Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων
Εργαστηριο Υπολογιστικων Συστηματων

Implementing a TLB Generator with Chisel for RISC-V architecture

Διπλωματικη Εργασια

του

ΝΙΚΟΛΑΟΥ ΧΑΡΑΛΑΜΠΟΥ ΠΑΠΑΔΟΠΟΥΛΟΥ

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π

Εργαστηριο Υπολογιστικων Συστηματων
Αθήνα, Νοέμβριος 2019

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Implementing a TLB Generator with Chisel for RISC-V architecture

Διπλωματικη Εργασια

του

ΝΙΚΟΛΑΟΥ ΧΑΡΑΛΑΜΠΟΥ ΠΑΠΑΔΟΠΟΥΛΟΥ

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 11 Νοεμβρίου 2019.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Νεκτάριος Κοζύρης	Γεώργιος Γκούμας	Διονύσιος Πνευματικάτος
Καθηγητής Ε.Μ.Π	Επ. Καθηγητής Ε.Μ.Π	Καθηγητής Ε.Μ.Π

Αθήνα, Νοέμβριος 2019

(Υπογραφή)

.....

ΝΙΚΟΛΑΟΣ ΧΑΡΑΛΑΜΠΟΣ ΠΑΠΑΔΟΠΟΥΛΟΣ

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Περίληψη

Ο RISC-V είναι μία ανοιχτή Αρχιτεκτονική Συνόλου Εντολών που αναπτύχθηκε από το Πανεπιστήμιο της Καλιφόρνια, Μπέρκλεϋ. Αρχικά ο RISC-V σχεδιάστηκε για ερευνητικούς και εκπαιδευτικούς σκοπούς, αλλά η ανοιχτή φύση του οδηγεί μέρα με την μέρα στην ευρεία αποδοχή του και από την βιομηχανία. Ο RISC-V έχει υλοποιηθεί σε μικροαρχιτεκτονική από οργανισμούς ανά τον κόσμο, με τον Rocket Chip Generator να είναι μία από τις βασικότερες υλοποιήσεις. Ο Rocket Chip Generator είναι μία βιβλιοθήκη παραμετροποιήσιμων μερών επεξεργαστών τα οποία συνδυάζονται για να παράξουν μία ευρεία γκάμα υπολογιστικών συστημάτων, από μικρούς ενσωματωμένους επεξεργαστές μέχρι πολύπλοκα πολυπύρηνα συστήματα. Είναι υλοποιημένος στην Chisel, μία γλώσσα ανάπτυξης υλικού που επιτρέπει την δημιουργία πολύπλοκων αλλά ευέλικτων σχεδίων κυκλωμάτων για ASIC καθώς και για FPGA. Στην παρούσα εργασία ερευνούμε την ροή ανάπτυξης υλικού στον Rocket Chip Generator με εργαλεία προσομοίωσης υλικού και συγκεκριμένα τον Verilator καθώς και FPGA για γρήγορο έλεγχο. Επικεντρωνόμαστε στην μονάδα διαχείρισης μνήμης του Rocket Chip Generator, και ειδικότερα στον Translation Lookaside Buffer (TLB). Το TLB είναι μία μικρή κρυφή μνήμη που κρατάει τις μεταφράσεις από εικονικές σε φυσικές διευθύνσεις μνήμης για τις οποίες υπεύθυνη είναι η μονάδα διαχείρισης μνήμης. Το TLB του Rocket Chip Generator είναι πλήρως-συσχετιστικό και παραμετροποιήσιμο ως προς τον αριθμό των θέσεων. Το πρόβλημα που προκύπτει από τα πλήρως-συσχετιστικά TLB εντοπίζεται στην μεγάλη αύξηση του μεγέθους του κυκλώματος αναζήτησης μετάφρασης εντός της κρυφής μνήμης όσο αυξάνονται οι θέσεις. Επειδή το TLB είναι στο Critical Path του επεξεργαστή μπορεί να μειώσει δραστικά την επίδοση του, με την περίπτωση του Rocket Chip Generator να εμφανίζει μείωση 50% στον χρονισμό του Xilinx ZCU102 με αύξηση μεγέθους από 32 σε 512 θέσεις. Παραμετροποιούμε το TLB του Rocket Chip Generator μετατρέποντας το σε συχετιστικό-ανά-σετ με ευέλικτους τους αριθμούς set και ways. Η γεννήτρια TLB που σχεδιάσαμε μπορεί να παράξει από άμεσης-απεικόνισης έως πλήρως-συσχετιστικό TLB αναλόγως την εφαρμογή και τις επιδόσεις που μας ενδιαφέρουν. Τέλος, εξετάζουμε την σχεδίαση του παραμετροποιήσιμου TLB που υλοποιήσαμε έναντι του αρχικού σε σχέση με την κατανάλωση πόρων του FPGA, το critical path καθώς και την απόδοση τους χρησιμοποιώντας μετροπρογράμματα της σουίτας SPEC2006.

Λέξεις Κλειδιά

RISC-V, Σχεδίαση Υλικού, Rocket Chip Generator, Chisel, FPGA, Verilator, TLB, Μονάδα Διαχείριση Μνήμης

Abstract

RISC-V is an open Instruction Set Architecture (ISA) developed by UC Berkeley. Initially designed for research and education, the open nature of RISC-V promotes collaboration both in industry and academia. RISC-V has been implemented and fabricated by many institutions around the world, one of those implementation being the open-source Rocket Chip Generator. Rocket Chip Generator is a library of parametrized processor parts put together to form a powerful tool that can produce a wide variety of processor designs ranging from tiny embedded processors to complex multi-core systems. It is written in Chisel, a hardware design language that features advanced circuit design and generation for both ASIC and FPGA targets. In this thesis we will explore the hardware design flow using Rocket Chip Generator with hardware emulation tools and FPGA for fast testing. We will demonstrate the powerful features of Chisel by analyzing, implementing and testing a Translation Lookaside Buffer (TLB) generator. The TLB acts as a fast cache for virtual to physical address translations handled by the Memory Management Unit (MMU) of a processor. Rocket Chip Generator uses a fully-associative TLB with customizable number of entries. When the TLB entries are many, the tag matching circuit becomes complex and large because of the fully-associative mechanism . The TLB lies on the critical path of the processor and can quickly degrade perfrormance: There is 50% reduction in clock rate from 32 to 512 entries on the Rocket Chip Generator. We modify the TLB and implement a TLB generator with variable sets and ways. Our TLB Generator can produce from direct-mapped to fully-associative designs depending on our needs on performance and area. We assess our design against the original TLB in terms of FPGA resources, critical path and performance based on benchmarks of the SPEC2006 suite.

Keywords

RISC-V, Hardware Design, Rocket Chip Generator, Chisel, FPGA, Spike Simulator, TLB, Memory Management

στην οικογένεια μου

Ευχαριστίες

Θα ήθελα καταρχήν να ευχαριστήσω τον καθηγητή κ.Νεκτάριο Κοζύρη για την επίβλεψη αυτής της διπλωματικής εργασίας και για την δυνατότητα που μου έδωσε να την εκπονήσω στο Εργαστήριο Υπολογιστικών Συστημάτων. Τον ευχαριστώ ιδιαίτερα για την εμπιστοσύνη που μου έδειξε καθώς και για τις ευκαιρίες που μου παρείχε.

