ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ
UNIVERSITY OF PATRAS

Πολυτεχνική Σχολή
Τμήμα Μηχανικών Η/Υ & Πληροφορικής

Διπλωματική Εργασία

Μια ενδεικτική υλοποίηση **RISC-V** επεξεργαστή και ενός υποστηρικτικού **Assembler**

UPatras RISC-V

Βάιος Λασκαρέλιας
A.M.: 1054432

Επιβλέπων
Χαρίδημος Βέργος, Καθηγητής

Μέλη Εξεταστικής Επιτροπής
Δημήτριος Νικολός, Καθηγητής
Νικόλαος Σκλάβος, Αναπληρωτής Καθηγητής

Πάτρα, Οκτώβριος 2021

Ευχαριστίες

Θα ήθελα να εκφράσω την εκτίμησή μου για τους ανθρώπους που με στήριξαν έμπρακτα σε όλη την πορεία των σπουδών μου. Χωρίς αυτούς, η εκπόνηση αυτής της διπλωματικής εργασίας δεν θα ήταν δυνατή.

Thanks

I would like to express my appreciation to each and every one who supported me during my studies. Without them, the completion of this thesis would not be possible.

Περίληψη

Η παρούσα διπλωματική εργασία υλοποιεί ένα λειτουργικό θεωρητικό μοντέλο ενός διασωληνωμένου επεξεργαστή βασισμένου στην αρχιτεκτονική RISC-V, υλοποιημένο σε Icarus Verilog, με έμφαση στην απλότητα της τελικής σχεδίασης. Το μοντέλο συνοδεύεται από έναν συμβατό, επεκτάσιμο Assembler για την διευκόλυνση του προγραμματισμού του επεξεργαστή, γραμμένο σε C, με την βοήθεια των εργαλείων Flex και Bison. Ο Assembler είναι συμβατός με οποιονδήποτε επεξεργαστή βασισμένο στην αρχιτεκτονική RV32I.

Για την επιβεβαίωση της σωστής και κατά των προδιαγραφών λειτουργίας του μοντέλου, εκτελέστηκε μεγάλος αριθμός πιθανών ακραίων περιπτώσεων μεμονωμένων εντολών και ακολουθιών εντολών. Τα παραγόμενα αποτελέσματα των εντολών συγκρίθηκαν με αυτά που περιγράφονται στις προδιαγραφές κατά την διάρκεια, και μετά το τέλος της εκτέλεσής τους. Για τον έλεγχο του Assembler, παρήχθησαν περίπλοκα διανύσματα εντολών και κωδικοποιήσεων άμεσων δεδομένων, τα οποία ελέγθηκαν χειροκίνητα και διασταυρώθηκαν με άλλους συμβατούς RISC-V Assembler. Τελικά, ο Assembler παράγει τα ορθά διανύσματα κωδικοποιήσεων των εντολών και το μοντέλο του επεξεργαστή εκτελεί τα προγράμματα με τα αναμενόμενα αποτελέσματα, κάνοντας το σύνολο των υλοποιήσεων συμβατό με την αρχιτεκτονική RISC-V.

Abstract

The processor is a timeless concept of Computer Science, since it materialises the theoretical concepts of Information Theory. Starting from the theoretical Turing Machine and the mechanical Analytical Engine of Charles Babbage, to todays chaotic Artificial Intelligence models and interconnected clusters of supercomputers, the processor is an integral piece of every computation system today.

This thesis implements a working theoretical model of a RISC-V compatible CPU, implemented in Icarus Verilog, designed with simplicity in mind. This CPU model is accompanied by a compatible, extensible RISC-V Assembler, written in C with the help of Flex and Bison tools, in order to make the model's CPU programming easier. The Assembler is compatible with every RV32I CPU implementation.

For verification of the correctness and specification compliance of the CPU model a large number of instructions and streams of instructions was executed, covering possible edge cases. Results of the intructions execution were observed during, and after their execution and was made sure they followed the specification's descriptions. The assembler was checked by manually checking generated complex instruction and immediate data encodings and also comparing against other RISC-V compatible assembler implementations.

Περιεχομενα

Κατάλογος Διαγραμματων

Κατάλογος Πινάκων

Κεφαλαιο 1
ΕΙΣΑΓΩΓΗ

1.1 Περί Επεξεργαστών

Οι επεξεργαστής είναι ένα διαχρονικό κομμάτι της επιστήμης της πληροφορικής καθώς ενσαρκώνει την πρακτική υλοποίηση της θεωρίας υπολογισμού. Ξεκινώντας από τις θεωρητικές μηχανές Turing και την μηχανική Αναλυτική Μηχανή του Charles Babbage, μέχρι τα σημερινά χαοτικά μοντέλα τεχνητής νοημοσύνης και τα διασυνδεδεμένα συμπλέγματα υπερυπολογιστών, ο επεξεργαστής αποτελεί αναπόσπαστο μέρος κάθε υπολογιστικού συστήματος.

Η ανάγκη που οδήγησε στην δημιουργία των επεξεργαστών ξεκινάει ακόμα και πριν την επιστήμη της πληροφορικής. Οι μαθηματικές πράξεις εκτελούνταν από ανθρώπινο δυναμικό, απαιτώντας πολύωρη επαναληπτική εργασία και οδηγώντας πολλές φορές σε λάθη. Οι πρώτοι επεξεργαστές, πριν ακόμα οριστεί ο όρος, κατασκευάστηκαν για να εκτελούν επαναλαμβανόμενες μαθηματικές πράξεις και αλγόριθμους σε δεδομένα, παράγοντας ένα χρήσιμο μαθηματικό αποτέλεσμα [For14]. Ακόμα και σήμερα, η λειτουργία ενός επεξεργαστή μπορεί να περιγραφεί από την επεξεργασία δεδομένων εισόδου που παράγει ένα χρήσιμο αποτέλεσμα. Η γενική αυτή διαδικασία της επεξεργασίας δεδομένων χρησιμοποιείται πλέον ευρέως σε κάθε τομέα της τεχνολογίας, καθιστώντας τον επεξεργαστή ένα διαχρονικό στοιχείο της επιστήμης των υπολογιστών.

1.2 Αρχιτεκτονική Συνόλου Εντολών

Αρχιτεκτονική συνόλου εντολών (Instruction Set Architecture - ISA) ονομάζεται ο ορισμός όλων των εντολών προς εκτέλεση από έναν επεξεργαστή, καθώς και ο ορισμός των συνεπειών από την εκτέλεση κάθε μίας από αυτές τις εντολές. Πρόκειται δηλαδή για το μέρος του επεξεργαστή το οποίο εκτίθεται στο λογισμικό και προσφέρει έναν καλώς ορισμένο και τυποποιημένο τρόπο επικοινωνίας του υλικού με το λογισμικό [HP11]. Οι συνέπειες των εκτελέσεων των εντολών μπορούν να είναι αριθμητικά αποτελέσματα, αλλαγή της ροής του προγράμματος με βάση μια συνθήκη ή κατάσταση, αλλαγή της κατάστασης που βρίσκεται ο επεξεργαστής. Οι ορισμοί αυτών των εντολών συνοδεύονται από την δυαδική κωδικοποίησή τους. Θεωρούμε πως ένας επεξεργαστής είναι συμβατός με μία αρχιτεκτονική συνόλου εντολών, εφόσον εκτελεί την ίδια ακολουθία δυαδικών κωδικοποιήσεων εντολών έχοντας τις συνέπειες που αναφέρονται από τον ορισμό των εντολών στην αρχιτεκτονική.

Οι αρχιτεκτονικές συνόλου εντολών μπορούν να κατηγοριοποιηθούν σε δύο τύπους:

Αρχιτεκτονικές RISC Reduced Instruction Set Computer

- Περιορισμένος αριθμός εντολών που εκτελούν βασικές λειτουργίες γενικού σκοπού.
- Κάθε μία εντολή είναι ορισμένη έτσι ώστε να εκτελεί μία απλουστευμένη λειτουργία.
- Οι λειτουργίες επεξεργασίας δεδομένων και προσπέλασης της μνήμης δεδομένων διαχωρίζονται σε ξεχωριστές εντολές.

Αρχιτεκτονικές CISC Complex Instruction Set Computer

- Μεγάλος αριθμός εντολών που είναι στοχευμένες σε λειτουργίες ειδικού σκοπού.
- Υπάρχει πληθώρα εντολών όπου η καθεμία από αυτές εκτελεί μια πολύ συγκεκριμένη, εξειδικευμένη λειτουργία.
- Μία εντολή μπορεί να κάνει ταυτόχρονα προσπέλαση της μνήμης δεδομένων και επεξεργασία δεδομένων.

Ένα μεγάλο σύνολο εντολών με περίπλοκες λειτουργίες δυσκολεύει τόσο την ανάπτυξη συμβατών επεξεργαστών, αλλά και αποδοτικών compiler. Αρχιτεκτονικές CISC, για παράδειγμα η x86, μπορεί να έχουν εντολές οι οποίες προσπελαύνουν την μνήμη μέχρι και 50 φορές κατά την εκτέλεσή τους[REF]. Οι προδιαγραφές που ορίζει μια CISC αρχιτεκτονική καθιστούν ακόμα και μια θεωρητική υλοποίηση ενός συμβατού με αυτές επεξεργαστή, πολύ πολύπλοκη. Αυτό έχει οδηγήσει πολλούς εμπορικούς x86 επεξεργαστές να μετατρέπουν εσωτερικά την εντολή x86 σε μια ακολουθία απλούστερων εντολών παρόμοιων με αυτές των RISC αρχιτεκτονικών [HP11].

1.3 Η αρχιτεκτονική RISC-V

Η αρχιτεκτονική συνόλου εντολών RISC-V είναι μια RISC αρχιτεκτονική, η οποία σχεδιάστηκε στο Berkeley University of California, αρχικά από τους Andrew Waterman, Yunsup Lee, David A. Patterson, Krste Asanovic, για ακαδημαϊκούς και εκπαιδευτικούς σκοπούς [Wat+16]. Η σχεδιαστική της φιλοσοφία βασίζεται στην απλότητα, την παραμετροποίηση και την επεκτασιμότητα με βάση πολλά μικρά, ανεξάρτητα σύνολα εντολών. Η ύπαρξη επεκτάσεων σε ένα βασικό σύνολο εντολών και το ευρύ πεδίο εφαρμογών δεν αποτελεί την ειδοποιό διαφορά όσον αφορά την αρχιτεκτονική RISC-V σε σχέση με άλλες, αλλά πρόκειται για τον κύριο στόχο της αρχιτεκτονικής.

Το πλεονέκτημα της αρχιτεκτονικής RISC-V βρίσκεται στο γεγονός πως σχεδιάστηκε από την αρχή για τους παραπάνω σκοπούς. Η παραμετροποίηση και επέκτασή της δεν αποτελεί απόρροια δημιουργίας νέων αναγκών ή ανάγκη αναπροσαρμογής μιας αρχιτεκτονικής για λύση νέων προβλημάτων.

1.3.1 Υλοποιήσεις RISC-V

Ένας από τους βασικούς στόχους της αρχιτεκτονικής RISC-V είναι η αποφυγή περιορισμών με σκοπό να επιτρέψει πολλές διαφορετικές εφαρμογές τεχνικών σχεδίασης στις συμβατές υλοποιήσεις. Η απλότητα της επιλογής των εντολών που βρίσκονται στο σύνολο καθώς και οι κωδικοποιήσεις τους, εξασφαλίζουν ευκολία όσον αφορά τις σχεδιάσεις υλικού και εξομοιωτών για ακαδημαϊκούς σκοπούς. Ο διαχωρισμός της αρχιτεκτονικής RISC-V σε μικρότερα προαιρετικά υποσύνολα εντολών διευκολύνει περαιτέρω την ανάπτυξη σε εκπαιδευτικά περιβάλλοντα. Αρκετά πανεπιστήμια συμμετέχουν στο ίδρυμα RISC-V ως μέλη [Int].

Ακαδημαϊκές Υλοποιήσεις:

BOOM (Berkeley Out-of-Order Machine) University of California, Berkeley [CPA16]
Παραμετροποιήσιμος συνθέσιμος (synthesizable) πυρήνας RISC-V RV64GC, με υποστήριξη Linux. Κύριο χαρακτηριστικό της μικροαρχιτεκτονικής του είναι η out-of-order εκτέλεση εντολών.

Riscy Processors και RiscyOO MIT [CSAa], [CSAb]
Οικογένεια RISC-V επεξεργαστών του MIT με υποστήριξη Linux. Βασισμένη σε προηγούμενες εργασίες των εργαστηρίων αρχιτεκτονικής του πανεπιστημίου σε MIPS Pipelined και Out-of-order επεξεργαστές.

Lizard Cornell University [JG]
Παραμετροποιήσιμος συνθέσιμος RV64IM πυρήνας. Χρησιμοποιεί κατά κόρον out-of-order εκτέλεση εντολών και την τεχνική μετονομασίας Register. Παρέχει υποστήριξη εκτέλεσης για στατικά προγράμματα C.

Η παραμετροποίηση που προσφέρει το σύνολο εντολών RISC-V καθιστά την αρχιτεκτονική κατάλληλη και για εμπορική χρήση, καθώς η χρήση των υπάρχοντων και η δημιουργία custom συμπληρωματικών υποσυνόλων εντολών (επεκτάσεων), δίνει πρακτικά απεριόριστες δυνατότητες για την κάλυψη κάθε ανάγκης. Όπως προαναφέρθηκε, πολύπλοκες λειτουργίες που εκτελούνται από τους εμπορικά διαδεδομένους x86 επεξεργαστές ήδη μετατρέπονται σε ακολουθίες εντολών RISC, γεγονός που αποδεικνύει την ανταγωνιστικότητα που μπορεί να έχει ένας RISC-V επεξεργαστής στην αγορά. Κορυφαίες εταιρίες του τομέα του hardware όπως οι Google, NVidia, έχουν δείξει ενδιαφέρον για την αρχιτεκτονική συμμετέχοντας ενεργά στον οργανισμό RISC-V [Int]. Άξια αναφοράς αποτελεί η εταιρία SiFive, η οποία παρέχει παραμετροποιήσιμες υλοποιήσεις οικογενειών επεξεργαστών για χρήσεις γενικού σκοπού, νευρωνικών δικτύων και ενσωματωμένων συστημάτων [SiFa].

Εμπορικές Υλοποιήσεις:

SiFive P550 SiFive, Inc. [SiFb]
: Ο γρηγορότερος RISC-V επεξεργαστής γενικού σκοπού, με υποστήριξη Linux. Βασισμένος στην αρχιτεκτονική RV64GC με out-of-order pipeline 13 σταδίων και τριπλό instruction issuing.

SweRV Core EH1 Western Digital[Dig]
: Ένας RV32IMC επεξεργαστής για χρήση ως ελεγκτή στα αποθηκευτικά μέσα (σκληροί δίσκοι, κάρτες μνήμης) που παράγει η εταιρία. Dual-issue pipeline 9 σταδίων.

NEOX Graphics Think Silicon[Sil]
: Οικογένεια επεξεργαστών γραφικών (GPU) βασισμένη στην αρχιτεκτονική RV64GC. Υποστήριξη παραμετροποίησης αριθμών πυρήνων (4-64) και την οργάνωση αυτών, multithreading, μεγέθη και οργάνωση κρυφής μνήμης.

1.3.2 Σύνολο Εντολών RISC-V και Επεκτάσεις

Το κύριο σχεδιαστικό ρεύμα της αρχιτεκτονικής RISC-V είναι ο διαχωρισμός της σε περαιτέρω μικρά υποσύνολα εντολών [Wat+16]. Η παραπάνω σχεδιαστική νοοτροπία προσφέρει την δυνατότητα παραμετροποίησης των δυνατοτήτων του επεξεργαστή ανάλογα με τις ανάγκες της κάθε υλοποίησης. Η ίδια η αρχιτεκτονική ενθαρρύνει την δημιουργία περαιτέρω, custom επεκτάσεων για την ικανοποίηση οποιασδήποτε ανάγκης η οποία δεν καλύπτεται από τις ήδη υπάρχουσες επεκτάσεις. Η παραμετροποίηση επιτρέπει ελευθερία επιλογής μέχρι και στο μέγεθος του εντέλου του επεξεργαστή, δηλαδή το μήκος σε bit των αριθμών στους οποίους εφαρμόζονται οι αριθμητικές πράξεις, το οποίο αντίστοιχα ορίζει και το μήκος των διευθύνσεων της μνήμης.

Προς το παρόν ορίζεται ένα βασικό σύνολο εντολών, το οποίο θα πρέπει να εκτελείται από κάθε επεξεργαστή αρχιτεκτονικής RISC-V, και ορισμένα στανταρισμένα υποσύνολα επεκτάσεων, για λειτουργίες που μπορεί να είναι πιθανό να φανούν χρήσιμες από πολλές υλοποιήσεις.

RV32I/RV64I/RV128I Base
: Το βασικό υποσύνολο εντολών
Συμπεριλαμβάνει αριθμητικές και λογικές πράξεις ακεραίων, προσπελάσεις (εγγραφές και αναγνώσεις) μνήμης, εντολές διακλάδωσης και ορισμένες εντολές διαχείρισης των προσπελάσεων της μνήμης. Κάθε μήκος εντέλου ορίζει μια διαφορετική βάση επειδή ανάλογα με το μέγεθος του εντέλου τροποποιούνται ορισμένες εντολές προσπέλασης μνήμης και εκτέλεσης πράξεων.

Επέκταση M Multiplication
: Πολλαπλασιασμός - Διαίρεση ακεραίων
Οι πράξεις πολλαπλασιασμού και διαίρεσης έχει νόημα να οριστούν ως επέκταση καθώς απλουστεύουν την υλοποίηση μιας Αριθμητικής - Λογικής Μονάδας στις περιπτώσεις όπου δεν απαιτούνται.

Επέκταση A Atomics

Ατομικές προσπελάσεις μνήμης

Παρέχει εντολές οι οποίες χρησιμοποιούνται για την οργάνωση ατομικών προσπέλασεων κοινών μνημών μεταξύ των πυρήνων ενός RISC-V επεξεργαστή ή πολλών RISC-V επεξεργαστών.

Επέκταση F Floating Point Arithmetic

Αριθμητική κινητής υποδιαστολής

Η αριθμητική κινητής υποδιαστολής ορίζεται ως επέκταση για τους ίδιους λόγους με τις πράξεις πολλαπλασιασμού - διαίρεσης. Είναι συμβατή με την αριθμητική IEEE 754-2008.

Επέκταση D Double Precision Floating Point Arithmetic

Αριθμητική κινητής υποδιαστολής διπλής ακρίβειας

Συμπληρωματική της επέκτασης F. Προσφέρει ακριβέστερα αριθμητικά αποτελέσματα στις πράξεις αριθμών κινητής υποδιαστολής.

Επέκταση Q Quad Precision Floating Point Arithmetic

Αριθμητική κινητής υποδιαστολής τετραπλής ακρίβειας

Συμπληρωματική των επεκτάσεων F και D, προσφέρει τετραπλή ακρίβεια στις πράξεις κινητής υποδιαστολής.

Επέκταση Zicsr Control and Status Registers

Καταχωρητές ελέγχου και κατάστασης

Η επέκταση ορίζει ένα σύνολο καταχωρητών κατάστασης και λειτουργίες ατομικής προσπέλασης τους. Χρησιμοποιείται για την οργάνωση της επικοινωνίας μεταξύ πυρήνων ή επεξεργαστών, καθώς και για σκοπούς των επιπέδων του περιβάλλοντος εκτέλεσης (Machine - Hypervisor - Supervisor - User).

Επέκταση Zifencei Instruction Fencing

Συμπληρωματική επέκταση συνέπειας μνήμης

Προσθέτει εντολές για την οργάνωση των προσπελάσεων της μνήμης εντολών σε επίπεδο ενός πυρήνα RISC-V.

Επέκταση G General Purpose

Σύνολο επεκτάσεων γενικού σκοπού

Συμπεριλαμβάνει την βάση και τις επεκτάσεις M, A, F, D, Zicsr, Zifencei. Το σύνολο εντολών που προκύπτει με τον συνδυασμό αυτών, προορίζεται για χρήση σε επεξεργαστές γενικού σκοπού, γι αυτό και στο συγκεκριμένο υποσύνολο επεκτάσεων έχει οριστεί ξεχωριστή ονομασία.

Κεφαλαιο 2
ΤΟ ΒΑΣΙΚΟ ΣΥΝΟΛΟ ΕΝΤΟΛΩΝ RV32I - ΑΝΑΛΥΣΗ

Το σύνολο των εντολών που περιγράφει η βασική αρχιτεκτονική RISC-V (RV32I) μπορεί να χωριστεί περαιτέρω σε 4 κατηγορίες.

2.1 Εντολές Αριθμητικών - Λογικών Πράξεων

Πράξεις μεταξύ δύο καταχωρητών (Register - Register)

Σε αυτή την κατηγορία κατατάσσονται εντολές οι οποίες εκτελούν μια μαθηματική ή λογική πράξη μεταξύ των δεδομένων δύο καταχωρητών, και αποθηκεύουν το αποτέλεσμα σε έναν τρίτο καταχωρητή. Ο καταχωρητής αποθήκευσης του αποτελέσματος μπορεί να είναι οποιοσδήποτε, συμπεριλαμβανομένων των καταχωρητών ανάγνωσης των δεδομένων. Η κατηγορία περιλαμβάνει τις παρακάτω εντολές:

Εντολή	Αποτέλεσμα	Περιγραφή
add x3, x1, x2	x3 ← x1 + x2	Αριθμητική πρόσθεση των $(x1)*, $(x2). Αποθήκευση του αποτελέσματος στον x3.
sub x3, x1, x2	x3 ← x1 - x2	Αριθμητική αφαίρεση του $(x2) από το $(x1). Αποθήκευση του αποτελέσματος στον x3.
sll x3, x1, x2	x3 ← x1 << x2	Λογική ολίσθηση του $(x1) κατά X θέσεις αριστερά. Η τιμή X ορίζεται ως τα 5 λιγότερο σημαντικά bit της τιμής $(x2). Αποθήκευση του αποτελέσματος στον καταχωρητή x3.
slt x3, x1, x2	(x1 < x2) ? x3 ← 1 : 0	Το $(x3) παίρνει την τιμή 1 αν $(x1) < $(x2), διαφορετικά παίρνει την τιμή 0.
sltu x3, x1, x2	(x1 < x2) ? x3 ← 1 : 0	Ίδιο με το παραπάνω, αλλά η σύγκριση $(x1) < $(x2), γίνεται θεωρώντας πως οι αριθμοί δεν έχουν πρόσημο.
xor x3, x1, x2	x3 ← x1 ^ x2	Λογική πράξη XOR στα $(x1), $(x2) ανά bit (bitwise). Αποθήκευση του αποτελέσματος στον x3.
srl x3, x1, x2	x3 ← x1 >> x2	Λογική ολίσθηση του $(x1) κατά X θέσεις δεξιά. Η τιμή X ορίζεται ως τα 5 λιγότερο σημαντικά bit της τιμής $(x2). Αποθήκευση του αποτελέσματος στον καταχωρητή x3.
sra x3, x1, x2	x3 ← x1 >>> x2	Αριθμητική ολίσθηση του $(x1) κατά X θέσεις δεξιά. Η τιμή X ορίζεται ως τα 5 λιγότερο σημαντικά bit της τιμής $(x2). Αποθήκευση του αποτελέσματος στον καταχωρητή x3.
or x3, x1, x2	x3 ← x1 & x2	Λογική πράξη AND στα $(x1), $(x2) ανά bit (bitwise). Αποθήκευση του αποτελέσματος στον x3.
and x3, x1, x2	x3 ← x1 | x2	Λογική πράξη OR στα $(x1), $(x2) ανά bit (bitwise). Αποθήκευση του αποτελέσματος στον x3.

Πίνακας 1: Εντολές τύπου Register - Register

*Αναφέρεται πως ως "$(x1)" ορίζεται ως το περιεχόμενο του καταχωρητή x1, και ως "x1" ο ίδιος ο καταχωρητής. Παρομοίως για τους υπόλοιπους καταχωρητές.

Πράξεις μεταξύ ενός καταχωρητή - ενός άμεσου δεδομένου (Register - Immediate)

Αυτή η κατηγορία εμπεριέχει τις εντολές που εκτελούν μια πράξη μεταξύ ενός καταχωρητή και του άμεσου δεδομένου που βρίσκεται κωδικοποιημένο στο διάνυσμα της εντολής. Αποθηκεύουν το αποτέλεσμά τους σε έναν δεύτερο καταχωρητή, ο οποίος μπορεί να είναι ίδιος με τον πρώτο. Η κατηγορία περιλαμβάνει τις ίδιες εντολές με την κατηγορία Register - Register, με την διαφορά πως ο καταχωρητής x2 αντικαθίσταται με το άμεσο δεδομένο.

Εντολή	Αποτέλεσμα	Περιγραφή
addi x3, x1, imm	x3 ← x1 + imm	Αριθμητική πρόσθεση των $(x1)*, immediate. Αποθήκευση του αποτελέσματος στον x3.
slti x3, x1, imm	(x1 < imm) ? x3 ← 1 : 0	Το $(x3) παίρνει την τιμή 1 αν $(x1) < immediate, διαφορετικά παίρνει την τιμή 0.
sltiu x3, x1, imm	(x1 < imm) ? x3 ← 1 : 0	Ίδιο με το παραπάνω, αλλά η σύγκριση $(x1) < immediate, γίνεται θεωρώντας πως οι αριθμοί δεν έχουν πρόσημο.
xori x3, x1, imm	x3 ← x1 ^ imm	Λογική πράξη XOR στα $(x1), immediate ανά bit (bitwise). Αποθήκευση του αποτελέσματος στον x3.
ori x3, x1, imm	x3 ← x1 & imm	Λογική πράξη AND στα $(x1), immediate ανά bit (bitwise). Αποθήκευση του αποτελέσματος στον x3.
andi x3, x1, imm	x3 ← x1 | imm	Λογική πράξη OR στα $(x1), immediate ανά bit (bitwise). Αποθήκευση του αποτελέσματος στον x3.
slli x3, x1, imm	x3 ← x1 << imm	Λογική ολίσθηση του $(x1) κατά X θέσεις αριστερά. Η τιμή X ορίζεται ως τα 5 λιγότερο σημαντικά bit του immediate. Αποθήκευση του αποτελέσματος στον καταχωρητή x3.
srli x3, x1, imm	x3 ← x1 >> imm	Λογική ολίσθηση του $(x1) κατά X θέσεις δεξιά. Η τιμή X ορίζεται ως τα 5 λιγότερο σημαντικά bit του immediate. Αποθήκευση του αποτελέσματος στον καταχωρητή x3.
srai x3, x1, imm	x3 ← x1 >>> imm	Αριθμητική ολίσθηση του $(x1) κατά X θέσεις δεξιά. Η τιμή X ορίζεται ως τα 5 λιγότερο σημαντικά bit του immediate. Αποθήκευση του αποτελέσματος στον καταχωρητή x3.

