HY-225: Οργάνωση Υπολογιστών
Άνοιξη 2023

Τμ. Επ. Υπολογιστών
© Πανεπιστήμιο Κρήτης

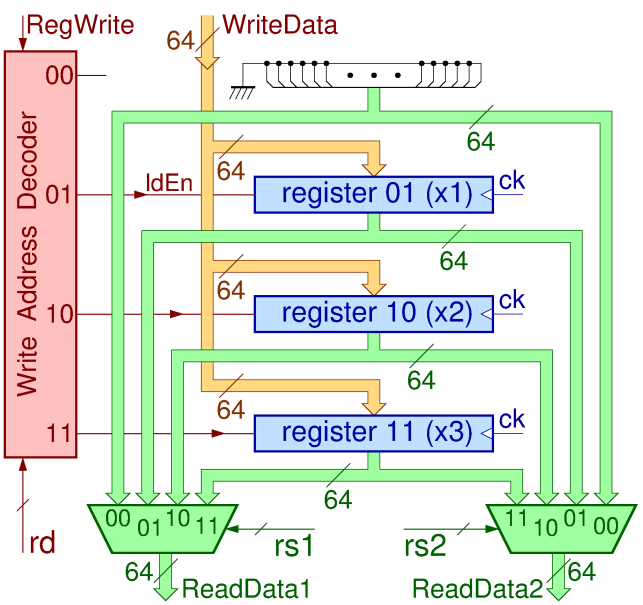
Ασκήσεις 8: Μία Απλή Υλοποίηση του RISC-V σε Έναν Κύκλο Ρολογιού ανά Εντολή

κάντε τις έως Τε. 29 Μαρτίου 2023 (βδ. 8.2)
Υπενθύμιση Διαγωνισμού Προόδου: Σάββατο 18 Μαρτίου 2023, ώρα 11-1

Βιβλίο: (α) Υλοποίηση Επεξεργαστή: Διαβάστε την §4.4 (σελίδες 374-389 στο Ελληνικό βιβλίο, pp. 251-262 στο Αγγλικό)· επιπροσθέτως, οι ενότητες 4.1 - 4.3 (σελ. 356-374, pp. 236-251) περιέχουν εισαγωγικό υλικό, χρήσιμο γιά την αρχική κατανόηση, αλλά υπερκαλύπτονται, όσον αφορά την τελική εκμάθηση, από την §4.4. (β) Immediate fields in instructions: σχετικές είναι οι σελίδες 113-121 του Αγγλικού βιβλίου.

Σε αυτό το δεύτερο μέρος του μαθήματος θα δούμε πώς υλοποιείται σε υλικό (hardware ψηφιακά κυκλώματα) ένας επεξεργαστής RISC-V με ένα αντιπροσωπευτικό υποσύνολο των εντολών του πραγματικού RISC-V. Ξεκινάμε, σε αυτήν εδώ τη σειρά 8 σημειώσεων και ασκήσεων με μιά πολύ απλή υλοποίηση, ανάλογη εκείνης του απλού υπολογιστή του μαθήματος της Ψηφιακής Σχεδίασης, όπου όλες οι εντολές εκτελούνται σε έναν (μεν) μόνον κύκλο ρολογιού, ο οποίος όμως είναι πολύ *μακρύς* (μεγάλη περίοδος ρολογιού = μικρή συχνότητα ρολογιού). Όπως και τότε, θα χρειαστούμε το Μετρητή Προγράμματος (PC - Program Counter), τη Μνήμη Εντολών (Instruction Memory), τη Μνήμη Δεδομένων (Data Memory), μιάν Μονάδα γιά τις Αριθμητικές & Λογικές Πράξεις (ALU - Arithmetic-Logic Unit), και κάμποσους πολυπλέκτες (αντί και του Bus, που σήμερα σπανίως χρησιμοποιείται)· στη θέση του ενός, τότε, Συσσωρευτή (ACC - Accumulator) τώρα θα έχουμε μιά μικρή (και γρήγορη) μνήμη με τους 32 καταχωρητές –ξεκινάμε με αυτήν.

8.1 Πολύπορτο Αρχείο Καταχωρητών



Όπως όλες οι μνήμες (§9.3 Ψηφιακής Σχεδίασης), έτσι και το αρχείο των καταχωρητών (Register File - RF) αποτελείται από μανταλωτές –εδώ ας υποθέσουμε flip-flops– οργανωμένα σε λέξεις (καταχωρητές), με έναν αποκωδικοποιητή γιά τις διευθύνσεις και πολυπλέκτες γιά την ανάγνωση. Όμως, λόγω του μικρού μεγέθους και της επιδιωκόμενης υψηλής ταχύτητας αυτής της μικρής μνήμης –του αρχείου των καταχωρητών– στους περισσότερους σημερινούς επεξεργαστές αυτό υλοποιείται σε μορφή **πολύπορτης μνήμης**, δηλαδή προσφέρει τη δυνατότητα *πολλαπλών ταυτόχρονων* προσπελάσεων, σε πολλαπλές λέξεις, σε αυθαίρετες, ανεξάρτητες διευθύνσεις η κάθε μία. Εμείς θα χρησιμοποιήσουμε ένα **τρίπορτο** αρχείο καταχωρητών, που επιτρέπει δηλαδή τρείς (3) ταυτόχρονες, ανεξάρτητες προσπελάσεις: δύο αναγνώσεις, και μία εγγραφή. Το κύκλωμα ενός τέτοιου Αρχείου Καταχωρητών φαίνεται στο σχήμα δίπλα,

μόνο γιά να χωράει δείχνουμε μόνον τους 4 πρώτους καταχωρητές, αντί των 32 που έχει ο RISC-V. Όπως και το βιβλίο, υποθέτουμε ότι φτιάχνουμε 64-μπιτο RISC-V, άρα ο κάθε καταχωρητής είναι 64-μπιτος. Σε αυτή την υλοποίηση του ενός κύκλου ρολογιού ανά εντολή, χρειάζεται οι καταχωρητές να είναι ακμοπυροδότητοι, αν και στους πραγματικούς επεξεργαστές (με ομοχειρία (pipelining) που θα δούμε μετά) αυτοί αρκεί να είναι απλοί μανταλωτές. Ο καταχωρητής υπ' αριθμόν μηδέν είναι ειδικός, όπως ξέρουμε: δεν χρειάζεται flip-flops –υλοποιείται σαν 64 σύρματα γείωσης, όπως δείχνει το σχήμα στο επάνω μέρος, αφού πάντοτε όταν τον διαβάζουμε δίνει περιεχόμενο μηδέν.