Επίσης, ευχαριστώ από καρδιάς τους ερευνητές Βασίλη Καρακώστα, Χλόη Αλβέρτη και Κωνσταντίνο Νίκα που με εισήγαγαν στον εξαιρετικά ενδιαφέροντα χώρο του RISC-V, για τις πολύτιμες γνώσεις, την διαρκή καθοδήγησή, και την ενθάρρυνση τους κατά την διάρκεια της εργασίας αυτής.

Για τα χρόνια που πέρασα φοιτητής στην Αθήνα, θέλω να ευχαριστήσω όλους τους φίλους και γνωστούς για τις ωραίες στιγμές που περάσαμε! Ευχαριστώ την οικογένεια μου για την υποστήριξη και την αγάπη που μου προσέφεραν όλα αυτά τα χρόνια, και που είναι πάντα δίπλα μου σε ότι και αν κάνω. Τέλος, ευχαριστώ πολύ την Άλμπα για την αμέριστη κατανόηση, υπομονή και αγάπη που μου έδειξε στα δύσκολα και στα εύκολα, κατά την διάρκεια των σπουδών μου και όχι μόνο.

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

Κεφάλαιο 1

Εισαγωγή

Η ιδέα των επεξεργαστών RISC (Reduced Instruction Set Computer) ξεκίνησε από τον David Patterson το 1980 στο UC Berkeley. Έπειτα από παρατήρηση, διαπιστώθηκε ότι το λειτουργικό σύστημα Unix όταν μεταφραζόταν για τον CISC (Complex Instruction Set Computer) μικροεπεξεργαστή Motorola 68000, χρησιμοποιούσε μονάχα το 30% του συνόλου των δυνατών εντολών [18]. Μεγάλο ποσοστό λοιπόν από το κύκλωμα αποκωδικοποίησης εντολών έμενε αδρανές καθώς οι εντολές αυτές δεν εκτελούνταν ποτέ. Η πρόταση του RISC συνοψίζεται στην δημιουργία όσο το δυνατόν μικρότερου συνόλου εντολών, με ανταλλαγή των πόρων που απελευθερώνονται σε διαφορετικές μονάδες του επεξεργαστή όπως είναι το αρχείο καταχωρητών.

Το 2010, έπειτα από 30 χρόνια της πρώτης έκδοσης RISC-I, ξεκίνησε η ανάπτυξη του RISC-V από το UC Berkeley ενώ από το 2016 είναι διεθνής προσπάθεια. Ο RISC-V είναι μία ανοιχτή Αρχιτεκτονική Συνόλου Εντολών, Instruction Set Architecture (ISA), ένα standard το οποίο ορίζει τις εντολές, τους καταχωρητές και τις βασικές λειτουργίες ενός επεξεργαστή.

Κατά την διάρκεια της ανάπτυξης του RISC-V ISA η απόδειξη της ορθής λειτουργίας γίνεται με τον Rocket Chip Generator, έναν παραμετροποιήσιμο μικροεπεξεργαστή η σχεδίαση του οποίου γίνεται σύμφωνα με τα επίσημα Specifications του RISC-V ISA. Ο Rocket Chip Generator έχει εξελιχθεί σε μία ευέλικτη γεννήτρια συστημάτων System-on-Chip (SoC) η οποία μπορεί να παράξει από χαμηλής κατανάλωσης και επίδοσης συστήματα μέχρι πολύπλοκα και υψηλών επιδόσεων συστήματα. Την υλοποίηση του Rocket Chip Generator ευνόησε η γλώσσα περιγραφής υλικού Chisel, η οποία είναι πρακτικά μία βιβλιοθήκη περιγραφής κυκλωμάτων ενσωματωμένη στην γλώσσα Scala. Η σχεδίαση της Chisel βασίζεται σε προγραμματιστικές τεχνικές και μοντέλα γλωσσών προγραμματισμού υψηλού επιπέδου για την παραγωγικότερη δημιουργία ευέλικτων και παραμετροποιήσιμων γεννητριών κυκλωμάτων.

1.1 Αντικείμενο της διπλωματικής

Στην εργασία αυτή θα ασχοληθούμε με την μονάδα διαχείρισης μνήμης του Rocket Chip Generator, και ειδικότερα με το Translation Lookaside Buffer (TLB). Το TLB είναι μία μικρή κρυφή μνήμη που κρατάει τις μεταφράσεις από εικονικές σε φυσικές διευθύνσεις μνήμης για τις

οποίες υπεύθυνη είναι η μονάδα διαχείρισης μνήμης. Το TLB του Rocket Chip Generator είναι πλήρως συσχετιστικό και παραμετροποιήσιμο ως προς τον αριθμό των θέσεων. Το πρόβλημα που προκύπτει από τα πλήρως συσχετιστικά TLB εντοπίζεται στην μεγάλη αύξηση μεγέθους και πολυπλοκότητας του κυκλώματος αναζήτησης μετάφρασης εντός της κρυφής μνήμης όσο αυξάνονται οι θέσεις. Το κύκλωμα αυτό πρέπει να ελέγξει όλες τις θέσεις του TLB καθώς η αναζητούμενη μετάφραση εικονικής σελίδας μπορεί να βρίσκεται σε οποιαδήποτε θέση. Επειδή το TLB είναι στο critical path του επεξεργαστή μπορεί να μειώσει δραστικά την απόδοση του επεξεργαστή, λόγω χειρότερου χρονισμού. Παραμετροποιούμε το TLB του Rocket Chip Generator μετατρέποντας το σε set-associative με ευέλικτους τους αριθμούς set και ways. Η οργάνωση set-associative βελτιώνει την καθυστέρηση λόγω άμεσης επιλογής του set όπου πρέπει να βρίσκεται η μετάφραση που αναζητούμε, και πλήρης αναζήτησης εντός των λίγων θέσεων του. Η γεννήτρια TLB που σχεδιάσαμε μπορεί να παράξει από άμεσης-απεικόνισης έως πλήρως-συσχετιστικό TLB αναλόγως την εφαρμογή και τις επιδόσεις που μας ενδιαφέρουν. Τέλος, εξετάζουμε την σχεδίαση του παραμετροποιήσιμου TLB που υλοποιήσαμε έναντι του αρχικού, σε σχέση με την κατανάλωση πόρων του FPGA, το Critical Path καθώς και την επίδοση τους χρησιμοποιώντας μετροπρογράμματα της σουίτας SPEC2006.

1.2 Οργάνωση του τόμου

Η παρούσα εργασία είναι οργανωμένη στα παρακάτω κεφάλαια.