Πίνακας 2: Εντολές τύπου Register - Immediate

2.2 Εντολές προσπέλασης μνήμης

Οι εντολές προσπέλασης μνήμης αφορούν οποιαδήποτε μεταφορά δεδομένων μεταξύ καταχωρητών και μνήμης δεδομένων. Οι εντολές σε αυτήν την κατηγορία είναι οι μοναδικές που προσπελαύνουν την μνήμη, αφού η RISC-V πρόκειται για μια load-store αρχιτεκτονική [Wat+16].

Εντολές ανάγνωσης της μνήμης (Load)
Αυτή η κατηγορία περιλαμβάνει εντολές για την ανάγνωση της μνήμης δεδομένων, δηλαδή την μεταφορά δεδομένων από την μνήμη δεδομένων στους καταχωρητές. Η μνήμη έχει οριστεί να είναι **διευθυνσιοδοτημένη ανά byte**. Οι εντολές είναι οι:

Εντολή	Αποτέλεσμα	Περιγραφή
lb x2, imm(x1)	x2 ← $(x1 + imm) (Byte)	Ανάγνωση του περιεχομένου στην θέση μνήμης (x1 + imm), μεγέθους 1 byte και αποθήκευσή του στον x2. Στο περιεχόμενο εφαρμόζεται επέκταση προσήμου πριν αποθηκευτεί στον x2.
lh x2, imm(x1)	x2 ← $(x1 + imm) (Halfword)	Ανάγνωση του περιεχομένου στις θέσεις μνήμης (x1 + imm) και (x1 + imm + 1)*, μεγέθους 2 byte και αποθήκευσή του στον x2. Στο περιεχόμενο εφαρμόζεται επέκταση προσήμου πριν αποθηκευτεί στον x2.
lw x2, imm(x1)	x2 ← $(x1 + imm) (Word)	Ανάγνωση του περιεχομένου στις θέσεις μνήμης (x1 + imm), (x1 + imm + 1), (x1 + imm + 2), (x1 + imm + 3), μεγέθους 4 byte και αποθήκευσή του στον x2.
lbu x2, imm(x1)	x2 ← $(x1 + imm) (Byte Unsigned)	Ίδια με την εντολή lb, αλλά δεν εφαρμόζεται επέκταση προσήμου.
lhu x2, imm(x1)	x2 ← $(x1 + imm) (Halfword Unsigned)	Ίδια με την εντολή lh, αλλά δεν εφαρμόζεται επέκταση προσήμου.

Πίνακας 3: Εντολές τύπου Load

*Για την συγκεκριμένη **Litte Endian** υλοποίηση, μια λέξη έχει οριστεί να έχει τα πιο σημαντικά byte στις θέσεις μνήμης με μεγαλύτερη διεύθυνση. Για μια Big Endian υλοποίηση, θα πρέπει να τροποποιηθούν οι εντολές ανάγνωσης Halfword και Byte στο επίπεδο του Hardware έτσι ώστε να προσπελαύνονται οι αντίστοιχες θέσεις.

Εντολές εγγραφής στην μνήμη (Store)

Αυτή η κατηγορία εντολών περιέχει εντολές για εγγραφή στην μνήμη δεδομένων, δηλαδή την μεταφορά δεδομένων από τους καταχωρητές στην μνήμη δεδομένων.

Οι εντολές αυτού του τύπου είναι οι:

Εντολή	Αποτέλεσμα	Περιγραφή
sb x2, imm(x1)	$(x1 + imm) ← $(x2) (Byte)	Εγγραφή του λιγότερου σημαντικού byte του $(x2) στην θέση μνήμης (x1 + imm).
sh x2, imm(x1)	$(x1 + imm) ← $(x2) (Halfword)	Εγγραφή του λιγότερου σημαντικού byte του $(x2) στην θέση μνήμης (x1 + imm), και του αμέσως επόμενου λιγότερου σημαντικού στην θέση (x1 + imm + 1)*.
sw x2, imm(x1)	$(x1 + imm) ← $(x2) (Word)	Εγγραφή των byte του $(x2) στις θέσεις μνήμης (x1 + imm), (x1 + imm + 1), (x1 + imm + 2), (x1 + imm + 3), με αντίστοιχη σειρά.

Πίνακας 4: Εντολές τύπου Store

2.3 Εντολές φόρτωσης ειδικού δεδομένου

Ειδική κατηγορία εντολών που χρησιμοποιούν την κωδικοποίηση U - type. Σε αυτή την κατηγορία κατατάσσονται οι εντολές lui και auipc. Ακολουθούν οι εξηγήσεις τους.

Εντολή	Αποτέλεσμα	Περιγραφή
lui x1, upper-imm†	x1 ← upper-imm	Αποθήκευση του upper-imm στον καταχωρητή x1.
auipc x1, upper-imm	x1 ← upper-imm + PC	Πρόσθεση της τιμής του μετρητή προγράμματος με το upper-imm και αποθήκευση του αποτελέσματος στον καταχωρητή x1.

Πίνακας 5: Εντολές τύπου Register - Immediate

*Ισχύουν οι ίδιες παρατηρήσεις περί Endianness όπως και στις εντολές τύπου Load.

†Τα 20 πιο σημαντικά ψηφία του άμεσου δεδομένου imm.

2.4 Εντολές διακλάδωσης

Σε αυτή την κατηγορία κατατάσσονται εντολές οι οποίες τροποποιούν το περιεχόμενο του μετρητή προγράμματος. Με την αλλαγή της τιμής του μετρητή προγράμματος αλλάζει επίσης η επόμενη εντολή προς εκτέλεση.

Εντολές διακλάδωσης υπό συνθήκη (Conditional Branches)
Οι εντολές διακλάδωσης υπό συνθήκη ελέγχουν πρώτα την σχέση μεταξύ των περιεχομένων δύο καταχωρητών και τροποποιούν τον μετρητή προγράμματος εφόσον η σχέση τηρεί μια συνθήκη. Οι εντολές είναι οι ακόλουθες:

Εντολή	Αποτέλεσμα	Περιγραφή
beq x1, x2, label*	$(x1) == $(x2) ? PC[†]← label	Αν $(x1) ίσο με $(x2), αλλαγή του μετρητή προγράμματος στην θέση που ορίζει το label.
bne x1, x2, label	$(x1) != $(x2) ? PC ← label	Αν $(x1) διάφορο του $(x2), αλλαγή του μετρητή προγράμματος στην θέση που ορίζει το label.
blt x1, x2, label	$(x1) < $(x2) ? PC ← label	Αν $(x1) μικρότερο του $(x2), αλλαγή του μετρητή προγράμματος στην θέση που ορίζει το label.
bge x1, x2, label	$(x1) >= $(x2) ? PC ← label	Αν $(x1) μεγαλύτερο ή ίσο του $(x2), αλλαγή του μετρητή προγράμματος στην θέση που ορίζει το label.
bltu x1, x2, label	$(x1) < $(x2) ? PC ← label	Αντίστοιχο της blt, με την διαφορά πως η σύγκριση των $(x1), $(x2) γίνεται θεωρώντας πως οι αριθμοί δεν έχουν πρόσημο.
bgeu x1, x2, label	$(x1) >= $(x2) ? PC ← label	Αντίστοιχο της bge, με την διαφορά πως η σύγκριση των $(x1), $(x2) γίνεται θεωρώντας πως οι αριθμοί δεν έχουν πρόσημο.

Πίνακας 6: Εντολές τύπου Branch

[†]Η έννοια και η λειτουργία του label εξηγείται σε βάθος στο μέρος του Assembler.
[†]Program Counter - Μετρητής Προγράμματος.

Εντολές άλματος (Unconditional Jumps)

Οι εντολές άλματος τροποποιούν τον μετρητή προγράμματος ανεξαρτήτως συνθήκης. Αποθηκεύουν την τιμή του μετρητή προγράμματος πριν την τροποποίησή του. Οι εντολές άλματος είναι:

Εντολή	Αποτέλεσμα	Περιγραφή
jal x1, label	x1 ← PC + 4, PC ← label	Αποθήκευση της διεύθυνσης της επόμενης εντολής στον x1, και θέση της τιμής του μετρητή προγράμματος στην τιμή του label.
jalr x1, label(x2)	x1 ← PC + 4, PC ← label + $(x2)	Αποθήκευση της διεύθυνσης της επόμενης εντολής στον x1, και ανάθεση της τιμής του μετρητή προγράμματος στην τιμή (label + $(x2)).

Πίνακας 7: Εντολές τύπου Jump

2.5 Εντολές Διαχείρισης Κλήσεων Λογισμικού

Εκ των οποίων υπάρχουν:

ecall[*] Environment Call

Αλλαγή ροής του εκτελούμενου προγράμματος για κλήση ειδικής ρουτίνας του λειτουργικού συστήματος ή του περιβάλλοντος εκτέλεσης. Μπορεί να χαρακτηριστεί και σαν ειδική μορφή διακοπής (interrupt).

ebreak[†] Environment Break

Αλλαγή ροής του εκτελούμενου προγράμματος για κλήση του αποσφαλματωτή (debugger).

2.6 Εντολή Διαχείρισης Προσπελάσεων Μνήμης

Συμπεριλαμβάνει την παρακάτω εντολή:

fence Barrier προσπέλασης Μνήμης

Σε περίπτωση πολυπύρηνων ή πολυεπεξεργαστικών μοντέλων υπάρχει η ανάγκη οργάνωσης της σειράς των αναγνώσεων και εγγραφών της κοινής μνήμης μεταξύ αυτών. Από τις προδιαγραφές, η εκτέλεση της εντολής fence ορίζει πως οι προσπελάσεις μνήμης από τον συγκεκριμένο πυρήνα / επεξεργαστή μέχρι εκείνο το σημείο θα γίνουν ορατές και στους υπόλοιπους πυρήνες / επεξεργαστές ή συνδεδεμένες δευτερεύουσες συσκευές. Δεν υπάρχει ανάγκη της εκτέλεσης αυτής της εντολής από το συγκεκριμένο μοντέλο, καθώς υλοποιεί έναν μόνο πυρήνα / επεξεργαστή και in-order εκτέλεση εντολών.

[*]Στην παρούσα υλοποίηση δεν υπάρχει ανάγκη εκτέλεσης αυτών των εντολών, συνεπώς ο επεξεργαστής τις εκτελεί ως NOP.

[†]Πρακτικά, ecall με διαφορετική διεύθυνση.

2.7 Δυαδικές Κωδικοποιήσεις

Η αρχιτεκτονική RISC-V ορίζει 6 κατηγορίες για τις δυαδικές κωδικοποιήσεις των εντολών [Wat+16]. Κάθε κατηγορία έχει διαφορετικό αριθμό πεδίων που προορίζονται για διαφορετικές χρήσεις. Ωστόσο, η θέση των κοινών πεδίων μεταξύ των εντολών παραμένει ίδια, κάνοντας την παραγωγή των διανυσμάτων απλούστερη.

Τόσο στον επίσημο ορισμό των προδιαγραφών της αρχιτεκτονικής RISC-V, όσο και στην παρούσα εργασία, χρησιμοποιούνται οι παρακάτω ονομασίες πεδίων των εντολών της αρχιτεκτονικής RISC-V.

Ονομασία	Σημασία	Χρήση	Μέγεθος	Θέση
opcode*	Operation Code	Κωδικός της εντολής προς εκτέλεση	7 bit	Bits 7 - 0
rs1	Register Select 1	Διεύθυνση καταχωρητή προς ανάγνωση - θύρα 1	5 bit	Bits 19 -15
rs2	Register Select 2	Διεύθυνση καταχωρητή προς ανάγνωση - θύρα 2	5 bit	Bits 24 -20
rd	Destination Register	Διεύθυνση καταχωρητή για αποθήκευση του αποτελέσματος	5 bit	Bits 11 - 7
funct3	Function (3 bits)	Παραμετροποίηση λειτουργίας της εντολής προς εκτέλεση	3 bit	Bits 14 - 12
funct7	Function (7 bits)	Περαιτέρω παραμετροποίηση λειτουργίας της εντολής προς εκτέλεση	7 bit	Bits 31 - 25
imm	Immediate Data	Άμεσο δεδομένο - Δεδομένο ενσωματωμένο στο διάνυσμα εντολής	Ανάλογα με τον τύπο κωδικοποίησης	
shamt	Shift Amount	Αριθμός θέσεων ολίσθησης - για εντολές ολίσθησης με άμεσο δεδομένο	5 bit	Bits 24 - 20

Πίνακας 8: Τα πεδία εντολών της αρχιτεκτονικής RISC-V

*Το πεδίο opode υπάρχει σε όλες τις εντολές, ανεξαρτήτως τύπου κωδικοποίησης. Όλα τα άλλα πεδία εμφανίζονται μόνο σε συγκεκριμένες κωδικοποιήσεις.

Στο βασικό υποσύνολο εντολών εμφανίζονται οι παρακάτω τύποι κωδικοποιήσεων εντολών. Κάθε τύπος αξιοποιεί διαφορετικά πεδία και ο καθένας προορίζεται για διαφορετικό τύπο εντολών. Πιο συγκεκριμένα:

R - type Αξιοποιεί δύο πεδία καταχωρητών προς ανάγνωση rs1 και rs2, το πεδίο καταχωρητή προς εγγραφή rd, και τα δύο πεδία παραμετροποίησης λειτουργίας funct3 και funct7. Χρησιμοποιείται από τις εντολές πράξεων μεταξύ δύο καταχωρητών (Register - Register).

I - type Αξιοποιεί ένα πεδίο καταχωρητή προς ανάγνωση rs1, το πεδίο καταχωρητή προς εγγραφή rd, ένα πεδίο παραμετροποίησης λειτουργίας funct3 και ένα πεδίο άμεσου δεδομένου imm. Χρησιμοποιείται από τις εντολές πράξεων μεταξύ ενός καταχωρητή κι ενός άμεσου δεδομένου (Register - Immediate) καθώς και από τις εντολές ανάγνωσης (Load).

S - type Αξιοποιεί δύο πεδία καταχωρητών προς ανάγνωση rs1 και rs2, ένα πεδίο παραμετροποίησης λειτουργίας funct3 και δύο πεδία για την κωδικοποίηση του άμεσου δεδομένου imm*. Χρησιμοποιείται από τις εντολές εγγραφής στην μνήμη δεδομένων (Store).

B - type Αξιοποιεί δύο πεδία καταχωρητών προς ανάγνωση rs1 και rs2, ένα πεδίο παραμετροποίησης λειτουργίας funct3 και δύο πεδία για την κωδικοποίηση του άμεσου δεδομένου imm. Χρησιμοποιείται από τις εντολές διακλάδωσης υπό συνθήκη (Branch).

U - type Αξιοποιεί το πεδίο καταχωρητή προς εγγραφή rd, και ένα πεδίο κωδικοποίησης του άμεσου δεδομένου imm. Χρησιμοποιείται από τις ειδικές εντολές φόρτωσης lui και auipc.

J - type Αξιοποιεί το πεδίο καταχωρητή προς εγγραφή rd, και ένα πεδίο κωδικοποίησης του άμεσου δεδομένου imm. Αν και τα πεδία έχουν την ίδια θέση και μέγεθος με τον τύπο I - type, η κωδικοποίηση του άμεσου δεδομένου διαφέρει μεταξύ τους. Χρησιμοποιείται από τις εντολές άλματος (Unconditional Jumps).

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R - type
imm[11:0]		rs1	funct3	rd	opcode	I - type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S - type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B - type
imm[31:12]				rd	opcode	U - type
imm[20|10:1|11|19:12]				rd	opcode	J - type

Πίνακας 9: Τύποι κωδικοποιήσεων εντολών της αρχιτεκτονικής RISC-V (Από [Wat+16])

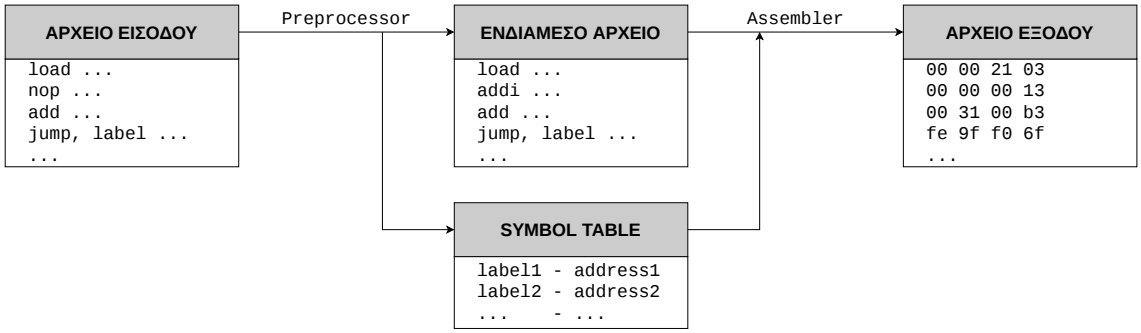
*Το άμεσο δεδομένο καταλαμβάνει τα σημεία του διανύσματος της εντολής που δεν καταλαμβάνονται από άλλα πεδία. Σε αυτή την περίπτωση, το άμεσο δεδομένο διασπάται σε 2 μέρη.

Κεφαλαιο 3
ASSEMBLER - ΣΧΕΔΙΑΣΤΙΚΕΣ ΑΠΟΦΑΣΕΙΣ

Η ανάπτυξη ενός RISC-V Assembler βοήθησε πολύ στην κατανόηση των κωδικοποίησεων των πεδίων των εντολών της αρχιτεκτονικής RISC-V, και συνεπώς η γνώση που αποκτήθηκε συντέλεσε σε σχεδιαστικές αποφάσεις για την υλοποίηση του μοντέλου του επεξεργαστή.

Πιο συγκεκριμένα, ο Assembler αποτέλεσε την βάση κατανόησης των σταθερών πεδίων κωδικοποιήσεων που πρόκειται για μια από τις βασικές σχεδιαστικές τεχνικές της RISC-V αρχιτεκτονικής. Ο Assembler επίσης χρησιμοποιήθηκε και κατά την ανάπτυξη του μοντέλου του επεξεργαστή για την παραγωγή εκτελέσιμων προγραμμάτων με σκοπό την αποσφαλμάτωση τους. Αποτέλεσε σημαντικό εργαλείο για την αντιμετώπιση προβλημάτων της εκτέλεσης όλων των τύπων εντολών, και ειδικότερα για τις εντολές διακλάδωσης και της επίλυσης των προβλημάτων της διασωλήνωσης. Αν και τα προβλήματα της διασωλήνωσης επιλύθηκαν στο επίπεδο του υλικού, ο Assembler επίτρεψε τον γρήγορο έλεγχο των πιθανών ακολουθιών εντολών που μπορεί να προκαλέσουν προβλήματα κατά την εκτέλεσή τους στο μοντέλο διασωλήνωσης που αναπτύχθηκε.

Η λειτουργία του Assembler είναι να μετατρέπει ένα αρχείο Assembly RISC-V σε Bytecode προς εκτέλεση από τον επεξεργαστή. Το αρχείο εξόδου που παράγει ο Assembler πρόκειται για ένα αρχείο μνήμης που περιέχει τις δεκαεξαδικές κωδικοποιήσεις των εντολών στο αρχείο εισόδου. Ο Assembler συνοδεύεται από έναν προ-επεξεργαστή (Preprocessor), ο οποίος παράγει ένα αρχείο Assembly με απλούστερη μορφή το οποίο προορίζεται για περαιτέρω επεξεργασία από τον Assembler. Ο Preprocessor χρησιμοποιείται για την αποκωδικοποίηση ψευδοεντολών σε αντίστοιχες απλές, τον υπολογισμό offset μνήμης και συμπληρώνει τον Assembler ως προς την υποστήριξη της χρήσης Labels στις εντολές διακλάδωσης. Μαζί με το αρχείο δίνεται στον Assembler πληροφορία για τις θέσεις μνήμης των Label, για την παραγωγή των σωστών Offset για τις εντολές διακλάδωσης.



Διάγραμμα 1: Διαδικασία μετατροπής αρχείου Assembly σε εκτελέσιμο αρχείο - αρχείο με την δυαδική κωδικοποίηση των εντολών.

3.1 Μορφές Εντολών Γλώσσας Μηχανής

Στον Preprocessor ορίζονται οι μορφές των εντολών που μπορεί να συναντηθούν στο αρχείο εισόδου. Πιο συγκεκριμένα:

- Instruction Register, Immediate(Register)
- Instruction Register, Register, Register
- Instruction Register, Register, Immediate
- Instruction Register, Register, Label
- Instruction Register, Immediate
- Instruction Register, Label

Για παράδειγμα, αντίστοιχα για κάθε μορφή εντολής:

- sw s0, 0(s1)
- add s0, ra, sp
- add s1, sp, 1
- beq s0, zero, overflow_handler
- lui t1, 0xff
- jal ra, overflow_handler

Στον Preprocessor επίσης ορίζεται η μορφή των δηλώσεων Label ως εξής:

- Label:

Για παράδειγμα:

- overflow_handler:

3.1.1 Ονοματολογία Καταχωρητών

Οι προδιαγραφές ορίζουν συγκεκριμένα ονόματα για κάθε καταχωρητή, καθώς και προτεινόμενες χρήσεις για τον καθένα από αυτούς*. Ο Assembler παρέχει υποστήριξη για την χρήση οποιουδήποτε από τα παρακάτω ονόματα για αναφορά σε κάποιον συγκεκριμένο καταχωρητή.

Καταχωρητής	Ονομασίες
x0	zero
x1	ra
x2	sp
x3	gp
x4	tp
x5	t0
x6-7	t1-2
x8	s0 / fp
x9	s1
x10-11	a0-1
x12-17	a2-7
x18-27	s2-11
x28-31	t3-6

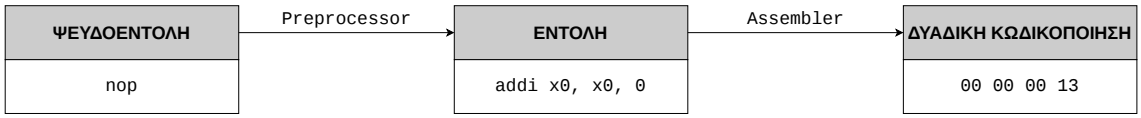
Πίνακας 10: Ονομασίες καταχωρητών (Από [Wat+16]).

Ο καταχωρητής x0 / zero είναι ορισμένος από την αρχιτεκτονική RISC-V να επιστρέφει σταθερά την τιμή 0 στην ανάγνωσή του, και το περιεχόμενό του να παραμένει αμετάβλητο στις εγγραφές. Συνεπώς, οι εντολές που χρησιμοποιούν αυτόν τον καταχωρητή για αποθήκευση του αποτελέσματός τους δεν αλλάζουν την κατάσταση του επεξεργαστή. Τέτοιου τύπου εντολές ορίζονται από την αρχιτεκτονική RISC-V ως προαιρετικές εντολές HINT για την μέτρηση της απόδοσης του επεξεργαστή, αφού δεν επιφέρουν αλλαγές στην κατάστασή του. Θεωρούμε πως η κατάσταση του επεξεργαστή ορίζεται από τα περιεχόμενα των μνημών δεδομένων και τα περιεχόμενα των καταχωρητών. Αφού οι εντολές με καταχωρητή αποτελέσματος x0 δεν μεταβάλλουν την τιμή σε κάποιον καταχωρητή ή μνήμη, δεν αλλάζουν και την κατάσταση του επεξεργαστή [Wat+16].

*Από τις επίσημες προδιαγραφές προτείνονται συγκεκριμένες χρήσεις για κάθε καταχωρητή, για σκοπούς μεταφραστών και compiler υψηλότερου επιπέδου γλωσσών προγραμματισμού. Η στανταρισμένη χρήση καταχωρητών θα μπορούσε να αξιοποιηθεί από πολυπλοκότερα μοντέλα. Στην παρούσα διπλωματική εργασία δεν ασχολούμαστε με την μετάφραση γλωσσών υψηλότερου επιπέδου.

3.2 Ψευδοεντολές

Ο Preprocessor μπορεί να παραμετροποιηθεί για την υποστήριξη ψευδοεντολών. Αναλαμβάνει την μετατροπή της ψευδοεντολής σε μια εντολή - ή μια ακολουθία εντολών - κατάλληλη για περαιτέρω μετατροπή της στην δυαδική κωδικοποίησή της από τον Assembler. Μία τέτοια ψευδοεντολή που έχει υλοποιηθεί είναι η εντολή "nop" (No Operation), που είναι ορισμένη ως την εντολή "addi x0, x0, 0" (add immediate - x0 ← x0 + 0) με βάση τις προδιαγραφές της αρχιτεκτονικής RISC-V. Ο Preprocessor, εντοπίζοντας την ψευδοεντολή "nop", καταγράφει στο ενδιάμεσο αρχείο την αποκωδικοποίηση της ψευδοεντολής σε μια απλή μορφή εντολής Assembly, δηλαδή "addi x0, x0, 0", που μπορεί να επεξεργαστεί ο Assembler. Για ακολουθίες εντολών που παράγονται από μία μόνο ψευδοεντολή είναι απαραίτητο να εφαρμοστεί διόρθωση στην τιμή του μετρητή θέσεων μνήμης, καθώς ο Preprocessor αυξάνει τον μετρητή κατά 4 για κάθε εντολή ή ψευδοεντολή που υπάρχει στο αρχείο εισόδου. Είναι σημαντικό να αναφερθεί πως τα Labels δεν αποτελούν εντολές προς εκτέλεση, οπότε δεν μεταβάλλουν τον μετρητή θέσεων μνήμης.



Διάγραμμα 2: Διαδικασία αποκωδικοποίησης ψευδοεντολής και μετατροπή της σε δυαδική κωδικοποίηση.

3.3 Ο Ρόλος του Assembler στις Εντολές Διακλάδωσης - Υλοποίηση των Label

Στις προδιαγραφές των RISC-V επεξεργαστών (πέραν της επέκτασης C - Compressed, στην οποία κάθε εντολή μπορεί να έχει διαφορετικό μήκος) κάθε εντολή έχει μήκος 32 bit (4 byte), ανεξάρτητα από το μέγεθος του εντέλου του επεξεργαστή. Η διευθυνσιοδότηση της μνήμης εντολών γίνεται ανά byte, συνεπώς κάθε μία εντολή καταλαμβάνει 4 θέσεις μνήμης. Ο Preprocessor αναγιγνώσκει το περιεχόμενο του αρχείου εισόδου χωρίζοντάς το σε "λέξεις". Εφόσον ένα σύνολο λέξεων ακολουθεί την γραμματική και την δομή μίας από τις μορφές των εντολών, εγγράφεται η αντίστοιχη μερικώς επεξεργασμένη εντολή στο παραγόμενο από τον Preprocessor αρχείο από τον Assembler. Ο Preprocessor χρησιμοποιεί έναν μετρητή θέσεων μνήμης για κάθε εντολή ο οποίος ξεκινάει από το 0 και για κάθε εντολή που επεξεργάζεται, αυξάνεται κατά 4.