Γιά να διαβάσουμε (επιλέξουμε) έναν από τους 4 ή 32 καταχωρητές, όπως σε κάθε μνήμη, χρειαζόμαστε έναν 64-μπιτο πολυπλέκτη 4-σε-1 ή 32-σε-1, ο οποίος φυσικά χρειάζεται 2 ή 5 εισόδους (2 ή 5 bits) επιλογής γιά να του λένε ποιόν από τους 4 ή 32 καταχωρητές θέλουμε να διαβάζουμε (επιλέγουμε) κάθε στιγμή· αυτά τα 2 (στο σχήμα) ή 5 (στον RISC-V) bits είναι η *διεύθυνση ανάγνωσης*. Εάν τώρα προσθέσουμε και έναν *δεύτερο* ανάλογο πολυπλέκτη (64-μπιτο και αυτόν), και τον ελέγξουμε με 2 ή 5 άλλα bits "διεύθυνσης" (που έχουν εν γένει διαφορετική τιμή από τα πρώτα, χωρίς και να απαγορεύεται μερικές φορές να παίρνουν και την ίδια τιμή), τότε αυτός ο δεύτερος πολυπλέκτης θα μας διαβάζει (επιλέγει) έναν άλλον, δεύτερο καταχωρητή από τους 32 (που είναι εν γένει άλλος από τον πρώτον, χωρίς όμως και να αποκλείεται μερικές φορές να είναι ο ίδιος, εάν έτσι δοθούν οι δύο διευθύνσεις ανάγνωσης). Έτσι φτιάξαμε ένα αρχείο καταχωρητών **Δίπορτης Ανάγνωσης** (2-port read), δηλαδή, ανά πάσα στιγμή, μπορούμε να επιλέγουμε και βλέπουμε, στις δύο (64-μπιτες) εξόδους δεδομένων, τα περιεχόμενα δύο οιωνδήποτε από τους 32 καταχωρητές (διαφορετικών ή ίδιων, μεταξύ τους). Από άποψη καθυστέρησης, οι δύο πολυπλέκτες δουλεύουν εν παραλλήλω –ταυτόχρονα δηλαδή– και άρα μέσα στο χρόνο μιάς και μόνο ανάγνωσης από μονόπορτο RF, εδώ επιτυγχάνουμε *δύο* αναγνώσεις αντί μίας.

Ταυτόχρονα με τα παραπάνω, εάν υπάρχει και ένας τρίτος αποκωδικοποιητής 2-σε-4 ή 5-σε-32 (τρίτο τον λέμε επειδή οι δύο πολυπλέκτες ανάγνωσης κρύβουν μέσα τους και από έναν αποκωδικοποιητή διεύθυνσης, καθένας), και εάν αυτόν τον τρίτο αποκωδικοποιητή τον χρησιμοποιήσουμε γιά να ελέγξουμε τα 4 ή 32 σήματα ενεργοποίησης φόρτωσης (load enable) των 4 ή 32 καταχωρητών, τότε ταυτόχρονα με την ανάγνωση των δύο καταχωρητών, θα μπορούμε να έχουμε και εγγραφή σε έναν τρίτο (διαφορετικό ή και τον ίδιο με τους δύο πρώτους). Φυσικά, η έξοδος 00 ή 00000 του αποκωδικοποιητή *δεν* χρησιμοποιείται, αφού τυχόν εγγραφή στον x0 δεν έχει κανένα αποτέλεσμα. Γιά να λειτουργήσει καθώς πρέπει η εγγραφή, και να μην εγγράφουμε σε κάποιον καταχωρητή σε *κάθε* κύκλο ρολογιού, καταστρέφοντας έτσι τα παλαιά περιεχόμενά του ακόμα κι αν δεν έχουμε τίποτα χρήσιμο να γράψουμε, χρειαζόμαστε κι ένα συνολικό σήμα ελέγχου εγγραφής, RegWrite, που όταν είναι μηδέν αδρανοποιεί την εγγραφή γενικά, σε όλους τους καταχωρητές.

Όπως είπαμε στην §4.3, ο RISC-V έχει σε σταθερή πάντα θέση μέσα στην εντολή τα πεδία `rs1` και `rs2` των καταχωρητών πηγής και ομοίως του καταχωρητή προορισμού `rd` –όταν αυτοί υπάρχουν. Ακριβώς αυτά τα τρία πεδία της εντολής είναι οι είσοδοι "διεύθυνσης" γιά τις τρείς πόρτες του RF: δύο γιά τους πολυπλέκτες ανάγνωσης, και το τρίτο γιά τον αποκωδικοποιητή εγγραφής. Οι εντολές που δεν έχουν ένα ή και τα δύο από τα πεδία `rs1`, `rs2`, απλώς διαβάζουν τον έναν ή και τους δύο "τυχαίους" καταχωρητές που τυχαίνει να υποδεικνύουν εκείνα τα bits όποιων άλλων πεδίων βρίσκονται στις θέσεις εκείνες, και στη συνέχεια αγνοούν αυτή τη μία ή και τις δύο αναγνωσθείσες τιμές. Αυτό είναι ταχύτερο από το να περιμένουμε πρώτα να αποκωδικοποιήσουμε τον opcode γιά να διαπιστώσουμε εάν

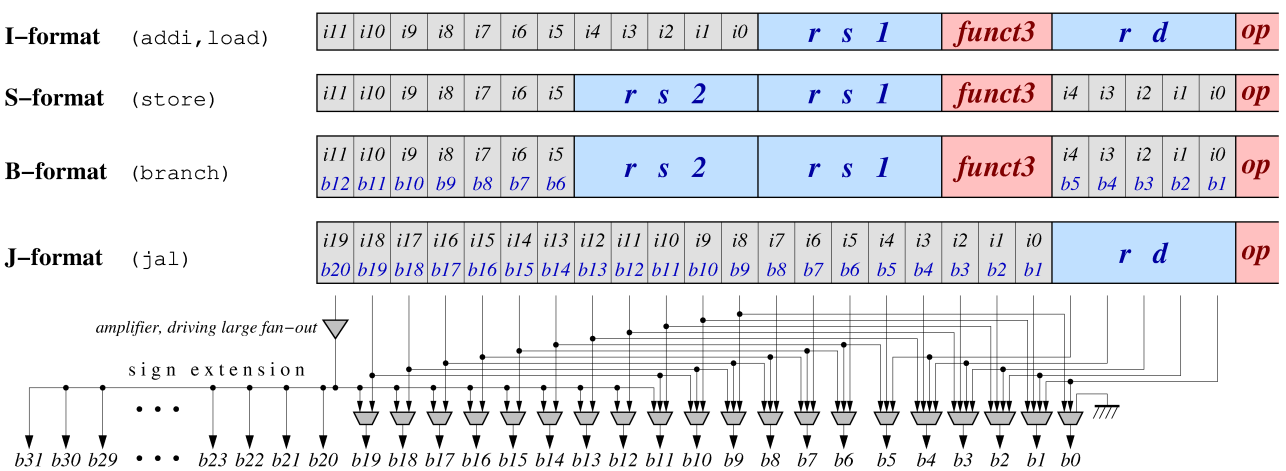
και πόσους καταχωρητές χρειαζόμαστε και στη συνέχεια να διαβάσουμε μόνον όσους χρειαζόμαστε, έστω και ελαφρώς σε βάρος της κατανάλωσης ενέργειας γιά την ανάγνωση όταν δεν τους χρειαζόμαστε. Οι εντολές που δεν γράφουν κανένα αποτέλεσμα σε καταχωρητή θα έχουν σβηστό το συνολικό σήμα ελέγχου εγγραφής, RegWrite, άρα θα αγνοούν τα 5 bits του πεδίου "rd" της εντολής, που στην πραγματικότητα, σε αυτές τις εντολές, είναι "τυχαίο" κομάτι άλλου πεδίου.

8.2 Θέση των bits των Σταθερών Immediate στον RISC-V

[Δείτε την παράγραφο αυτή σαν ένα παράδειγμα της αρχής "Ποτέ μην αναβάλετε γιά run-time ό,τι μπορείτε να κάνετε σε compile-time" μάλλον, παρά σαν κάτι που πρέπει να θυμάστε τις λεπτομέρειές του]. Οι πληροφοριές εδώ προέρχονται από την §2.3 (σελ. 16-17) του εγχειριδίου του RISC-V, riscv-spec-20191213.pdf .