- Στο κεφάλαιο 2 παρουσιάζουμε το απαραίτητο θεωρητικό υπόβαθρο των τεχνολογιών που θα χρησιμοποιήσουμε οι οποίες είναι η αρχιτεκτονική RISC-V, η γλώσσα περιγραφής υλικού Chisel και η γεννήτρια μικροεπεξεργαστών Rocket Chip Generator. Στο τέλος του κεφαλαίου παρουσιάζουμε αναλυτικά την μονάδα διαχείρισής μνήμης του Rocket Chip Generator.

- Στο κεφάλαιο 3 παρουσιάζουμε αναλυτικά τη μεθοδολογία που χρησιμοποιήσαμε για την ανάλυση, την ανάπτυξη και τον έλεγχο του σχεδιασμού στον Rocket Chip Generator. Θα μιλήσουμε για τα εργαλεία που είναι απαραίτητα όπως τον Verilator, το Xilinx ZCU102 και το Buildroot μεταξύ άλλων.

- Στο κεφάλαιο 4 θα εξετάσουμε αναλυτικά την σχεδίαση του αρχικού TLB του Rocket Chip Generator. Έπειτα θα σχεδιάσουμε και θα υλοποιήσουμε ένα παραμετροποιήσιμο TLB με βάση τις παραδοχές και τις σχεδιαστικές επιλογές μας.

- Στο κεφάλαιο 5 θα παρουσιάσουμε την επίδοση του περαμετροποιήσιμου TLB σε σχέση με το αρχικό TLB σε σχέση με μετρικές πόρων της πλατφόρμας FPGA και μετρικές επίδοσης χρησιμοποιώντας την σουίτα μετροπρογραμμάτων SPEC2006.

- Τέλος, στο κεφάλαιο 6 θα μελετήσουμε τις πιθανές μελλοντικές προεκτάσεις της εργασίας αυτής.

Κεφάλαιο 2

Θεωρητικό υπόβαθρο

Στο κεφάλαιο αυτο θα παρουσιαστούν αναλυτικά οι τεχνολογίες που χρησιμοποιήθηκαν στην παρούσα διπλωματική, δηλαδή η ανοιχτή Αρχιτεκτονική Συνόλου Εντολών RISC-V, η γλώσσα περιγραφής υλικού Chisel καθώς και η γεννήτρια μικροεπεξεργαστών Rocket Chip Generator η οποία αποτελεί υλοποίηση της αρχιτεκτονικής RISC-V με χρήση της γλώσσας Chisel. Τέλος, θα παρουσιαστεί η λειτουργία του συστήματος διαχείρισης εικονικής μνήμης από τη σκοπιά του RISC-V ISA και του Rocket Chip Generator.

2.1 RISC-V ISA

2.1.1 Ανοιχτή Αρχιτεκτονική Συνόλου Εντολών RISC-V

Η Αρχιτεκτονική Συνόλου Εντολών (Instruction Set Architecture, ISA) αποτελεί την διεπαφή μεταξύ λογισμικού και υλικού σε ένα υπολογιστικό σύστημα. Είναι ένα σύνολο εντολών και συμβάσεων οι οποίες αφορούν τον προγραμματισμό, τους καταχωρητές, τους τύπους δεδομένων καθώς και την διαχείριση μνήμης [19]. Η μικροαρχιτεκτονική ενός συστήματος η οποία αφορά τις τεχνικές σχεδίασης ενός επεξεργαστή διαφέρει από το ISA: Το ISA ορίζει ένα μοντέλο το οποίο η υλοποιείται σε κάποια μικροαρχιτεκτονική.

Το RISC-V ISA ξεκίνησε να αναπτύσσεται στο Πανεπιστήμιο της Καλιφόρνια, Μπέρκλεϋ το 2010, και από το 2016 λαμβάνει διεθνή προσοχή. Όπως αναφέρει το όνομα, είναι μία αρχιτεκτονική RISC (Reduced Instruction Set Computer), λίγων, απλών και γενικών εντολών με το **RV32I: RISC-V Base Integer ISA** να περιέχει μόλις 48 εντολές. Το RISC-V ISA είναι πρακτικά ένα Standard καθώς απομονώνει την αρχιτεκτονική από την υλοποίηση της υποκείμενης μικροαρχιτεκτονικής. Είναι ανοικτό, με την έννοια ότι δεν υπόκειται σε περιοριστικές άδειες είτε αντίτιμο για την απόκτηση/υλοποίηση του. Η ανάπτυξη και διαχείριση του RISC-V ISA γίνεται μέσω του **RISC-V Foundation**[1].

Το RISC-V Instruction Set Manual ορίζει 2 Specifications, το **Volume I: User-Level ISA** [6] στο οποίο περιέχονται τα απαραίτητα καθώς και προαιρετικά σετ εντολών που μπορεί να περιέχει ένα πρόγραμμά που τρέχει σε χώρο χρήστη, καθώς και το **Volume II: Priv-**

[1] https://www.riscv.org

ileged Architecture στο οποίο περιγράφονται επιπλέον επίπεδα εκτέλεσης καθώς και τα χαρακτηριστικά τους για την διαχείρισή ενός συστήματος.

2.1.2 RISC-V User-Level ISA

Το User-Level ISA αποτελείται από τα **RV32/64/128 Integer Base ISA** με address space 32, 64 και 128 bit αντίστοιχα. Αναλόγως του address space που επιλέγεται για κάποιο σύστημα, το Integer Base ISA είναι απαραίτητο για την υλοποίηση εφαρμογών χώρου χρήστη. Το User-level ISA Specification ορίζει επιπλέον **Extensions** τα οποία είτε προσθέτουν εξειδικευμένες εντολές είτε προσθέτουν επιπλέον λειτουργικότητα με αποτέλεσμα την βελτίωση της επίδοσης του συστήματος. Να σημειωθεί ότι τα επίσημα Extensions υλοποιούνται και ελέγχονται από το RISC-V Foundation, υπάρχει όμως επιπλέον **reserved opcode space** για υλοποίηση **custom Extensions**, δηλαδή οποιοσδήποτε επιθυμεί να προσθέσει λειτουργικότητα για την βελτίωση της επίδοσης συγκεκριμένης εφαρμογής μπορεί να την υλοποιήσει παράλληλα στο official ISA.