Σε περίπτωση που βρεθεί ο ορισμός ενός Label, καταγράφεται το όνομά του και η θέση μνήμης του, δηλαδή η διεύθυνση μνήμης της εντολής στην οποία αναφέρεται. Ο Preprocessor δηλαδή, κατασκευάζει μια απλουστευμένη μορφή ενός symbol table, με τα ονόματα και τις διευθύνσεις μνήμης των ονοματισμένων σημείων των υπο-ρουτινών στο αρχείο εισόδου. Είναι απαραίτητο το ίδιο το Label να μη προσμετρηθεί στην καταμέτρηση των θέσεων μνήμης καθώς δεν αποτελεί εντολή προς τον επεξεργαστή και δεν καταλαμβάνει χώρο στην μνήμη εντολών. Τα labels αξιοποιούνται για να διευκολυνθεί ο προγραμματισμός του επεξεργαστή. Χρησιμοποιούνται για τον "ονοματισμό" θέσεων του κώδικα και αυτοματοποιούν τον υπολογισμό των θέσεων μνήμης για τις εντολές διακλάδωσης. Αφού ολοκληρωθεί η προ-επεξεργασία του αρχικού αρχείου εισόδου, έχει δημιουργηθεί το ενδιάμεσο αρχείο που θα χρησιμοποιηθεί ως είσοδος του Assembler.

Πέραν του ενδιάμεσου αρχείου, ο Assembler αξιοποιεί το symbol table που δημιουργεί ο Preprocessor για την αποκωδικοποίηση των εντολών που χρημιοποιούν Label. Οι εντολές διακλάδωσης της αρχιτεκτονικής RISC-V χρησιμοποιούν σχετική διευθυνσιοδότηση για να δίνουν την δυνατότητα στα προγράμματα να βρίσκονται σε οποιαδήποτε θέση μνήμης. Η διεύθυνση μνήμης στην οποία πρόκεται να γίνει η διακλάδωση υπολογίζεται με βάση την τρέχουσα διεύθυνση μνήμης που βρίσκεται στον μετρητή προγράμματος. Ο Assembler, έχοντας την πληροφορία που παρέχεται από το symbol table, παράγει το σωστό Offset το οποίο

θα προστεθεί στην διεύθυνση μνήμης του μετρητή προγράμματος, έτσι ώστε να αποκτήσει την διεύθυνση της επόμενης εντολής που υποδεικνύεται από την εντολή διακλάδωσης. Το offset πρόκειται για την διαφορά της διεύθυνσης μνήμης της τρέχουσας εντολής από την διεύθυνση μνήμης της επιθυμητής εντολής της οποίας πρόκειται να εκτελεστεί εφόσον η συνθήκη διακλάδωσης αληθεύει. Με την τεχνική του σχετικού υπολογισμού της θέσης μνήμης διακλάδωσης εξασφαλίζεται με απλό τρόπο πως το πρόγραμμα που αποκωδικοποιεί ο Preprocessor και ο Assembler θα εκτελεστεί σωστά, ανεξαρτήτως της θέσης που βρίσκεται στην μνήμη προγράμματος. Αυτό συμβαίνει λόγω του γεγονότος πως οι διευθύνσεις των διακλαδώσεων υπολογίζονται σε σχέση με την διεύθυνση της τρέχουσας εντολής.

Όνομα Label	**Θέση μνήμης**
__start	0x8000
overflow_handler	0x8040
stack_pop	0x8100
...	...

Πίνακας 11: Ένα απλό παράδειγμα του symbol table.

3.4 Παραγωγή Διανυσμάτων Εντολών

Τα διανύσματα εντολών που παράγονται από τον Assembler αξιοποιούν τον τύπο δεδομένων **unsigned int** μήκους **32 bit**, και δημιουργούνται με ολισθήσεις και προσθέσεις. Η αποθήκευση των παραγόμενων διανυσμάτων εντολών στο αρχείο εξόδου μπορεί να γίνει είτε σε δυαδική μορφή για την παραγωγή δυαδικών εκτελέσιμων αρχείων με μια μικρή παραμετροποίηση του Assembler, είτε σε μορφή ASCII που χρησιμοποιείται στο τρέχον μοντέλο, για την φόρτωσή τους στην μνήμη εντολών.

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R - type
0000000	00010	00001	000	00011	0110011	add x3, x1, x2

Αρχικά, δεν έχει οριστεί κάποια τιμή στο διάνυσμα της εντολής.

xxxxxxx	xxxxx	xxxxx	xxx	xxxxx	xxxxxxx	Αρχικά

Θέτουμε την τιμή του opcode στο διάνυσμα της εντολής.

xxxxxxx	xxxxx	xxxxx	xxx	xxxxx	**0110011**	1 - opcode

Προσθέτουμε την τιμή του επόμενου πεδίου ($\text{rd} = 00011_{\text{b}}$) **αριστερά ολισθημένη κατά το μέγεθος του προηγούμενου πεδίου** στο αποτέλεσμα του βήματος 1. Στην συγκεκριμένη περίπτωση το πεδίο opcode έχει μέγεθος 7 bit συνεπώς θα γίνει ολίσθηση στην τιμή rd κατά 7 ψηφία αριστερά. Αυτή η διαδικασία ολίσθησης και πρόσθεσης μεταβάλλει μόνο το επόμενο πεδίο του διανύσματος διατηρώντας τα δεξιότερα ψηφία αμετάβλητα.

xxxxxxx	xxxxx	xxxxx	xxx	**00011**	0110011	2 - rd

Επαναλαμβάνουμε την παραπάνω διαδικασία ολίσθησης και πρόσθεσης για τα υπόλοιπα πεδία της εντολής.

xxxxxxx	xxxxx	xxxxx	**000**	00011	0110011	3 - funct3
xxxxxxx	xxxxx	**00001**	000	00011	0110011	4 - rs1
xxxxxxx	**00010**	00001	000	00011	0110011	5 - rs2
0000000	00010	00001	000	00011	0110011	6 - funct7

Πίνακας 12: Διαδικασία αποκωδικοποίησης και παραγωγής του διανύσματος της εντολής add x3, x1, x2

3.5 Περαιτέρω Παραμετροποίηση

Αξιοποιώντας την υπάρχουσα τεχνική ολίσθησης - πρόσθεσης δίνεται η δυνατότητα να προστεθεί στον Assembler υποστήριξη για οποιαδήποτε μορφή εντολής στο επίπεδο της δυαδικής κωδικοποίησης. Με τροποποίηση του Preprocessor μπορούν επίσης να υλοποιηθούν ευκολότερα μνημονικά και μορφές εντολών της γλώσσας Assembly για την διευκόλυνση του προγραμματισμού του επεξεργαστή.

Ένα παράδειγμα είναι η χρήση απλών μαθηματικών τύπων για αντικατάσταση των μνημονικών των εντολών αριθμητικών πράξεων. Μια εντολή με πιο ασυνήθιστη γραμματική μορφή "x3 = x1 + x2" θα μπορούσε να μετατρέπεται από τον Preprocessor στην ήδη υποστηριζόμενη από τον Assembler μορφή "add x3, x1, x2".

Κεφαλαιο 4
ΕΠΕΞΕΡΓΑΣΤΗΣ - ΣΧΕΔΙΑΣΤΙΚΕΣ ΑΠΟΦΑΣΕΙΣ

Σε αυτό το κεφάλαιο παρουσιάζεται η συλλογιστική πορεία που ακολουθήθηκε στην σχεδίαση του μοντέλου του επεξεργαστή. Οι αποφάσεις αιτιολογούνται με βάση την ανάλυση των εντολών της βασικής αρχιτεκτονικής συνόλου εντολών RV32I, καθώς και με την ακολούθηση ορισμένων σχεδιαστικών τάσεων και τεχνικών από άλλες υλοποιήσεις RISC επεξεργαστών και επεξεργαστών γενικότερα.

4.1 Η τεχνική της διασωλήνωσης

Οι περισσότεροι RISC επεξεργαστές αποφεύγουν την χρήση μικροκώδικα και αξιοποιούν την τεχνική της διασωλήνωσης στην μικροαρχιτεκτονική τους. Η τεχνική της διασωλήνωσης χωρίζει τις επιμέρους υπολειτουργίες μιας εντολής σε στάδια, τα οποία εκτελούνται το ένα μετά το άλλο. Η εκτέλεση ενός σταδίου είναι ανεξάρτητη από τα άλλα, και έτσι επιτυγχάνεται ένας βαθμός παραλληλοποίησης σε επίπεδο εντολής [HP11].

Τα πέντε στάδια της κλασσικής διασωλήνωσης RISC είναι τα εξής [Nik17]:

IF - Instruction Fetch Ανάκτηση εντολής
: Σε αυτο το στάδιο γίνεται ανάγνωση της εντολής προς εκτέλεση. Ο μετρητής προγράμματος επίσης αυξάνεται ανάλογα, έτσι ώστε να διευθυνσιοδοτεί την επόμενη εντολή προς εκτέλεση και να αναγνωστεί στον επόμενο κύκλο.

ID - Instruction Decode Αποκωδικοποίηση εντολής
: Μετά την ανάγνωση της εντολής, γίνεται ανάγνωση του περιεχομένου των καταχωρητών τους οποίους ορίζει η εντολή. Τα δεδομένα ανάγνωσης θα προχωρήσουν προς το επόμενο στάδιο για περαιτέρω επεξεργασία.

EX - Execute Εκτέλεση
: Στο στάδιο της εκτέλεσης εφαρμόζεται η πράξη που έχει οριστεί από την εντολή στα δεδομένα από τους καταχωρητές που έχουν αναγνωστεί. Το αποτέλεσμα μπορεί να είναι ένα αριθμητικό δεδομένο ή ένας υπολογισμός διεύθυνσης για την προσπέλαση της μνήμης δεδομένων. Στην περίπτωση που εκτελείται πράξη με αριθμητικό αποτέλεσμα, θα χρειαστεί να αποθηκευτεί σε έναν καταχωρητή σε επόμενο στάδιο της διασωλήνωσης (στάδιο WB). Στην περίπτωση υπολογισμού διεύθυνσης, το αποτέλεσμα του υπολογισμού προχωράει στο στάδιο MEM.

MEM - Memory Access Προσπέλαση Μνήμης
: Σε αυτό το στάδιο γίνεται η προσπέλαση της μνήμης δεδομένων, αφότου έχει υπολογιστεί η διεύθυνση προσπέλασης στο στάδιο EX. Σε περίπτωση που εκτελείται εντολή ανάγνωσης, το δεδομένο αποθηκεύεται σε έναν καταχωρητή αποτελέσματος για να εγγραφεί στο αρχείο καταχωρητών κατά το επόμενο στάδιο εκτέλεσης. Σε περίπτωση εντολής εγγραφής στην μνήμη, το δεδομένο προς εγγραφή είναι προετοιμασμένο από τους προηγούμενους κύκλους.

WB - Write Back* Επανεγγραφή
: Στο τελευταίο στάδιο της διασωλήνωσης γίνεται η εγγραφή του αριθμητικού αποτελέσματος της πράξης που ορίζει η εντολή (από το στάδιο EX) ή το δεδομένο από την ανάγνωση της μνήμης (στο στάδιο MEM) στους καταχωρητές.

*Στην συγκεκριμένη υλοποίηση, το στάδιο MEM συγχωνεύεται με το στάδιο WB. Αξιοποιώντας την προκαθορισμένη συμπεριφορά του εξομοιωτή Icarus Verilog παρατηρήθηκε πως η ανάγνωση της μνήμης και η εγγραφή του αποτελέσματος στον καταχωρητή μπορούν να εκτελεστούν ταυτόχρονα.

Καθώς οι λειτουργίες της διασωλήνωσης είναι μερικώς επικαλυπτόμενες, ο σχεδιασμός του μοντέλου του επεξεργαστή οφείλει να αντιμετωπίζει τυχόντα προβλήματα που δημιουργούνται από αυτή την επικάλυψη λειτουργιών. Ειδικότερα στην περίπτωση που θέλουμε ο επεξεργαστής να είναι απόλυτα συμβατός με την αρχιτεκτονική RV32I, η οποία δεν ορίζει καμία ακολουθία εντολών εκτός προδιαγραφών, τα προβλήματα που δημιουργούνται από την διασωλήνωση θα πρέπει να αντιμετωπίζονται σε κάθε περίπτωση στο επίπεδο του υλικού.

4.2 Επιμέρους Λειτουργικές Μονάδες

Θεωρείται πως όλες οι μονάδες υλοποιούνται ως ακολουθιακά κυκλώματα και λειτουργούν με σήμα ρολογιού, εκτός αν αναφέρεται διαφορετικά. Για κάθε λειτουργική μονάδα έχουμε θεωρήσει πως χρειάζεται έναν κύκλο ρολογιού για να εκτελέσει την λειτουργία της, και πως είναι θετικά ακμοπυροδότητη. Η θύρα εισόδου του σήματος ρολογιού παραλείπεται από τις περιγραφές.

4.2.1 Μνήμη

Αρχικά, χρειαζόμαστε μια μονάδα για την ανάγνωση των εντολών προς εκτέλεση. Με αυτήν την μονάδα, ο επεξεργαστής αλληλεπιδρά μόνο μέσω του μετρητή προγράμματος ο οποίος την διευθυνσιοδοτεί έτσι ώστε να αναγνωστεί η εντολή προς εκτέλεση. Χρειαζόμαστε επίσης μια μονάδα μνήμης για την ανάγνωση και την εγγραφή δεδομένων κατά την εκτέλεση του προγράμματος.

Παρατηρώντας το σύνολο εντολών που αφορούν την μνήμη (κατηγορία εντολών Load - Store), συμπεραίνουμε πως δεν υπάρχει καμία εντολή η οποία χρειάζεται να κάνει εγγραφή και ανάγνωση της μνήμης δεδομένων ταυτόχρονα. Συνεπώς, μπορούμε να χρησιμοποιήσουμε μια θύρα δεδομένων, η οποία εκτελεί μία λειτουργία ανάγνωσης ή εγγραφής σε κάθε κύκλο. Για τον καθορισμό της λειτουργίας της μνήμης χρησιμοποιείται ένα σήμα εγγραφής ή ανάγνωσης, ανάλογα με τις ανάγκες της εντολής. Η προσπέλαση της μνήμης προϋποθέτει επίσης τον ορισμό της διεύθυνσης προς προσπέλαση, άρα και μια θύρα διεύθυνσης.

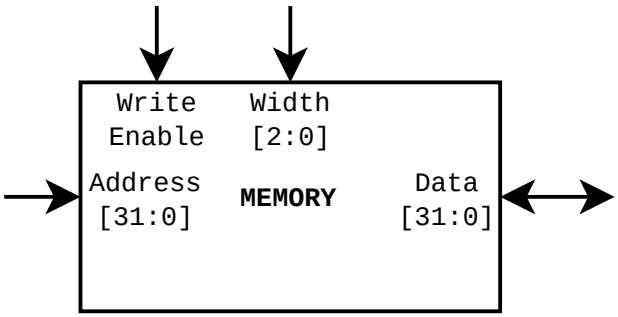
Υπάρχουν πολλές εντολές εγγραφής (sb, sh, sw), με ειδοποιό διαφορά το μέγεθος του δεδομένου προς εγγραφή. Η διαφορά αυτή είναι κβαντισμένη ανά byte όσο αναφορά το μέγεθος της εγγραφής και κωδικοποιείται στο πεδίο funct3 του διανύσματος της εντολής. Αυτό οδηγεί αρχικά στην απόφαση διευθυνσιοδότησης της μνήμης ανά byte, και επίσης στον ορισμό μιας θύρας που καθορίζει το μέγεθος της εγγραφής.

Με τον ίδιο τρόπο ορίζονται από τις προδιαγραφές και οι εντολές ανάγνωσης της μνήμης (lb, lh, lw), με την προσθήκη όμως μιας απλής επέκτασης προσήμου πριν την εγγραφή του αναγιγνωσμένου δεδομένου στον ορισμένο καταχωρητή. Υπάρχουν και οι αντίστοιχες εντολές οι οποίες δεν κάνουν επέκταση προσήμου (lbu, lhu). Αφού και αυτές οι λειτουργίες κωδικοποιούνται στο πεδίο funct3 της εντολής, η μνήμη αναλαμβάνει επίσης και την επέκταση προσήμου στις λειτουργίες ανάγνωσης που επιβάλλεται, με βάση την τιμή αυτής της θύρας.

Ο τρέχων σχεδιασμός της μνήμης δεδομένων υπερκαλύπτει τις ανάγκες της μνήμης εντολών, άρα χρησιμοποιούμε την ίδια λειτουργική μονάδα και για τις δύο μνήμες. Η μνήμη εντολών είναι ορισμένη να εκτελεί μόνο λειτουργίες ανάγνωσης μεγέθους 32 bit*, δηλαδή την εντολή lw, μέσω κατάλληλα ορισμένων τιμών στις θύρες της.

*Το σταθερό μέγεθος των 32 bit είναι ορισμένο από τις προδιαγραφές ως το μέγεθος των διανυσμάτων των εντολών της αρχιτεκτονικής RV32I.

Καταλήγουμε στον παρακάτω σχεδιασμό:



Διάγραμμα 3: Η μονάδα μνήμης

Ο επεξεργαστής διαθέτει δύο μνήμες, Μνήμη Εντολών και Μνήμη Δεδομένων, βασιζόμενες στην ίδια λειτουργική μονάδα. Η μονάδα μνήμης διευθυνσιοδοτείται ανά byte και μπορεί να αποθηκεύσει και να αναγνώσει λέξεις μεταβλητού μεγέθους 1, 2, 4 και 8 byte σε έναν κύκλο. Η μονάδα μπορεί να εφαρμόσει επέκταση προσήμου στην θύρα εξόδου των δεδομένων, εφόσον το μέγεθος της ανάγνωσης δεν καλύπτει το μέγιστο μέγεθος των 4 ή 8 byte (Ίσο με το μέγεθος του εντέλου του επεξεργαστή).

Διαθέτει τις εξής θύρες εισόδου:

Address Μεταβλητό μέγεθος, ανάλογο με το μέγεθος της μνήμης.
Χρησιμοποιείται για την επιλογή της διεύθυνσης της λέξης.

Width Σήμα μεγέθους λειτουργίας μεγέθους 3 bit.
Χρησιμοποιείται για την επιλογή του μεγέθους της ανάγνωσης ή της εγγραφής. Η λειτουργία (εγγραφή ή ανάγνωση) γίνεται στην δοθείσα διεύθυνση [Address] και στις αμέσως επόμενες ανάλογα με την τιμή του μεγέθους που δίνεται. Εφαρμόζεται επίσης επέκταση προσήμου ανάλογα με την τιμή του [Width].

- Εάν το πιο σημαντικό bit έχει την τιμή 0, εφαρμόζεται επέκταση προσήμου στην λέξη προς ανάγνωση.
- Διαφορετικά, οι κενές θέσεις τιμοδοτούνται με 0.

Δεν εφαρμόζεται κάποια επέκταση προσήμου στις λειτουργίες εγγραφής.

Write Enable Σήμα λειτουργίας ανάγνωσης/εγγραφής ενός bit.
Χρησιμοποιείται για την επιλογή της λειτουργίας της μνήμης. Οι ενέργειες αφορούν το περιεχόμενο στην διεύθυνση μνήμης που δίνεται στην θύρα [Address].

- Εάν το σήμα έχει τιμή 0, γίνεται ανάγνωση του περιεχομένου της θέσης μνήμης [Address], καθώς και στις αμέσως επόμενες ανάλογα με την τιμή της θύρας [Width].
- Εάν το σήμα έχει τιμή 1, γίνεται εγγραφή του περιεχομένου της θύρας [Data] στην θέση μνήμης [Address], καθώς και στις αμέσως επόμενες ανάλογα με την τιμή της θύρας [Width].

Διαθέτει την εξής θύρα εισόδου/εξόδου:

Data Θύρα εισόδου/εξόδου δεδομένου μεταβλητού μεγέθους.
Η λειτουργικότητα της θύρας καθορίζεται από το σήμα [Write Enable]. Το μέγεθος της λέξης ανάγνωσης ή της λέξης προς εγγραφή καθορίζεται από την τιμή στην θύρα [Width].

- Εάν το σήμα [Write Enable] έχει τιμή 0, λειτουργεί ως θύρα εξόδου και παίρνει την τιμή του περιεχομένου της θέσης μνήμης [Address].
- Εάν το σήμα [Write Enable] έχει τιμή 1, λειτουργεί ως θύρα εισόδου και γίνεται εγγραφή του περιεχομένου της στην θέση μνήμης [Address].

Σημειώνεται πως δεν υπάρχει δυνατότητα ταυτόχρονης ανάγνωσης και εγγραφής.

4.2.2 Αριθμητική - Λογική Μονάδα

Σε κάθε περίπτωση των εντολών εκτέλεσης αριθμητικών - λογικών πράξεων, υπάρχουν μόνο δύο δεδομένα και μια εκτέλεση πράξης. Αυτό ορίζει δύο θύρες για τα δεδομένα που θα εκτελεστεί η πράξη και μια θύρα αποτελέσματος της πράξης. Η εκτέλεση της πράξης ορίζεται από το πεδίο funct3 και προαιρετικά το πεδίο funct7 όπου χρειάζεται, άρα η μονάδα θα χρειαστεί επίσης μια θύρα για τον ορισμό της πράξης προς εκτέλεση.

Η διαφορά των πράξεων Register - Register και Register - Immediate καθορίζεται από το πεδίο opcode, ωστόσο δεν επηρεάζει την λειτουργία της Αριθμητικής - Λογικής μονάδας καθώς εξακολουθεί να εφαρμόζει μια πράξη σε 2 δεδομένα.

Υπάρχουν περιπτώσεις όπου απαιτείται υπολογισμός πράξης σε εντολές που δεν ανήκουν στην κατηγορία αριθμητικών - λογικών πράξεων:

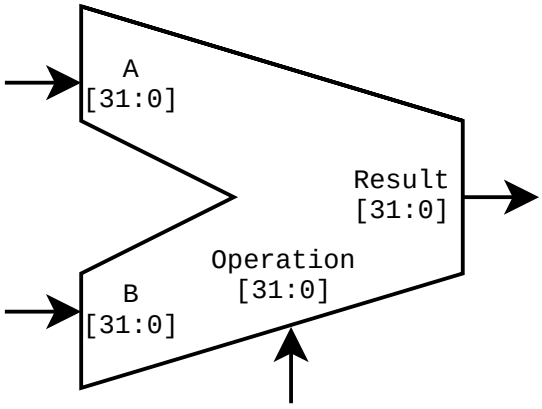
- Σε κάθε περίπτωση διευθυνσιοδότησης της μνήμης, δηλαδή στις εντολές **Load, Store**, η αριθμητική - λογική μονάδα αξιοποιείται για τον υπολογισμό της διεύθυνσης κάνοντας την πρόσθεση του άμεσου δεδομένου με το δεδομένο που αναγιγνώσθηκε από τον καταχωρητή που ορίζει η εντολή.

- Στην εντολή **jal** απαιτείται επίσης μια πρόσθεση του άμεσου δεδομένου label που βρίσκεται κωδικοποιημένο στο διάνυσμα της εντολής με το δεδομένου του καταχωρητή που ορίζεται από την εντολή.

- Στην εντολή **auipc** απαιτείται η πρόσθεση του περιεχομένου του μετρητή προγράμματος στο δεδομένο ενός καταχωρητή που ορίζει η εντολή.

Σε κάθε περίπτωση δηλαδή που δεν εφαρμόζεται μια ορισμένη από την εντολή αριθμητική - λογική πράξη, μπορούμε να αποφασίσουμε πως η αριθμητική - λογική μονάδα θα εκτελεί την πράξη της πρόσθεσης στα δύο δεδομένα που της δίνονται.

Συνεπώς, η σχεδίαση της αριθμητικής - λογικής μονάδας είναι:



Διάγραμμα 4: Η αριθμητική - λογική μονάδα

Η μονάδα χρησιμοποιείται για αριθμητικές και λογικές πράξεις μεταξύ δύο έντελων. Υποστηρίζει αριθμητικές πράξεις ακεραίων (integer operations), λογικές πράξεις ανά bit (bitwise logic operations) συμπεριλαμβανομένων αριθμητικών και λογικών ολισθήσεων (arithmetic - logical shift).

Διαθέτει τις εξής θύρες εισόδου:

Θύρα A και Θύρα B Θύρα εισόδου εντέλου σταθερού μεγέθους, ανάλογο με το μέγεθος του εντέλου του επεξεργαστή (32 ή 64 bit).
Σε αυτές τις θύρες δίνονται οι τιμές των εντέλων για την εκτέλεση της πράξης.

Operation Θύρα εισόδου πράξης μεγέθους 32 bit.
Ανάλογα με την τιμή της θύρας, εφαρμόζεται μια συγκεκριμένη πράξη μεταξύ των εντέλων A και B. Σε περίπτωση που δεν έχει οριστεί συγκεκριμένη πράξη, εκτελείται η πράξη της πρόσθεσης.

Διαθέτει την εξής θύρα εξόδου:

Result Θύρα εξόδου αποτελέσματος σταθερού μεγέθους, ανάλογο με το μέγεθος του εντέλου του επεξεργαστή (32 ή 64 bit).
Αυτή η θύρα τιμοδοτείται με το αποτέλεσμα της πράξης [A] [Operation] [B], στον επόμενο κύκλο ρολογιού.

4.2.3 Αρχείο Καταχωρητών

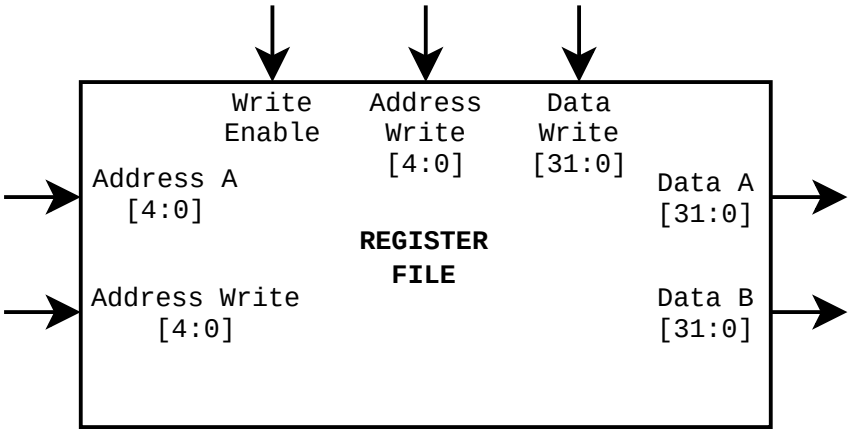
Σχεδόν σε όλες τις κωδικοποιήσεις εντολών υπάρχει το πεδίο rs1, το οποίο ορίζει έναν καταχωρητή προς ανάγνωση, και το πεδίο rd που ορίζει τον καταχωρητή προς εγγραφή του αποτελέσματος. Σε αρκετές κωδικοποιήσεις επίσης υπάρχει το πεδίο rs2, το οποίο ορίζει έναν δεύτερο καταχωρητή προς ανάγνωση. Με βάση τα παραπάνω, ορίζουμε στο αρχείο καταχωρητών 3 θύρες διευθύνσεων, 2 για ανάγνωση και 1 για εγγραφή. Αντίστοιχα ορίζονται 2 θύρες δεδομένων ανάγνωσης των αντίστοιχων καταχωρητών και 1 θύρα δεδομένων εγγραφής σε καταχωρητή. Τα μεγέθη των θυρών διευθύνσεων ορίζονται στα 5 bit, ίδιο μέγεθος με τα πεδία rs1, rs2, rd, αφού οι προδιαγραφές ορίζουν 32 καταχωρητές. Ορίζουμε επίσης μια θύρα ενεργοποίησης εγγραφής, καθώς δεν απαιτείται εγγραφή του αποτελέσματος σε καταχωρητή σε κάθε εντολή.