Όπως είχαμε δεί στην §4.3, σταθερές "Immediate" υπάρχουν στα format I, B, S (σταθερές των 12 bits) και J, U (σταθερές των 20 bits) του RISC-V. Οι μορφές B και S είναι παραλλαγές της I που απλώς κόβουν το 12-μπιτο Immediate σε δύο κομάτια προκειμένου να μένουν πάντα σε σταθερή θέση τα πεδία των καταχωρητών. Οι B και S είναι ίδιες μεταξύ τους εκτός από το ότι η B χρησιμοποιείται στις διακλαδώσεις (Branch), όπου το Immediate12 πολλαπλασιάζεται επί 2 πριν προστεθεί στον PC, ενώ η S χρησιμοποιείται στις εγγραφές μνήμης (Store), όπου δεν υπάρχει πολλαπλασιασμός πριν τη χρήση. Αντίστοιχα, η μορφή J χρησιμοποιείται από την εντολή jal, όπου το Immediate20 πολλαπλασιάζεται επί 2 πριν προστεθεί στον PC, ενώ η μορφή U (Upper) χρησιμοποιείται από τις εντολές lui και auipc, όπου το Immediate20 πολλαπλασιάζεται επί 2^{12} πριν χρησιμοποιηθεί.

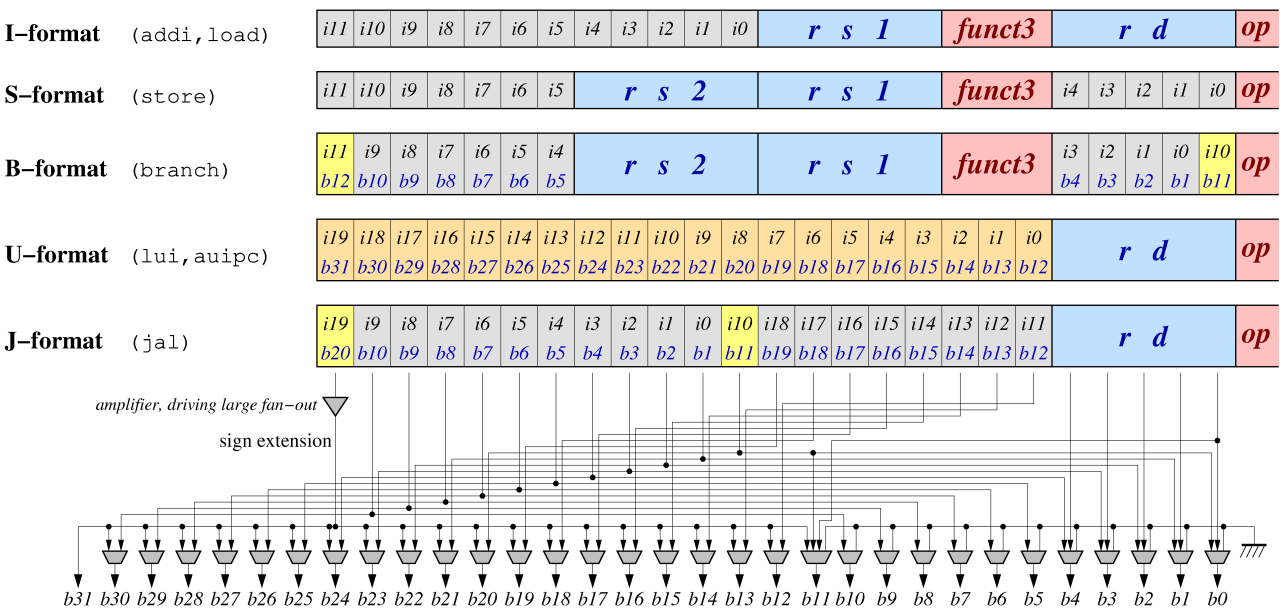
Με δεδομένο το πού μέσα στην εντολή μπαίνουν οι σταθερές ποσότητες Imm12 και Imm20 όπως είδαμε στην §4.3, θα περίμενε κανείς τα bits τους να γράφονται με την "φυσιολογική" τους σειρά μέσα στην εντολή, όπως δείχνει το πρώτο σχήμα εδώ, που όπως θα εξηγήσουμε όμως *δεν* είναι αυτό που κάνει ο RISC-V. Στα σχήματα παριστάνουμε τα 12 bits του Imm12 ως *i11, i10,... i2, i1, i0* όπου *i11* είναι το περισσότερο σημαντικό (most significant - MS) bit, και *i0* είναι το λιγότερο σημαντικό (least significatn - LS) bit. Ομοίως παριστάνουμε τα 20 bits του Imm20 ως *i19, i18,... i2, i1, i0* όπου *i19* είναι το MS, και *i0* είναι το LS.



Τα bits αυτά, μετά από επέκταση προσήμου (§7.7), τροφοδοτούν συνήθως τη δεύτερη είσοδο μιάς αριθμητικής/λογικής μονάδας (ALU) όπου την πρώτη είσοδο τροφοδοτεί ο

καταχωρητής rs1 ή ο PC. Οι βελτιστοποιήσεις αυτής της παραγράφου αφορούν ιδιαίτερα επεξεργαστές μικρού κόστους, όπου ο ίδιος αθροιστής μπορεί να χρησιμοποιείται και γιά τις πράξεις όπως addi, load/store address, και γιά τον υπολογισμό της διεύθυνσης προορισμού των branch/jump. Όταν συμβαίνει αυτό, η δεύτερη είσοδος αυτού του αθροιστή πρέπει να τροφοδοτηθεί με 5 διαφορετικές παραλλαγές της σταθερής ποσότητας, ανάλογα εάν η εντολή είναι τύπου I, S, B, J, ή U. Έστω ότι αυτή η δεύτερη είσοδος του αθροιστή αποτελείται από τα 32 bits: *b31, b30,... b2, b1, b0* όπου *b31* είναι το MS, και *b0* είναι το LS. Όπως δείχνει το σχήμα (μπλέ γράμματα), στις περιπτώσεις B και J όπου απαιτείται πολλαπλασιασμός επί 2, κάθε bit *i* πρέπει να ολισθήσει μία θέση αριστερά και να τροφοδοτήσει το διπλανό του προς τα αριστερά bit *b*. Οι ολισθήσεις αυτές, και οι παραλλαγές θέσης των Immediates θα απαιτούσαν αρκετούς πολυπλέκτες στην είσοδο *b31,... b0* του αθροιστή, όπως φαίνεται στο σχήμα· γιά απλότητα, το πρώτο αυτό σχήμα *δεν* περιλαμβάνει τις εντολές U.

Αντί γιά την παραπάνω απλοϊκή/διαισθητική τοποθέτηση των bits των Immediates μέσα στις εντολές, ο RISC-V παρατηρεί ότι δεν υπάρχει κανένας λόγος ο πολλαπλασιασμός επί 2, ή άλλες μετακινήσεις bits, να γίνονται από το hardware την *κάθε* φορά και στον *κάθε* κύκλο ρολογιού που χρησιμοποιείται ένα Immediate, αλλά αντιθέτως αρκεί πολλά από αυτά να τα κάνει *μία φορά* ο Assembler και να τα βρίσκει έτοιμα το υλικό (hardware)! Το αποτέλεσμα είναι η τοποθέτηση των bits των Immediates μέσα στην εντολή που δείχνει το δεύτερο σχήμα, όπως δηλαδή ορίζει ο RISC-V. Η τοποθέτηση αυτή ναι μεν μοιάζει παράξενη και ανακατωμένη στον ανθρώπινο παρατηρητή, πλήν όμως ελαχιστοποιεί τις εισόδους πολυπλεκτών που απαιτούνται στο hardware, ενώ το κόστος της γιά τον Assembler είναι μηδαμινό.