Επεκτασιμότητα

Κάποια από τα σημαντικότερα Extensions τα οποία χρησιμοποιούνται στην παρούσα εργασία είναι:

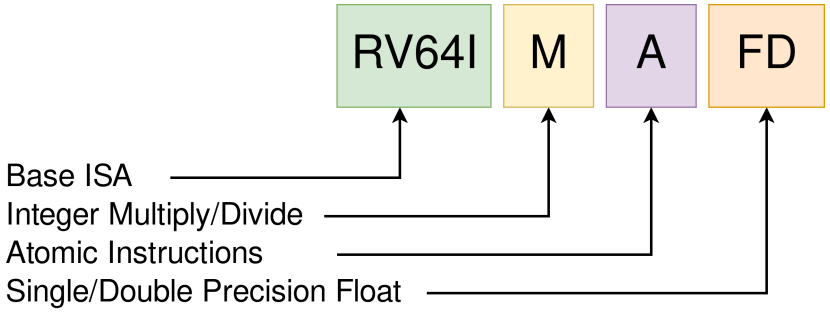
Extension	Περιγραφή
M	Integer Multiplication and Division
A	Atomic Instructions
F	Single-Precision Floating Point Instructions
D	Double-Precision Floating Point Instructions

Πίνακας 2.1: Βασικότερα Extensions

Σε περίπτωση που επιθυμούμε να τρέξουμε μία εφαρμογή χώρου χρήστη η οποία έχει μεταφραστεί (compiled) με extensions τα οποία δεν είναι υλοποιημένα από την μικροαρχιτεκτονική που στοχεύουμε, θα πρέπει τα extensions αυτά να υλοποιούνται σε **emulation routines** οι οποίες θα τρέχουν έπειτα από illegal instruction exception. Παραδείγματος χάρη, το V (Vector) extension προσθέτει υποστήριξη αριθμητικών πράξεων και διαχείρισης διανυσμάτων για αύξηση της επίδοσης λόγω παραλληλισμού δεδομένων. Σε περίπτωση που δεν είναι υλοποιημένο το V extension στην μικροαρχιτεκτονική και χρειαστεί να τρέξουμε εφαρμογή που το χρησιμοποιεί, τότε μία ρουτίνα V Extension emulation θα εξομοιώσει τις Vector εντολές με απλές εντολές του base ISA, θυσιάζοντας προφανώς την επίδοση με την συμβατότητα. Τα παραπάνω δεν ισχύουν για το A Extension (Atomic Instructions) για το οποίο πρέπει να υπάρχει υποστήριξη στην μικροαρχιτεκτονική.

Ο συμβολισμός ενός RISC-V Core σε σχέση με τα extensions που περιέχει φαίνεται στο σχήμα 2.1.

Τα Extensions που είναι απαραίτητα για την υποστήριξη ενός Λειτουργικού Συστήματος



Σχήμα 2.1: Συμβολισμός RV64IMAFD Core

όπως για παράδειγμα το Linux είναι τα **I, M, A, F, D** με συμβολισμό **G (General)**. Το **RV64G ISA** είναι το βασικό ISA που θα χρησιμοποιηθεί κατά την διάρκεια της παρούσας εργασίας.

2.1.3 RISC-V Privileged Architecture

Εκτός από την υποστήριξη του υπολογιστικού μοντέλου γενικού σκοπού, των εφαρμογών δηλαδή που τρέχουν στον χώρο χρήστη, υπάρχει ανάγκη για επιπλέον λειτουργικότητα για την διαχείριση του συστήματος. Οι σημαντικότεροι λόγοι για την ύπαρξη επιπρόσθετων επιπέδων εκτέλεσης με επιπλέον προνόμια είναι η διαχείριση και προστασία κοινών πόρων όπως η μνήμη και οι συσκευές, η υποστήριξη multitasking, καθώς και η απόκρυψη της υποκείμενης υλοποίησης του υλικού, για μεγαλύτερη ευχέρεια ανάπτυξης λογισμικού[2].

Privilege Levels

Το Privileged Architecture Specification ορίζει επιπλέον Privilege Levels, όπως βλέπουμε στον παρακάτω πίνακα:

Επίπεδο	Όνομα	Συντομογραφία
2	Machine	M-mode
1	Supervisor	S-mode
0	User	U-mode

Πίνακας 2.2: Privilege Levels

Το κάθε επίπεδο προσθέτει επιπλέον **λειτουργικότητα, εντολές και καταχωρητές κατάστασης και ελέγχου, Control and Status Registers (CSRs)**. Το κάθε επίπεδο έχει πλήρη πρόσβαση στα χαμηλότερα επίπεδα ενώ το αντίθετο δεν ισχύει γενικά: υπό προϋποθέσεις είτε για την προώθηση των διακοπών σε χαμηλότερα επίπεδα [3], είτε την πρόσβαση στους performance counters.

[2]Βλέπε παράδειγμα για το V (Vector) Extension παράγραφος 2.1.2, υποενότητα Επεκτασιμότητα
[3]Βλέπε N Extension: User-Level Interrupt Handling

Συνδυάζοντας τα Privilege Levels μπορούμε να κατασκευάσουμε από απλά και μικρά συστήματα, μέχρι υψηλής πολυπλοκότητας και λειτουργικότητας συστήματα. Ο πίνακας 2.3 συνοψίζει τους συνδυασμούς αναλόγως των Privilege Levels.

Supported Modes	Παράδειγμα Συστήματος
M	Απλά Ενσωματωμένα Συστήματα
M + U	Ασφαλή Ενσωματωμένα Συστήματα
M + U + S	Συστήματα με υποστήριξη λειτουργικού συστήματος
M + Virtual[U + S]	Συστήματα με υποστήριξη για πολλαπλα λειτουργικά συστήματα

Πίνακας 2.3: Πιθανά σενάρια στησίματος ενός RISC-V συστήματος

Machine-Mode

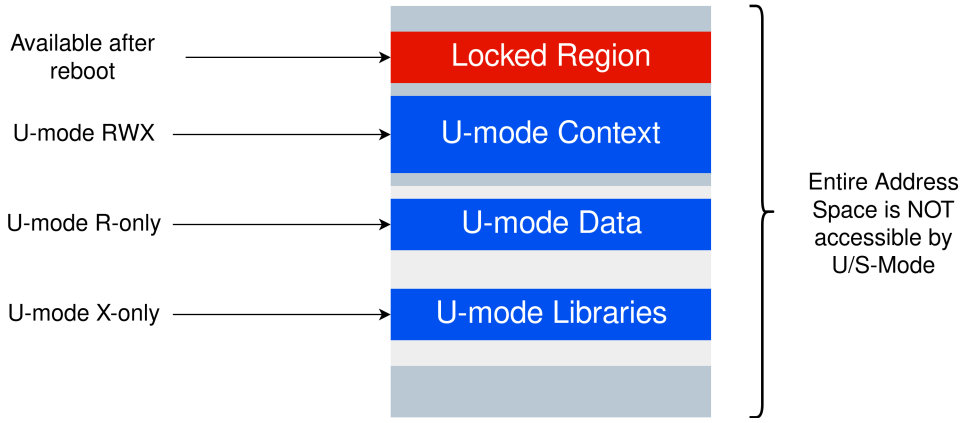
Το **M-mode** πρέπει να υπάρχει σε κάθε υλοποίηση καθώς είναι το μόνο επίπεδο το οποίο έχει πλήρη πρόσβαση στην αρχιτεκτονική (πρακτικά στους CSRs). Επειδή έχει πλήρη έλεγχο όμως, κώδικας είτε λάθος είτε κακόβουλος που τρέχει σε M-mode καθιστά το σύστημα ανασφαλές. Το M-mode προσθέτει βασικούς καταχωρητές για τον έλεγχο του συστήματος, Control and Status Registers (CSRs), καθώς και επιπλέον καταχωρητές μέτρησης επίδοσης (Performance Counters) μαζί με αντίστοιχους Hardware Performace Monitor Event Registers οι οποίοι φορτώνονται με συγκεκριμένο κωδικό γεγονότος υλικού για το κάθε hardware performance event ενημερώνοντας τους performance counters. Επιπλέον, το M-mode προσθέτει υποστήριξη για διακοπές/traps απαραίτητες για την ύπαρξη συσκευών και για την διαχείριση σφαλμάτων και γεγονότων όπως τα:

