HY-225: Οργάνωση Υπολογιστών
Άνοιξη 2020

Τμ. Επ. Υπολογιστών
© Πανεπιστήμιο Κρήτης

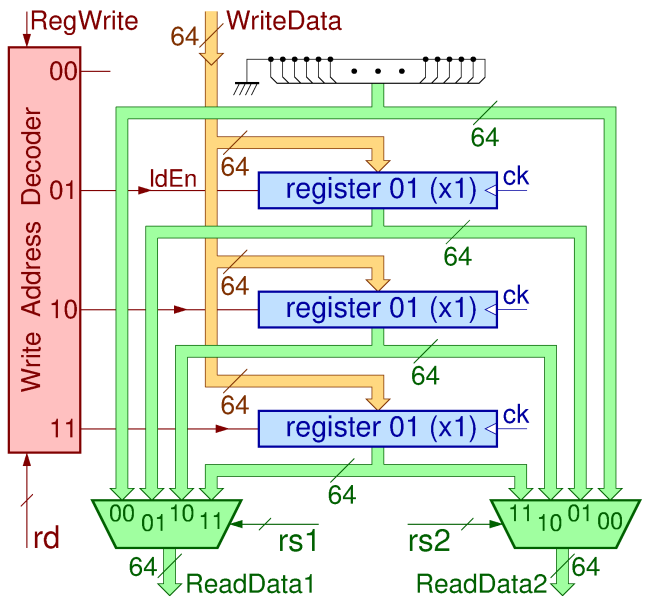
Ασκήσεις 8: Μία Απλή Υλοποίηση του RISC-V σε Έναν Κύκλο Ρολογιού ανά Εντολή

Προθεσμία έως Δευτέρα 23 Μαρτίου 2020, ώρα 23:59 (βδ. 8.1)
Υπενθύμιση Διαγωνισμού Προόδου: Σάββατο 14 Μαρτίου 2020, ώρα 12-2

Βιβλίο: (α) Προσημασμένοι αριθμοί: Διαβάστε την §2.4 (σελίδες 127-134 στο Ελληνικό βιβλίο, pp. 74-81 στο Αγγλικό), και ξαναθυμηθείτε το Εργαστήριο 6 της Ψηφιακής Σχεδίασης. (β) 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. (γ) 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 Προσημασμένοι Αριθμοί, Επέκταση Προσήμου

Οι σημερινοί επεξεργαστές παριστάνουν τους προσημασμένους (signed) αριθμούς σε κωδικοποίηση συμπληρώματος ως-προς-2, όπως είχαμε δεί στο μάθημα της Ψηφιακής Σχεδίασης (HY-120, ενότητα 6.3). Σε αυτή την αναπαράσταση, η μετατροπή προσημασμένου (signed) αριθμού από λιγότερα σε περισσότερα bits γίνεται με την εξής τεχνική, που ονομάζεται "*επέκταση προσήμου*" *(sign extension)* και αποδεικνύεται μαθηματικά ως εξής:

Έστω ο προσημασμένος ακέραιος $A_{s,k}$ με *k bits*, τον οποίο θέλουμε να μετατρέψουμε στον ίδιο αριθμό $A_{s,n}$ με *n bits*: $A_{s,n} = A_{s,k}$ όπου n>k. Εάν τα bits του $A_{s,k}$ τα ερμηνεύσουμε σαν μη προσημασμένο (unsigned) ακέραιο, τότε θα μοιάζουν με (θα δηλώνουν) έναν αριθμό που ας το ονομάσουμε $A_{u,k}$ και ομοίως εάν τα bits του $A_{s,n}$ τα ερμηνεύσουμε σαν unsigned τότε θα μοιάζουν με τον $A_{u,n}$. Εάν ο $A_{s,k} = A_{s,n}$ είναι μη αρνητικός (δηλ. θετικός ή μηδέν), τότε, κατά τον ορισμό της κωδικοποίησης συμπληρώματος ως-προς-2, (α) το αριστερό bit τους θα είναι μηδέν, και (β) οι απρόσημες ερμηνίες τους θα είναι: $A_{u,k} = A_{s,k}$ και $A_{u,n} = A_{s,n}$, και αφού $A_{s,n} = A_{s,k}$ τότε θα είναι και: $A_{u,n} = A_{u,k}$. Αυτοί οι δύο