Παρατηρήστε σε αυτό το δεύτερο σχήμα τα εξής. Συγκρίνοντας τη μορφή (format) B προς την S, βλέπουμε ότι στην B τα bits του Imm12 είναι ήδη ολισθημένα μία θέση αριστερά από τον Assembler, άρα έτοιμα στην θέση που πρέπει γιά την είσοδο *b31,... b0*. Εξαιρέσεις, σημειωμένες με κίτρινο, αποτελούν τα εξής: Το MS bit, *b12 = i11*, που θα τροφοδοτήσει την επέκταση προσήμου, παραμένει πάντα σε σταθερή θέση –την αριστερότερη: το bit αυτό τροφοδοτεί πολλούς ακροατές, δηλαδή έχει μεγάλο "fan-out", άρα απαιτεί ενίσχυση προκειμένου να μην είναι πολύ αργό, άρα θέλουμε να αποφύγουμε έξτρα πολυπλέκτη πριν

τον ενισχυτή που θα αύξανε την καθυστέρηση (βλ. HY-120 §12.8). Το επόμενο bit, *b11 = i10*, που "εκδιώχθηκε" από τη θέση του λόγω αριστερής ολίσθησης, τοποθετείται στη "κενωθείσα" θέση του bit *b0* που είναι πάντα 0, άρα περιττό να γραφτεί στην εντολή.

Στη συνέχεια, παρατηρήστε τις εντολές U: αυτές χρησιμοποιούν το Imm20 στα 20 "αριστερά" (MS) bits του αθροιστή (πρόσθεση γιά την auipc, απλό πέρασμα γιά την lui), άρα, όπως δείχνει το σχήμα, το MS bit *i19* του Imm20 πρέπει να οδηγηθεί στο MS bit *b31* του αθροιστή, το LS bit *i0* στο bit *b12* του αθροιστή, ενώ τα υπόλοιπα 12 LS bits του αθροιστή, *b11,... b0*, πρέπει να είναι 0. Γιά τις εντολές U, τα bits του Imm20 τοποθετούνται στις "φυσιολογικές" θέσεις που θα περίμενε κανείς.

Τώρα κοιτάξτε τη θέση των bits του Imm20 στην εντολή jal, μορφής J: μοιάζουν πολύ ανακατωμένα, αλλά η θέση τους είναι η βέλτιστη όταν την συγκρίνουμε με τις θέσεις των bits στις εντολές U και I. Πρώτα παρατηρούμε ότι επειδή η εντολή jal χρησιμοποιεί το Imm20 πολλαπλασιασμένο επί 2, το MS bit *i19* του Imm20 πρέπει να οδηγηθεί στο bit *b20* του αθροιστή, το LS bit *i0* στο bit *b1* του αθροιστή, κ.ο.κ., όπως δείχνει το σχήμα. Τώρα δείτε ότι: (α) το bit *i19=b20*, που θα τροφοδοτήσει την επέκταση προσήμου, βρίσκεται στη στάνταρ θέση γι' αυτό το σκοπό, αριστερά· (β) τα επόμενα 8 bits, *i18,... i11*, που θα τροφοδοτήσουν τα bits *b19,... b12* του αθροιστή, βρίσκονται στην ακριβώς ίδια θέση όπως και τα bits των εντολών U που τροφοδοτούν τα ίδια bits του αθροιστή· (γ) τα 10 LS bits του Imm20, *i9,... i0*, που θα τροφοδοτήσουν τα bits *b10,... b1* του αθροιστή, βρίσκονται στην ακριβώς ίδια θέση όπως και τα bits των εντολών I (καθώς και των εντολών S και B γιά τα 6 αριστερά τους) που τροφοδοτούν τα ίδια bits του αθροιστή· και (δ) το ένα bit *i10=b11*, που περίσσεψε, τοποθετείται στην κενή θέση του *i0* των εντολών I, η οποία εδώ είναι κενή αφού το *b0=0* λόγω πολλαπλασιασμού επί 2.

Με αυτή την τοποθέτηση των bits των σταθερών ποσοτήτων λοιπόν, που επέλεξε σκόπιμα ο RISC-V, οι πολυπλέκτες που απαιτούνται στη δεύτερη είσοδο του αθροιστή, γιά να τροφοδοτήσουν το κάθε bit του από τα όποια διαφορετικά bits της εντολής (ή μηδέν) απαιτείται, είναι αυτοί που φαίνονται στο δεύτερο σχήμα. Όπως βλέπουμε, χρειάζονται 31 πολυπλέκτες, από τους οποίους οι 25 είναι μεγέθους 2-σε-1, οι 5 είναι μεγέθους 3-σε-1, και ένας είναι 4-σε-1. Αυτοί είναι πολύ οικονομικότεροι από το τι θα χρειάζονταν εάν τα bits των σταθερών ποσοτήτων ετοποθετούντο μέσα στις εντολές με τον απλοϊκό, διαισθητικά "προφανή" τρόπο που έδειχνε το πρώτο σχήμα. Στο πρώτο εκείνο σχήμα είχαμε παραλείψει, γιά απλότητα, τις εντολές U. Εάν φανταστούμε τώρα ότι περιλαμβάνμουμε και τις U, αυτές θα προσέθεταν μία ακόμα είσοδο πολύπλεξης στα bits από *b30* έως και *b1*. Το αποτέλεσμα θα ήταν 31 πολυπλέκτες, εκ των οποίων 11 μεγέθους 2-σε-1 (γιά τα bits *b30,... b20*), 9 μεγέθους 3-σε-1 (γιά τα bits *b19,... b12* και το *b0*), 7 μεγέθους 4-σε-1 (γιά τα bits *b11,... b5*), και 4 μεγέθους 5-σε-1 (γιά τα bits *b4,... b1*), δηλαδή συνολικά σημαντικά μεγαλύτεροι πολυπλέκτες απ' όσο χρειάζεται με τις βελτιστοποιήσεις τοποθέτησης των bits των σταθερών ποσοτήτων που κάνει ο RISC-V, έστω και εάν αυτές μοιάζουν σαν "ανακάτωμα" των bits.

8.3 Απλό Datapath του RISC-V Ενός Κύκλου ρολογιού ανά εντολή

Βασικό σας βοήθημα εδώ είναι οι διαφάνειες 8-12 καθώς και οι αντίστοιχες βιντεοσκοπημένες διαλέξεις. Επικουρικά, μπορείτε να διαβάσετε από το Αγγλικό βιβλίο τις σελίδες 255 εώς και 260 (ή ακόμα και τις σελίδες 243-251 του Αγγλικού βιβλίου, ή και 365-374 του Ελληνικού). Αυτά τα datapaths υλοποιοούν μόνον ένα αντιπροσωπευτικό υποσύνολο των εντολών του βασικού RISC-V. Στο βιβλίο, το εκεί datapath υλοποιεί μόνο μία

εντολή load (την **`ld`** - load double), μία εντολή store (την **`sd`** - store double), μία εντολή branch (την **`beq`** - branch if equal), και τέσσερεις εντολές **τύπου R**, τις *add, sub, and, or* (καμία εντολή πράξεων με Immediate, και κανένα άλμα). Στις διαφάνειες του μαθήματος υπάρχει και η πρόβλεψη γιά εντολές πράξεων με Immediate, και γιά άλλες τρείς διακλαδώσεις· μερικοί δρόμοι που λείπουν γιά τα άλματα επαφίενται ως ασκήσεις εδώ παρακάτω.

8.4 Opcodes και τα πεδία funct3 και funct7 στον RISC-V

Θυμηθείτε από την §4.3 ότι ο "βασικός" opcode του RISC-V, δεξιά σε κάθε εντολή, αποτελείται από 7 bits, και καθορίζει την κατηγορία, άρα και το format, της εντολής –και γιά τις εντολές με format J/U και την ίδια την εντολή. Γιά εμάς που εξετάζουμε μόνον τις 32-μπιτες εντολές του RISC-V, τα δύο δεξιότερα bits του βασικού opcode είναι πάντα "11". Ο πλήρης κατάλογος με τους βασικούς opcodes του RISC-V υπάρχει στις σελίδες 103-107 του εγχειριδίου riscv-spec-v2.2.pdf γιά όποιον ενδιαφέρεται· εδώ, ας αναφερθούμε στις βασικότερες μόνον από τις εντολές του –δείτε και τις διαφάνειες 14-16 των διαλέξεων.