- **Access Faults**: Σφάλματα σελίδας και ελλιπών δικαιωμάτων πρόσβασης στην μνήμη
- **Breakpoints**: Χρησιμεύουν στην ευκολότερη αποσφαλμάτωση
- **Environment Calls**: Κλήσεις συστήματος για διαχείριση ενεργειών που απαιτούν περισσότερα δικαιώματα από ανώτερη βαθμίδα
- **Illegal Instructions**: Εντολές που προσπαθούν να εκτελεστούν με λιγότερα δικαιώματα, ανύπαρκτες εντολές καθώς και εντολές που δεν είναι υλοποιημένες στην μικροαρχιτεκτονική. Έγκυρες αλλά όχι υλοποιημένες στην μικροαρχιτεκτονική εντολές εξυπηρετούνται από ρουτίνες εξομοίωσης.
- **Misaligned address accesses**: Σφάλματα στοίχισης διευθύνσεων, στην αρχιτεκτονική RISC-V δεν επιτρέπονται misaligned διευθύνσεις για την απλοποίηση της μικροαρχιτεκτονικής.

User-mode

Το U-mode επιτρέπει την εκτέλεση του αναξιόπιστου κώδικα στο επίπεδο του χώρου χρήστη για την προστασία της λειτουργίας του συστήματος. Κώδικας που τρέχει σε U-mode

δεν έχει πρόσβαση στους privileged CSRs καθώς και δεν επιτρέπεται να εκτελέσει privileged εντολές. Το U-mode προσθέτει τον μηχανισμό της φυσικής προστασίας μνήμης, **Physical Memory Protection (PMP)**. Το PMP χωρίζει την μνήμη του συστήματος σε περιοχές στις οποίες ο κώδικας του U-mode έχει συγκεκριμένα δικαιώματα πρόσβασης όπως φαίνεται στο σχήμα 2.2 [16]. Μία περιοχή PMP μπορεί επίσης να κλειδωθεί και για το M-mode, με αποτέλεσμα να είναι προσβάσιμη έπειτα από επανεκκίνηση του συστήματος. Ο μηχανισμός PMP επομένως προσθέτει επιπλέον βαθμίδες ασφαλείας για το σύστημα. Πέρα από το PMP ορίζεται και ο μηχανισμός Physical Memory Attributes (PMA) ο οποίος οριοθετεί περιοχές μνήμης και ορίζει τα χαρακτηριστικά τους[4].



Σχήμα 2.2: Μηχανισμός φυσικής προστασία μνήμης (PMP)

Supervisor-Mode

Το S-mode προσθέτει τα βασικά χαρακτηριστικά που χρειάζεται ένα σύστημα για να υποστηρίξει ένα μοντέρνο λειτουργικό σύστημα όπως είναι η διαχείριση εικονικής μνήμης, οι κλήσεις συστήματος και λοιπά. Στην υποενότητα 2.4 θα αναλύσουμε εκτενώς το σύστημα διαχείρισης εικονικής μνήμης.

2.2 Η γλώσσα περιγραφής υλικού Chisel

2.2.1 Βασικά στοιχεία της Chisel

Η **Chisel (Constructing Hardware In a Scala Embedded Language)** [2] είναι μία γλώσσα ειδικού σκοπού (Domain Specific Language) στοχευμένη στην περιγραφή υλικού. Πρακτικά είναι μία βιβλιοθήκη κτισμένη επί της γλώσσας προγραμματισμού **Scala** επομένως οι Compilers/Build Tools είναι κοινοί.

Η Chisel είναι μία πειραματική προσπάθεια να παρακαμφθούν διάφορα προβλήματα των κλασσικών γλωσσών περιγραφής κυκλωμάτων (Verilog, VHDL) με την χρήση προγραμματιστικών μοντέλων περιγραφής κυκλωμάτων υψηλού επιπέδου. Διαφέρει όμως από το High

[4] Atomicity PMAs, Memory-Ordering PMAs, Coherency and Cachebility PMAs και λοιπά. Βλέπε RISC-V Instruction Set Manual, Volume II: Privileged Architecture version 1.10 παράγραφος 3.5

Level Synthesis (HLS) [7] στην έννοια ότι το HLS είναι μία τεχνική/μέθοδος μετατροπής προγραμμάτων γραμμένων σε γλώσσα υψηλού επιπέδου σε κύκλωμα, ενώ η Chisel είναι γλώσσα περιγραφής κυκλωμάτων που χρησιμοποιεί προγραμματιστικές τεχνικές και μοντέλα γλωσσών προγραμματισμού υψηλού επιπέδου όπως φαίνεται στην λίστα που ακολουθεί.

Προτερήματα/Χαρακτηριστικά της Chisel:

- Java JVM
- Συναρτησιακός Προγραμματισμός
- Αντικειμενοστραφής Προγραμματισμός
- Στατικό Σύστημα Τύπων
- Συμπερασμός Τύπων

Στην Chisel ορίζεται ένα απλό και συγκεκριμένο σετ από Design Construction Primitives, το ελάχιστο δυνατό για **Register-Transfer-Level (RTL)** σχεδίαση. Κάποια από αυτά είναι τα **Wire** (Καλώδιο), **Vec** (Διάνυσμα), **Reg** (Καταχωρητής) και **Mem** (Μνήμη).

Με την χρήση των ανωτέρω τύπων, των συνδυασμό τους, καθώς και της εκτενής βιβλιοθήκης της Chisel διευκολύνεται και αυτοματοποιείται η περιγραφή κυκλωμάτων και καθίσταται δυνατή η δημιουργία **Generators** κυκλωμάτων.

Ευέλικτη ενδιάμεση αναπαράσταση RTL

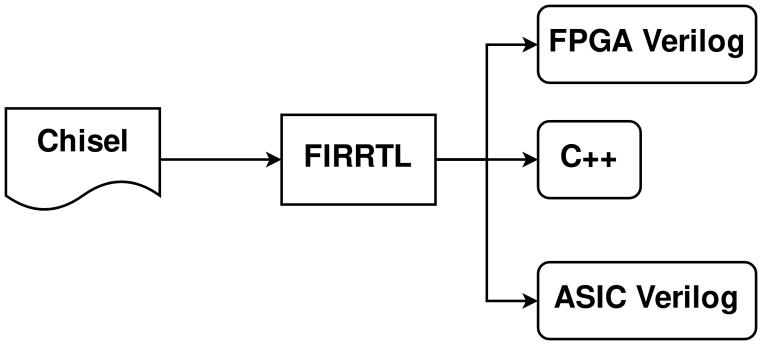
Η Chisel παράγει είτε Verilog είτε C++. Η διαδικασία μετάφρασης αποτελείται από αρκετά στάδια: Στο πρώτο στάδιο, η Chisel μεταφράζεται σε κυκλωματική ενδιάμεση αναπαράσταση RTL η οποία ονομάζεται **Ευέλικτη ενδιάμεση αναπαράσταση RTL, Flexible Intermediate Representation for RTL (FIRRTL)**. Στο δεύτερο στάδιο γίνεται ο έλεγχος και εφαρμόζονται βελτιστοποιήσεις πάνω στο FIRRTL, και στο τελευταίο στάδιο παράγεται Verilog/C++ με βάση το βελτιστοποιημένο FIRRTL (σχήμα 2.3). Εάν ο έλεγχος του FIRRTL είναι επιτυχής τότε το κύκλωμα είναι Synthesizable, δηλαδή μπορεί να υλοποιηθεί είτε σε FPGA (Field-Programmable Gate Array είτε σε ASIC (Application Specific Integrated Circuit.