τελευταίοι, αφού είνα unsigned ακέραιοι, με n και k bits αντίστοιχα, και είναι ίσοι μεταξύ τους, προκύπτει ότι ο $A_{u,n}$ που έχει περισσότερα bits θα είναι ίδιος με τον $A_{u,k}$ αλλά με n-k μηδενικά προστεθημένα αριστερά από τον $A_{u,k}$.

Αλλιώς, εάν οι $A_{s,n} = A_{s,k}$ είναι αρνητικοί, τότε, κατά τον ορισμό μας, (α) το αριστερό bit τους θα είναι ένα, και (β) οι απρόσημες ερμηνίες τους θα είναι: $A_{u,k} = A_{s,k} + 2^k$ και $A_{u,n} = A_{s,n} + 2^n$. Δεδομένου ότι: $A_{s,n} = A_{s,k}$ προκύπτει ότι θα είναι και: $A_{u,n} - 2^n = A_{u,k} - 2^k$. Επομένως, η αναπαράσταση που ψάχνουμε είναι η: $A_{u,n} = A_{u,k} - 2^k + 2^n = A_{u,k} + (2^n - 2^k) = (2^{n-k} - 1) \cdot 2^k + A_{u,k}$. Σε αυτήν την τελευταία έκφραση, ο μεν αριστερός προσθετέος, $(2^{n-k} - 1) \cdot 2^k$, αποτελείται από (n-k) το πλήθος άσσους (που είναι ο αριθμός $2^{n-k} - 1$) ολισθημένους αριστερά κατά k θέσεις bits (που είναι ο πολλαπλασιασμός επί 2^k), ο δε δεξιός προσθετέος, $A_{u,k}$, είναι τα αρχικά k bits του αρχικού αριθμού που μας δόθηκε. Δεδομένου ότι ο πρώτος προσθετέος έχει όλο μηδενικά στις k δεξιές θέσεις, ο δε δεύτερος προσθετέος έχει όλο μηδενικά στις (n-k) αριστερές θέσεις, το άθροισμά τους θα είναι προφανώς απλώς η "συγκόλληση" (concatenation) των δύο αυτών ποσοτήτων. Επομένως, η αναπαράσταση με n bits του αρχικού αριθμού που μας δόθηκε θα αποτελείται από τα αρχικά k bits, δεξιά, μαζί με (n-k) άσσους κολλημένους ακριβώς αριστερά τους.

Συνολικά λοιπόν, γιά να μετατρέψουμε ένα προσημασμένο (signed) ακέραιο από k (λιγότερα) bits σε n (περισσότερα) bits, δεν έχουμε παρά να κάνουμε το εξής: Τοποθετούμε αριστερά από τα bits που μας δόθηκαν (n-k) επιπλέον bits τα οποία είναι όλα τους *αντίγραφα του αριστερού* (most significant) bit του αριθμού που μας δόθηκε, δηλαδή αντίγραφα του bit που υποδεικνύει το πρόσημο του δοθέντος αριθμού (0 γιά θετικούς ή το μηδεν, 1 γιά αρνητικούς). Η πράξη αυτή ονομάζεται επομένως, προφανώς, *επέκταση προσήμου* (sign extension).