Στο format J/U, που έχει μόνον τον βασικόν opcode, υπάρχουν τρείς μόνον εντολές στον RISC-V, οι γνωστές μας **`jal`** (opcode: 1101111), **`lui`** (opcode: 0110111), και **`auipc`** (opcode: 0010111). Αφού γιά τις 32-μπιτες εντολές ο βασικός opcode (των 7 bits) έχει πάντα τα δύο δεξιά του bits = 11, του "περισσεύουν 5 "χρήσιμα" bits, δηλαδή 32 συνδυασμοί. Από αυτούς, οι τέσσερεις (4) της μορφής "xx11111" δηλώνουν εντολές μεγέθους 48 ή περισσοτέρων bits, άλλοι τρείς (3) είναι γιά τις εντολές με format J/U που μόλις είπαμε, άλλοι τρείς (3) είναι κρατημένοι γιά μελλοντική χρήση (reserved for future use - RFU), τέσσερεις (4) άλλοι είναι αφημένοι γιά ειδικές, ιδιωτικές χρήσεις (custom/proprietary use), και επομένως απομένουν δεκαοκτώ (18) ακόμα βασικοί opcodes που υποδηλώνουν κατηγορίες 32-μπιτων εντολών με τα υπόλοιπα formats, I, B/S, και R. Από αυτούς τους 18 εναπομένοντες βασικούς opcodes, ενδιαφέρον γιά εμάς έχουν οι εξής έξι:

- **Opcode=0000011**: εντολές **load** (**I** format): σε αυτό το format, ο βασικός opcode επεκτείνεται με τα 3 bits του πεδίου funct3. Οι 7 από τους 8 συνδυασμούς αυτού του πεδίου είναι εν χρήσει στον RISC-V, και ορίζουν τις εξής εντολές load: *000*: load byte (lb), *001*: load half (lh), *010*: load word (lw), *011*: load double (ld); *100*: load byte unsigned (lbu), *101*: load half unsigned (lhu), *110*: load word unsigned (lwu). Παρατηρήστε ότι το μεν αριστερό bit του funct3 δηλώνει signed/unsigned, τα δε δύο δεξιά δηλώνουν το μέγεθος της ποσόοτητας που μεταφέρεται από τη μνήμη στον καταχωρητή.

- **Opcode=0100011**: εντολές **store** (**S** format): ομοίως με το format I, ο βασικός opcode επεκτείνεται και εδώ με τα 3 bits του πεδίου funct3. Οι 4 από τους 8 συνδυασμούς αυτού του πεδίου είναι εν χρήσει στον RISC-V, και ορίζουν τις εξής εντολές store: *000*: store byte (sb), *001*: store half (sh), *010*: store word (sw), *011*: store double (sd). Παρατηρήστε ότι αυτοί οι 4 κώδικες funct3 είναι ακριβώς ίδιοι με τους αντίστοιχους γιά τις εντολές load γιά data ιδίου μεγέθους: η ομοιομορφία απλοποιεί τα κυκλώματα. Υπενθυμίζεται ότι οι εντολές store δεν χρειάζονται παραλλαγή unsigned, διότι γράφουν στη μνήμη *μόνον* όσα Bytes ορίζει το μέγεθος του τελεστέου, άρα *δεν* τα επεκτείνουν αριστερά ούτε με μηδενικά ούτε με το bit προσήμου.

- **Opcode=1100011**: εντολές **branch** (**B** format): ομοίως με τα formats S και I, ο βασικός opcode επεκτείνεται και εδώ με τα 3 bits του πεδίου funct3. Οι 6 από τους 8 συνδυασμούς αυτού του πεδίου είναι εν χρήσει στον RISC-V, και ορίζουν τις εξής

εντολές branch: *000*: branch if equal (beq), *001*: branch if not equal (bne); *100*: branch if less than (blt), *101*: branch if greater or equal (bge), *110*: branch if less than unsigned (bltu), *111*: branch if greater or equal unsigned (bgeu). Παρατηρήστε και εδώ τις ομοιομορφίες: το δεξιό bit του funct3 επιλέγει τη θετική συνθήκη ή την άρνησή της, το μεσαίο bit επιλέγει σύγκριση signed/unsigned (που γιά την ισότητα δεν διαφέρουν μεταξύ τους άρα περιττεύει η δεύτερη), και το αριστερό bit δηλώνει σύγκριση ισότητας ή ανισότητας.

- **Opcode=0010011**: εντολές **πράξεων Immediate** (**I** format): ομοίως με τις προηγούμενες κατηγορίες εντολών, ο βασικός opcode επεκτείνεται και εδώ με τα 3 bits του πεδίου funct3. Οι 8 συνδυασμοί αυτού του πεδίου ορίζουν τις εξής εντολές αριθμητικών/λογικών πράξεων μεταξύ καταχωρητή και σταθερής ποσότητας Imm12: *000*: add immediate (addi), *010*: set if less than immediate (slti), *011*: set if less than immediate unsigned (sltiu); *100*: exclusive-OR immediate (xori), *110*: OR immediate (ori), *111*: AND immediate (andi); *001*: shift left logical immediate (slli), *101*: shift right immediate (σε δύο παραλλαγές). Παρατηρήσετε τις ομοιομορφίες στον κωδικό funct3 με την επόμενη κατηγορία εντολών.

- **Opcode=0110011**: εντολές **πράξεων Register-to-Register** (**R** format): σε αυτό το format, ο βασικός opcode επεκτείνεται και με τα 3 bits του πεδίου funct3 και με τα 7 bits του πεδίου funct7. Με όλα αυτά τα 10 επιπλέον bits, υπάρχει πληθώρα συνδυασμών διαθέσιμων γιά εντολές με format R, ένα μικρό ποσοστό από τους οποίους είναι εν χρήσει από τον βασικό RISC-V, ενώ οι υπόλοιποι παραμένουν διαθέσιμοι γιά τις προαιρετικές επεκτάσεις. Θα αναφέρουμε μερικούς από τους συνδυασμούς, προκειμένου να παρατηρήσετε τις ομοιομορφίες στον κωδικό funct3 με την προηγούμενη κατηγορία εντολών, καθώς και τη χρήση του κωδικού funct7 σαν επέκταση του funct3 γιά ομοειδείς κατηγορίες εντολών:
  - *funct3=000*: αριθμητικές πράξεις: αυτές διαφοροποιούνται μεταξύ τους με βάση το **funct7**: 0000000: add, 0100000: subtract, 0000001: multiply. Παρατηρήστε ότι η πρόσθεση από την αφαίρεση διαφοροποιούνται με το δεύτερο από αριστερά bit του κωδικού funct7, το οποίο, παρεμπιπτώντως είναι ακριβώς το ίδιο με το σήμα ελέγχου "add'/sub" του κυκλώματος προσθαφαιρέτη που είχαμε δεί στην Ψηφιακή Σχεδίαση.
  - *funct3=010*: set if less than (slt), *funct3=011*: set if less than unsigned (sltu); *funct3=100*: xor, *funct3=110*: or, *funct3=111*: and; *funct3=001*: shift left logical (sll). Σε όλες αυτές τις εντολές, το μεν funct3 είναι το ίδιο με εκείνο της αντίστοιχης εντολής Immediate, το δε funct7 είναι "0000000" γιά όλες τους.
  - *funct3=101*: δεξιές ολισθήσεις: αυτές διαφοροποιούνται μεταξύ τους με βάση το **funct7**: 0000000: shift right logical (δηλ. unsigned), 0100000: shift right arithmetic (δηλ. signed).