Η διασωλήνωση επιβάλλει ταυτόχρονη εγγραφή και ανάγνωση του αρχείου καταχωρητών, για την αποφυγή δομικών κινδύνων. Οι δομικοί κίνδυνοι προκύπτουν όταν μία μονάδα αξιοποιείται σε παραπάνω από ένα στάδιο της διασωλήνωσης [Nik17]. Στην συγκεκριμένη

περίπτωση, τα στάδια όπου αξιοποιείται ταυτόχρονα κάποια μονάδα, είναι τα στάδια MEM/WB και ID.

Στην συγκεκριμένη υλοποίηση αξιοποιείται η προκαθορισμένη συμπεριφορά του εξομοιωτή Icarus Verilog για ταυτόχρονη ανάγνωση και εγγραφή στον ίδιο καταχωρητή, η οποία επιτρέπει την ταυτόχρονη ανάγνωση και εγγραφή στον ίδιο κύκλο, με προτεραιότητα στο δεδομένο προς εγγραφή. Η συμπεριφορά αυτή λύνει το δομικό πρόβλημα που παρουσιάζεται, ενώ ταυτόχρονα προσομοιώνει πραγματικές υλοποιήσεις επεξεργαστών [HP11].

Τα παραπάνω οδηγούν σε αυτήν την σχεδίαση αρχείου καταχωρητών:



Διάγραμμα 5: Το αρχείο καταχωρητών

Το αρχείο καταχωρητών πρόκειται για ένα σύνολο 32 διευθυνσιοδοτημένων καταχωρητών, όπως είναι ορισμένο στις προδιαγραφές [Wat+16]. Δίνεται η δυνατότητα ανάγνωσης του καταχωρητή στον ίδιο κύκλο εγγραφής του.

Διαθέτει τις εξής θύρες εισόδου:

Address A και Address B Θύρες εισόδου διευθύνσεων καταχωρητών προς ανάγνωση, μεγέθους 5 bit.

Address Write Θύρα εισόδου διεύθυνσης καταχωρητή προς εγγραφή, μεγέθους 5 bit.

Data Write Θύρα εισόδου δεδομένων προς εγγραφή στον καταχωρητή με διεύθυνση [Address].
Το μέγεθος της θύρας καθορίζεται ανάλογα με το μέγεθος του εντέλου του επεξεργαστή (32 ή 64 bit).

Write Enable Σήμα λειτουργίας εγγραφής ενός bit.
Χρησιμοποιείται για την επιλογή της ενέργειας στον καταχωρητή με διεύθυνση [Address Write].

- Εάν το σήμα έχει τιμή 1, γίνεται εγγραφή του περιεχομένου της θύρας [Data Write] στον καταχωρητή με διεύθυνση [Address Write].
- Διαφορετικά, η τιμή του καταχωρητή παραμένει ως έχει.

Διαθέτει τις εξής θύρες εξόδου:

Data A και Data B Θύρες εξόδου δεδομένων των καταχωρητών, μεγέθους αντίστοιχο με αυτό του εντέλου του επεξεργαστή (32 ή 64 bit).
Αυτές οι θύρες τιμοδοτούνται στον επόμενο κύκλο ρολογιού με τα περιεχόμενα των καταχωρητών στις διευθύνσεις [Address A] και [Address B] αντίστοιχα.

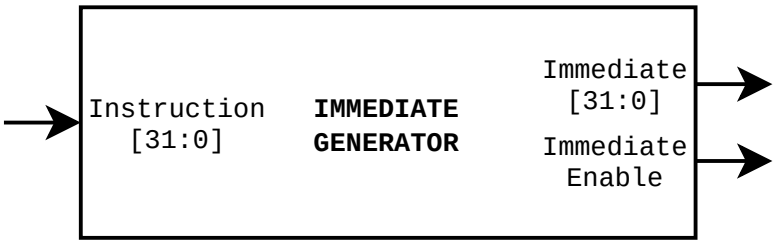
4.2.4 Μονάδα Διαχείρισης Άμεσων Δεδομένων

Η μονάδα αναλαμβάνει την αποκωδικοποίηση των δεδομένων που βρίσκονται ενσωματωμένα στα διανύσματα ορισμένων τύπων κωδικοποίησης εντολών, όπως επίσης και την απαραίτητη επέκταση προσήμου αυτών. Ορίζονται 5 τύποι εντολών (I, S, U, B, J type) οι οποίοι κωδικοποιούν το δεδομένο σε διαφορετικά σημεία της εντολής με διαφορετικό τρόπο. Ανάλογα με τον τύπο της εντολής η μονάδα μετατρέπει το κωδικοποιημένο ενσωματωμένο δεδομένο στην κατάλληλη μορφή προς περαιτέρω επεξεργασία του. Οι παραπάνω προτάσεις ορίζουν μια θύρα εισόδου του διανύσματος εντολής και μια θύρα αποτελέσματος της αποκωδικοποίησης. Χρειάζεται επίσης η χρήση ενός σήματος ενεργοποίησης άμεσου δεδομένου καθώς το άμεσο δεδομένο δεν είναι απαραίτητο για την εκτέλεση κάθε τύπου εντολής.

Σχετικά με την θέση της μονάδας στην διαδικασία της διασωλήνωσης:

- Το αποτέλεσμα που παράγει πρέπει να οδηγηθεί στην εκτέλεση της πράξης, συνεπώς πρέπει να βρίσκεται **πριν το στάδιο EX**.
- Αυτονόητα η μονάδα πρέπει να τοποθετηθεί σε στάδιο **μετά το IF** καθώς επιδρά στο διάνυσμα της εντολής η οποία θα αναγνωστεί ως συνέπεια του σταδίου IF.
- Η μονάδα **τοποθετείται στο στάδιο ID** καθώς αντικαθιστά την ανάγνωση ενός καταχωρητή με την ανάγνωση του άμεσου δεδομένου.

Καταλήγουμε στην παρακάτω σχεδίαση της μονάδας διαχείρισης άμεσων δεδομένων:



Διάγραμμα 6: Η μονάδα διαχείρισης άμεσων δεδομένων

Ανάλογα με τον τύπο της εντολής, η μονάδα μετατρέπει το κωδικοποιημένο ενσωματωμένο δεδομένο σε μήκος 32 ή 64 bit, ίσο με το μέγεθος του εντέλου του επεξεργαστή, διατηρώντας το πρόσημό του.

Διαθέτει την εξής θύρα εισόδου:

Instruction Θύρα εισόδου διανύσματος εντολής, μεγέθους 32 bit.
Σε αυτό το διάνυσμα βρίσκεται το κωδικοποιημένο άμεσο δεδομένο προς μετατροπή.

Διαθέτει τις εξής θύρες εξόδου:

Immediate Θύρες εξόδου αποτελέσματος της αποκωδικοποίησης και επέκτασης προσήμου, μεγέθους αντίστοιχο με αυτό του εντέλου του επεξεργαστή (32 ή 64 bit).

Immediate Enable Θύρες εξόδου σήματος ενεργοποίησης της λειτουργίας άμεσου δεδομένου, μεγέθους ενός bit.
Ενεργοποιείται ανάλογα με τον τύπο της εντολής.

- Εάν το σήμα έχει τιμή 1, το αποτέλεσμα [Immediate] οδηγείται στην θύρα εισόδου [B] της αριθμητικής - λογικής μονάδας.
- Διαφορετικά, η θύρα εισόδου [B] της αριθμητικής - λογικής μονάδας οδηγείται από το αποτέλεσμα ανάγνωσης του καταχωρητή rs2.

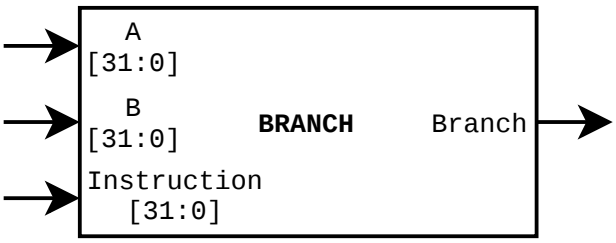
4.2.5 Μονάδα Διαχείρισης Διακλαδώσεων

Η μονάδα διακλαδώσεων αναλαμβάνει τον υπολογισμό των συνθηκών με βάση τις τιμές των ορισμένων από την εντολή καταχωρητών, προκειμένου να αποφασίσει εάν θα εκτελεστεί ή όχι η επιθυμητή διακλάδωση. Η παραπάνω λειτουργία δημιουργεί την ανάγκη για την ύπαρξη μιας θύρας αποτελέσματος της συνθήκης και δύο θύρες για την σύγκριση των δεδομένων. Εφόσον δεν εκτελούν όλες οι εντολές κάποια διακλάδωση, υπάρχει και μια θύρα εισόδου της εντολής που ενεργοποιούν αντίστοιχα τις λειτουργίες σύγκρισης. Άξιο αναφοράς είναι το γεγονός πως αυτή η μονάδα δεν υπολογίζει την διεύθυνση της διακλάδωσης*, παρά μόνο αποφασίζει εάν θα πραγματοποιηθεί ή όχι.

Σχετικά με την θέση της μονάδας στην διαδικασία της διασωλήνωσης:

- Η μονάδα λειτουργεί με δεδομένα που αναγιγνώσκονται από τους καταχωρητές, άρα θα πρέπει να τοποθετηθεί **πριν το στάδιο ID**.
- Η λειτουργία της μονάδας είναι όμοια με την αριθμητική - λογική μονάδα επειδή και αυτή εκτελεί μια πράξη - στην συγκεκριμένη περίπτωση μια σύγκριση - σε δύο έντελα. Αυτό σημαίνει ότι μπορεί να τοποθετηθεί παράλληλα με την αριθμητική - λογική μονάδα **στο στάδιο EX**.

Από τα παραπάνω, καταλήγουμε στην σχεδίαση:



Διάγραμμα 7: Η μονάδα διαχείρισης διακλαδώσεων

Η μονάδα διαχείρισης διακλαδώσεων εκτελεί την σύγκριση των δύο εντέλων που ορίζονται από την εντολή διακλάδωσης και έχει ως έξοδο ένα σήμα που υποδεικνύει εάν θα εκτελεστεί η διακλάδωση.

*Ο υπολογισμός της διεύθυνσης γίνεται από την αριθμητική - λογική μονάδα με την εκτέλεση της κατάλληλης πρόσθεσης, πράγμα που έχει περιγραφεί παραπάνω.

Διαθέτει τις εξής θύρες εισόδου:

Θύρα A και Θύρα B Θύρα εισόδου εντέλου σταθερού μεγέθους, ανάλογο με το μέγεθος του εντέλου του επεξεργαστή (32 ή 64 bit).
Σε αυτές τις θύρες δίνονται οι τιμές των εντέλων για την εκτέλεση της σύγκρισης.

Instruction Θύρα εισόδου διανύσματος εντολής, μεγέθους 32 bit.
Με βάση την τιμή του opcode του διανύσματος της εντολής αποφασίζεται η λειτουργία της σύγκρισης και γενικότερα η λειτουργία της μονάδας.

Διαθέτει τις εξής θύρες εξόδου:

Branch Θύρα εξόδου σήματος διακλάδωσης, μεγέθους 1 bit.

- Εφόσον το σήμα έχει την τιμή 1, θα πρέπει να αλλάξει τιμή ο μετρητής προγράμματος, συνεπώς να γίνει εκτέλεση της διακλάδωσης.
- Στην τιμή 0, η μέθοδος της αλλαγής της τιμής του μετρητή προγράμματος δεν μεταβάλλεται.

4.2.6 Μονάδα Διαχείρισης Κινδύνων

Σκοπός αυτής της μονάδας είναι να ανιχνεύσει εντολές οι οποίες δεν θα έχουν το επιθυμητό αριθμητικό αποτέλεσμα εάν εκτελεστούν με μια συγκεκριμένη ακολουθία. Η διαφορά στο αποτέλεσμα της εκτέλεσης συγκεκριμένων ακολουθιών εντολών οφείλεται στην τεχνική της διασωλήνωσης που εφαρμόζεται για την σχεδίαση του συγκεκριμένου μοντέλου επεξεργαστή.

Η τεχνική της διασωλήνωσης δημιουργεί κινδύνους δεδομένων σε περιπτώσεις ακολουθιών εντολών οι οποίες:

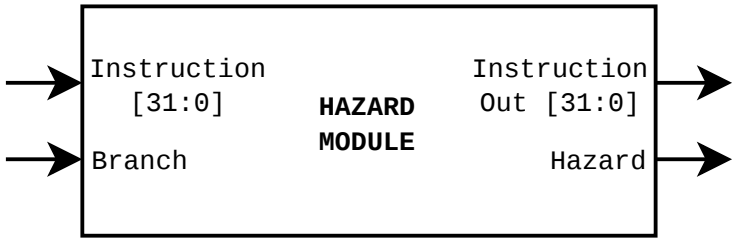
1. Αρχικά, εγγράφουν το αποτέλεσμα μιας πράξης σε έναν καταχωρητή.
2. Στην συνέχεια, αναγιγνώσκουν τον καταχωρητή με το αποτέλεσμα.

Οι κίνδυνοι παρουσιάζονται όταν η εντολή της ανάγνωσης φτάσει στο στάδιο ID της διασωλήνωσης, όπου γίνενται ανάγνωση των καταχωρητών. Στη σχεδίαση με την τεχνική της διασωλήνωσης ωστόσο, η εντολή όπου αποθηκεύει ένα αποτέλεσμα σε καταχωρητή μπορεί να βρίσκεται σε στάδιο προηγούμενου του WB, έχοντας ως συνέπεια να μην έχει εκτελεστεί ακόμα η λειτουργία της εγγραφής του αποτελέσματος στους καταχωρητές. Συνεπώς, η εντολή η οποία θα αναγνώσει το περιεχόμενο του καταχωρητή, θα συνεχίσει την εκτέλεσή της με την προηγούμενη τιμή, και όχι με το αποτέλεσμα της προηγούμενης εντολής όπως θα ήταν αναμενόμενο από την ακολουθιακή εκτέλεση των δύο αυτών εντολών.

Σχετικά με την θέση της μονάδας στην διαδικασία της διασωλήνωσης:

- Αξιοποιεί το διάνυσμα της εντολής προς εκτέλεση, συνεπώς θα πρέπει να τοποθετηθεί **μετά το στάδιο IF**.
- Η μονάδα ελέγχει ποιες εντολές θα προχωρήσουν με την λειτουργία ανάγνωσης καταχωρητών, άρα θα πρέπει να βρίσκεται **προτού το στάδιο ID**.
- Εφόσον δεν ορίζεται κάποιο στάδιο πριν το IF και μετά το ID, θα πρέπει να οριστεί ένα **νέο στάδιο ανάμεσα στο IF και στο ID**, το οποίο θα ονομάζουμε **στάδιο HAZ** (Hazard).

Καταλήγουμε στην παρακάτω σχεδίαση της μονάδας διαχείρισης κινδύνων:



Διάγραμμα 8: Η μονάδα διαχείρισης κινδύνων

Η μονάδα αυτή ανιχνεύει τους καταχωρητές που πρόκειται να χρησιμοποιήσει κάθε εντολή και με βάση αυτούς παύει την εκτέλεση της επόμενης εντολής, εφόσον προκαλεί κίνδυνο δεδομένου. Μια τέτοια μονάδα θα μπορούσε να παραλειφθεί, αφήνοντας την διαχείριση τέτοιων προβλημάτων στον Assembler ή στον προγραμματιστή*.

Διαθέτει τις εξής θύρες εισόδου:

Instruction Θύρα εισόδου διανύσματος εντολής, μεγέθους 32 bit.

Branch Θύρα εισόδου σήματος διακλάδωσης του ενός bit.
Σε περίπτωση που πρόκειται να γίνει διακλάδωση, το διάνυσμα στην έξοδο εντολής της μονάδας γίνεται 0, για να αποφευχθεί η εκτέλεση του λάθος τμήματος του προγράμματος.

Διαθέτει τις εξής θύρες εξόδου:

Hazard Θύρα εξόδου σήματος ύπαρξης Hazard, μεγέθους 1 bit.

- Εφόσον το σήμα έχει την τιμή 1, ο μετρητής προγράμματος δεν μεταβάλλει την τιμή του, αφήνοντας έναν κύκλο ρολογιού πριν προχωρήσει στην επόμενη εντολή. Πρακτικά, δημιουργείται ένα Pipeline Bubble.
- Στην τιμή 0, η μέθοδος της αλλαγής της τιμής του μετρητή προγράμματος δεν μεταβάλλεται.

Instruction Out Θύρες εξόδου διανύσματος εντολής, μεγέθους 32 bit.
Η εντολή που βρίσεται σε αυτή την θύρα προωθείται στις μονάδες των επόμενων σταδίων διασωλήνωσης προς εκτέλεση.

4.3 Τελικό Μοντέλο - Ένα Απλό RISC-V RV(32/64)I Hardware Thread

Συνδυάζοντας την τεχνική της διασωλήνωσης, τις παραπάνω μονάδες και τις εντολές που βρίσκονται στο σύνολο RV32I μπορούμε να οδηγηθούμε στην οργάνωση των σταδίων της διασωλήνωσης και στην λεπτομερέστερη διασύνδεση των λειτουργικών μονάδων. Η διασύνδεση θα χρειαστεί να αξιοποιήσει περαιτέρω στοιχεία λογικού σχεδιασμού για να καταφέρει να αποτελεί μια ολοκληρωμένη, λειτουργική σχεδίαση.

*Η αρχιτεκτονική RISC-V δεν αναφέρει καμία ακολουθία εντολών προς αποφυγή, άρα οποιοσδήποτε κίνδυνος προκύπτει θα πρέπει να αντιμετωπίζεται σε επίπεδο υλικού, έτσι ώστε να εκτελούνται με τα ορισμένα από τις προδιαγραφές αποτελέσματα.

Οι σχεδιαστικές αποφάσεις στο επίπεδο της μικροαρχιτεκτονικής είναι οι εξής:

- Η τεχνική της διασωλήνωσης απαιτεί κάθε στάδιο να εκτελεί διαφορετική εντολή, με τα στάδια να τροφοδοτούν το επόμενό τους. Η πρόταση αυτή ορίζει καταχωρητές οι οποίοι θα ολισθαίνουν το διάνυσμα της εντολής από το ένα στάδιο στο επόμενο. Στην συγκεκριμένη σχεδίαση, οι καταχωρητές αυτοί χρησιμοποιούν την ονοματολογία με **πρόθεμα** `P_`.

- Λόγω της διασωλήνωσης ορίζουμε επίσης καταχωρητές οι οποίοι αποθηκεύουν την τιμή του μετρητή προγράμματος για την εντολή που εκτελεί κάθε στάδιο, με την ονοματολογία με **πρόθεμα** `PC_`.

- Τα σταθερά πεδία κωδικοποιήσεων επιβάλλουν την σταθερή διασύνδεση των πεδίων του διανύσματος της εντολής με τις αντίστοιχες λειτουργικές μονάδες.

- Η απουσία μικροκώδικα και μονάδας ελέγχου για την οργάνωση των υπολειτουργιών λύνεται με την αποκωδικοποίηση του πεδίου `opcode` σε κάθε στάδιο της διασωλήνωσης από τις λειτουργικές μονάδες.

- Ο αποκλεισμός των αχρησιμοποίητων πεδίων κάθε κωδικοποίησης γίνεται με την χρήση πολυπλεκτών και με σήματα ελέγχου που παράγονται ή αναγιγνώσκονται από τις μονάδες.

Οι σχεδιαστικές αποφάσεις στο επίπεδο της διασύνδεσης των μονάδων είναι οι εξής:

- Οι εντολές πράξεων χωρίζονται σε εντολές πράξεων καταχωρητή - καταχωρητή και καταχωρητή - άμεσου δεδομένου. Η ειδοποιός διαφορά των εντολών καταχωρητή - καταχωρητή και καταχωρητή - άμεσου δεδομένου βρίσκεται στην αντικατάσταση ή όχι του πεδίου `rs2` με το πεδίο `imm`, ενώ το πεδίο `rs1` παραμένει ως έχει. Η μονάδα διαχείρισης άμεσων δεδομένων διαθέτει σήμα ενεργοποίησης για τις εντολές που χρησιμοποιούν άμεσο δεδομένο. Από τα παραπάνω οδηγούμαστε στην τοποθέτηση ενός πολυπλέκτη για τον καθορισμό της εισόδου `B` της αριθμητικής - λογικής μονάδας, με σήμα επιλογής την έξοδο `Immediate Enable`, για την επιλογή του καταχωρητή `rs2` ή του άμεσου δεδομένου `imm` αντίστοιχα.

- Για την εκτέλεση των εντολών διακλάδωσης και της εντολής `auipc`, χρειάζεται να υπολογιστεί η πρόσθεση ενός άμεσου δεδομένου με την τιμή του μετρητή προγράμματος της εντολής. Σκοπός του πολυπλέκτη στην είσοδο `A` της αριθμητικής - λογικής μονάδας είναι να ενεργοποιεί την είσοδο του δεδομένου της τιμής του μετρητή προγράμματος, εντοπίζοντας τέτοιου τύπου εντολές από το πεδίο `opcode` του καταχωρητή `P_ID`.

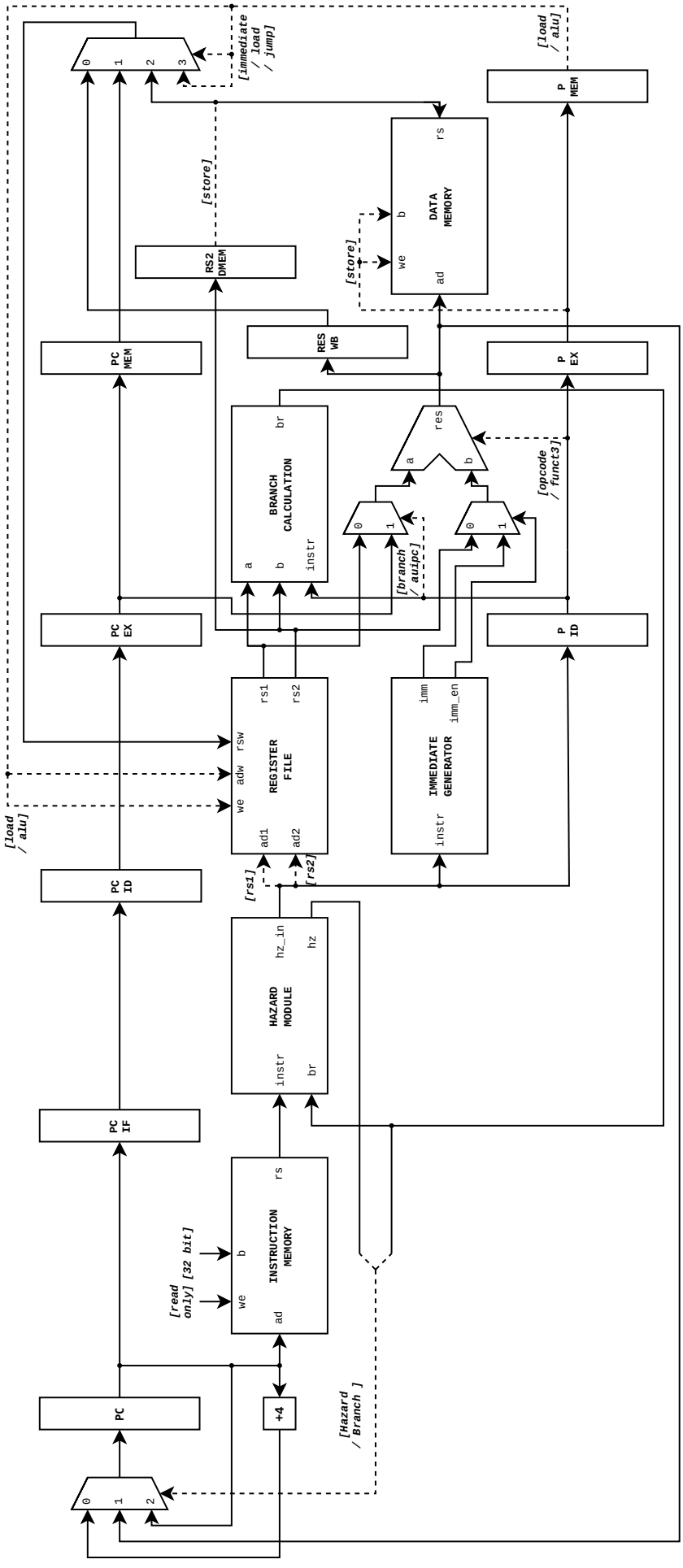
- Ο καταχωρητής `rs2_dmem` χρησιμοποιείται για να αποθηκεύσει στο στάδιο EX, το δεδομένο που αναγνώστηκε από τον δεύτερο καταχωρητή προς ανάγνωση από το αρχείο καταχωρητών. Το δεδομένο προχωράει στην κατάλληλη θύρα της μνήμης δεδομένων για εγγραφή του, εφόσον η εντολή είναι τύπου Store.

- Ο καταχωρητής `res_wb` χρησιμοποιείται για να αποθηκεύσει στο στάδιο MEM/WB το αποτέλεσμα της αριθμητικής πράξης το οποίο θα εγγραφεί στον καταχωρητή που ορίζει το πεδίο `rd`.

- Ορίζεται ένας πολυπλέκτης για την επιλογή του δεδομένου προς εγγραφή στον καταχωρητή που ορίζει το πεδίο `rd`. Πιο συγκεκριμένα, η επιλογή της πηγής του δεδομένου προς εγγραφή ορίζεται από τον τύπο της εντολής, άρα από το πεδίο `opcode` του καταχωρητή `P_MEM`. Υπάρχουν 4 πηγές δεδομένων προς εγγραφή στον καταχωρητή που ορίζει το πεδίο `rd`:
  - Οι **εντολές αριθμητικών πράξεων** χρειάζεται να εγγράψουν στον καταχωρητή που ορίζει το πεδίο `rd` το περιεχόμενο του καταχωρητή αποτελέσματος `res_wb`.
  - Οι **εντολές φόρτωσης** χρειάζεται να εγγράψουν στον καταχωρητή που ορίζει το πεδίο `rd` το δεδομένο που αναγιγνώσκεται από την θύρα `rs` (θύρα εισόδου / εξόδου `Data`) της μνήμης δεδομένων.
  - Η **εντολή φόρτωσης άμεσου δεδομένου** `lui` έχει υλοποιηθεί έτσι ώστε να φορτώνεται από το περιεχόμενο του καταχωρητή `P_MEM`. Δεν έχει οριστεί ειδική πράξη στην αριθμητική - λογική μονάδα για τον υπολογισμό του ορθού αποτελέσματος και εγγραφή του μέσω του καταχωρητή `res_wb`, όπως τις παραπάνω πράξεις, γι αυτό χρησιμοποιείται κατευθείαν ο παραπάνω καταχωρητής.
  - Οι **εντολές άλματος** αποθηκεύουν την τρέχουσα τιμή του μετρητή προγράμματος στον καταχωρητή που ορίζει το πεδίο `rd`. Συνεπώς, πηγή του δεδομένου προς εγγραφή είναι επίσης ο καταχωρητής `PC_MEM`.
- Οι εντολές βρόχου και άλματος τροποποιούν το περιεχόμενο του μετρητή προγράμματος, όπου στο συγκεκριμένο διάγραμμα μικροαρχιτεκτονικής αναφέρεται ως `PC`.
  - Σε περίπτωση που δεν υπάρχει αλλαγή από τέτοιου τύπου εντολές, η τιμή του μετρητή προγράμματος αυξάνεται κατά 4 έτσι ώστε να δεικτοδοτεί την επόμενη εντολή προς εκτέλεση.
  - Εάν αποφασιστεί από την μονάδα διαχείρισης διακλαδώσεων πως θα πρέπει να αλλάξει η ροή του προγράμματος, θα πρέπει να εγγραφεί στον μετρητή προγράμματος η τιμή του αποτελέσματος της πρόσθεσης που θα εκτελέσει η αριθμητική - λογική μονάδα.
  - Σε περίπτωση που υπάρχει κίνδυνος δεδομένου, ο μετρητής προγράμματος θα πρέπει να διατηρήσει την τιμή του αμετάβλητη, έχοντας ως αποτέλεσμα μια προσωρινή παύση της εκτέλεσης νέων εντολών από τον επεξεργαστή.