8.3 Εντολές "Upper Immediate" (lui, auipc) γιά αυθαίρετες 32-μπιτες Σταθερές

Η εντολή **`lui rd, Imm20`** (load upper immediate), με format "**U**" (βλ. §4.3), στον 32-μπιτο RISC-V, γράφει τα 20 bits του πεδίου Imm20 της στα αριστερά 20 bits του καταχωρητή rd, και μηδενίζει τα δεξιά 12 bits του rd. Στον 64-μπιτο RISC-V, βάζει τα παραπάνω 32 bits στο δεξιό (LS) ήμισυ του rd, και τα κάνει sign-extend στο αριστερό (MS) ήμισυ του rd. Παρ' ότι ονομάζεται "load" όμως **δεν** είναι εντολή προσπέλασης της μνήμης δεδομένων –δεν ανήκει στην κατηγορία των εντολών load.

Στη συνήθη χρήση της ακολουθείται από μιάν εντολή `addi` (add immediate), η οποία προσθέτει στον ίδιο καταχωρητή το 12-μπιτο Immediate της. Δεδομένου ότι η `lui` έβαλε 12 μηδενικά δεξιά στον καταχωρητή, η πρόσθεση καταλήγει στο να αφήσει στα 12 δεξιά bits τη σταθερή ποσότητα από την εντολή `addi`. Στον 32-μπιτο RISC-V, στα 20 αριστερά bits του καταχωρητή, η μεν εντολή `lui` έβαλε την 20-μπιτη σταθερά της, η δε εντολή `addi` προσθέτει σε αυτήν είτε (α) 20 μηδενικά εάν η (12-μπιτη) σταθερά της έχει 0 στο αριστερό bit της, είτε (β) 20 άσους εάν η (12-μπιτη) σταθερά της έχει 1 στο αριστερό bit της. Στη μεν περίπτωση (α) τα 20 μηδενικά αφήνουν αναλοίωτα τα 20 αριστερά bits, στη δε περίπτωση (β) η πρόσθεση 20 άσων ισοδυναμεί με την πρόσθεση του αριθμού -1 (μείον 1) σε αυτά τα 20 bits. Έτσι τελικά μπορούμε να συνθέσουμε την οιαδήποτε αυθαίρετη 32-μπιτη σταθερά, βάζοντας τα μεν 12 δεξιά bits της στο Immediate της `addi`, τα δε 20 αριστερά bits της, είτε αυτούσια είτε αυξημένα κατά +1, στο Immediate της `lui` – όπου η επιλογή "αυτούσια" ή "αυξημένα κατά +1" γίνεται όταν τα 12ο από δεξιά bit είναι 0 ή 1, αντίστοιχα. Στο βιβλίο η εντολή αυτή, καθώς και η τυπική χρήση της, περιγράφονται στην αρχή της §2.10.

Επίσης 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 rellocate), 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 βελησνεκές ±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 βελησνεκές ±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 επίσης "**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 $PC+HI{\times}2^{12}$ 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 παραπάνω 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 $PC+HI{\times}2^{12}+LO = PC+Const32$:

- 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)`**

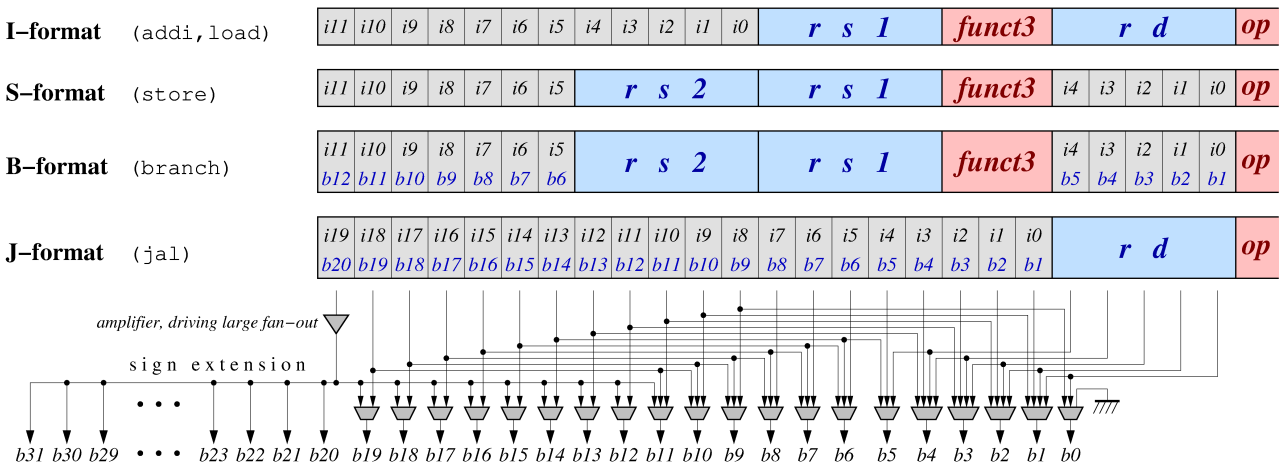
8.4 Θέση των bits των Σταθερών Immediate στον RISC-V