- **Opcode=1100111**: εντολή **jalr** (jump-and-link-register) (**I** format): σε αυτήν το funct3 είναι 000, ενώ οι υπόλοιποι 7 συνδυασμοί τιμών του funct3, με αυτόν τον opcode, *δεν* χρησιμοποιούνται.

Παρατηρήστε πόσο πολύ οι opcodes / function-codes γιά συναφείς/ομοειδείς κατηγορίες εντολών, ή γιά εντολές αυτές καθευατές, μοιάζουν μεταξύ τους, π.χ. διαφέρουν πολλές φορές κατά ένα μόλις bit: τέτοιοι opcodes είναι γειτονικοί στο Χάρτη Karnaugh, άρα οδηγούν σε απλοποιήσεις στο κύκλωμα ελέγχου, που σημαίνουν *μικρό και γρήγορο* κύκλωμα.

Άσκηση 8.5: Το (Συνδυαστικό) Κύκλωμα Ελέγχου

Αφού όλες οι εντολές στην παρούσα απλή υλοποίηση εκτελούνται σε έναν μόνον (αλλά αρκετά μακρύ!) κύκλο ρολογιού, ο έλεγχος θα είναι *Συνδυαστικό* Κύκλωμα, μόνον: οι έξοδοί του (τα σήματα ελέγχου) θα εξαρτώνται από την *τρέχουσα* τιμή των εισόδων του (opcode, funct3, funct7), δηλαδή από την τρέχουσα εντολή και μόνον, που την εκτελούμε στον τρέχοντα κύκλο ρολογιού, και όχι από οιαδήποτε πληροφορία από το παρελθόν, δηλαδή από προηγούμενους κύκλους ρολογιού δηλαδή από προηγούμενες εντολές.

Δείτε την διαφάνεια 13 των διαλέξεων. Τα σήματα ελέγχου (έξοδοι του κυκλώματος ελέγχου) σημειώνονται με κόκκινο. Το σήμα **immMd** (immediate constant Mode) θα το *αγνοήσουμε* σε αυτή την άσκηση: αυτό πρέπει να ελέγξει τους πολυπλέκτες της §8.2, παραπάνω, αλλά σε αυτή την άσκηση δεν θέλουμε να μπούμε σε τόση λεπτομέρεια. Από τα υπόλοιπα σήματα ελέγχου, θα διαπιστώσετε ότι, με τις υποθέσεις αυτής της άσκησης, όλα εκτός από δύο εξαρτώνται τελικά από τον βασικό Opcode και μόνον, και όχι από τα funct3 και funct7. Το πεδίο funct3 το χρειάζονται μόνον οι εντολές διακλάδωσης γιά το σήμα **pcSrc** (to branch or not to branch) και η ALU γιά το τι πράξη ακριβώς να κάνει –σήμα **aluMd**· γιά το τελευταίοα υτό σήμα, χρειαζόμαστε και το πεδίο funct7. Οι εντολές που θα υλοποιήσετε σε αυτή την άσκηση, και οι υποθέσεις γιά τα Opcodes/funct3/funct7 τους –και ιδιαίτερα γιά το τι θα κάνουμε γιά τους υπόλοιπους συνδυασμούς αυτών– έχουν ως εξής:

- **lw** (load word): Μεταξύ των 5 εντολών load (lw, lh, lb, lhu, lbu) του 32-μπιτου RISC-V (υποθέτουμε 32-μπιτο σε αυτή την άσκηση), εμείς θα υλοποιήσουμε μόνον την load-word (lw), και μάλιστα γιά όλες τις άλλες –άλλες τιμές του funct3, ακόμα και τιμές που αντιστοιχούν σε μη ορισμένες εντολές (undefined/reserved)– εμείς θα τις βλέπουμε σαν lw και θα εκτελούμε εντολή lw. Αυτό ισοδυναμεί με αγνόηση του πεδίου funct3 όταν **Opcode == 00.000.11** (που σηματοδοτεί εντολές load).

- **sw** (store word): Μεταξύ των 3 εντολών store (sw, sh, sb) του 32-μπιτου RISC-V, εμείς θα υλοποιήσουμε μόνον την store-word (sw), και μάλιστα γιά όλες τις άλλες –άλλες τιμές του funct3, ακόμα και τιμές που αντιστοιχούν σε μη ορισμένες εντολές (undefined/reserved)– εμείς θα τις βλέπουμε σαν sw και θα εκτελούμε εντολή sw. Αυτό ισοδυναμεί με αγνόηση του πεδίου funct3 όταν **Opcode == 01.000.11** (που σηματοδοτεί εντολές store).

- **Διακλαδώσεις**: Μεταξύ των 6 εντολών branch του RISC-V, εμείς θα υλοποιήσουμε μόνον τις 4 που κάνουν προσημασμένη σύγκριση. Στη θέση των δύο άλλων που κάνουν απρόσημη σύγκριση, καθώς και στις δύο περιπτώσεις όπου το funct3 είναι undefined/reserved, εμείς θα αγνοούμε τη διαφορά και θα κάνουμε προσημασμένη σύγκριση και πάλι. Την σύγκριση θα την κάνουμε μέσω αφαίρεσης (rs1)-(rs2) στην ALU, υποθέτοντας ότι δεν προκαλείται υπερχείλιση κατά την αφαίρεση –κάτι που ένας πραγματικός επεξεργαστής δεν θα έπρεπε να το υποθέτει και αγνοεί. Έτσι, οι εντολές διακλάδωσης που θα υλοποιήσουμε θα είναι:
  ```
  funct3==0x0 ⇒beq    funct3==0x1 ⇒bne
  funct3==1x0 ⇒blt    funct3==1x1 ⇒bge
  ```
- Σήμα **pcSrc**: αυτό καθορίζει εάν επόμενη εντολή θα είναι η "από κάτω", ή μιά άλλη όπως συμβαίνει γιά τις επιτυχημένες διακλαδώσεις (και γιά τα άλματα, αλλά σε αυτή την άσκηση δεν θα ασχοληθούμε με άλματα). Η κεντρική μονάδα ελέγχου ("Main Ctrl" στο σχήμα) ανιχνεύει εάν πρόκειται γιά εντολή διακλάδωσης, όποτε **Opcode ==**

11.000.11. Σε αυτή την περίπτωση ανάβει το σήμα **brOp** (1 bit). Η δευτερεύουσα μονάδα ελέγχου διακλαδώσεων –"br ctrl" στο σχήμα– όταν και μόνον όταν **brOp==1**, βάσει του πεδίου funct3 και των εξόδων zero και sign της ALU, πρέπει να αποφασίσει εάν η διακλάδωση είναι επιτυχής και τότε να στρίψει τον πολυπλέκτη από την πλευρά pcSrc=1. Υπόδειξη: η λύση υπήρχε σχεδόν έτοιμη στο κύλκωμα ελέγχου του "Απλού Υπολογιστή" του μαθήματος της Ψηφιακής Σχεδίασης.