  Τα σήματα που οδηγούν τον πολυπλέκτη του μετρητή προγράμματος είναι τα σήματα `br (Branch)` και `hz (Hazard)`, που παράγουν οι μονάδες διαχείρισης διακλαδώσεων και κινδύνων αντίστοιχα.

Οι περιγραφές των λειτουργικών μονάδων στις παραπάνω υποενότητες σε συνδυασμό με τις περιγραφές των προδιαγραφών [Wat+16] για την εκτέλεση των εντολών στο σύνολο RV32I οδηγούν στην παρακάτω μικροαρχιτεκτονική:



Διάγραμμα 9: Τελική μικροαρχιτεκτονική του επεξεργαστή

4.4 Επεκτασιμότητα του μοντέλου

Προαναφέρεται πως ορισμένες μονάδες παρέχουν υποστήριξη για μεταβλητό μέγεθος εντέλου για την εκτέλεση των λειτουργιών τους. Με μια παραμετροποίηση των μεγεθών των καταχωρητών δεδομένων των μονάδων και με επέκταση των λειτουργιών της μνήμης δίνεται η δυνατότητα επιλογής της υποστηριζόμενης αρχιτεκτονικής συνόλου εντολών. Το βασικό μοντέλο παρέχει υποστήριξη για 2 αρχιτεκτονικές συνόλου εντολών χωρίς καμία αλλαγή στην μικροαρχιτεκτονική:

- Μέγεθος εντέλου 32-bit, συνεπώς υποστήριξη αρχιτεκτονικής συνόλου εντολών RV32I.
- Μέγεθος εντέλου 64-bit, συνεπώς υποστήριξη αρχιτεκτονικής συνόλου εντολών RV64I.

4.5 Ενδεικτική σύνθεση της περιγραφής του επεξεργαστή

Εκτελέστηκε μια διαδικασία σύνθεσης των περιγραφών του υλικού του επεξεργαστή σε FPGA. Σκοπός της διαδικασίας σύνθεσης ήταν η απόδειξη της υλοποιησιμότητας του μοντέλου σε έναν βασικό βαθμό.

Οι περιγραφές υλικού αποδείχθηκαν συνθέσιμες, χωρίς ωστόσο να ελεγχθούν μετά την σύνθεσή τους για την ορθότητα της λειτουργικότητας του σχεδιασμού. Ο θεωρητικός σχεδιασμός είναι απολύτως λειτουργικός, έχοντας τα αναμενόμενα από τις προδιαγραφές αποτελέσματα μετά την εκτέλεση εντολών και ακολουθιών εντολών.

Για την σύνθεση χρειάστηκε να οριστούν θύρες εισόδου και εξόδου στο μοντέλο του επεξεργαστή, έτσι ώστε να μην απλοποιηθεί σε βαθμό κατακερματισμού από την διαδικασία της σύνθεσης. Αποφασίστηκε πως ο υλοποιήσιμος μικροεπεξεργαστής θα δέχεται ως είσοδο ένα σήμα ρολογιού και ως έξοδο θα εκθέτει τα σήματα αποτελέσματος της αριθμητικής - λογικής μονάδας `res` και την θύρα δεδομένων της μνήμης δεδομένων `rs`, για προσομοίωση μιας απλής διεπαφής με μία εξωτερική μνήμη δεδομένων. Επιλέχθηκαν επίσης αρκετά μικρά μεγέθη μνημών δεδομένων και εντολών, έτσι ώστε να εξαλειφθεί τυχούσα υπερκάλυψη των μονάδων (μπλοκ) του FPGA.

Αξιοποιούνται τα αναμενόμενα μπλοκ ειδικού σκοπού του FPGA για την υλοποίηση της αριθμητικής - λογικής μονάδας, του αρχείου καταχωρητών, τις μνήμες εντολών και δεδομένων. Οι υπόλοιπες μονάδες συντέθηκαν από μπλοκ γενικού σκοπού του FPGA. Η σύνθεση είχε ως αποτέλεσμα χρονισμού ρολόι με συχνότητα 148 MHz, χωρίς καμία βελτιστοποίηση, αφού οι περιγραφές υλικού δεν έχουν σχεδιαστεί με γνώμονα την διαδικασία της σύνθεσης.

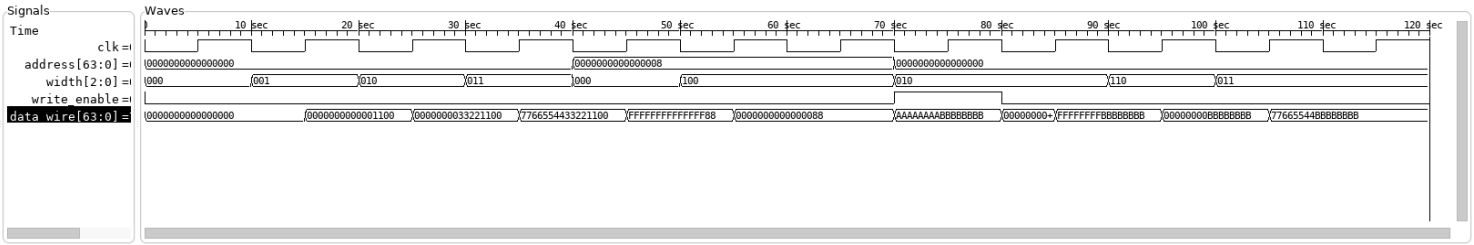
Κεφαλαιο 5
ΕΛΕΓΧΟΣ ΛΕΙΤΟΥΡΓΙΩΝ - ΚΥΜΑΤΟΜΟΡΦΕΣ

5.1 Μονάδα Μνήμης

Η μονάδα μνήμης αρχικοποιείται με υποστήριξη για τις εντολές RV64I, έτσι ώστε να αναδειχθούν όλες οι λειτουργίες της. Φορτώνεται με δεδομένα από το αρχείο `memtest.mem`, κατ' αυτόν τον τρόπο:

Θέση Μνήμης	Περιεχόμενο
0x0000	00 11 22 33
0x0004	44 55 66 77
0x0008	88 99 aa bb
0x000A	cc dd ee ff
0x0010	01 23 45 67

Παραδείγματα λειτουργιών:



Κυματομορφή 10: Κυματομορφή Λειτουργιών της Μονάδας Μνήμης.

Χρονική στιγμή 0

Προετοιμασία ανάγνωσης ενός byte στην θέση μνήμης με διεύθυνση 0x0000 και επέκταση προσήμου.

- Σήμα εισόδου `address`
  Ορισμός διεύθυνσης προσπέλασης στην θέση 0x0000.
- Σήμα εισόδου `width`
  Ορισμός μεγέθους προσπέλασης στην τιμή 000, άρα 1 byte με επέκταση προσήμου.
- Σήμα εισόδου `write_enable`
  Ορισμός σήματος εγγραφής στην τιμή 0, άρα λειτουργία ανάγνωσης.

Χρονική στιγμή 5

Αποτέλεσμα ανάγνωσης ενός byte στην θέση 0x0000 με επέκταση προσήμου.

- Σήμα εξόδου `data_wire`
  Από την μνήμη διαβάζεται το περιεχόμενο 0x00 της θέσης 0x0000, με επέκταση προσήμου.

Χρονική στιγμή 10

Προετοιμασία ανάγνωσης δύο byte στην θέση 0x0000 και επέκταση προσήμου.

- Σήμα εισόδου `width`
  Ορισμός μεγέθους προσπέλασης στην τιμή 001, άρα 2 byte με επέκταση προσήμου.
- Άλλα σήματα εισόδου
  Παραμένουν ως έχουν.

Χρονική στιγμή 15

Αποτέλεσμα ανάγνωσης δύο byte στην θέση 0x0000 με επέκταση προσήμου.

- Σήμα εξόδου `data_wire`
  Από την μνήμη διαβάζεται το περιεχόμενο 0x1100 της θέσης 0x0000, με επέκταση προσήμου.

Χρονικές στιγμές 20, 25, 30, 35

Διαδοχικές αναγνώσεις της θέσης μνήμης 0x0000 και αποτελέσματά τους με επαύξηση μεγέθους ανάγνωσης στα 4 και 8 byte αντίστοιχα, με επέκταση προσήμου.
Ακολουθείται η ίδια διαδικασία αλλάζοντας μόνο το σήμα `width`, με τα αντίστοιχα αποτελέσματα.

Χρονική στιγμή 40

Προετοιμασία ανάγνωσης ενός byte στην θέση 0x0008 και επέκταση προσήμου με εφαρμογή των αντίστοιχων σημάτων.

Χρονική στιγμή 45

Αποτέλεσμα ανάγνωσης του byte στην θέση 0x0000 με επέκταση προσήμου.

Χρονική στιγμή 50

Προετοιμασία ανάγνωσης ενός byte στην θέση 0x0008 χωρίς επέκταση προσήμου.

- Σήμα εισόδου `width`
  Ορισμός μεγέθους προσπέλασης στην τιμή 100, άρα 1 byte χωρίς επέκταση προσήμου.
- Άλλα σήματα εισόδου
  Παραμένουν ως έχουν.

Χρονική στιγμή 55

Αποτέλεσμα ανάγνωσης του byte στην θέση 0x0008 χωρίς επέκταση προσήμου.

Χρονική στιγμή 70

Προετοιμασία για εγγραφή του δεδομένου 0xAAAAAAAABBBBBBBB στην διεύθυνση μνήμης 0x0000.

- Σήμα εισόδου `address`
  Ορισμός διεύθυνσης προσπέλασης στην θέση 0x0000.
- Σήμα εισόδου `width`
  Ορισμός μεγέθους προσπέλασης στην τιμή 010, άρα 4 byte.
- Σήμα εισόδου `write_enable`
  Ορισμός σήματος εγγραφής στην τιμή 1, άρα λειτουργία εγγραφής.
- Σήμα εισόδου `data_enable`
  Ορισμός του δεδομένου 0xAAAAAAAABBBBBBBB προς εγγραφή.

Χρονική στιγμή 80

Προετοιμασία ανάγνωσης 4 byte στην θέση 0x0000 και επέκταση προσήμου, με εφαρμογή των αντίστοιχων σημάτων.

Χρονική στιγμή 85

Αποτέλεσμα ανάγνωσης 4 byte στην θέση 0x0000 με επέκταση προσήμου.

Χρονική στιγμή 90

Προετοιμασία ανάγνωσης 4 byte στην θέση 0x0000 χωρίς επέκταση προσήμου, με εφαρμογή των αντίστοιχων σημάτων.

Χρονική στιγμή 95

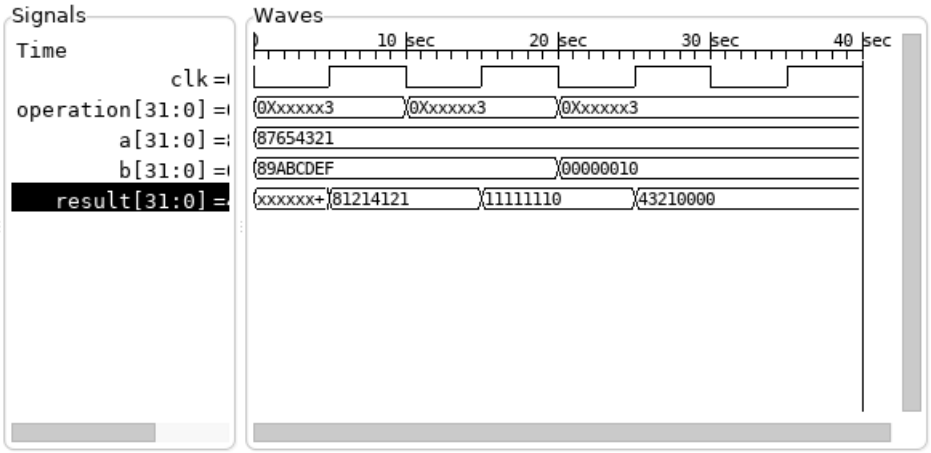
Αποτέλεσμα ανάγνωσης 4 byte στην θέση 0x0000 χωρίς επέκταση προσήμου.

Χρονική στιγμή 100
Προετοιμασία ανάγνωσης 8 byte στην θέση `0x0000`, με εφαρμογή των αντίστοιχων σημάτων.

Χρονική στιγμή 105
Αποτέλεσμα ανάγνωσης 8 byte στην θέση `0x0000`.

5.2 Αριθμητική - Λογική Μονάδα

Παραδείγματα λειτουργιών:



Κυματομορφή 11: Κυματομορφή λειτουργιών της Αριθμητικής - Λογικής μονάδας.

Χρονική στιγμή 0
Προετοιμασία των σημάτων εισόδου για εκτέλεση της λογικής πράξης AND ως εξής:

- Σήμα εισόδου `A`
  Ορισμός δεδομένου A ως `0x87654321`.
- Σήμα εισόδου `B`
  Ορισμός δεδομένου B ως `0x89ABCDEF`.
- Σήμα εισόδου `Operation`
  Ορισμός της πράξης προς εκτέλεση ως λογικό AND, με αδιάφορη τιμή x στα bit που δεν επηρεάζουν τον ορισμό της πράξης.

Χρονική στιγμή 5
Αποτέλεσμα της πράξης `0x87654321 & 0x89ABCDEF` που είναι `0x81214121`.

Χρονική στιγμή 10
Προετοιμασία του σήματος εισόδου `Operation` για την εκτέλεση της αριθμητικής πράξης ADD. Η υπερχείλιση αγνοείται.

Χρονική στιγμή 15
Αποτέλεσμα της πράξης `0x87654321 + 0x89ABCDEF` το οποίο είναι `0x11111110`.

Χρονική στιγμή 20

Προετοιμασία των σημάτων εισόδου για εκτέλεση της λογικής πράξης αριστερής ολίσθησης (εντολή sll) ως εξής:

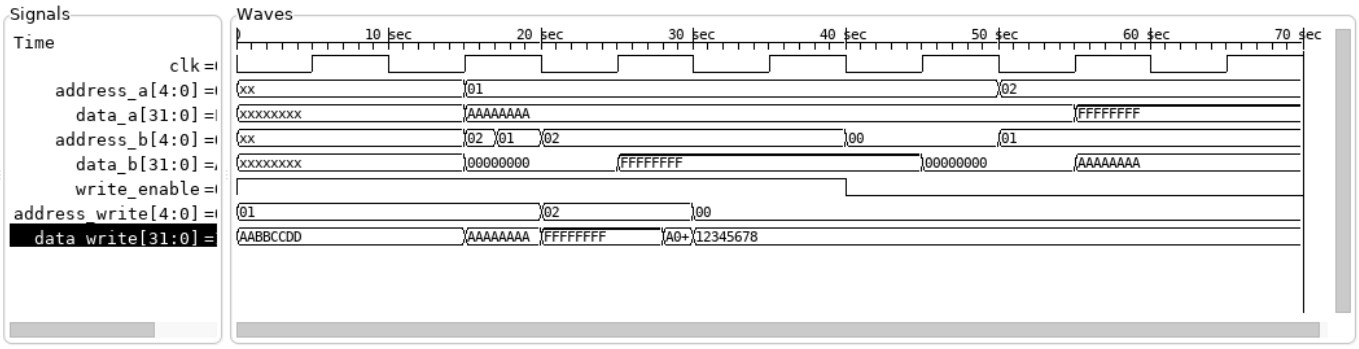
- Σήμα εισόδου `A`
  Ορισμός δεδομένου A ως `0x87654321`.
- Σήμα εισόδου `B`
  Ορισμός δεδομένου B ως `0x00000010`, για ολίσθηση της εισόδου A κατά 4.
- Σήμα εισόδου `Operation`
  Ορισμός της πράξης προς εκτέλεση ως αριστερή ολίσθηση, με αδιάφορη τιμή x στα bit που δεν επηρεάζουν τον ορισμό της πράξης.

Χρονική στιγμή 25

Αποτέλεσμα της πράξης `0x87654321` $<<$ `4` το οποίο είναι `0x43210000`.

5.3 Αρχείο Καταχωρητών

Παραδείγματα λειτουργιών:



Κυματομορφή 12: Κυματομορφή λειτουργιών του Αρχείου Καταχωρητών.

Χρονική στιγμή 0

Προετοιμασία των σημάτων εισόδου για εγγραφή δεδομένου σε καταχωρητή.

- Σήμα εισόδου `address_write`
  Ορισμός διεύθυνσης καταχωρητή προς εγγραφή ως `0x01`.
- Σήμα εισόδου `data_write`
  Ορισμός δεδομένου προς εγγραφή ως `0xAABBCCDD`.
- Σήμα εισόδου `write_enable`
  Ορισμός σήματος εγγραφής στην τιμή `1`, έτσι ώστε να πραγματοποιηθεί η εγγραφή του δεδομένου στον ορισμένο καταχωρητή.

Χρονική στιγμή 5

Εγγραφή του δεδομένου `0xAABBCCDD` στον καταχωρητή με διεύθυνση `0x01`.

Χρονική στιγμή 15

Προετοιμασία των σημάτων εισόδου για εγγραφή δεδομένου σε καταχωρητή και αναγνώσεις δύο καταχωρητών.

- Σήμα εισόδου `address_write`
  Ορισμός διεύθυνσης καταχωρητή προς εγγραφή ως `0x01`.
- Σήμα εισόδου `data_write`
  Ορισμός δεδομένου προς εγγραφή ως `0xAAAAAAAA`.
- Σήμα εισόδου `write_enable`
  Ορισμός σήματος εγγραφής στην τιμή `1`, έτσι ώστε να πραγματοποιηθεί η εγγραφή του δεδομένου στον ορισμένο καταχωρητή.
- Σήμα εισόδου `address_a`
  Ορισμός διεύθυνσης πρώτου καταχωρητή προς ανάγνωση ως `0x01`.
- Σήμα εισόδου `address_b`
  Ορισμός διεύθυνσης δεύτερου καταχωρητή προς ανάγνωση ως `0x02`.

Ταυτόχρονα, παρατηρείται στα σήματα εξόδου:

- Σήμα εξόδου `data_a`
  Ανάγνωση της τιμής `0xAAAAAAAA` από τον καταχωρητή με διεύθυνση `0x01`. Στον συγκεκριμένο καταχωρητή συμβαίνει αυτή την στιγμή η εγγραφή του δεδομένου `0xAAAAAAAA`, το οποίο διαβάζουμε. Σε περίπτωση που οι λειτουργίες εγγραφής και ανάγνωσης δεν υποστηρίζονταν ταυτόχρονα στον ίδιο καταχωρητή, θα αναγιγνωσκόταν το δεδομένο `0xAABBCCDD` που βρισκόταν μέχρι πρότινος στον καταχωρητή με διεύθυνση `0x01`.
- Σήμα εξόδου `data_b`
  Ανάγνωση της τιμής `0x00000000` από τον καταχωρητή με διεύθυνση `0x02`.

Χρονική στιγμή 17

Απρόσμενη μεταβολή του σήματος εισόδου `address_b` το οποίο δεν αλλάζει την τρέχουσα τιμή που είχε αναγνωστεί στην θετική ακμή του ρολογιού, την στιγμή 15.

Χρονική στιγμή 20

Προετοιμασία των σημάτων εισόδου που αφορούν την λειτουργία εγγραφής, έτσι ώστε να εγγραφεί το δεδομένο `0xFFFFFFFF` στον καταχωρητή με διεύθυνση `0x02`.

Χρονική στιγμή 25

Ταυτόχρονη λειτουργία εγγραφής και ανάγνωσης του δεδομένου `0xFFFFFFFF` που αφορά τον καταχωρητή με διεύθυνση `0x02`.

Χρονική στιγμή 28

Απρόσμενη μεταβολή του σήματος εισόδου `data_write` το οποίο δεν αλλάζει την προηγούμενη τιμή εγγραφής στην θετική ακμή του ρολογιού, την στιγμή 25.

Χρονική στιγμή 30

Προετοιμασία των σημάτων εισόδου που αφορούν την λειτουργία εγγραφής, έτσι ώστε να εγγραφεί το δεδομένο `0x12345678` στον καταχωρητή με διεύθυνση `0x00`.

Χρονική στιγμή 35

Εγγραφή του δεδομένου `ox12345678` στον καταχωρητή με διεύθυνση `0x00`.

Χρονική στιγμή 40

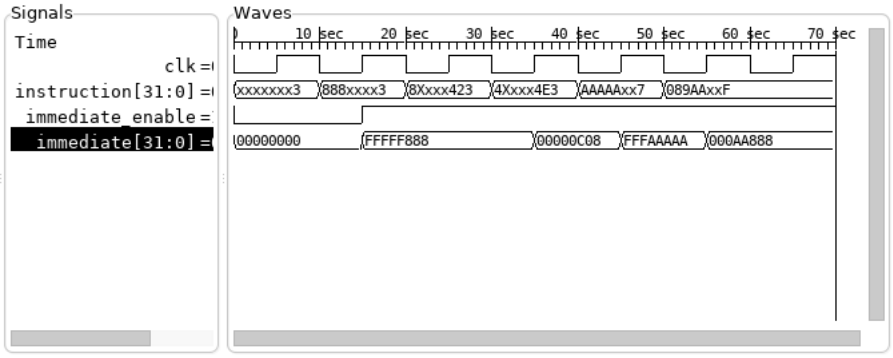
Προετοιμασία σημάτων εισόδου για ανάγνωση των περιεχομένων των καταχωρητών των διευθύνσεων `0x01` και `0x00`.

Χρονική στιγμή 45

Ανάγνωση των τιμών των καταχωρητών με διευθύνσεις0x01 και 0x00. Παρατηρείται ότι παρά την εγγραφή του δεδομένου 0x12345678 στον καταχωρητή με διεύθυνση 0x00, η ανάγνωσή του επιστρέφει σταθερά την τιμή 0x00000000, όπως ορίζεται από τις προδιαγραφές [Wat+16].

5.4 Μονάδα Διαχείρισης Άμεσων Δεδομένων

Παραδείγματα Λειτουργιών:



Κυματομορφή 13: Κυματομορφή λειτουργιών της μονάδας Διαχείρισης Άμεσων Δεδομένων

Χρονική στιγμή 0

Προετοιμασία σήματος εισόδου διανύσματος εντολής τύπου R για αποκωδικοποίηση, με αδιάφορη τιμή x στα bit που δεν επηρεάζουν την λειτουργία της μονάδας.

Χρονική στιγμή 5

Αποτέλεσμα αποκωδικοποίησης άμεσου δεδομένου. Στις εντολές τύπου R δεν υπάρχει άμεσο δεδομένο και το σήμα ενεργοποίησης παίρνει την τιμή 0.

Χρονική στιγμή 10

Προετοιμασία σήματος εισόδου διανύσματος εντολής τύπου I για αποκωδικοποίηση, με αδιάφορη τιμή x στα bit που δεν επηρεάζουν την λειτουργία της μονάδας. Το διάνυσμα κωδικοποιεί το άμεσο δεδομένο 0x888.

Χρονική στιγμή 15

Αποτέλεσμα αποκωδικοποίησης του άμεσου δεδομένου 0x888. Με την απαραίτητη επέκταση προσήμου η αποκωδικοποίηση παίρνει την τιμή 0xFFFFF888 και επιπλέον το σήμα ενεργοποίησης παίρνει την τιμή 1.

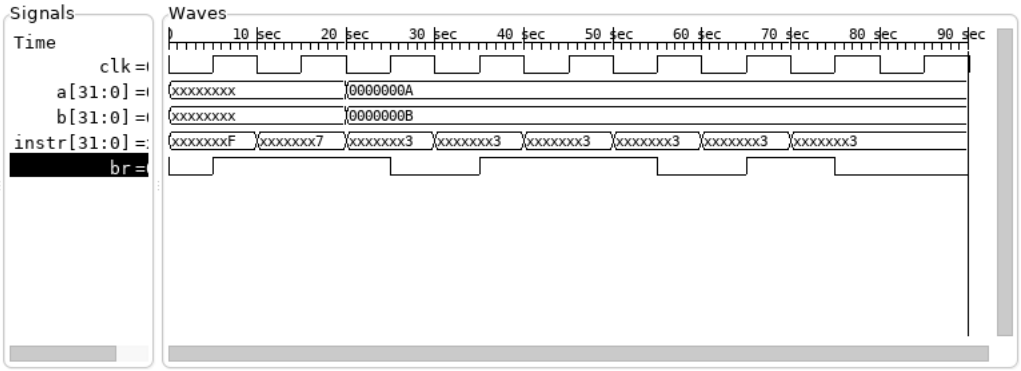
Χρονικές στιγμές 20, 25, 30, 35, 40, 45, 50, 55

Προετοιμασία του σήματος εισόδου διανυσμάτων εντολών τύπου S, B, U, J και αποτελέσματά τους αντίστοιχα. Το σήμα ενεργοποίησης παίρνει την τιμή 1 για κάθε μία από τις παρακάτω κωδικοποιήσεις εντολών, καθώς κωδικοποιούν άμεσο δεδομένο.
Κωδικοποιούνται με την σειρά οι αριθμοί:

- 0x888 στην κωδικοποίηση S - type, όπου με επέκταση προσήμου παράγει το αποτέλεσμα 0xFFFFF888.
- 0x0C08 στην κωδικοποίηση B - type, όπου με επέκταση προσήμου παράγει το αποτέλεσμα 0x00000C08.
- 0xAAAAA στην κωδικοποίηση U - type, όπου με επέκταση προσήμου παράγει το αποτέλεσμα 0xFFFAAAAA.
- 0x0AA888 στην κωδικοποίηση J - type, όπου με επέκταση προσήμου παράγει το αποτέλεσμα 0x000AA888.

5.5 Μονάδα Διαχείρισης Διακλαδώσεων

Παραδείγματα Λειτουργιών:



Κυματομορφή 14: Κυματομορφή λειτουργιών της μονάδας Διαχείρισης Διακλαδώσεων

Χρονική στιγμή 0

Προετοιμασία του σήματος εισόδου διανύσματος εντολής `jal`, με αδιάφορη τιμή x στα bit που δεν επηρεάζουν την λειτουργία της μονάδας.