2.3 Rocket Chip Generator

2.3.1 Υλοποιήσεις του RISC-V ISA

Το υψηλό κόστος ανάπτυξης μικροεπεξεργαστών αλλά και των συχνά περιοριστικών αδειών που συνοδεύουν την αγορά τους δημιουργούν ένα αρκετά δυσκίνητο περιβάλλον στην έρευνα/βιομηχανία με αποτέλεσμα το RISC-V ISA να λάβει γρήγορα προσοχή και μέσα στο διάστημα 2012-2019 να υπάρχουν ήδη αρκετές υλοποιήσεις:

- Rocket Chip - UC Berkeley



Σχήμα 2.3: Παραγωγή κώδικα για διαφορετικές πλατφόρμες από την Chisel

- Berkeley Out-Of-Order Machine (BOOM) - UC Berkeley [4]
- PULP Project - ETH Zurich [15]
- SHAKTI Processor Program - IIT Madras [8]

2.3.2 Rocket Chip Generator

Ο Rocket Chip Generator [1] είναι ένα ανοιχτού κώδικα **System-on-Chip (SoC) design Generator** που παράγει Synthesizable RTL. Είναι υλοποιημένος στην γλώσσα Chisel, η οποία καθιστά εύκολη την συγγραφή πολύπλοκων και παραμετροποιήσιμων γεννητριών για επεξεργαστικούς πυρήνες, κρυφές μνήμες και διασυνδέσεις εντός ενός SoC. Ο Rocket Chip Generator παράγει γενικού σκοπού μικροεπεξεργαστές οι οποίοι χρησιμοποιούν το RISC-V ISA, παρέχοντας μία γεννήτρια επεξεργαστικού πυρήνα in-order (Rocket) καθώς και μία γεννήτρια επεξεργαστικού πυρήνα out-of-order (BOOM). Επίσης, ο Rocket Chip Generator υποστηρίζει την ενσωμάτωση επιταχυντών (accelerators) στην μορφή extension εντολών.

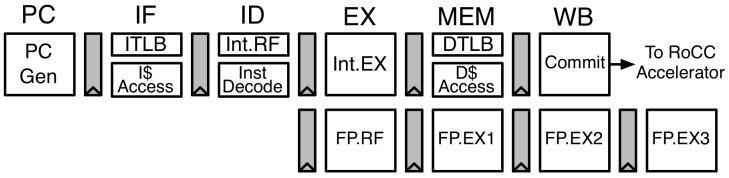
Παρακάτω παρουσιάζεται μία σύνοψη των δυνατοτήτων του Rocket Chip Generator:

- **Core**: Ο βαθμωτός in-order Rocket core και ο υπερβαθμωτός BOOM out-of-order core generator υποστηρίζουν προαιρετικό Floating Point Unit (FPU), παραμετροποιήσιμους branch predictors και παραμετροποιήσιμες λειτουργικές μονάδες (functional units).
- **Caches**: Μία συλλογή Generators κρυφών μνημών με ρυθμιζόμενο μέγεθος, associativity και πολίτικη αντικατάστασης, καθώς και ένα fully-associative TLB με ρυθμιζόμενο μέγεθος.
- **RoCC**: Το Rocket Custom Coprocessor interface, ένα template για επιταχυντές ειδικού σκοπού.
- **Tile**: Ένα Tile-Generator template για tiles με cache-coherent μνήμες. Ο αριθμός και ο τύπος των πυρήνων και των επιταχυντών είναι παραμετροποιήσιμος, καθώς και η οργάνωση των private κρυφών μνημών.

- **TileLink**: Μία γεννήτρια cache-coherency δικτύων ελεγκτών. Ρυθμιζόμενος ο αριθμός των tiles, των cache-coherency protocols μεταξύ των tiles, την ύπαρξη διαμοιραζόμενου χώρου αποθήκευσης καθώς και την φυσική υλοποίηση των υποκείμενων δικτύων.

- **Peripherals**: Γεννήτρια για AMBA-συμβατά[5] buses για χρήση με περιφερειακές συσκευές.

Rocket Core



Σχήμα 2.4: Στάδια διοχέτευσης στον Rocket Core

Ο Rocket είναι μία 5-σταδίων βαθμωτή γεννήτρια επεξεργαστικών πυρήνων που υλοποιούν το RV32G και RV64G ISA. Το σύστημα διαχείρισης εικονικής μνήμης, Memory Management Unit (MMU), υποστηρίζει σελιδοποίηση εικονικής μνήμης (page-based virtual memory). Ο Rocket έχει παραμετροποιήσιμη non-blocking κρυφή μνήμη δεδομένων καθώς και παραμετροποιήσιμο σύστημα πρόβλεψης διακλάδωσης (branch predictor) στο πρώτο στάδιο (front-end). Για την επεξεργασία αριθμών κινητής υποδιαστολής ο Rocket χρησιμοποιεί τις υλοποιήσεις μονάδων κινητής υποδιαστολής σε Chisel (berkeley-hardfloat[6]). Ο Rocket επίσης υποστηρίζει τα RISC-V Machine, User, και Supervisor privilege levels. Υπάρχει δυνατότητα παραμετροποίησης των περιεχόμενων extensions (M, A, F, D), των σταδίων σωλήνωσης των μονάδων κινητής υποδιαστολής, των χαρακτηριστικών των κρυφών μνημών (associativity, μέγεθος) καθώς και του μεγέθους του fully associative TLB. Τέλος, ο Rocket μπορεί να παρουσιαστεί σαν βιβλιοθήκη εξαρτημάτων επεξεργαστών, καθώς αρκετά μέρη του όπως οι κρυφές μνήμες, το TLB, ο Page Table Walker και το Control and Status Register File χρησιμοποιούνται από άλλες υλοποιήσεις όπως ο BOOM.

2.4 Διαχείριση Εικονικής Μνήμης

Σε αυτή την ενότητα θα ασχοληθούμε με την μονάδα διαχείρισης εικονικής μνήμης θεωρητικά αλλά και από την σκοπιά του RISC-V ISA [13], όπως ορίζεται από το Privileged Architecture Specification και την υλοποίηση του στον Rocket Chip.