- **Πράξεις με Immediates**: Μεταξύ των κάμποσων τέτοιων εντολών του RISC-V, εμείς θα υλοποιήσουμε μόνον τρείς, και θα αντικαταστήσουμε όλες τις υπόλοιπες με την βασική από τις τρείς, την addi, ως εξής: Όταν **Opcode == 00.100.11**, δηλαδή πράξεις με Immediates, τότε:
  `funct3==110 ⇒ori` `funct3==111 ⇒andi`
  Όλοι οι υπόλοιποι συνδυασμοί του `funct3 ⇒addi`

- **Πράξεις Register-to-Register**: Μεταξύ των κάμποσων τέτοιων εντολών του RISC-V, εμείς θα υλοποιήσουμε μόνον πέντε, και θα αντικαταστήσουμε όλες τις υπόλοιπες με την βασική μεταξύ τους, την add, ως εξής: Όταν **Opcode == 01.100.11**, δηλαδή πράξεις register-to-register, τότε:
  `funct3==110 ⇒or` `funct3==111 ⇒and` *με οιαδήποτε τιμή στο funct7*
  funct3 == 000 **και**: `funct7==0100000 ⇒sub` `funct7==0000001 ⇒mul`
  Όλοι οι υπόλοιποι συνδυασμοί των `funct3` και `funct7 ⇒add`
- Σήμα **aluMd** (4-μπιτο): παρ' ότι θα αρκούσαν 3 bits γιά να ελέγξουν τις 5 πράξεις που θέλουμε, εντούτοις γιά απλούστευση του κυκλώματος υποθέστε ότι η ALU ελέγχεται από 4 bits, όπου το αριστερό ελέγχει τον πολλαπλασιαστή, τα δύο μεσαία ελέγχουν την λογική υπομονάδα (or, and), και το δεξιό ελέγχει τον προσθαφαιρέτη, ως εξής:
  `aluMd==00x0 ⇒add` `aluMd==00x1 ⇒sub`
  `aluMd==010x ⇒ or` `aluMd==011x ⇒and`
  `aluMd==1xxx ⇒mul`
- Σήμα **aluOp** (3-μπιτο): παρ' ότι θα αρκούσαν 2 bits γιά να ελέγξουν τις 4 παρακάτω περιπτώσεις, εντούτοις γιά απλούστευση του κυκλώματος θα χρησιμοποιήσουμε 3 σύρματα γιά να καθοδηγήσει ο κεντρική μονάδα ελέγχου την δευτερεύουσα μονάδα ελέγχου της ALU, όπου: το αριστερό, όταν είναι 0, σημαίνει "αγνόησε το πεδίο funct7"· το μεσαίο, όταν είναι 0, σημαίνει "αγνόησε το πεδίο funct3"· και το δεξιό ελέγχει τον προσθαφαιρέτη, ως εξής:
  `aluOp==110` ⇒ πράξη register-to-register: κάνε ό,τι σου λένε τα πεδία funct7, funct3
  `aluOp==010` ⇒ πράξη με Immediate: κάνε ό,τι σου λέει το πεδίο funct3
  `aluOp==001` ⇒ διακλάδωση: κάνε αφαίρεση, αγνοώντας τα πεδία funct7, funct3
  `aluOp==000` ⇒ άλλη εντολή: κάνε πρόσθεση, αγνοώντας τα πεδία funct7, funct3

Άσκηση:

(α) Καταγράψτε τον Πίνακα Αληθείας και στη συνέχεια σχεδιάστε το κύκλωμα της κεντρικής μονάδας ελέγχου ("Main Ctrl" στο σχήμα). Είσοδος αυτού του κυκλώματος είναι ο Opcode (7 bits). Έξοδοί του είναι τα σήματα: aluSrc, rfSrc, mRd, mWr, rfWr, brOp, και aluOp (6+3 = 9 bits). Οι γραμμές του πίνακα δεν θα είναι 128 (2^7) αλλά πολύ λιγότερες: αυτές που αντιστοιχούν στις εντολές load, store, branch, πράξεις με Immediates, πράξεις register-to-register, καθώς και μία επιπλέον γραμμή γιά όλους τους *υπόλοιπους* συνδυασμούς των bits εισόδου. Γιά αυτούς όλους τους υπόλοιπους συνδυασμούς, θέλουμε να εξασφαλίσουμε ότι ο απλός αυτός επεξεργαστής μας θα τις βλέπει και θα τις εκτελεί σαν να ήσαν **noop** (no operation), δηλαδή δεν θα γράφει τίποτα, ούτε σε καταχωρητή ούτε στη μνήμη (ούτε θα

προσπαθεί να διαβάσει από άγνωστες/τυχαίες θέσεις μνήμης) –απλώς θα αυξάνει τον PC κατά 4, προχωρώντας έτσι στην επόμενη εντολή (χρησιμοποιήστε "don't care (x)" όπου μπορείτε, εδώ). Βάσει αυτού του πίνακα αληθείας, σχεδιάστε το κύκλωμα αυτό χρησιμοποιώντας πύλες not, and, or οσωνδήποτε εισόδων.

(β) Καταγράψτε ομοίως τον Πίνακα Αληθείας και στη συνέχεια σχεδιάστε το κύκλωμα της δευτερεύουσας μονάδας ελέγχου διακλαδώσεων ("br ctrl" στο σχήμα). Είσοδοι είναι τα σήματα brOp, zero, sign, και το πεδίο funct3 της εντολής (3+3 = 6 bits)· αξιοποιήστε ενδείξεις "x (don't care)" στην πλευρά εισόδων του πίνακα προκειμένου να μειώστε το πλήθος γραμμών του (δείτε και το εργαστήριο/άσκηση 12 του μαθήματος της Ψηφιακής Σχεδίασης). Εδώ, το κύκλωμά σας επιτρέπεται να περιέχει και πύλες xor (καθώς και έναν πολυπλέκτη, αν τον προτιμάτε αντί των ισοδυνάμων του δύο πυλών and, μίας not, και μίας or).

(γ) Τέλος, κάντε το ίδιο και γιά την δευτερεύουσα μονάδα ελέγχου της ALU ("alu ctrl" στο σχήμα). Και εδώ, μειώστε το πλήθος γραμμών του πίνακα αληθείας αξιοποιώντας ενδείξεις "x (don't care)" στην πλευρά εισόδων, καθώς και γραμμών του τύπου "υπόλοιποι συνδυασμοί πεδίου...". Βάσει αυτού του πίνακα αληθείας, σχεδιάστε το κύκλωμα αυτό χρησιμοποιώντας πύλες not, and, or οσωνδήποτε εισόδων.

Άσκηση 8.6: Προσθήκη των Εντολών jal, jalr στο Datapath ενός Κύκλου

Το datapath του μαθήματος (διαφάνειες 2 και 8-12 των διαλέξεων) δεν μπορεί να υλοποιήσει την εντολή jump-and-link (jal - κλήση διαδικασίας) διότι δεν έχει τρόπο να γράψει το PC+4 στον καταχωρητή προορισμού, ούτε την εντολή jump-and-link-register (jalr) διότι, επιπλέον, δεν έχει ούτε τρόπο να φέρει κάτι που πηγάζει από καταχωρητές γενικού σκοπού σαν επόμενη τιμή του PC. Γιά την jal, παρατηρήστε ότι τιμή PC+4 υπάρχει ήδη κάπου στο datapath, αλλά δεν υπάρχει ο δρόμος γιά να φτάσει εκεί που την χρειαζόμαστε. Γιά την jalr, παρατηρήστε ότι η διεύθυνση μνήμης στην οποία αυτή κάνει άλμα προκύπτει με ακριβώς τον ίδιο τρόπο όπως μιά ήδη υπάρχουσα εντολή: με *ποιάν* και ποιός είναι ο τρόπος;

Σχεδιάστε τις αλλαγές που απαιτούνται στο datapath της διαφάνειας 2 προκειμένου να μπορεί αυτό να εκτελεί και τις εντολές jal και jalr. Επίσης, σχολιάστε (σε απλό κείμενο) τις αλλαγές που θα χρειαστούν στον έλεγχο και στα σχετικά σήματα, **χωρίς** πάντως να ζητούνται αλλαγές/προσθήκες στους πίνακες αληθείας ή στα κυκλώματα του ελέγχου.