Χρονική στιγμή 5

Αποτέλεσμα απόφασης διακλάδωσης. Το σήμα ενεργοποίησης διακλάδωσης παίρνει την τιμή 1, ανεξαρτήτως των τιμών των εισόδων δεδομένων προς σύγκριση `A` και `B`.

Χρονικές στιγμές 10, 15

Προετοιμασία του σήματος εισόδου διανύσματος εντολής `jalr`, με αδιάφορη τιμή x στα bit που δεν επηρεάζουν την λειτουργία της μονάδας, έχοντας ίδιο αποτέλεσμα με το παραπάνω.

Χρονική στιγμή 20

Ανάθεση τιμών `0x0000000A` και `0x0000000B` στις εισόδους προς σύγκριση `A` και `B` αντίστοιχα. Προετοιμασία του σήματος εισόδου διανύσματος εντολής `beq`, με αδιάφορη τιμή x στα bit που δεν επηρεάζουν την λειτουργία της μονάδας.

Χρονική στιγμή 25

Αποτέλεσμα απόφασης διακλάδωσης με βάση την σύγκριση των εισόδων `A` και `B`. Το σήμα ενεργοποίησης διακλάδωσης παίρνει την τιμή 0.

Χρονικές στιγμές 30, 35, 40, 45, 50, 55, 60, 65, 70, 75

Διαδοχικές αλλαγές του σήματος εισόδου διανύσματος εντολής, έτσι ώστε να ελεγχθεί η λειτουργία όλων των εντολών διακλάδωσης, καθώς και τα αποτελέσματά τους. Το διάνυσμα μεταβάλλεται έτσι ώστε να κωδικοποιεί τις εντολές `bne`, `blt`, `bge`, `bltu`, `bgeu`, με αδιάφορη τιμή x στα bit που δεν επηρεάζουν την λειτουργία της μονάδας. Τα αποτελέσματα απόφασης διακλάδωσης της κάθε μίας από αυτές τις εντολές εμφανίζονται στην τιμή του σήματος ενεργοποίησης διακλάδωσης, με τις τιμές 1 και 0 για εκτέλεση της διακλάδωσης ή όχι αντίστοιχα.

5.6 Μονάδα Διαχείρισης Κινδύνων

Η μονάδα διαχείρισης κινδύνων δεν θα μπορούσε να εξομοιώσει τις λειτουργίες της ως αυτοτελής μονάδα. Οι λειτουργίες της παρουσιάζονται παρακάτω, ταυτόχρονα με τις ολοκληρωμένες κυματομορφές του επεξεργαστή κατά την εκτέλεση ενός ενδεικτικού προγράμματος.

Η λειτουργία της και ειδικότερα ο χρονισμός των σημάτων εξόδου της μονάδας διαχείρισης κινδύνων βασίζεται στα στάδια διασωλήνωσης του συγκερκιμένου μοντέλου επεξεργαστή καθώς και στην καθυστέρηση της ανάγνωσης της μνήμης εντολών κατά έναν κύκλο. Το σήμα ενεργοποίησης κινδύνου της μονάδας διαχείρισης κινδύνων συνδέεται άρρηκτα με τον μετρητή προγράμματος που διευθυνσιοδοτεί την μνήμη εντολών και με την λειτουργία της μνήμης εντολών. Το σήμα εξόδου εντολής προς εκτέλεση και ο χρόνος που παραμένει αμετάβλητο βασίζεται στην δομή και το βάθος της διασωλήνωσης του μοντέλου του επεξεργαστή, καθώς και την διαφορά μεταξύ των κύκλων που εμφανίστηκε ο κίνδυνος δεδομένων.

5.7 Μοντέλο Επεξεργαστή - Εκτέλεση Προγράμματος

Για τις ανάγκες επίδειξης βασικής λειτουργικότητας του μοντέλου επεξεργαστή έχει συνταχθεί και μεταφραστεί ένα απλό πρόγραμμα Assembly, αξιοποιώντας τον υλοποιημένο Assembler. Παρουσιάζονται οι εντολές του προγράμματος, καθώς και οι δυαδικές τους κωδικοποιήσεις, οι οποίες κατά την εκτέλεση αναγιγνώσκονται από την μνήμη εντολών.

Πρόγραμμα σε Assembly	Περιεχόμενο Μνήμης Εντολών*

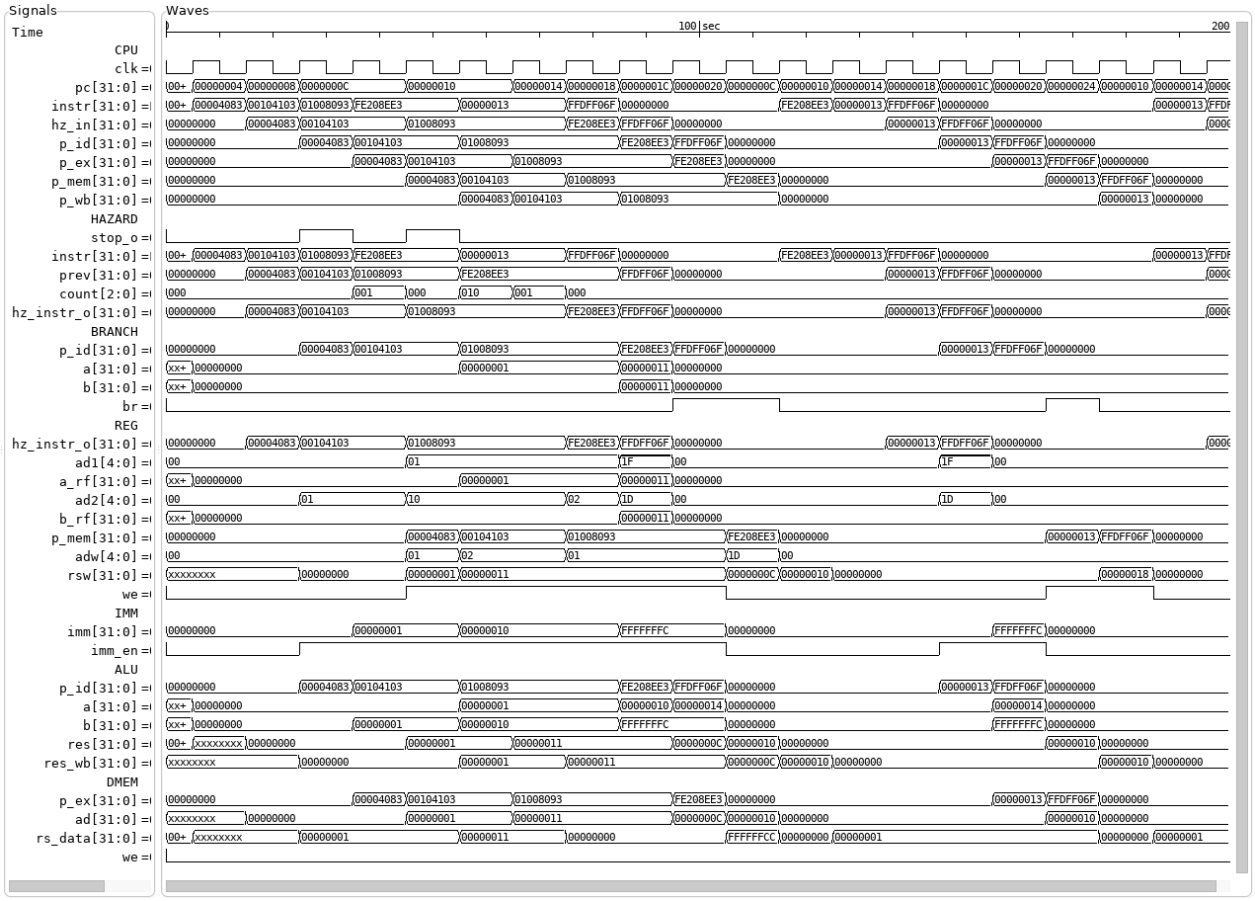
`1 lbu x1, 0(zero)`
`2 lbu x2, 1(zero)`
`3 loop:`
`4  addi x1, x1, 0x010`
`5  beq x1, x2, loop`
`6 halt:`
`7  nop`
`8  jal x0, halt`
	`1 @0`
	`2 83 40 00 00`
	`3 03 41 10 00`
	`4 93 80 00 01`
	`5 e3 8e 20 fe`
	`6 13 00 00 00`
	`7 6f f0 df ff`
	`8 // SYMBOL TABLE`
	`9 // [0x00000008] loop`
	`10 // [0x00000010] halt`

Πίνακας 13: Ένα απλό παράδειγμα μετάφρασης προγράμματος και αποθήκευσής του στην μνήμη εντολών

Στην τρέχουσα παραμετροποίηση, το μοντέλο επεξεργαστή είναι ορισμένο στο μέγεθος εντέλου 32-bit, άρα με υποστήριξη αρχιτεκτονικής συνόλου εντολών RV32I. Το παραπάνω πρόγραμμα θα μπορούσε να εκτελεστεί και στο μοντέλο επεξεργαστή με παραμετροποίηση για υποστήριξη RV64I.

*Η αποθήκευση των κωδικοποίησεων των εντολών στην μνήμη εντολών γίνεται με Little Endian τρόπο, ενώ στις κυματομορφές εμφανίζονται με Big Endian τρόπο.

Παράδειγμα εκτέλεσης βασικού προγράμματος:



Κυματομορφή 15: Κυματομορφή παραδείγματος εκτέλεσης προγράμματος στον επεξεργαστή

- Το πρόγραμμα έχει κίνδυνο δεδομένου μεταξύ των εντολών μεταξύ των γραμμών 1 και 4, 2 και 5. Το πρόβλημα παρουσιάζεται με την ανάγνωση μετά την εγγραφή των καταχωρητών `x1`, `x2` αντίστοιχα. Παρατηρείται στα σήματα των καταχωρητών διασωλήνωσης η καθυστέρηση που δημιουργείται στην ανάθεση εντολών στα επόμενα στάδια διασωλήνωσης για 2 ή 3 κύκλους, για την αντιμετώπιση του κινδύνου.

- Παρατηρείται επίσης η εκκαθάριση των καταχωρητών της διασωλήνωσης για την αποφυγή εκτέλεσης του λάθος μέρους του προγράμματος μετά από την διακλάδωση. Τα στάδια που εκκαθαρίζονται είναι αρχικά τα στάδια ID και EX, και σε δεύτερο χρόνο το στάδιο MEM / WB. Με αυτόν τον τρόπο εξασφαλίζεται ότι η εντολή όπου λανθασμένα στάλθηκε προς εκτέλεση, δεν θα έχει επιρροή πέραν της ανάγνωσης καταχωρητών, πράγμα το οποίο δεν αλλάζει την κατάσταση του επεξεργαστή. Με την καθυστέρηση της εκκαθάρισης του σταδίου MEM / WB δίνεται ο απαραίτητος κύκλος στην προηγούμενη εντολή να εκτελέσει τις λειτουργίες αυτού του σταδίου. Η υλοποιημένη συμπεριφορά είναι συμβατή με τις προδιαγραφές, δηλαδή δεν υπάρχει παράθυρο εκτέλεσης εντολής καθυστερημένης διακλάδωσης. Επίσης, σε περίπτωση που αποφασιστεί να μην εκτελεστεί η διακλάδωση, δεν γίνεται εκκαθάριση των καταχωρητών και η εκτέλεση συνεχίζεται χωρίς κάποια καθυστέρηση για αναπροσαρμογή των καταχωρητών της διασωλήνωσης.

Κεφαλαιο 6
ΜΕΛΛΟΝΤΙΚΕΣ ΒΕΛΤΙΣΤΟΠΟΙΗΣΕΙΣ

Οι υλοποιήσεις που έχουν αναπτυχθεί στην διπλωματική εργασία είναι χρήσιμες για εκπαιδευτικούς σκοπούς όπως:

- Εξομοίωση του βασικού υποσυνόλου εντολών RISC-V RV32I.
- Παράδειγμα υλοποίησης ενός λειτουργικού και συμβατού με προδιαγραφές Assembler.
- Παρουσίαση των λειτουργιών ενός διασωληνωμένου επεξεργαστή σε επίπεδο εσωτερικών σημάτων.
- Ο συνδυασμός όλων των παραπάνω ως ένα ολοκληρωμένο λειτουργικό μοντέλο.

Η συγκεκριμένη διπλωματική εργασία ορίζει ένα βασικό θεμέλιο για την ανάπτυξη και ιδανικά την υλοποίηση ενός λειτουργικού μικροεπεξεργαστή. Σε κάθε περίπτωση, υπάρχουν πολλές πτυχές της οι οποίες δύνανται να αναπτυχθούν περαιτέρω στο μέλλον, προσεγγίζοντας έναν πραγματικό μικροεπεξεργαστή.

6.1 Υλοποίηση Επεκτάσεων

Η αρχιτεκτονική συνόλου εντολών RISC-V ορίζει προαιρετικά υποσύνολα εντολών. Θα μπορούσαν να αναπτυχθούν σε επίπεδο εξομοίωσης και να ενσωματωθούν στο συγκεκριμένο μοντέλο επεξεργαστή.

Ο Assembler επίσης είναι σχεδιασμένος με βάση την εύκολη επεκτασιμότητα, συνεπώς θα μπορούσε να χρησιμοποιηθεί για την παραγωγή συμβατών εκτελέσιμων με τον περαιτέρω ανεπτυγμένο επεξεργαστή. Επεκτείνοντας παράλληλα τον επεξεργαστή και τον Assembler, δεν χρειάζεται να γίνει αλλαγή του τρόπου αποσφαλμάτωσης και ανάπτυξης που ακολουθήθηκε μέχρι στιγμής.

6.2 Μικροαρχιτεκτονικές Βελτιώσεις

Πολλοί υλοποιημένοι επεξεργαστές RISC-V αξιοποιούν τεχνικές όπως η out-of-order εκτέλεση εντολών και πολυπύρηνες μικροαρχιτεκτονικές. Το μοντέλο προς το παρόν ορίζει έναν μονοπύρηνο in-order επεξεργαστή, η σχεδίασή του ωστόσο δεν περιορίζει την επέκταση των δυνατοτήτων του για ενσωμάτωση νέων σχεδιαστικών τεχνικών.

Πιο συγκεκριμένα, το στάδιο αντιμετώπισης Hazard που έχει προστεθεί στην κλασσική RISC διασωλήνωση, θα μπορούσε να χαρακτηριστεί ως μια πρώιμη μονάδα Instruction Issuing και Memory Ordering. Μπορούν να υλοποιηθούν σχεδιαστικές τεχνικές όπως:

- Instruction window για υποστήριξη out-of-order εκτέλεσης εντολών
- Register renaming για την γρηγορότερη διαχείριση των κινδύνων δεδομένων και εκτέλεση των προγραμμάτων
- Branch prediction για πρόβλεψη των αποφάσεων διακλαδώσεων

6.3 Υλοποίηση σε Υλικό

Έχει ήδη γίνει μια ενδεικτική σύνθεση των βασικών λειτουργικών μονάδων του επεξεργαστή σε FPGA, χωρίς να είναι ο κύριος στόχος αυτής της διπλωματικής εργασίας. Φυσικά, μια περιγραφή υλικού γίνεται με σκοπό την πρακτική, λειτουργική υλοποίησή της.

Η ανάπτυξη ωστόσο ενός ολοκληρωμένου μικροεπεξεργαστή απέχει πολύ από μια διαδικασία εξομοίωσης και θεωρητικής υλοποίησης. Αποτελεί πολύπλοκο, σύνθετο και χρονοβόρο εγχείρημα, που απαιτεί εξειδικευμένο εξοπλισμό, ανθρώπινο δυναμικό (πέραν του ενός ατόμου) τόσο για την σχεδίαση όσο και για την επιβεβαίωση της λειτουργικότητας της τελικής υλοποιημένης σύνθεσης υλικού.

6.4 Υλοποίηση Βασικού Compiler

Συμπληρωματικά με την λειτουργική υλοποίηση του μικροεπεξεργαστή, θα μπορούσε να αναπτυχθεί ένας C Compiler για την διευκόλυνση εκτέλεσης στατικών προγραμμάτων αρχικά, με τελικό στόχο την εκτέλεση ενός λειτουργικού συστήματος στον υλοποιημένο πλέον μικροεπεξεργαστή.

6.5 Υποστήριξη συσκευών

Φυσική απόρροια των ρεαλιστικών λειτουργικών υλοποιήσεων υλικού και της επιτυχούς εκτέλεσης ενός λειτουργικού συστήματος είναι η υποστήριξη διασύνδεσης συσκευών στο πλέον χρήσιμο σύστημα, μέρος του οποίου θα αποτελεί ο μικροεπεξεργαστής. Το τελικό σύστημα θα μπορούσε να είναι ένας λειτουργικός μικροελεγκτής, ή ακόμα και μια πρώιμη μονάδα κεντρικής επεξεργασίας ενός ολοκληρωμένου υπολογιστικού συστήματος.

Κεφαλαιο 7
ΠΑΡΑΡΤΗΜΑ - ΥΛΟΠΟΙΗΣΕΙΣ

Όλες οι υλοποιήσεις βρίσκονται εδώ: https://github.com/laskarelias/upatras-riscv

7.1 Περιγραφή Υλικού - Verilog

Ακολουθεί η περιγραφή υλικού κάθε μίας λειτουργικής μονάδας και του μοντέλου του επεξεργαστή, σε γλώσσα Verilog.

7.1.1 Περιγραφή Μονάδας Μνήμης - mem.v

```
module mem
#(parameter BITS = 32)
(clk, ad, we, b, rs);

    input            clk;
    input [BITS-1:0] ad;   // mem address
    input            we;   // write enable
    input [2:0]      b;    // width
    inout [BITS-1:0] rs;   // mem data - read write port
    parameter file = "";
    integer i = 0;

    reg [7:0]      memory[4095:0];   // 4096 x 8-bit
    reg [BITS-1:0] rs_data;

    initial begin
        for (i = 0; i < 4096; i = i + 1) begin
            memory[i] = 'b0;
        end
        rs_data = 'b0;
        $readmemh(file, memory, 0, 4095);
    end

    always @(posedge clk)
        if (we) begin
            memory[ad]     <= rs[7:0];
            memory[ad + 1] <= (b == 3'b001) || (b == 3'b010) || (b == 3'b011) ?
                              rs[15:8]  : memory[ad + 1];
            memory[ad + 2] <= (b == 3'b010) || (b == 3'b011) ?
                              rs[23:16] : memory[ad + 2];
            memory[ad + 3] <= (b == 3'b010) || (b == 3'b011) ?
                              rs[31:24] : memory[ad + 3];
            if (BITS > 32) begin
                memory[ad + 4] <= (b == 3'b011) ? rs[39:32] : memory[ad + 4];
                memory[ad + 5] <= (b == 3'b011) ? rs[47:40] : memory[ad + 5];
                memory[ad + 6] <= (b == 3'b011) ? rs[55:48] : memory[ad + 6];
                memory[ad + 7] <= (b == 3'b011) ? rs[63:56] : memory[ad + 7];
            end
        end
        else begin
            case ({b})
                3'b000  : begin // LB
                              rs_data <= $signed(memory[ad]);
                          end
                3'b001  : begin // LH
                              rs_data <= $signed({memory[ad + 1], memory[ad]});
                          end
                3'b010  : begin // LW
                              rs_data <= $signed({memory[ad + 3], memory[ad + 2],
                                                  memory[ad + 1], memory[ad]});
                          end
                3'b011  : begin // LD
                              rs_data <= $signed({memory[ad + 7], memory[ad + 6],
                                                  memory[ad + 5], memory[ad + 4],
                                                  memory[ad + 3], memory[ad + 2],
                                                  memory[ad + 1], memory[ad]});
                          end
                3'b100  : begin // LBU
                              rs_data <= {memory[ad]};
```

```
60                          end
61                  3'b101  : begin // LHU
62                              rs_data <= {memory[ad + 1], memory[ad]};
63                          end
64                  3'b110  : begin // LWU
65                              rs_data <= {memory[ad + 3], memory[ad + 2],
66                                          memory[ad + 1], memory[ad]};
67                          end
68                  default : begin
69                              rs_data <= 'b0;
70                          end
71              endcase
72          end
73
74      assign rs = (we) ? 'bz : rs_data;
75
76  endmodule
```

7.1.2 Περιγραφή Αριθμητικής - Λογικής Μονάδας - alu.v

```
1   module alu
2  #(parameter BITS = 32)
3  (clk, instr, a, b, res_r);
4
5       input              clk;
6       input  [BITS-1:0] a;         // rs1
7       input  [BITS-1:0] b;         // rs2
8       input  [31:0]      instr;
9       output [BITS-1:0] res_r;
10      reg    [BITS-1:0] res;        //result register
11      reg    [BITS-1:0] res_r;
12
13      initial begin
14          res = 0;
15      end
16
17      always @(posedge clk) begin
18          casex ({instr[31:25], instr[14:12], instr[6:0]})
19              'b00000000000110011:          // ADD
20                  res = a + b;
21              'b01000000000110011:          // SUB
22                  res = a - b;
23              'b00000000010110011:          // SLL
24                  res = a << b[4:0];
25              'b00000000100110011:          // SLT
26                  res = ($signed(a) < $signed(b)) ? 'b1 : 'b0;
27              'b00000000110110011:          // SLTU
28                  res = (a < b) ? 'b1 : 'b0;
29              'b00000001000110011:          // XOR
30                  res = a ^ b;
31              'b00000001010110011:          // SRL
32                  res = a >> b[4:0];
33              'b01000001010110011:          // SRA
34                  res = a >>> b[4:0];
35              'b00000001100110011:          // OR
36                  res = a | b;
37              'b00000001110110011:          // AND
38                  res = a & b;
39              'bxxxxxxx0000010011:          // ADDI
40                  res = a + b;
41              'bxxxxxxx0100010011:          // SLTI
42                  res = ($signed(a) < $signed(b)) ? 'b1 : 'b0;
43              'bxxxxxxx0100010011:          // SLTIU
44                  res = (a < b) ? 'b1 : 'b0;
45              'bxxxxxxx1000010011:          // XORI
46                  res = a ^ b;
47              'bxxxxxxx1100010011:          // ORI
48                  res = a | b;
49              'bxxxxxxx1110010011:          // ANDI
50                  res = a & b;
51              'b00000000010010011:          // SLLI
52                  res = a << b[4:0];
53              'b00000001010010011:          // SRLI
54                  res = a >> b[4:0];
55              'b01000001010010011:          // SRAI
56                  res = a >>> b[4:0];
```

```
57              default:
58                  res = a + b;
59          endcase
60          res_r <= res;
61      end
62
63  endmodule
```

7.1.3 Περιγραφή Αρχείου Καταχωρητών - regfile.v

```
1  module regfile
2  #(parameter BITS = 32)
3  (clk, ad1, ad2, adw, we, rs1, rs2, rsw);
4
5      input              clk;
6      input  [4:0]       ad1;  // rs1 address - read
7      input  [4:0]       ad2;  // rs2 address - read
8      input  [4:0]       adw;  // write port address
9      input              we;   // write enable
10     output [BITS-1:0] rs1;  // rs1 data
11     output [BITS-1:0] rs2;  // rs2 data
12     input  [BITS-1:0] rsw;  // write port data
13
14     // Register file storage
15     reg [BITS-1:0] registers[31:0]; // 32 x 32-bit
16     reg [BITS-1:0] rs1;
17     reg [BITS-1:0] rs2;
18
19     integer i;
20
21     initial begin
22         for (i = 0; i < 32; i = i + 1) begin
23             registers[i] <= 'b0;
24         end
25     end
26
27     always @(posedge clk) begin
28             registers[adw] = we ? rsw : registers[adw];
29             rs1 <= (ad1) ? registers[ad1] : 'b0;
30             rs2 <= (ad2) ? registers[ad2] : 'b0;
31     end
32 endmodule
```

7.1.4 Περιγραφή Μονάδας Διαχείρισης Άμεσων Δεδομένων - immgen.v

```
1  module immgen
2  #(parameter BITS = 32)
3  (clk, instr, imm_en, imm);
4
5      input              clk;
6      input  [31:0]      instr;
7      output             imm_en;
8      output [BITS-1:0] imm;
9
10     reg                imm_en;
11     reg    [BITS-1:0] imm;
12
13     initial begin
14         imm <= 'b0;
15         imm_en <= 0;
16     end
17
18     always @(posedge clk) begin
19         casex (instr[6:0])
20             7'b00x0011 : begin          // I-type - LB, LH, LW, ALU imm
21                              imm <= $signed(instr[31:20]);
22                              imm_en <= 1;
23                          end
24             7'b1100111: begin           // I-type - JALR
25                              imm <= $signed(instr[31:20]);
26                              imm_en <= 1;
27                          end
28             7'b0100011 : begin          // S-type - SW, SH, SB
29                              imm <= $signed({instr[31:25], instr[11:7]});
30                              imm_en <= 1;
```

```
                        end
            7'b0x10111 : begin          // U-type - AUIPC, LUI
                            imm <= $signed(instr[31:12]);
                            imm_en <= 1;
                        end
            7'b1100011 : begin          // B-type - BRANCH
                            imm <= $signed({instr[31], instr[7], instr[30:25],
                                            instr[11:8], 1'b0});
                            imm_en <= 1;
                        end
            7'b1101111 : begin          // J-type - JAL
                            imm <= $signed({instr[31], instr[19:12], instr[20],
                                            instr[30:21], 1'b0});
                            imm_en <= 1;
                        end
            default    : begin
                            imm <= 32'b0;
                            imm_en <= 0;
                        end
        endcase
    end

endmodule
```

7.1.5 Περιγραφή Μονάδας Διαχείρισης Διακλαδώσεων - branch.v

```
module branch
#(parameter BITS = 32)
(clk, instr, a, b, br);

    input         clk;
    input  [BITS-1:0] a;
    input  [BITS-1:0] b;
    input  [31:0]     instr;
    output        br;

    reg           br;

    initial
        br = 0;

    always @(posedge clk) begin
        casex ({instr[14:12], instr[6:0]})
            10'b0001100011 : begin // BEQ
                                br <= (a == b) ? 1 : 0;
                             end
            10'b0011100011 : begin // BNE
                                br <= (a != b) ? 1 : 0;
                             end
            10'b1001100011 : begin // BLT
                                br <= ($signed(a) <  $signed(b)) ? 1 : 0;
                             end
            10'b1011100011 : begin // BGE
                                br <= ($signed(a) >= $signed(b)) ? 1 : 0;
                             end
            10'b1101100011 : begin // BLTU
                                br <= (a <  b) ? 1 : 0;
                             end
            10'b1111100011 : begin // BGEU
                                br <= (a >= b) ? 1 : 0;
                             end
            10'bxxx1101111 : begin // JAL
                                br <= 1;
                             end
            10'b0001100111 : begin // JALR
                                br <= 1;
                             end
            default        : begin
                                br <= 0;
                             end
        endcase
    end

endmodule
```