[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 .

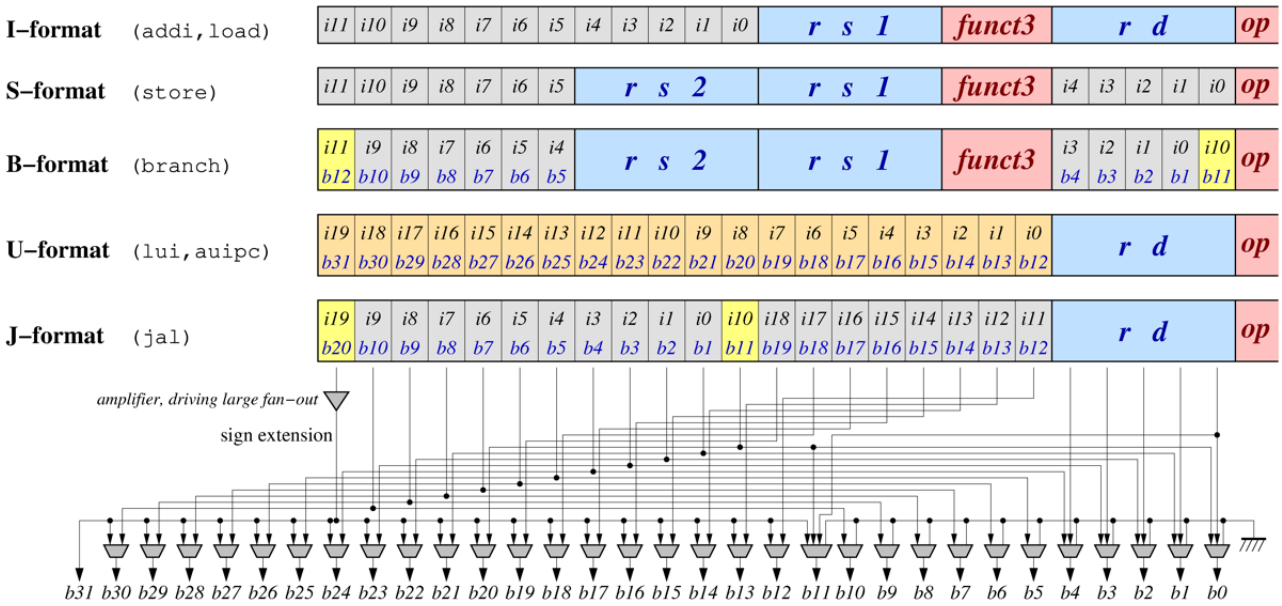
Όπως είχαμε δεί στην §4.3, σταθερές "Immediate" υπάρχουν στα format I, B, S (σταθερές των 12 bits) και J, U (σταθερές των 20 bits) του RISC-V. Οι μορφές B και S είναι παραλλαγές της I που απλώς κόβουν το 12-μπιτο Immediate σε δύο κομάτια προκειμένου να μένουν πάντα σε σταθερή θέση τα πεδία των καταχωρητών. 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 (§8.2), 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.