[5]Advanced Microcontroller Bus Architecture
[6]https://github.com/ucb-bar/berkeley-hardfloat

2.4.1 Η διαχείριση εικονικής μνήμης στο RISC-V ISA

Το S-mode όπως αναφέραμε στην ενότητα 2.1.3 ορίζει σύστημα εικονικής μνήμης το οποίο χωρίζει την μνήμη σε σταθερού μεγέθους σελίδες με σκοπό την διαχείριση και την προστασία της μνήμης. Όταν το σύστημα σελιδοποίησης είναι ενεργοποιημένο οι περισσότερες διευθύνσεις είναι εικονικές (virtual address) και πρέπει να μεταφραστούν σε φυσικές διευθύνσεις (physical address) έτσι ώστε να γίνει προσπέλαση στην φυσική μνήμη. Οι εικονικές διευθύνσεις μεταφράζονται σε φυσικές διαβαίνοντας μία δενδρική δομή δεδομένων η οποία ονομάζεται Page Table. Οι τελευταίοι κόμβοι (leaf-nodes) του Page Table καθορίζουν κατά πόσον μία εικονική διεύθυνσή χαρτογραφείται σε μία φυσική διεύθυνσή και αν ισχύει αυτό ορίζει ποια Privilege levels επιτρέπεται να έχουν πρόσβαση στην σελίδα αυτή. Προσπάθεια πρόσβασης σε μία σελίδα η οποία είτε δεν είναι χαρτογραφημένη (mapped), είτε δεν υπάρχουν αρκετά δικαιώματα για την προσπέλαση της οδηγεί σε σφάλμα σελίδας (Page Fault).

Ο RISC-V ορίζει διάφορα σχήματα σελιδοποίησης αναλόγως του RV32/64 Base ISA, όπως βλέπουμε:

- **RV32 Paging Scheme** - 4KB base page
  - Sv32: 4GB virtual address space (2-level Page Table)
- **RV64 Paging Scheme** - 4KB base page
  - Sv39: 512GB virtual address space (3-level Page Table)
  - Sv48: 256TB virtual address space (4-level Page Table)

Στην παρούσα εργασία θα ασχοληθούμε με το σχήμα Sv39 του RV64.

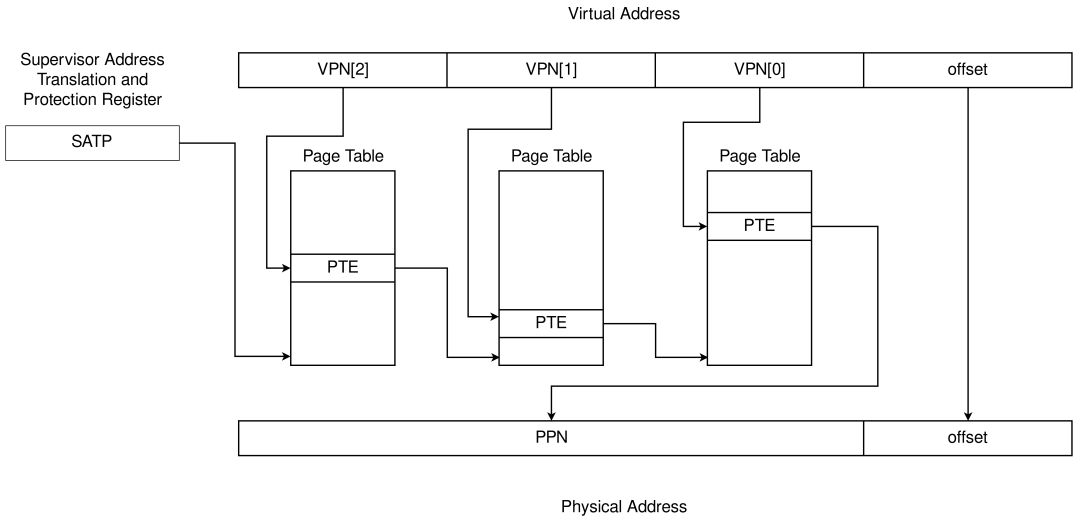
2.4.2 Σχήμα διαχείρισης εικονικής μνήμης Sv39 (RV64)

Όπως αναφέρθηκε παραπάνω, το σχήμα sv39 του RV64 ορίζει μέγεθος σελίδας 4KB και δομή Page Table 3-επιπέδων. Ο καταχωρητής συστήματος Supervisor Address Translation and Protection (`satp`) ελέγχει το σύστημα σελιδοποίησης κρατώντας την φυσική διεύθυνση της ρίζας του Page Table. Η οργάνωση του Page Table του Sv39 φαίνεται στο σχήμα 2.5.

Τα Page Tables του σχήματος Sv39 περιέχουν 2^9 Page Table Entries. Οποιοδήποτε επίπεδο PTE μπορεί να είναι τελικός κόμβος (leaf node), επομένως πέρα από σελίδες μεγέθους 4KB το σχήμα Sv39 υποστηρίζει 2MB Megapages και 1GB Gigapages.

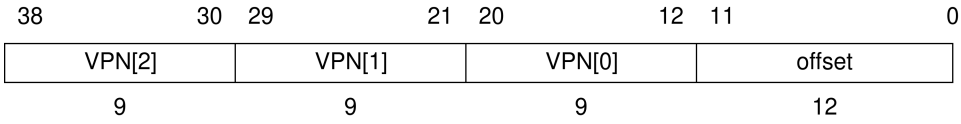
Τα σχήματα 2.6, 2.7, 2.8 παρουσιάζουν την δομή μίας εικονικής σελίδας, μίας φυσικής σελίδας και αντίστοιχα ενός Page Table Entry. Τα 10 πρώτα bits ενός Page Table Entry είναι reserved για μελλοντική χρήση. Πέρα από τον αριθμό της φυσικής σελίδας τα υπόλοιπα πεδία του PTE είναι τα εξής:

- **RSW** : Reserved πεδίο για μελλοντική χρήση από το S-mode
- **D** : Dirty bit, σημαίνει ότι έχει γίνει κάποια εγγραφή στην σελίδα
- **A** : Accessed bit, σημαίνει ότι η σελίδα έχει διαβαστεί η εγγραφεί (προσπελαστεί γενικά0

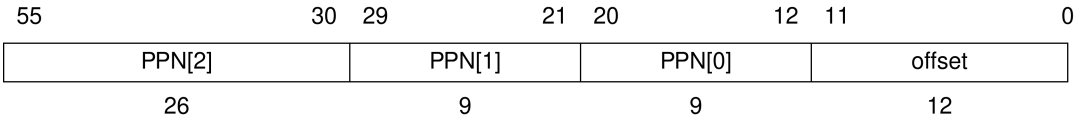


Σχήμα 2.5: Η οργάνωση του Page Table στο σχήμα sv39 (RV64)

- **G** : Global mapping
- **U** : Σελίδα που ανήκει στο U-mode
- **X** : Στην σελίδα επιτρέπεται εκτέλεση κώδικα
- **W** : Στην σελίδα επιτρέπεται εγγραφή
- **R** : Στην σελίδα επιτρέπεται διάβασμα δεδομένων
- **V** : Valid bit, η σελίδα είναι έγκυρη



