.:
  `auipc t0, HI` (ή `HI+1`); **`jalr ra, LO(t0)`**
- Άλμα PC-relative σε αυθαίρετη απόσταση:
  `auipc t0, HI` (ή `HI+1`); **`jalr x0, LO(t0)`**
- Load PC-relative σε αυθαίρετη απόσταση:
  `auipc t0, HI` (ή `HI+1`); **`ld rd, LO(t0)`**
- Store PC-relative σε αυθαίρετη απόσταση:
  `auipc t0, HI` (ή `HI+1`); **`sd rs2, LO(t0)`**

5.10 Πράξεις Σύγκρισης με αποτέλεσμα Boolean

Εκτός από τα if-then-else, στις γλώσσες προγραμματισμού, αριθμητικές συγκρίσεις εμφανίζονται και σε εκχωρήσεις (assignments) του τύπου: `myBool = (a<b)`, όπου οι μεν μεταβλητές a και b είναι αριθμοί, το δε αποτέλεσμα της σύγκρισης που εκχωρείται είναι τύπου *Boolean*. Στην C, ο τύπος Boolean κρατιέται μεν μέσα σε μεταβλητές τύπου ακεραίου, αλλά οι μόνες νόμιμες τιμές του είναι το 0 γιά "ψευδές" και το 1 γιά "αληθές" (αυτό έχει πολλά μηδενικά αριστερά, και έναν μόνον άσο δεξιά). Εκχωρήσεις

όπως η παραπάνω θα μπορούσαν βέβαια να υλοποιηθούν μέσω `if (a<b) {myBool=1;} else {myBool=0;}`, όμως μιά τέτοια υλοποίηση αφ' ενός θα απαιτούσε κάμποσες εντολές, και αφ' ετέρου θα προκαλούσε σημαντικές καθυστερήσεις κατά την εκτέλεση του προγράμματος επειδή οι εντολές διακλάδωσης είναι αργές ή και δύσκολες στους επεξεργαστές με ομοχειρία (pipelining) όπως θα δούμε αργότερα στο μάθημα. Γιά να υποστηρίξει εκχωρήσεις όπως η παραπάνω, ο RISC-V έχει την εντολή **set if less than (slt)**, μαζί με άλλες τρείς παραλλαγές της, που υλοποιεί ουσιαστικά την παραπάνω εκχώρηση. Πρόκειται γιά εντολές ανάλογες προς την πρόσθεση ή την αφαίρεση, μόνο που το αποτέλεσμά τους είναι τύπου Boolean αντί τύπου ακέραιος:

- **`slt  rd, rs1, rs2`** # rd ← (rs1<rs2) – signed comparison, Boolean result
- **`sltu  rd, rs1, rs2`** # rd ← (rs1<rs2) – unsigned comparison, Boolean result
- **`slti  rd, rs1, Imm12`** # rd ← (rs1<Imm12) – signed Immediate, signed comparison
- **`sltiu rd, rs1, Imm12`** # signed Immediate (sign-extended), but unsigned comparison

Οι εντολές αυτές γράφουν στον καταχωρητή rd είτε τον αριθμό 1 (πολλά μηδενικά bits αριστερά και ένας άσος δεξιά) εάν το αποτέλεσμα της σύγκρισης είναι αληθές, είτε τον αριθμό 0 (όλο μηδενικά bits) εάν το αποτέλεσμα είναι ψευδές. Οι δύο πρώτες παραπάνω εντολές συγκρίνουν δύο καταχωρητές μεταξύ τους, και ακολουθούν το R-format, οι δε δύο επόμενες συγκρίνουν καταχωρητή (αριστερά από το <) προς σταθερή ποσότητα (δεξιά από το <), και ακολουθούν το I-format. Η μεν πρώτη και η τρίτη θεωρούν τους τελεστέους σαν προσημασμένους (signed) ακεραίους όταν κάνουν τη σύγκριση, οι δε δεύτερη και τέταρτη κάνουν απρόσημη (unsigned) σύγκριση. Στον RISC-V οι σταθερές ποσότητες "Immediate" θεωρούνται πάντα προσημασμένες, άρα το Imm12 επεκτείνεται από 12 σε 32 ή 64 bits πάντα σαν προσημασμένη (signed) σταθερά, ακόμα και όταν η σύγκριση, στη συνέχεια, είναι unsigned, δηλαδή θεωρεί τον 32-μπιτο ή 64-μπιτο δεύτερο τελεστέο σαν unsigned. Όσον αφορά εκφράσεις όπως `(a<b)&&(c<d)`, είτε σε εκχωρήσεις Booolean είτε μέσα σε if, εάν μεν η γλώσσα ή οι περιστάσεις δεν απαιτούν short-circuit evaluation τότε αυτές μπορούν να υλοποιηθούν με δύο εντολές set-if-less-than ακολουθούμενες από μία εντολή (bit-wise) AND, που δίνει συνήθως και την γρηγορότερη εκτέλεση· εάν, από την άλλη, απαιτείται short-circuit evaluation, τότε θα χρειαστεί και η χρήση διακλαδώσεων.