7.1.6 Περιγραφή Μονάδας Διαχείρισης Κινδύνων - hazard.v

```
1  module hazard
2  #(parameter BITS = 32)
3  (clk, instr, br, stop_o, hz_instr_o);
4      input             clk;
5      input      [31:0] instr;
6      input             br;
7      output            stop_o;
8      output     [31:0] hz_instr_o;
9
10     wire [31:0] rd1;
11     wire [31:0] rd2;
12     wire [31:0] wr1;
13     wire [31:0] wr2;
14     reg  [31:0] wr3;
15     wire [31:0] prd1;
16     wire [31:0] prd2;
17     wire        halt1;
18     wire        halt2;
19     wire        halt_p;
20     wire        stop;
21     reg         br2;
22
23     reg  [31:0] prev;
24     reg  [31:0] prev2;
25     reg   [2:0] count;
26
27     initial begin
28         prev  <= 'b0;
29         count <= 'b0;
30         wr3   <= 'b0;
31         br2   <= 'b0;
32     end
33
34     always @(posedge clk) begin
35         wr3   <= wr2;
36         prev  <= br | br2   ? 32'b0 :
37                  count > 0 ? prev  : instr;
38         count <= (halt1 && count == 0) ? 'd2 :
39                  (halt2 && count == 0) ? 'd1 :
40                  count                 ? count - 1 : 'b0;
41         br2   <= br;
42     end
43
44     assign wr2 =  (count == 0) ?
45                   (prev[6:0]  == 7'b0110111 |   // LUI
46                    prev[6:0]  == 7'b0010111 |   // AUIPC
47                    prev[6:0]  == 7'b1101111 |   // JAL
48                    prev[6:0]  == 7'b1100111 |   // JALR
49                    prev[6:0]  == 7'b0000011 |   // LOAD
50                    prev[6:0]  == 7'b0010011 |   // ALU Imm
51                    prev[6:0]  == 7'b0110011 |   // ALU R-R
52                    prev[6:0]  == 7'b0001111) ?  // FENCE
53                                  ('b1 << prev[11:7])  & 'hFFFFFFFE : 'b0 : 'b0;
54
55     assign rd1  = (instr[6:0] == 7'b1100111 |   // JALR
56                    instr[6:0] == 7'b1100011 |   // BRANCH
57                    instr[6:0] == 7'b0000011 |   // LOAD
58                    instr[6:0] == 7'b0100011 |   // STORE
59                    instr[6:0] == 7'b0010011 |   // ALU Imm
60                    instr[6:0] == 7'b0110011 |   // ALU R-R
61                    instr[6:0] == 7'b0001111) ?  // FENCE
62                                  ('b1 << instr[19:15]) & 'hFFFFFFFE : 'b0;
63
64     assign prd1 =  (prev[6:0] == 7'b1100111 |   // JALR
65                     prev[6:0] == 7'b1100011 |   // BRANCH
66                     prev[6:0] == 7'b0000011 |   // LOAD
67                     prev[6:0] == 7'b0100011 |   // STORE
68                     prev[6:0] == 7'b0010011 |   // ALU Imm
69                     prev[6:0] == 7'b0110011 |   // ALU R-R
70                     prev[6:0] == 7'b0001111) ?  // FENCE
71                                  ('b1 << prev[19:15]) & 'hFFFFFFFE : 'b0;
72
73     assign rd2  = (instr[6:0] == 7'b1100011 |   // BRANCH
74                    instr[6:0] == 7'b0100011 |   // STORE
```

```
75                    instr[6:0] == 7'b0110011) ?  // ALU R-R
76                                  ('b1 << instr[24:20]) & 'hFFFFFFFE : 'b0;
77
78      assign prd2 = (prev[6:0]  == 7'b1100011 |   // BRANCH
79                     prev[6:0]  == 7'b0100011 |   // STORE
80                     prev[6:0]  == 7'b0110011) ?  // ALU R-R
81                                  ('b1 << prev[24:20]) & 'hFFFFFFFE : 'b0;
82
83
84
85      assign halt1      = (rd1 | rd2) & (wr2) ? 1 : 0;
86      assign halt2      = (rd1 | rd2) & (wr3) ? 1 : 0;
87      assign stop       = (halt1 | halt2 | (count != 0));
88      assign stop_o     = (halt1 | (halt2 & (count != 1)));
89      assign hz_instr_o = (br | br2)          ? 32'b0 :
90                          count > 0           ? hz_instr_o : prev;
91
92
93  endmodule
```

7.2 Εξομοίωση Υλικού και Testbench

Ακολουθεί η περιγραφή των αρχείων εξομοίωσης των λειτουργικών μονάδων, σε γλώσσα Verilog.

7.2.1 Testbench Μονάδας Μνήμης - memtest.v

```
1   module memtest();
2       reg         clk;
3       reg  [63:0] address;
4       reg  [63:0] data;
5       wire [63:0] data_wire;
6       reg  [2:0]  width;
7       reg         write_enable;
8
9       initial begin
10          $dumpfile("memtest.vcd");
11          $dumpvars(0, memtest);
12      end
13
14      always #5 clk = !clk;
15
16      mem #(.BITS(64)) memtst(clk, address, write_enable, width, data_wire);
17      defparam memtst.file = "memtest.mem";
18
19      assign data_wire = write_enable ? data : 'bz;
20
21      initial begin
22          clk = 0;
23          #0  address = 'h000000000000000;
24              write_enable = 0;
25              width = 'b000;
26          #10 width += 'b1;
27          #10 width += 'b1;
28          #10 width += 'b1;
29          #10 address += 'h8;
30              width = 'b000;
31          #10 width = 'b100;
32          #20 address = 'b0;
33              width = 'b010;
34              write_enable = 'b1;
35              data  = 'hAAAAAAAABBBBBBBB;
36          #10 write_enable = 'b0;
37          #10 width = 'b110;
38          #10 width = 'b011;
39          #20 $finish;
40      end
41  endmodule
```

7.2.2 Testbench Αριθμητικής - Λογικής Μονάδας - alutest.v

```
module alutest();
    reg         clk;
    reg  [31:0] operation;
    reg  [31:0] a;
    reg  [31:0] b;
    wire [31:0] result;

    initial begin
        $dumpfile("alutest.vcd");
        $dumpvars(0, alutest);
    end

    always #5 clk = !clk;

    alu #(.BITS(32)) alutst(clk, operation, a, b, result);

    initial begin
        clk = 0;
        #0  a = 'h87654321;
            b = 'h89abcdef;
            operation = 'b0000000xxxxxxxxxx111xxxxx0110011;
        #10 operation = 'b0000000xxxxxxxxxx000xxxxx0110011;
        #10 b = 'h00000010;
            operation = 'b0000000xxxxxxxxxx001xxxxx0110011;
        #20 $finish;
    end
endmodule
```

7.2.3 Testbench Αρχείου Καταχωρητών - regtest.v

```
module regtest();
    reg         clk;
    reg  [4:0]  address_a;
    reg  [4:0]  address_b;
    reg  [4:0]  address_write;
    reg         write_enable;
    wire [31:0] data_a;
    wire [31:0] data_b;
    reg  [31:0] data_write;

    initial begin
        $dumpfile("regtest.vcd");
        $dumpvars(0, regtest);
    end

    always #5 clk = !clk;

    regfile #(.BITS(32)) rftst(clk, address_a, address_b, address_write,
    write_enable, data_a, data_b, data_write);

    initial begin
        clk = 0;
        #0  data_write = 'haabbccdd;
            address_write = 'h01;
            write_enable = 'b1;
        #15 address_a = 'h01;
            address_b = 'h02;
            data_write = 'haaaaaaaa;
        #2  address_b = 'h01;
        #3  address_b = 'h02;
            data_write = 'hffffffff;
            address_write = 'h02;
            write_enable = 'b1;
        #8  data_write = 'ha0a0a0a0;
        #2  data_write = 'h12345678;
            address_write = 'h00;
        #10 address_b = 'h0;
            write_enable = 'b0;
        #10 address_a = 'h02;
            address_b = 'h01;
        #20 $finish;
    end
endmodule
```

7.2.4 Testbench Μονάδας Διαχείρισης Άμεσων Δεδομένων - immtest.v

```
module immtest();
    reg         clk;
    reg  [31:0] instruction;
    wire [31:0] immediate;
    wire        immediate_enable;

    initial begin
        $dumpfile("immtest.vcd");
        $dumpvars(0, immtest);
    end

    always #5 clk = !clk;

    immgen #(.BITS(32)) immtst(clk, instruction, immediate_enable, immediate);

    initial begin
        clk = 0;
        #0  instruction = 'bxxxxxxxxxxxxxxxxxxxxxxxxx0110011;  // R-type
        #10 instruction = 'b100010001000xxxxxxxxxxxxx0000011;  // I-type
        #10 instruction = 'b1000100xxxxxxxxxxxxx010000100011;  // S-type
        #10 instruction = 'b0100000xxxxxxxxxxxxx010011100011;  // B-type
        #10 instruction = 'b10101010101010101010xxxxx0110111;  // U-type
        #10 instruction = 'b00001000100110101010xxxxx1101111;  // J-type
        #20 $finish;
    end
endmodule
```

7.2.5 Testbench Μονάδας Διαχείρισης Διακλαδώσεων - branchtest.v

```
module branchtest();
    reg         clk;
    reg  [31:0] instruction;
    reg  [31:0] a;
    reg  [31:0] b;
    wire        branch;

    initial begin
        $dumpfile("branchtest.vcd");
        $dumpvars(0, branchtest);
    end

    always #5 clk = !clk;

    branch #(.BITS(32)) branchtst(clk, instruction, a, b, branch);

    initial begin
        clk = 0;
        #0  instruction = 'bxxxxxxxxxxxxxxxxxxxxxxxxx1101111;
        #10 instruction = 'bxxxxxxxxxxxxxxxxxxxxxxxxx1100111;
        #10 instruction = 'bxxxxxxxxxxxxxxxxx000xxxxx1100011;
            a = 'hA;
            b = 'hB;
        #10 instruction = 'bxxxxxxxxxxxxxxxxx001xxxxx1100011;
        #10 instruction = 'bxxxxxxxxxxxxxxxxx100xxxxx1100011;
        #10 instruction = 'bxxxxxxxxxxxxxxxxx101xxxxx1100011;
        #10 instruction = 'bxxxxxxxxxxxxxxxxx110xxxxx1100011;
        #10 instruction = 'bxxxxxxxxxxxxxxxxx111xxxxx1100011;
        #20 $finish;
    end

endmodule
```

7.2.6 Testbench Μοντέλου Επεξεργαστή - tb.v

```
module test();
    reg clk;

    initial begin
        $dumpfile("test.vcd");
        $dumpvars(0, test);
    end

    always #5 clk = !clk;

    cpu2 #(.BITS(32)) t1(clk);

    initial begin
            clk = 0;

      #200 $finish;

    end

endmodule
```

7.3 Αποτέλεσμα σύνθεσης

Ακολουθούν τα αξιόλογα μέρη της εξόδου του εργαλείου Xilinx ISE κατά την διαδικασία της σύνθεσης των περιγραφών υλικού.

```
=========================================================================
*                      Synthesis Options Summary                        *
=========================================================================
---- Source Parameters
Input File Name                    : "cpu2.prj"
Ignore Synthesis Constraint File   : NO

---- Target Parameters
Output File Name                   : "cpu2"
Output Format                      : NGC
Target Device                      : xc7a200t-3-fbg484

---- Source Options
Top Module Name                    : cpu2
Automatic FSM Extraction           : YES
FSM Encoding Algorithm             : Auto
Safe Implementation                : No
FSM Style                          : LUT
RAM Extraction                     : Yes
RAM Style                          : Auto
ROM Extraction                     : Yes
Shift Register Extraction          : YES
ROM Style                          : Auto
Resource Sharing                   : YES
Asynchronous To Synchronous        : NO
Shift Register Minimum Size        : 2
Use DSP Block                      : Auto
Automatic Register Balancing       : No

---- Target Options
LUT Combining                      : Auto
Reduce Control Sets                : Auto
Add IO Buffers                     : YES
Global Maximum Fanout              : 100000
Add Generic Clock Buffer(BUFG)     : 32
Register Duplication               : YES
Optimize Instantiated Primitives   : NO
Use Clock Enable                   : Auto
Use Synchronous Set                : Auto
Use Synchronous Reset              : Auto
Pack IO Registers into IOBs        : Auto
Equivalent register Removal        : YES

---- General Options
Optimization Goal                  : Speed
Optimization Effort                : 1
Power Reduction                    : NO
Keep Hierarchy                     : Yes
```

```
Netlist Hierarchy                   : As_Optimized
RTL Output                          : Yes
Global Optimization                 : AllClockNets
Read Cores                          : YES
Write Timing Constraints            : NO
Cross Clock Analysis                : NO
Hierarchy Separator                 : /
Bus Delimiter                       : <>
Case Specifier                      : Maintain
Slice Utilization Ratio             : 100
BRAM Utilization Ratio              : 100
DSP48 Utilization Ratio             : 100
Auto BRAM Packing                   : NO
Slice Utilization Ratio Delta       : 5

=========================================================================
=========================================================================
HDL Synthesis Report

Macro Statistics
# RAMs                                               : 4
 128x8-bit dual-port Read Only RAM                   : 2
 32x32-bit dual-port RAM                             : 2
# Adders/Subtractors                                 : 10
 32-bit adder                                        : 8
 32-bit subtractor                                   : 1
 4-bit subtractor                                    : 1
# Registers                                          : 151
 1-bit register                                      : 4
 3-bit register                                      : 1
 32-bit register                                     : 18
 8-bit register                                      : 128
# Latches                                            : 32
 1-bit latch                                         : 32
# Comparators                                        : 8
 3-bit comparator greater                            : 1
 32-bit comparator equal                             : 1
 32-bit comparator greater                           : 4
 5-bit comparator equal                              : 2
# Multiplexers                                       : 40
 32-bit 2-to-1 multiplexer                           : 32
 32-bit 7-to-1 multiplexer                           : 1
 8-bit 128-to-1 multiplexer                          : 4
 8-bit 2-to-1 multiplexer                            : 3
# Logic shifters                                     : 5
 32-bit shifter logical left                         : 4
 32-bit shifter logical right                        : 1
# Tristates                                          : 608
 1-bit tristate buffer                               : 96
 8-bit tristate buffer                               : 512
# Xors                                               : 1
 32-bit xor2                                         : 1

=========================================================================

Device utilization summary:
---------------------------

Selected Device : 7a200tfbg484-3


Slice Logic Utilization:
 Number of Slice Registers:            1498  out of  269200     0%
 Number of Slice LUTs:                 5316  out of  134600     3%
    Number used as Logic:              5220  out of  134600     3%
    Number used as Memory:               96  out of  46200     0%
       Number used as RAM:               64
       Number used as SRL:               32

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:   5559
   Number with an unused Flip Flop:    4061  out of   5559    73%
   Number with an unused LUT:           243  out of   5559     4%
   Number of fully used LUT-FF pairs:  1255  out of   5559    22%
   Number of unique control sets:       141
```

```
IO Utilization:
 Number of IOs:                          65
 Number of bonded IOBs:                  65  out of    285    22%

Specific Feature Utilization:
 Number of Block RAM/FIFO:                2  out of    365     0%
    Number using Block RAM only:          2
 Number of BUFG/BUFGCTRLs:                2  out of     32     6%

Primitive and Black Box Usage:
------------------------------
# BELS                           : 5533
#      GND                       : 5
#      INV                       : 5
#      LUT1                      : 29
#      LUT2                      : 107
#      LUT3                      : 209
#      LUT4                      : 327
#      LUT5                      : 608
#      LUT6                      : 3935
#      MUXCY                     : 168
#      MUXF7                     : 41
#      VCC                       : 5
#      XORCY                     : 94
# FlipFlops/Latches              : 1498
#      FD                        : 185
#      FDE                       : 1120
#      FDR                       : 129
#      FDRE                      : 32
#      LDC_1                     : 32
# RAMS                           : 35
#      RAM32X1D                  : 32
#      RAMB18E1                  : 3
# Shift Registers                : 32
#      SRLC16E                   : 32
# Clock Buffers                  : 2
#      BUFG                      : 1
#      BUFGP                     : 1
# IO Buffers                     : 64
#      OBUF                      : 64

Timing Summary:
---------------
Speed Grade: -3

   Minimum period: 6.700ns (Maximum Frequency: 149.245MHz)
   Minimum input arrival time before clock: No path found
   Maximum output required time after clock: 2.415ns
   Maximum combinational path delay: No path found
```

7.4 Κώδικας Assembler - Flex, Bison, C

Ακολουθεί ο πηγαίος κώδικας σε Flex, Bison, C του προγράμματος του Assembler.

7.4.1 Λεξικό Preprocessor - preproc.l

```
%option prefix="xx"
%option noyywrap
%option yylineno
%option nounput
%option noinput

%{
#include "preproc.tab.h"
// #define YY_DECL int yylex()
#define XX_DECL int xxlex()

%}

%option noyywrap
%option yylineno

empty    ^\s*[\n]+
```

```
18 linecom  ^[;][^\n]*\n
19 comment  [;][^\n]*
20 ws       [ \t]+
21 nl       [\n]+
22
23 reg      zero|ra|sp|gp|tp|t[0-6]|s[0-9]+|fp|a[0-7]|x[0-9]+
24 bin      0b[01]+
25 hex      0x[0-9a-fA-F]+
26 dec      [0-9]+
27
28 instr    lui|auipc|jal|jalr|beq|bne|blt|bge|bltu|bgeu|lb|lh|lw|lbu|lhu|sb|sh|sw|
       addi|slti|sltiu|xori|ori|andi|slli|srli|srai|add|sub|sll|slt|sltu|xor|srl|sra|
       or|and|fence|ecall|ebreak
29 pseudo   nop
30
31 label    [_a-zA-Z][_a-zA-Z0-9]{1,31}
32
33 %%
34 {linecom} {};
35 {comment} {};
36 {empty}   {};
37 {ws}      {};
38 {nl}      return P_NL;
39 {bin}     { xxtext[xxleng] = '\0'; xxlval.sval = strdup(xxtext); return P_OP; }
40 {hex}     { xxtext[xxleng] = '\0'; xxlval.sval = strdup(xxtext); return P_OP; }
41 {dec}     { xxtext[xxleng] = '\0'; xxlval.sval = strdup(xxtext); return P_OP; }
42 {reg}     { xxtext[xxleng] = '\0'; xxlval.sval = strdup(xxtext); return P_REG; }
43
44 "("       return P_LPAR;
45 ")"       return P_RPAR;
46 ","       return P_COMMA;
47 ":"       return P_COLON;
48
49 {instr}   { xxtext[xxleng] = '\0'; xxlval.sval = strdup(xxtext); return P_INSTR; }
50 {pseudo}  { xxtext[xxleng] = '\0'; xxlval.sval = strdup(xxtext); return P_PSEUDO; }
51 {label}   { xxtext[xxleng] = '\0'; xxlval.sval = strdup(xxtext); return P_LABEL; }
```

7.4.2 Γραμματική Preprocessor - preproc.y

```
1  %define api.prefix {xx}
2
3  %{
4  #define XXERROR_VERBOSE 1
5  //#define YYERROR_VERBOSE 1
6
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <string.h>
10 #include "sym.h"
11
12 extern int xxlex();
13 extern int xxparse();
14 extern int xxlineno;
15
16 //extern int yylex();
17 //extern int yyparse();
18 //extern int yylineno;
19
20 extern FILE* xxin;
21 extern FILE* xxout;
22
23 // extern FILE* yyin;
24 // extern FILE* yyout;
25
26 extern void xxerror(const char* s);
27 //extern void yyerror(const char* s);
28
29 extern sym* table;
30 extern int symbols;
31
32 unsigned int mem_ad = 0;
33
34 void label_handler(char* l) {
35     table = (sym *)realloc(table, ++symbols * sizeof(sym));
36     table[symbols-1].ad = mem_ad;
37     strcpy(table[symbols-1].name, l);
```

```
}

void ptoi(char* p) {
    if (!strcmp(p, "nop")){
        fprintf(xxout, "addi x0, x0, 0\n");
    }
}

%}
%union {
    char* sval;
}

%token       P_NL;
%token       P_LPAR;
%token       P_RPAR;
%token       P_COMMA;
%token       P_COLON;

%token <sval> P_OP;
%token <sval> P_REG;
%token <sval> P_INSTR;
%token <sval> P_PSEUDO;
%token <sval> P_LABEL;


%%
file: instructions
;

instructions: instruction P_NL instructions
            | instruction P_NL
            | instruction
;

instruction: instr
           {
               mem_ad += 4;
           }
;

instr: P_INSTR P_REG P_COMMA P_OP P_LPAR P_REG P_RPAR // ok
     {
         fprintf(xxout, "%s %s, %s(%s)\n", $1, $2, $4, $6);
     }
     | P_INSTR P_REG P_COMMA P_REG P_COMMA P_REG // ok
     {
         fprintf(xxout, "%s %s, %s, %s\n", $1, $2, $4, $6);
     }
     | P_INSTR P_REG P_COMMA P_REG P_COMMA P_OP // ok
     {
         fprintf(xxout, "%s %s, %s, %s\n", $1, $2, $4, $6);
     }
     | P_INSTR P_REG P_COMMA P_REG P_COMMA P_LABEL // ok
     {
         fprintf(xxout, "%s %s, %s, %s\n", $1, $2, $4, $6);
     }
     | P_INSTR P_REG P_COMMA P_OP // ok
     {
         fprintf(xxout, "%s %s, %s\n", $1, $2, $4);
     }
     | P_INSTR P_REG P_COMMA P_LABEL // ok
     {
         fprintf(xxout, "%s %s, %s\n", $1, $2, $4);
     }
     | P_LABEL P_COLON // ok
     {
         fprintf(xxout, "; %s:\n", $1);
         label_handler($1);
         mem_ad -= 4;
     }
     | P_PSEUDO
     {
         ptoi($1);
     }
;
```

```
114
115 %%
116
117 void xxerror(const char *s) {
118   printf("[Preprocessor] [Line %d] %s\n", xxlineno, s);
119   exit(1);
120 }
```

7.4.3 Λεξικό Assembler - asm.l

```
1  %option noyywrap
2  %option yylineno
3  %option nounput
4  %option noinput
5
6
7  %{
8  #include "asm.tab.h"
9  #define YY_DECL int yylex()
10
11 static char* tmp;
12
13 // register to integer
14 int rtoi(char *yy){
15     if (!strcmp(yy, "zero")) { return 0; }
16     if (!strcmp(yy, "ra"))   { return 1; }
17     if (!strcmp(yy, "sp"))   { return 2; }
18     if (!strcmp(yy, "gp"))   { return 3; }
19     if (!strcmp(yy, "tp"))   { return 4; }
20     if (!strcmp(yy, "fp"))   { return 8; }
21     if (yy[0] == 't') {
22         if (atoi(yy+1) < 3) { return atoi(yy+1) + 5; }
23         if (atoi(yy+1) < 7) { return atoi(yy+1) + 25; }
24         return -1;
25     }
26     if (yy[0] == 's') {
27         if (atoi(yy+1) < 2)  { return atoi(yy+1) + 8; }
28         if (atoi(yy+1) < 12) { return atoi(yy+1) + 16; }
29         return -1;
30     }
31     if (yy[0] == 'a') {
32         if (atoi(yy+1) < 8)  { return atoi(yy+1) + 10; }
33         return -1;
34     }
35     if (yy[0] == 'x') {
36         if (atoi(yy+1) < 32) { return atoi(yy+1); }
37         return -1;
38     }
39     return -1;
40 }
41
42 %}
43
44 %option noyywrap
45 %option yylineno
46
47 empty    ^\s*[\n]+
48 linecom  ^[;][^\n]*\n
49 comment  [;][^\n]*
50 ws       [ \t]+
51 nl       [\n]+
52 reg      (zero|ra|sp|gp|tp|t[0-6]|s[0-9]+|fp|a[0-7]|x[0-9]+)
53 bin      0b[01]+
54 hex      0x[0-9a-fA-F]+
55 dec      [0-9]+
56 num      {bin}{dec}{hex}
57 label    [_a-zA-Z][_a-zA-Z0-9]{1,31}
58
59 %%
60 {linecom} {};
61 {comment} {};
62 {empty}   {};
63 {ws}      {};
64 {nl}      return NL;
65 {bin}     { yylval.imm = strtoul(yytext+2, &tmp, 2);  return NUM; }
66 {dec}     { yylval.imm = strtoul(yytext,   &tmp, 10); return NUM; }
```

```
{hex}     { yylval.imm = strtoul(yytext+2, &tmp, 16); return NUM; }
{reg}     { yylval.rs  = rtoi(yytext);                return REG; }


"("       return LPAR;
")"       return RPAR;
","       return COMMA;
":"       return COLON;

"lui"     return LUI;
"auipc"   return AUIPC;

"jal"     return JAL;
"jalr"    return JALR;
"beq"     return BEQ;
"bne"     return BNE;
"blt"     return BLT;
"bge"     return BGE;
"bltu"    return BLTU;
"bgeu"    return BGEU;

"lb"      return LB;
"lh"      return LH;
"lw"      return LW;
"ld"      return LD;
"lbu"     return LBU;
"lhu"     return LHU;
"lwu"     return LWU;

"sb"      return SB;
"sh"      return SH;
"sw"      return SW;

"addi"    return ADDI;
"slti"    return SLTI;
"sltiu"   return SLTIU;
"xori"    return XORI;
"ori"     return ORI;
"andi"    return ANDI;
"slli"    return SLLI;
"srli"    return SRLI;
"srai"    return SRAI;

"add"    return ADD;
"sub"    return SUB;
"sll"    return SLL;
"slt"    return SLT;
"sltu"   return SLTU;
"xor"    return XOR;
"srl"    return SRL;
"sra"    return SRA;
"or"     return OR;
"and"    return AND;

"fence"  return FENCE;
"ecall"  return ECALL;
"ebreak" return EBREAK;

{label}  { yytext[yyleng] = '\0'; yylval.sval = strdup(yytext); return LABEL; }
```