Άσκηση 8.7: Πόσο αργό είναι το ρολόϊ της Απλής Υλοποίησης;

Στην απλή υλοποίηση του ενός (μακρύ) κύκλου ρολογιού ανά εντολή της §4.4 του βιβλίου, υποθέστε ότι οι βασικές μονάδες υλικού (hardware) έχουν τις εξής καθυστερήσεις, η καθεμία:

- (Κρυφή) Μνήμη Εντολών: 500 ps
- (Κρυφή) Μνήμη Δεδομένων: 500 ps
- Αρχείο Καταχωρητών (ανάγνωση): 300 ps
- Αρχείο Καταχωρητών (εγγραφή): 200 ps
- ALU: 400 ps (το ίδιο και οι αθροιστές γιά τον PC)
- Συνδυαστική Λογική Ελέγχου: 200 ps
- Αγνοήστε τις καθυστερήσεις πολυπλεκτών και του PC

΄Εστω ότι την χρονική στιγμή t=0 ps έρχεται η ακμή ρολογιού που ξεκινά την εκτέλεση της τρέχουσας εντολής. Τότε πείτε σε ποιές χρονικές στιγμές, στην **χειρότερη περίπτωση**, θα είναι έγκυρες και έτοιμες οι εξής τιμές ή σήματα:

a. Η εντολή.
b. Τα σήματα ελέγχου.
c. Οι τιμές των καταχωρητών πηγής.
d. Η έξοδος της ALU.
e. Οι τιμές PC+4 και PC+2×Imm12.
f. Η τιμή γιά τον επόμενο PC (μετά τον πολυπλέκτη, στην είσοδο του PC).
g. Η έξοδος της Data Memory.
h. Η είσοδος Write Data του Αρχείου Καταχωρητών.
i. Ολοκλήρωση της τυχόν εγγραφής στο Αρχείο Καταχωρητών.

Συνολικά, το ρολόϊ (που έχει *σταθερή* συχνότητα και περίοδο), πρέπει να προσφέρει χρόνο επαρκή γιά να ολοκληρωθεί η *χειρότερη (αργότερη)* από τις λειτουργίες που θέλουμε να χωρούν σε έναν κύκλο ρολογιού. Βάσει των παραπάνω απαντήσεών σας, ποιά είναι η χειρότερη; Πόσο τουλάχιστο λοιπόν θα πρέπει να διαρκεί η περίοδος (κύκλος) του ρολογιού γιά αυτόν τον επεξεργαστή, σε ns; Πόση, το πολύ, επομένως θα μπορεί να είναι η συχνότητα του ρολογιού, σε MHz;

΄Εστω ότι προσθέτουμε και εντολή διαίρεσης, η οποία κάνει την ALU να έχει καθυστέρηση 4400 ps, αντί 400 ps προηγουμένως. Τότε, πόση θα γίνει η μέγιστη δυνατή συχνότητα ρολογιού; Πόσες φορές πιό αργός θα γίνει τότε ολόκληρος ο υπολογιστής, επειδή η *κάθε* εντολή, που παίρνει πάντα έναν ολόκληρο κύκλο ρολογιού, θα αργεί τόσες φορές περισσότερο όσο η συχνότητα του ρολογιού είναι τώρα χαμηλότερη;

8.8 Περί Σπατάλης Υπολογιστικών Πόρων

Διαβάστε τις τελευταίες δύο σελίδες της §4.4 του βιβλίου (σελ. 261-262 στο Αγγλικό βιβλίο): Σε αυτή την απλή υλοποίηση του ενός (μακρύ) κύκλου ρολογιού ανά εντολή, όπως καταλάβατε και από την άσκηση 8.7, βασικά, την κάθε στιγμή, *μία μόνο μονάδα (πόρος) υλικού* (hardware resource) δουλεύει, ενώ οι υπόλοιπες "κάθονται", περίπου σαν την χαρακτηριστική φωτογραφία δεξιά – οι μεν μονάδες *πριν* από αυτήν που εργάζεται "κάθονται" με σταθερές τις εισόδους τους προκειμένου να κρατούν σταθερές τις εξόδους τους γιά να μπορεί να εργάζεται με αυτές σαν εισόδους η μονάδα που εργάζεται, οι δε μονάδες *μετά* από αυτήν που εργάζεται "κάθονται" άπραγες διότι οι είσοδοί τους δεν έχουν σταθεροποιηθεί ακόμα

στην τελική, έγκυρη τιμή τους.

8.9 Υλοποίηση Πολλαπλών Κύκλων ανά Εντολή, γιά Εξοικονόμηση Πόρων

Το θέμα αυτό εμελετάτο εν εκτάσει (και ήταν αντικείμενο μικρού "project") σε παλαιότερες εκδόσεις αυτού του μαθήματος, και υπήρχε και στις προηγούμενες εκδόσεις του βιβλίου, αλλά δεν υπάρχει στην 4η έκδοση του βιβλίου, προκειμένου να επικεντρωθεί το μάθημα στην ομοχειρία (pipelining), και ομοίως και εμείς φέτος το καλύψαμε εξαιρετικά επίτροχάδην. Δείτε την διαφάνεια 21 του μαθήματος και τη σχετική διάλεξη γιά μιάν επιγραμματική παρουσίαση του θέματος: *(i)* Χρησιμοποιούμε ένα ρολόϊ περίπου 5 φορές πιό γρήγορο· η "χειρότερη" εντολή (load) παίρνει 5 κύκλους του νέου ρολογιού, άρα περίπου το ίδιο αργή· οι άλλες εντολές παίρνουν 4 ή και 3 κύκλους του νέου ρολογιού καθεμία, άρα λίγο πιό γρήγορες. *(ii)* Κάνουμε οικονομία στους πόρους υλικού (αθροιστές/ALU, μνήμες), επαναχρησιμοποιώντας τις μοναδικές σχετικές μονάδες που έχουμε τώρα, αξιοποιώντας προς τούτο το πιό "λεπτόκοκκο" σήμα χρονισμού που έχουμε τώρα, δηλαδή το ρολόϊ 5-πλάσιας συχνότητας. *(iii)* Γιά να πετύχουμε οι εντολές διακλάδωσης να τελειώνουν σε 3 μόλις κύκλους ρολογιού, προετοιμάζουμε (στον μοναδικό αθροιστή/ALU που έχουμε) τις τιμές PC+4 και PC+2*Imm όσο πιό νωρίς μπορούμε –πριν να μάθουμε ποιά είναι η εντολή (γιά το PC+4), και πριν να μάθουμε εάν η εντολή είναι διακλάδωση ή όχι (γιά το PC+2*Imm). Εάν θέλετε περισσότερες λεπτομέρειες και υλικό, ανατρέξτε σε προηγούμενες εκδόσεις του βιβλίου, ή μελετήστε τις σημειώσεις του μαθήματος του έτους 2009, ενότητες: 10, 11, και 12.

Τρόπος Παράδοσης:
Κάντε την άσκηση αυτή από τώρα (και μελετήστε την και πριν την εξέταση Προόδου για να προετοιμαστείτε γιά την Πρόοδο και) για να την έχετε έτοιμη, αλλά θα την παραδώσετε λίγο αργότερα, μαζί με τη σειρά ασκήσεων 9. Ετοιμάστε την σε μορφή **PDF** (μόνον) (μπορεί να είναι κείμενο μηχανογραφημένο ή/και "σκαναρισμένο" χειρόγραφο, αλλά θα την παραδώσετε σε μορφή PDF *μόνον*).