<u>Άσκηση 5.11:</u> Σύνθεση άλλων Συγκρίσεων

Ο RISC-V έχει μόνον τις παραπάνω τέσσερεις εντολές συγκρίσεων, που όλες τους κάνουν σύγκριση "less than", και δεν έχει συγκρίσεις ≤ ή > ή ≥ επειδή οι τελευταίες μπορούν να συντεθούν από τη σύγκριση less than –ας δούμε πώς. Έστω ότι η μεταβλητή **`i`** βρίσκεται στον καταχωρητή **x22** η μεταβλητή **`j`** στον καταχωρητή **x23**, η μεταβλητή **`B`** στον καταχωρητή **x26**, και ότι **`CONST`** σημαίνει μια αυθαίρετη προσημασμένη σταθερή ποσότητα που χωρά σε 12 bits. Συνθέστε τις εξής εκχωρήσεις (i)-(viii), όπου όλες θεωρούνται signed, χρησιμοποιώντας την εντολή **`slt`** ή την **`slti`**, και, όπου χρειάζεται, και την εντολή **`xori`** (ανάλογη της addi, αλλά κάνει exclusive-OR αντί γιά πρόσθεση) με σταθερή immediate τον αριθμό 1, η οποία επομένως έχει σαν συνέπεια να αντιστρέφει το δεξιό bit του άλλου τελεστέου της, δηλαδή ισοδυναμεί με Boolean NOT:

i. `B = ( i <  j )` (μικρότερο)
ii. `B = ( i >= j )` (μεγαλύτερο ή ίσο)
iii. `B = ( i >  j )` (μεγαλύτερο)
iv. `B = ( i <= j )` (μικρότερο ή ίσο)

v. `B = ( i < CONST )` (μικρότερο)
vi. `B = ( i >= CONST )` (μεγαλύτερο ή ίσο)
vii. `B = ( i > CONST )` (μεγαλύτερο)
viii. `B = ( i <= CONST )` (μικρότερο ή ίσο)

Υπόδειξη: παίξτε πρώτα με τη σειρά που βάζετε τους δύο τελεστέους πηγής όταν αυτοί είναι καταχωρητές, καθώς και με ενδεχόμενη άρνηση του αποτελέσματος. Όταν ο δεύτερος τελεστέος είναι σταθερή ποσότητα, τότε δεν μπορείτε μεν να αντιστρέψετε τη σειρά που τους δίνετε στην εντολή slti, μπορείτε όμως να παίξετε με τις σταθερές `CONST, CONST+1, CONST-1`. Θα εκτιμήστε το πόσα πολλά μπορεί να κάνει το software, εκμεταλλευόμενο λίγους, προσεκτικά επιλεγμένους δομικούς λίθους hardware!

Στη συνέχεια παρατηρήστε ότι η εντολή: **`sltu rd, x0, rs2`** ελέγχει εάν ο rs2 είναι μεγαλύτερος του μηδενός, *αλλά* τον συγκρίνει ως unsigned, δηλαδή σαν πάντα ≥0, άρα ουσιαστικά ελέγχει εάν ο rs2 είναι **διάφορος του μηδενός**. Επίσης η εντολή: **`sltiu rd, rs1, 1`** ελέγχει εάν ο rs1 είναι μικρότερος του 1, *αλλά* τον συγκρίνει ως unsigned, δηλαδή σαν πάντα ≥0, άρα ουσιαστικά ελέγχει εάν ο rs1 είναι **ίσος με μηδέν**.

Με βάση τις τελευταίες δύο παρατηρήσεις, χρησιμοποιήστε μιάν εντολή αφαίρεσης (sub) ή πρόσθεσης του `-CONST`, και στη συνέχεια μιάν από τις δύο αυτές συγκρίσεις ισότητας/ανισότητας προς μηδέν, γιά να συνθέσετε και τις εξής τέσσερεις εκχωρήσεις. Σαν προσωρινό καταχωρητή παρατηρήστε ότι μπορείτε να χρησιμοποιήσετε τον ίδιο τον καταχωρητή προορισμού, και ότι αυτό θα ίσχυε ακόμα και εάν τύχαινε ο καταχωρητής αυτός να ήταν ο ίδιος με έναν από τους καταχωρητές πηγής. Δώστε τις απαντήσεις σας γραπτά.

a. `B = ( i == j )` (ίσο)
b. `B = ( i != j )` (άνισο)
c. `B = ( i == CONST )` (ίσο)
d. `B = ( i != CONST )` (άνισο)

Τρόπος Παράδοσης:
Παραδώστε μαζί την προηγούμενη άσκηση 4 και αυτήν εδώ τη σειρά 5, on-line, σε μορφή **PDF** (μόνον) (μπορεί να είναι κείμενο μηχανογραφημένο ή/και "σκαναρισμένο" χειρόγραφο, αλλά *μόνον* σε μορφή PDF). Παραδώστε μέσω **`turnin ex045@hy225 [directoryName]`** ένα αρχείο ονόματι **`ex045.pdf`** που θα περιέχει τις απαντήσεις σας σε όλες τις ασκήσεις, 4 και 5.

(Σημείωση σε περίπτωση δυσλειτουργίας του `turnin`: εάν αυτό σας λέει "/home/misc/courses/hy225/TURNIN/ex...: No such file or directory" ή κάτι ανάλογο, ενδέχεται ο υπολογιστής όπου τρέχετε να μην μπορεί να δεί τον server που στην πραγματικότητα κρατά αυτό το directory του μαθήματος (ή το δικό σας home directory), λόγω δυσλειτουργίας του δικτύου που ελπίζει κανείς να είναι προσωρινή. Πολλές φορές, εκτελώντας την εντολή λίγο αργότερα, ή από άλλον υπολογιστή μπορεί να λύσει το πρόβλημα –κατάλογος των διαθέσιμων μηχανημάτων υπάρχει στην ιστοσελίδα του Τμήματος, κάτω από την ενότητα: Υπηρεσίες→Ψηφιακές Υπηρεσίες→Απομακρυσμένη πρόσβαση στα Debian μέσω VPN).