7.4.4 Γραμματική Assembler - asm.y

```
%{
#define YYERROR_VERBOSE 1

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "sym.h"

extern int yylex();
extern int yyparse();
extern int yylineno;

extern FILE* yyin;
extern FILE* yyout;
```

```
15
16 extern int symbols;
17 extern sym* table;
18
19 unsigned int ins = 0;
20 unsigned int mem = 0;
21
22 void yyerror(const char* s);
23
24 void writeins(unsigned int ins) {
25     for (int i = 0; i < 4; i++) {
26         fprintf(yyout, "%02x ", ins & 0xff);
27         ins >>= 8;
28     }
29     mem += 4;
30     fprintf(yyout, "\n");
31 }
32
33 unsigned int label_resolve(char* l) {
34     for (int i = 0; i < symbols; i++) {
35         if (!strcmp(table[i].name, l)) {
36             return table[i].ad - mem;
37         }
38     }
39     printf("[Assembler] [Line %d] Jump to Label %s - Label not declared\n",
       yylineno, l);
40     exit(1);
41     return 0;
42 }
43
44 %}
45 %union {
46     unsigned int imm;
47     unsigned int rs;
48     char* sval;
49 }
50
51 %token       NL;
52 %token       LPAR;
53 %token       RPAR;
54 %token       COMMA;
55 %token       COLON;
56
57 %token <imm> NUM;
58 %token <rs>  REG;
59
60 %token LUI;
61 %token AUIPC;
62 %token JAL;
63 %token JALR;
64 %token BEQ;
65 %token BNE;
66 %token BLT;
67 %token BGE;
68 %token BLTU;
69 %token BGEU;
70 %token LB;
71 %token LH;
72 %token LW;
73 %token LD;
74 %token LBU;
75 %token LHU;
76 %token LWU;
77 %token SB;
78 %token SH;
79 %token SW;
80 %token ADDI;
81 %token SLTI;
82 %token SLTIU;
83 %token XORI;
84 %token ORI;
85 %token ANDI;
86 %token SLLI;
87 %token SRLI;
88 %token SRAI;
89 %token ADD;
```

```
%token SUB;
%token SLL;
%token SLT;
%token SLTU;
%token XOR;
%token SRL;
%token SRA;
%token OR;
%token AND;
%token FENCE;
%token ECALL;
%token EBREAK;

%token <sval> LABEL;

%%
file: instructions
;

instructions: instructions instruction NL
            | instruction NL
            | instruction
;

instruction: label
           | lui
           | auipc
           | jal
           | jalr
           | beq
           | bne
           | blt
           | bge
           | bltu
           | bgeu
           | lb
           | lh
           | lw
           | ld
           | lbu
           | lhu
           | lwu
           | sb
           | sh
           | sw
           | addi
           | slti
           | sltiu
           | xori
           | ori
           | andi
           | slli
           | srli
           | srai
           | add
           | sub
           | sll
           | slt
           | sltu
           | xor
           | srl
           | sra
           | or
           | and
           | fence
           | ecall
           | ebreak
;

label: LABEL COLON NL
;

lui: LUI REG COMMA NUM
   {
       ins =  0b0110111;
       ins += $2 << 7;
```

```
166         ins += $4 << 12;
167         writeins(ins);
168     }
169 ;
170
171 auipc: AUIPC REG COMMA NUM
172      {
173           ins =  0b0010111;
174           ins += $2 << 7;
175           ins += $4 << 12;
176           writeins(ins);
177      }
178 ;
179
180 jal: JAL REG COMMA NUM
181    {
182         ins =  0b1101111;
183         ins += $2     << 7;
184         ins += (($4 >> 12) & 0xff) << 12;
185         ins += (($4 >> 11) & 0x1)  << 20;
186         ins += (($4 >> 1) & 0x3ff) << 21;
187         ins += (($4 >> 20) & 0x1)  << 31;
188         printf("[Assembler] [Line %d] [Warning] Jump with offset %x without using
      label\n", yylineno, (unsigned int)$4);
189         writeins(ins);
190    }
191    | JAL REG COMMA LABEL
192    {
193         unsigned int lr = label_resolve($4);
194         ins =  0b1101111;
195         ins += $2     << 7;
196         ins += ((lr >> 12) & 0xff) << 12;
197         ins += ((lr >> 11) & 0x1)  << 20;
198         ins += ((lr >> 1) & 0x3ff) << 21;
199         ins += ((lr >> 20) & 0x1)  << 31;
200         writeins(ins);
201    }
202 ;
203
204 jalr: JALR REG COMMA NUM LPAR REG RPAR
205     {
206          ins =  0b1100111;
207          ins += $2    << 7;
208          ins += 0b000 << 12;
209          ins += $6    << 15;
210          ins += $4    << 20;
211          printf("[Assembler] [Line %d] [Warning] Jump with offset %x without using
      label\n", yylineno, (unsigned int)$4);
212          writeins(ins);
213     }
214     | JALR REG COMMA LABEL LPAR REG RPAR
215     {
216          unsigned int lr = label_resolve($4);
217          ins =  0b1100111;
218          ins += $2    << 7;
219          ins += 0b000 << 12;
220          ins += $6    << 15;
221          ins += lr    << 20;
222          writeins(ins);
223     }
224
225 ;
226
227 beq: BEQ REG COMMA REG COMMA NUM
228    {
229         ins =  0b1100011;
230         ins += (($6 >> 11) & 0x1) << 7;
231         ins += (($6 >> 1)  & 0xf)  << 8;
232         ins += 0b000 << 12;
233         ins += $2    << 15;
234         ins += $4    << 20;
235         ins += (($6 >> 5)  & 0x3f) << 25;
236         ins += (($6 >> 12) & 0x1)  << 31;
237         printf("[Assembler] [Line %d] [Warning] Branch with offset %x without using
      label\n", yylineno, (unsigned int)$6);
238         writeins(ins);
```

```
239     }
240     | BEQ REG COMMA REG COMMA LABEL
241     {
242         unsigned int lr = label_resolve($6);
243         ins =  0b1100011;
244         ins += ((lr >> 11) & 0x1) << 7;
245         ins += ((lr >> 1)  & 0xf)  << 8;
246         ins += 0b000 << 12;
247         ins += $2    << 15;
248         ins += $4    << 20;
249         ins += ((lr >> 5)  & 0x3f) << 25;
250         ins += ((lr >> 12) & 0x1)  << 31;
251         writeins(ins);
252     }
253 ;
254
255 bne: BNE REG COMMA REG COMMA NUM
256     {
257         ins =  0b1100011;
258         ins += (($6 >> 11) & 0x1) << 7;
259         ins += (($6 >> 1)  & 0xf) << 8;
260         ins += 0b001 << 12;
261         ins += $2    << 15;
262         ins += $2    << 20;
263         ins += (($6 >> 5)  & 0x3f) << 25;
264         ins += (($6 >> 12) & 0x1)  << 31;
265         writeins(ins);
266     }
267     | BNE REG COMMA REG COMMA LABEL
268     {
269         unsigned int lr = label_resolve($6);
270         ins =  0b1100011;
271         ins += ((lr >> 11) & 0x1) << 7;
272         ins += ((lr >> 1)  & 0xf) << 8;
273         ins += 0b001 << 12;
274         ins += $2    << 15;
275         ins += $2    << 20;
276         ins += ((lr >> 5)  & 0x3f) << 25;
277         ins += ((lr >> 12) & 0x1)  << 31;
278         writeins(ins);
279     }
280 ;
281
282 blt: BLT REG COMMA REG COMMA NUM
283     {
284         ins =  0b1100011;
285         ins += (($6 >> 11) & 0x1) << 7;
286         ins += (($6 >> 1)  & 0xf) << 8;
287         ins += 0b100 << 12;
288         ins += $2    << 15;
289         ins += $2    << 20;
290         ins += (($6 >> 5)  & 0x3f) << 25;
291         ins += (($6 >> 12) & 0x1)  << 31;
292         printf("[Assembler] [Line %d] [Warning] Branch with offset %x without using
     label\n", yylineno, (unsigned int)$6);
293         writeins(ins);
294     }
295     | BLT REG COMMA REG COMMA LABEL
296     {
297         unsigned int lr = label_resolve($6);
298         ins =  0b1100011;
299         ins += ((lr >> 11) & 0x1) << 7;
300         ins += ((lr >> 1)  & 0xf) << 8;
301         ins += 0b100 << 12;
302         ins += $2    << 15;
303         ins += $2    << 20;
304         ins += ((lr >> 5)  & 0x3f) << 25;
305         ins += ((lr >> 12) & 0x1)  << 31;
306         writeins(ins);
307     }
308 ;
309
310 bge: BGE REG COMMA REG COMMA NUM
311     {
312         ins =  0b1100011;
313         ins += (($6 >> 11) & 0x1) << 7;
```

```
314         ins += (($6 >> 1)  & 0xf)  << 8;
315         ins += 0b101 << 12;
316         ins += $2    << 15;
317         ins += $4    << 20;
318         ins += (($6 >> 5)  & 0x3f) << 25;
319         ins += (($6 >> 12) & 0x1)  << 31;
320         printf("[Assembler] [Line %d] [Warning] Branch with offset %x without using
        label\n", yylineno, (unsigned int)$6);
321         writeins(ins);
322     }
323     | BGE REG COMMA REG COMMA LABEL
324     {
325         unsigned int lr = label_resolve($6);
326         ins =  0b1100011;
327         ins += ((lr >> 11) & 0x1) << 7;
328         ins += ((lr >> 1)  & 0xf) << 8;
329         ins += 0b101 << 12;
330         ins += $2    << 15;
331         ins += $2    << 20;
332         ins += ((lr >> 5)  & 0x3f) << 25;
333         ins += ((lr >> 12) & 0x1)  << 31;
334         writeins(ins);
335     }
336 ;
337
338 bltu: BLTU REG COMMA REG COMMA NUM
339     {
340         ins =  0b1100011;
341         ins += (($6 >> 11) & 0x1) << 7;
342         ins += (($6 >> 1)  & 0xf)  << 8;
343         ins += 0b110 << 12;
344         ins += $2    << 15;
345         ins += $4    << 20;
346         ins += (($6 >> 5)  & 0x3f) << 25;
347         ins += (($6 >> 12) & 0x1)  << 31;
348         printf("[Assembler] [Line %d] [Warning] Branch with offset %x without using
        label\n", yylineno, (unsigned int)$6);
349         writeins(ins);
350     }
351     | BLTU REG COMMA REG COMMA LABEL
352     {
353         unsigned int lr = label_resolve($6);
354         ins =  0b1100011;
355         ins += ((lr >> 11) & 0x1) << 7;
356         ins += ((lr >> 1)  & 0xf) << 8;
357         ins += 0b110 << 12;
358         ins += $2    << 15;
359         ins += $2    << 20;
360         ins += ((lr >> 5)  & 0x3f) << 25;
361         ins += ((lr >> 12) & 0x1)  << 31;
362         writeins(ins);
363     }
364 ;
365
366 bgeu: BGEU REG COMMA REG COMMA NUM
367     {
368         ins =  0b1100011;
369         ins += (($6 >> 11) & 0x1) << 7;
370         ins += (($6 >> 1)  & 0xf)  << 8;
371         ins += 0b111 << 12;
372         ins += $2    << 15;
373         ins += $4    << 20;
374         ins += (($6 >> 5)  & 0x3f) << 25;
375         ins += (($6 >> 12) & 0x1)  << 31;
376         printf("[Assembler] [Line %d] [Warning] Branch with offset %x without using
        label\n", yylineno, (unsigned int)$6);
377         writeins(ins);
378     }
379     | BGEU REG COMMA REG COMMA LABEL
380     {
381         unsigned int lr = label_resolve($6);
382         ins =  0b1100011;
383         ins += ((lr >> 11) & 0x1) << 7;
384         ins += ((lr >> 1)  & 0xf) << 8;
385         ins += 0b111 << 12;
386         ins += $2    << 15;
```

```
387         ins += $2    << 20;
388         ins += ((lr >> 5)  & 0x3f) << 25;
389         ins += ((lr >> 12) & 0x1)  << 31;
390         writeins(ins);
391     }
392 ;
393
394 lb: LB REG COMMA NUM LPAR REG RPAR
395   {
396       ins =  0b0000011;
397       ins += $2    << 7;
398       ins += 0b000 << 12;
399       ins += $6    << 15;
400       ins += $4    << 20;
401       writeins(ins);
402   }
403 ;
404
405 lh: LH REG COMMA NUM LPAR REG RPAR
406   {
407       ins =  0b0000011;
408       ins += $2    << 7;
409       ins += 0b001 << 12;
410       ins += $6    << 15;
411       ins += $4    << 20;
412       writeins(ins);
413   }
414 ;
415
416 lw: LW REG COMMA NUM LPAR REG RPAR
417   {
418       ins =  0b0000011;
419       ins += $2    << 7;
420       ins += 0b010 << 12;
421       ins += $6    << 15;
422       ins += $4    << 20;
423       writeins(ins);
424   }
425 ;
426
427 ld: LD REG COMMA NUM LPAR REG RPAR
428   {
429       ins =  0b0000011;
430       ins += $2    << 7;
431       ins += 0b011 << 12;
432       ins += $6    << 15;
433       ins += $4    << 20;
434       writeins(ins);
435   }
436 ;
437
438 lbu: LBU REG COMMA NUM LPAR REG RPAR
439   {
440       ins =  0b0000011;
441       ins += $2    << 7;
442       ins += 0b100 << 12;
443       ins += $6    << 15;
444       ins += $4    << 20;
445       writeins(ins);
446   }
447 ;
448
449 lhu: LHU REG COMMA NUM LPAR REG RPAR
450   {
451       ins =  0b0000011;
452       ins += $2    << 7;
453       ins += 0b101 << 12;
454       ins += $6    << 15;
455       ins += $4    << 20;
456       writeins(ins);
457   }
458 ;
459
460 lwu: LWU REG COMMA NUM LPAR REG RPAR
461   {
462       ins =  0b0000011;
```

```
463       ins += $2    << 7;
464       ins += 0b110 << 12;
465       ins += $6    << 15;
466       ins += $4    << 20;
467       writeins(ins);
468   }
469 ;
470
471 sb: SB REG COMMA NUM LPAR REG RPAR
472   {
473       ins =  0b0100011;
474       ins += ($4 & 0x1f)  << 7;
475       ins += 0b000 << 12;
476       ins += $6    << 15;
477       ins += $2    << 20;
478       ins += ($4 >> 5) << 25;
479       writeins(ins);
480   }
481 ;
482
483 sh: SH REG COMMA NUM LPAR REG RPAR
484   {
485       ins =  0b0100011;
486       ins += ($4 & 0x1f)  << 7;
487       ins += 0b001 << 12;
488       ins += $6    << 15;
489       ins += $2    << 20;
490       ins += ($4 >> 5) << 25;
491       writeins(ins);
492   }
493 ;
494
495 sw: SW REG COMMA NUM LPAR REG RPAR
496   {
497       ins =  0b0100011;
498       ins += ($4 & 0x1f)  << 7;
499       ins += 0b010 << 12;
500       ins += $6    << 15;
501       ins += $2    << 20;
502       ins += ($4 >> 5) << 25;
503       writeins(ins);
504   }
505 ;
506
507 addi: ADDI REG COMMA REG COMMA NUM
508     {
509         ins =  0b0010011;
510         ins += $2    << 7;
511         ins += 0b000 << 12;
512         ins += $4    << 15;
513         ins += $6    << 20;
514         writeins(ins);
515     }
516 ;
517
518 slti: SLTI REG COMMA REG COMMA NUM
519     {
520         ins =  0b0010011;
521         ins += $2    << 7;
522         ins += 0b010 << 12;
523         ins += $4    << 15;
524         ins += $6    << 20;
525         writeins(ins);
526     }
527 ;
528
529 sltiu: SLTIU REG COMMA REG COMMA NUM
530      {
531          ins =  0b0010011;
532          ins += $2    << 7;
533          ins += 0b011 << 12;
534          ins += $4    << 15;
535          ins += $6    << 20;
536          writeins(ins);
537      }
538 ;
```

```
539
540 xori: XORI REG COMMA REG COMMA NUM
541     {
542         ins =  0b0010011;
543         ins += $2    << 7;
544         ins += 0b100 << 12;
545         ins += $4    << 15;
546         ins += $6    << 20;
547         writeins(ins);
548     }
549 ;
550
551 ori: ORI REG COMMA REG COMMA NUM
552    {
553        ins =  0b0010011;
554        ins += $2    << 7;
555        ins += 0b110 << 12;
556        ins += $4    << 15;
557        ins += $6    << 20;
558        writeins(ins);
559    }
560 ;
561
562 andi: ANDI REG COMMA REG COMMA NUM
563     {
564         ins =  0b0010011;
565         ins += $2    << 7;
566         ins += 0b111 << 12;
567         ins += $4    << 15;
568         ins += $6    << 20;
569         writeins(ins);
570     }
571 ;
572
573 slli: SLLI REG COMMA REG COMMA NUM
574     {
575         ins =  0b0010011;
576         ins += $2    << 7;
577         ins += 0b001 << 12;
578         ins += $4    << 15;
579         ins += ($6 & 0x1f) << 20;
580         writeins(ins);
581     }
582 ;
583
584 srli: SRLI REG COMMA REG COMMA NUM
585     {
586         ins =  0b0010011;
587         ins += $2    << 7;
588         ins += 0b101 << 12;
589         ins += $4    << 15;
590         ins += ($6 & 0x1f) << 20;
591         writeins(ins);
592     }
593 ;
594
595 srai: SRAI REG COMMA REG COMMA NUM
596     {
597         ins =  0b0010011;
598         ins += $2    << 7;
599         ins += 0b101 << 12;
600         ins += $4    << 15;
601         ins += ($6 & 0x1f) << 20;
602         ins += 0b0100000 << 25;
603         writeins(ins);
604     }
605 ;
606
607 add: ADD REG COMMA REG COMMA REG
608    {
609        ins =  0b0110011;
610        ins += $2    << 7;
611        ins += 0b000 << 12;
612        ins += $4    << 15;
613        ins += $6    << 20;
614        writeins(ins);
```

```
615     }
616 ;
617
618 sub: SUB REG COMMA REG COMMA REG
619     {
620         ins =  0b0110011;
621         ins += $2    << 7;
622         ins += 0b000 << 12;
623         ins += $4    << 15;
624         ins += $6    << 20;
625         ins += 0b0100000 << 25;
626         writeins(ins);
627     }
628 ;
629
630 sll: SLL REG COMMA REG COMMA REG
631     {
632         ins =  0b0110011;
633         ins += $2    << 7;
634         ins += 0b001 << 12;
635         ins += $4    << 15;
636         ins += $6    << 20;
637         writeins(ins);
638     }
639 ;
640
641 slt: SLT REG COMMA REG COMMA REG
642     {
643         ins =  0b0110011;
644         ins += $2    << 7;
645         ins += 0b010 << 12;
646         ins += $4    << 15;
647         ins += $6    << 20;
648         writeins(ins);
649     }
650 ;
651
652 sltu: SLTU REG COMMA REG COMMA REG
653      {
654          ins =  0b0110011;
655          ins += $2    << 7;
656          ins += 0b011 << 12;
657          ins += $4    << 15;
658          ins += $6    << 20;
659          writeins(ins);
660      }
661 ;
662
663 xor: XOR REG COMMA REG COMMA REG
664     {
665         ins =  0b0110011;
666         ins += $2    << 7;
667         ins += 0b100 << 12;
668         ins += $4    << 15;
669         ins += $6    << 20;
670         writeins(ins);
671     }
672 ;
673
674 srl: SRL REG COMMA REG COMMA REG
675     {
676         ins =  0b0110011;
677         ins += $2    << 7;
678         ins += 0b101 << 12;
679         ins += $4    << 15;
680         ins += $6    << 20;
681         writeins(ins);
682     }
683 ;
684
685 sra: SRA REG COMMA REG COMMA REG
686     {
687         ins =  0b0110011;
688         ins += $2    << 7;
689         ins += 0b101 << 12;
690         ins += $4    << 15;
```

```
691         ins += $6    << 20;
692         ins += 0b0100000 << 25;
693         writeins(ins);
694     }
695 ;
696
697 or: OR REG COMMA REG COMMA REG
698   {
699       ins =  0b0110011;
700       ins += $2    << 7;
701       ins += 0b110 << 12;
702       ins += $4    << 15;
703       ins += $6    << 20;
704       writeins(ins);
705   }
706 ;
707
708 and: AND REG COMMA REG COMMA REG
709    {
710         ins =  0b0110011;
711         ins += $2    << 7;
712         ins += 0b111 << 12;
713         ins += $4    << 15;
714         ins += $6    << 20;
715         writeins(ins);
716    }
717 ;
718
719 fence: FENCE
720 ;
721
722 ecall: ECALL
723      {
724          ins  = 0b1110011;
725          writeins(ins);
726      }
727 ;
728
729 ebreak: EBREAK
730       {
731           ins  = 0b1110011;
732           ins += 1 << 20;
733           writeins(ins);
734       }
735 ;
736
737 %%
738
739 void yyerror(const char *s) {
740   printf("[Assembler] [Line %d] %s\n", yylineno, s);
741   exit(1);
742 }
```

7.4.5 Ορισμός symbol table - sym.h

```
1 #ifndef _SYM_H
2 #define _SYM_H
3 #endif
4
5 typedef struct smb {
6     unsigned int ad;
7     char name[32];
8 } sym;
```

7.4.6 Βασικό πρόγραμμα - main2.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include "sym.h"
5 #include "asm.tab.h"
6 #include "preproc.tab.h"
7 #define YY_DECL int yylex()
8 #define XX_DECL int xxlex()
9 #define DEBUG 0
```

```
10
11 extern int xxlex();
12 extern int xxparse();
13 extern int xxlineno;
14
15 extern int yylex();
16 extern int yyparse();
17 extern int yylineno;
18
19 extern FILE* yyin;
20 extern FILE* xxin;
21 extern FILE* xxout;
22 extern FILE* yyout;
23
24 sym* table;
25 int symbols;
26
27 int main(int argc, char** argv) {
28     #ifdef YYDEBUG
29         yydebug = DEBUG;
30     #endif
31     #ifdef XXDEBUG
32         xxdebug = DEBUG;
33     #endif
34
35     char fout[32];
36     strcpy(fout, argv[2]);
37     symbols = 0;
38     table = (sym *)malloc(1 * sizeof(sym));
39     xxin = fopen(argv[1], "r");
40     char* fname = strcat(argv[1], ".out");
41     //printf("fname is : %s\n", fname);
42     xxout = fopen(fname, "w");
43     //printf("Preproc...\n");
44     xxparse();
45     //for (int i = 0; i < symbols; i++) {
46         //printf("; [0x%08x] %s\n", table[i].ad, table[i].name);
47     //}
48     fclose(xxout);
49     //printf("Preproc Done\n");
50     //printf("Asm...\n");
51     yyin = fopen(fname, "r");
52     yyout = fopen(fout, "w");
53     fprintf(yyout, "@0\n");
54     yyparse();
55     fprintf(yyout, "// SYMBOL TABLE\n");
56     for (int i = 0; i < symbols; i++) {
57         fprintf(yyout, "// [0x%08x] %s\n", table[i].ad, table[i].name);
58     }
59     fclose(yyout);
60     //printf("Asm Done\n");
61
62     return 0;
63 }
```

Κεφαλαιο 8
ΠΑΡΑΡΤΗΜΑ - ΕΡΓΑΛΕΙΑ

Για την εκπόνηση της παραπάνω διπλωματικής εργασίας χρησιμοποιήθηκαν τα παρακάτω **εργαλεία**:

vim Επεξεργαστής κειμένου
Χρησιμοποιήθηκε κατά κόρον για την ανάπτυξη του πηγαίου κώδικα.

Visual Studio Code Επεξεργαστής κειμένου - Περιβάλλον προγραμματισμού
Χρησιμοποιήθηκε κατά την συγγραφή του παρόντος τεχνικού κειμένου.

GTKWave Πρόγραμμα προβολής κυματομορφών
Χρησιμοποιήθηκε κατά την εξομοίωση και αποσφαλμάτωση του μοντέλου του επεξεργαστή και για την παραγωγή των διαγραμμάτων κυματομορφών που βρίσκονται στο παρόν τεχνικό κείμενο.

Xilinx ISE Περιβάλλον ανάπτυξης, εξομοίωσης και σύνθεσης υλικού
Χρησιμοποιήθηκε για την διαδικασία της σύνθεσης του μοντέλου του επεξεργαστή.

draw.io Επεξεργαστής διαγραμμάτων
Χρησιμοποιήθηκε για την δημιουργία των αρχιτεκτονικών διαγραμμάτων και των διαγραμμάτων ροής του παρόντος τεχνικού κειμένου.

Github Σύστημα ελέγχου εκδόσεων
Χρησιμοποιήθηκε για την ασφαλή αποθήκευση της προόδου της διπλωματικής εργασίας και επιπλέον για τον έλεγχο των διάφορων εκδόσεων που προέκυπταν κατά την ανάπτυξη και υλοποίησή της.

Χρησιμοποιήθηκαν επίσης οι παρακάτω **τεχνολογίες και γλώσσες**:

Icarus Verilog Γλώσσα περιγραφής υλικού
Χρησιμοποιήθηκε η γλώσσα περιγραφής υλικού Verilog για την περιγραφή των μονάδων του μοντέλου του επεξεργαστή και συγκεκριμένα ο εξομοιωτής Icarus Verilog.

Flex - Bison - C Βιβλιοθήκες της C για ανάπτυξη parser
Χρησιμοποιήθηκαν για την ανάπτυξη του Assembler.

Latex (texlive) - pdflatex - bibtex Γλώσσα markup
Χρησιμοποιήθηκε για την συγγραφή του παρόντος τεχνικού κειμένου και πιο συγκεκριμένα, ο διερμηνευτής pdflatex. Για την διαχείριση της βιβλιογραφίας χρησιμοποιήθηκε το bibtex backend.

Makefile Αυτοματοποίηση εκτελέσεων εντολών
Χρησιμοποιήθηκε για την αυτοματοποίηση εκτελέσεων εντολών για την μεταγλώττιση του Assembler, την εξομοίωση του μοντέλου του επεξεργαστή και την αυτόματη δημιουργία του εγγράφου κατά την διάρκεια της συγγραφής.

Φάνηκε χρήσιμη η παρακάτω **ιστοσελίδα**:

venus RISC-V Assembler (https://www.kvakil.me/venus/ [kva])
Χρησιμοποιήθηκε πριν την ανάπτυξη του Assembler για παραγωγή απλών διανυσμάτων για εκτέλεση από τον επεξεργαστή και για την αποσφαλμάτωση και σύγκριση των αποτελεσμάτων του υλοποιημένου Assembler.

Κεφαλαιο 9
ΒΙΒΛΙΟΓΡΑΦΙΑ

[HP11] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[For14] Behrouz Forouzan. *Foundations of Computer Science*. Andover: Cengage Learning, 2014. ISBN: 9781408044117.

[CPA16] Christopher Celio, David Patterson, and Krste Asanovic. "The Berkeley Out-of-Order Machine (BOOM) Design Specification". In: *University of California, Berkeley* (2016).

[Wat+16] Andrew Waterman et al. "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1". In: (2016).

[Nik17] Dimitrios Nikolos. *Computer Architecture*. Pan. Papakonstantinou, 2017. ISBN: 978-618-83197-0-7.

[CSAa] MIT CSAIL. *Riscy Processors - Open-Sourced RISC-V Processors*. URL: https://github.com/csail-csg/riscy. (accessed: 11/10/2021).

[CSAb] MIT CSAIL. *RiscyOO: RISC-V Out-of-Order Processors*. URL: https://github.com/csail-csg/riscy-OOO. (accessed: 11/10/2021).

[Dig] Western Digital. *RISC-V — Western Digital*. URL: https://www.westerndigital.com/company/innovations/risc-v. (accessed: 11/10/2021).

[Int] RISC-V International. *Members - RISC-V International*. URL: https://riscv.org/members/. (accessed: 11/10/2021).

[JG] Aaron Wisner Jacob Glueck. *The Lizard Core*. URL: https://github.com/cornell-brg/lizard. (accessed: 11/10/2021).

[kva] kvakil.me. *venus*. URL: https://www.kvakil.me/venus/. (accessed: 11/10/2021).

[SiFa] SiFive. *RISC-V Core IP - SiFive*. URL: https://www.sifive.com/risc-v-core-ip. (accessed: 11/10/2021).

[SiFb] SiFive. *SiFive Performance P550 Core Sets New Standard as Highest Performance RISC-V Processor IP - SiFive*. URL: https://www.sifive.com/risc-v-core-ip. (accessed: 11/10/2021).

[Sil] Think Silicon. *NEOX GPUs - Think Silicon*. URL: https://www.think-silicon.com/neox-graphics. (accessed: 11/10/2021).