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

Διαβάστε από το Αγγλικό βιβλίο τις σελίδες 255 εώς και 260. Βοηθητικά μπορείτε επίσης να δείτε και τις σελίδες 243-251 του Αγγλικού βιβλίου, ή και 365-374 του Ελληνικού. Αυτό το datapath του βιβλίου υλοποιεί μόνο ένα αντιπροσωπευτικό υποσύνολο των εντολών του βασικού RISC-V: 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).

8.6 Το (Συνδυαστικό) Κύκλωμα Ελέγχου

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 εξατάζουμε 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 σε στις βασικότερες μόνον από τις εντολές του. 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 ποσότητας που μεταφέρεται από τη μνήμη στον καταχωρητή. Το βιβλίο, από όλες αυτές υλοποιεί μόνον την load double, την οποία την αναγνωρίζει κοιτώντας μόνον τον βασικό opcode (0000011), ενώ αγνοεί το funct3 –που φυσικά ένας πραγματικός RISC-V δεν θα μπορούσε να το αγνοήσει.
- **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. Το βιβλίο, από όλες αυτές υλοποιεί μόνον την store double, την οποία την αναγνωρίζει κοιτώντας μόνον τον βασικό opcode (0100011), ενώ αγνοεί το funct3 –που φυσικά ένας πραγματικός RISC-V δεν θα μπορούσε να το αγνοήσει.
- **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. Το βιβλίο, από όλες αυτές υλοποιεί μόνον την branch if equal, την οποία την αναγνωρίζει κοιτώντας μόνον τον βασικό opcode (1100011), ενώ αγνοεί το funct3 –που φυσικά ένας πραγματικός RISC-V δεν θα μπορούσε να το αγνοήσει.
- **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). Το βιβλίο *δεν* υλοποιεί καμία από αυτές τις εντολές· εμείς τις αναφέρουμε προκειμένου να 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 διαθλέσιμοι 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.
    - *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).

  Και πάλι, εδώ, παρατηρήστε τις ομοιομορφίες στις επιλογές κωδικών. Το βιβλίο, από όλες αυτές υλοποιεί μόνον τις add, sub, or, και and, τις οποίες μπορεί κανείς να αναγνωρίσει κοιτώντας τον βασικό opcode (0110011), τον κωδικό funct3, και το δεύτερο από αριστερά bit του κωδικού funct7

γιά να διαφοροποιηθεί η πρόσθεση από την αφαίρεση –το οποίο, παρεμπιπτώντως, είναι ακριβώς το ίδιο με το σήμα ελέγχου "add'/sub" του κυκλώματος προσθαφαιρέτη που είχαμε δεί στην Ψηφιακή Σχεδίαση.
- **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.

Όπως είπαμε, το βιβλιο υλοποιεί μόνον τις εντολές ld, sd, beq, και τέσσερεις εντολές πράξεων τύπου R (add, sub, and, or). Οι τέσσερεις εντολές πράξεων κάνουν ακριβώς τις ίδιες λειτουργίες και οι τέσσερεις στο υπόλοιπο datapath, εκτός στην ALU όπου η καθεμιά τους κάνει διαφορετική πράξη. Έτσι, όλα τα υπόλοιπα σήματα ελέγχου των ALU control lines είναι κοινά και γιά τις τέσσερεις. Με αυτό το σκεπτικό, όσον αφορά όλα τα υπόλοιπα σήματα ελέγχου, το βιβλίο είναι σαν να υλοποιεί μόνον τέσσερεις εντολές: ld, sd, beq, και "ALU", τις οποίες και μπορεί να ξεχωρίσει μεταξύ τους από τον βασικό opcode και μόνον, όπως αναφέραμε συγκεκριμένα πιό πάνω. Αυτό το κάνει το Αγγλικό βιβλίο στη σελ. 261, figure 4.22, που είναι ο Πίνακας Αληθείας του κυρίως κυκλώματος ελέγχου (Control) (σχεδιασμένος λίγο ανορθόδοξα, στριμένος κατά 90 μοίρες). Όσον αφορά τον έλεγχο της ALU, ο κυρίως έλεγχος γεννά 2 σήματα, *ALUop*, τα οποία λένε σε ένα δευτερεύον κύκλωμα ελέγχου ("ALU control"): είτε (α) "κάνε πρόσθεση ανεξαρτήτως της τιμής των funct3, funct7" (κωδικός 00), είτε (β) "κάνε αφαίρεση ανεξαρτήτως της τιμής των funct3, funct7" (κωδικός 01), είτε (γ) "κάνε ό,τι πράξη σου υποδεικνύουν τα funct3, funct7" (κωδικός 10). Με βάση αυτή την κωδικοποίηση, το σχήμα 4.13 (σελ. 253) του Αγγλικού βιβλίου δίνει τον πίνακα αληθείας του δευτερεύοντος κυκλώματος, ALU control (οι αριστερές στήλες θα έπρεπε να λένε "01", "10", και όχι "x1", "1x", τα οποία είναι διφορούμενα διότι κάνουν και τα δύο match την περίπτωση "11"). Με αυτές τις διευκρινίσεις, διαβάστε από το Αγγλικό βιβλίο την §4.4, με την εξής σειρά: figure 4.22 (p. 261), και μετά τις 4 πρώτες σελίδες (pp. 251-254).

Άσκηση 8.7: Προσθήκη Εντολών στον Απλό RISC-V ενός Κύκλου

(α) Το datapath και ο έλεγχος της απλή υλοποίησης της §4.4 του (Αγγλικού) βιβλίου (σελ. 251-261) *δεν* υλοποιούν την εντολή **`addi`** (add immediate). Γιά να την προσθέσουμε εμείς εδώ, απαιτούνται αλλαγές και στο datapath, ή μόνον στον έλεγχο; Απαντήστε επιχειρηματολογώντας ότι το μεν πρώτο μέρος της εντολής addi κάνει ακριβώς το ίδιο με ό,τι κάνει(ουν) και άλλη(ες) ήδη υπάρχουσα(ες) εντολή(ες), το δε αποτελείωμά της επίσης κάνει ακριβώς το ίδιο με κάποια(ες) άλλη(ες) εντολή(ες). Με ποιάν(ες) εντολή(ες), κάθε φορά, και ποιές λειτουργίες είναι αυτές που είναι ακριβώς ίδιες;

(β) Γιά να γίνει η προσθήκη (α) θα πρέπει να συμπληρωθεί ο πίνακας αληθείας του κυρίως κυκλώματος ελέγχου, σε σχέση με αυτόν που δίνει το (Αγγλικό) βιβλίο στο fig. 4.22 (p. 261). Επειδή η addi είναι η μόνη από τις εντολές πράξεων-Immediate –δηλαδή τις εντολές με βασικό Opcode = 0010011– που θέλουμε να προσθέσουμε, θα αρκεστούμε να την αναγνωρίζουμε από τον βασικό opcode, σύμφωνα με όσα είπαμε ακριβώς παραπάνω (§8.6), και θα αγνοήσουμε το funct3 (που κανονικά θα έπρεπε να το ελέγχουμε και να είναι 000 γιά την addi όπως είπαμε στην §8.6). Κάντε λοιπόν αυτή την αλλαγή στον πίνακα του σχ. 4.22 του βιβλίου, προσθέτοντας την περίπτωση του opcode των εντολών πράξεων-Immediate, και γιά αυτή την περίπτωση προδιαγράφοντας τις τιμές των 8 σημάτων έλεγχου που είναι οι έξοδοι αυτού του κυκλώματος ελέγχου του σχ. 4.22.

(γ) Στη συνέχεια θέλουμε να προσθέσουμε, εκτός από την εντολή beq που υπάρχει, και τις εντολές **`bne`**, **`blt`**, και **`bge`**, οι οποίες υλοποιούνται κατ' αντιστοιχία των όμοιών τους που είχαμε και στον "Απλό Υπολογιστή" του μαθήματος της Ψηφιακής Σχεδίασης. Γιά να το πετύχουμε, χρειαζόμαστε, εκτός από την έξοδο *Zero* της ALU, και άλλη μία έξοδο, *Sign*, που θα είναι το πρόσημο (bit_{63}) της εξόδου της ALU. Επίσης, γιά να ξέρουμε ποιά από τις 4 εντολές διακλάδωσης που θα έχουμε είναι η παρούσα, θα χρειαστεί να κοιτάμε δύο από τα bits του πεδίου funct3 της εντολής: πείτε *ποιά* δύο bits. Σχεδιάστε το νέο κύκλωμα που θα αντικαταστήσει στο σχ. 4.21 (σελ. 260) του (Αγγλικού) βιβλίου την πύλη AND που ελέγχειο τον πολυπλέκτη που γεννά τη νέα τιμή του PC (υπόδειξη: η λύση υπήρχε σχεδόν έτοιμη στο κύλκωμα ελέγχου του "Απλού Υπολογιστή" του μαθήματος της Ψηφιακής Σχεδίασης).

(δ) Τέλος, θέλουμε να προσθέσουμε και την εντολή **`jalr`** (jump-and-link-register - §6.4). Παρατηρήστε πρώτον 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; Γιά να μπορέσει να γίνει το άλμα θα πρέπει να προσθέσετε έναν ακόμη πολυπλέκτη, μετά τον ήδη υπάρχοντα γιά τις διακλαδώσεις, στο δρόμο δημιουργίας της νέας τιμής του PC, και να τον τροφοδοτήστε από το κάταλληλο σημείο: σχεδιάστε τα σχετικά κομάτια του datapath (δεν χρειάζεται ολόκληρο το datapath), και δείξτε τις προσθήκες. Ο πολυπλέκτης αυτός θα πρέπει να ελέγχεται από ένα σήμα που θα ενεργοποιείται όταν opcode = 1100111 και funct3 = 000, όπως είπαμε παραπάνω (§8.6) (*δεν* ζητείται να σχεδιάσετε αυτή την αποκωδικοποίηση). Επίσης η εντολή αυτή, σαν εντολή καλέσματος διαδικασίας, γράφει την τιμή PC+4 στον καταχωρητή προορισμού της, `rd`. Η τιμή PC+4 υπάρχει ήδη κάπου στο datapath; Γιά να μπορέσει η τιμή αυτή να έλθει και να γραφτεί στον καταχωρητή προορισμού, θα πρέπει να προσθέσετε έναν ακόμη πολυπλέκτη, μετά τον ήδη υπάρχοντα γιά τις εγγραφές, στο δρόμο προς την είσοδο Write data του αρχείου των καταχωρητών, και να τον ελέγξετε με το ίδιο παραπάνω σήμα ελέγχου. Δείξτε και αυτές τις προσθήκες στα σχετικά κομάτια του datapath που σχεδιάσατε και που θα παραδώσετε με τις απαντήσεις σας.

Άσκηση 8.8: 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 κύκλορολογιού. 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;

8.9 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 8.8, 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.

8.10 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. Εάν θέλετε, πέραν της διάλεξης, να βρείτε και λεπτομερές περαιτέρω περιγραφικό υλικό, ανατρέξτε σε προηγούμενες εκδόσεις του βιβλίου, ή μελετήστε τις σημειώσεις 10, 11, και 12 του μαθήματος του έτους 2009.

Τρόπος Παράδοσης:
Παραδώστε την άσκηση αυτή on-line, σε μορφή **PDF** (μόνον) (μπορεί να είναι κείμενο μηχανογραφημένο ή/και "σκαναρισμένο" χειρόγραφο, αλλά *μόνον* σε μορφή PDF). Παραδώστε μέσω **`turnin ex08@hy225 [directoryName]`** ένα αρχείο ονόματι **`ex08.pdf`** που θα περιέχει τις απαντήσεις σας σε όλες τις ερωτήσεις.