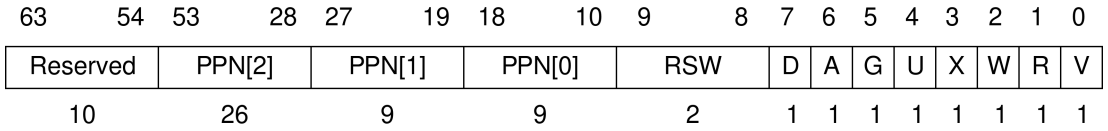
Σχήμα 2.6: Εικονική σελίδα Sv39



Σχήμα 2.7: Φυσική σελίδα Sv39

2.4.3 Η μονάδα διαχείρισης μνήμης στον Rocket Chip Generator

Ο Rocket Chip Generator υποστηρίζει τα RV32/RV64 Base ISAs, με βασικά σχήματα διαχείρισης μνήμης τα Sv32, Sv39. Στην εργασία αυτή καθώς ασχολούμαστε με το RV64



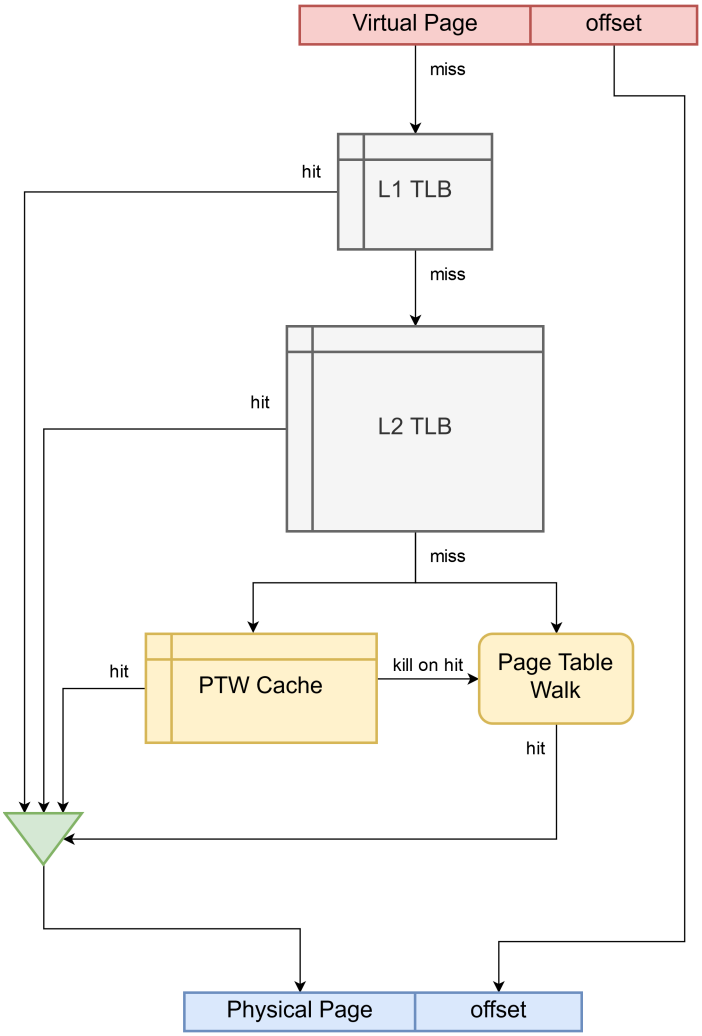
Σχήμα 2.8: Sv39 Page Table Entry

Base ISA, θεωρούμε από εδώ και στο εξής σύστημα διαχείρισης το Sv39. Το σύστημα διαχείρισης εικονικής μνήμης του Rocket Chip Generator έχει υλοποιημένη μονάδα Page Table Walk η οποία είναι υπεύθυνη για την εύρεση μεταφράσεων από την εικονική στην φυσική μνήμη. Επειδή οι αναζητήσεις εικονικών-φυσικών διευθύνσεων μπορούν να προσθέσουν μεγάλη καθυστέρηση στην μονάδα επεξεργασίας καλή πρακτική είναι η χρήση κρυφών μνημών οι οποίες κρατάνε τις ενδιάμεσες μεταφράσεις, βελτιώνοντας την επίδοση. Το σχήμα μετάφρασης εικονικών διευθύνσεων του Rocket Chip φαίνεται στο σχήμα 2.9. Με παραμετροποιήσιμο των αριθμό των Entries υποστηρίζονται:

1. **Fully-associative L1 TLB** : Μικρό και γρήγορο, πλήρως-συσχετιστικό πρώτου επιπέδου TLB με κόστος hit ένα κύκλο.

2. **Direct-mapped L2 TLB** : Μεγαλύτερο αλλά πιο αργό άμεσης-απεικόνισης δευτέρου επιπέδου TLB με κόστος hit 2 κύκλους.

3. **Fully-associative Page Table Walk Cache** : Πλήρως-συσχετιστική κρυφή μνήμη ενσωματωμένη στην μονάδα του Page Table Walker που κρατάει non-leaf μεταφράσεις των τριών επιπέδων του Page Table με κόστος hit ένα κύκλο ανά επίπεδο.



Σχήμα 2.9: Σχήμα μετάφρασης εικονικών διευθύνσεων στον Rocket Chip Generator

Κεφάλαιο 3

Μεθοδολογία

Στο κεφάλαιο αυτό θα αναλύσουμε τις πλατφόρμες υλικού/λογισμικού που θα χρησιμοποιούμε στην παρούσα εργασία. Αρχικά θα περιγράψουμε τα βήματα που ακολουθούμε κατά την διάρκεια σχεδίασης υλικού, τα απαραίτητα εργαλεία, και έπειτα το λογισμικό με το οποίο ελέγχουμε την ορθότητα και τις επιδόσεις του υλικού.

3.1 Ενσωμάτωση Rocket Chip Generator στο Xilinx ZCU102

Η πρώτη ενέργεια της παρούσας εργασίας ήταν η ενσωμάτωση του Rocket Chip Generator στο Xilinx ZCU102 FPGA. Το Xilinx ZCU102 βασίζεται στην ετερογενή πλατφόρμα επεξεργασίας Zynq Ultrascale+ MPSoC της Xilinx. Το συγκεκριμένο Multi-processor SoC περιέχει περιέχει 4 γενικούς επεξεργαστικούς πυρήνες ARM Cortex-A53 οι οποίοι δρουν ως το **Processing-System (PS)**, ενώ περιέχει **Programmable-Logic (PL)** μεγέθους 600K λογικών κελιών και 32.1Mb Block RAM. Τα χαρακτηριστικά του συγκεκριμένου Multi-processor SoC παρουσιάζονται αναλυτικότερα στον πίνακα 3.1. Όσον αφορά το λογισμικό, θα χρησιμοποιήσουμε το Xilinx Vivado 2018.1 καθώς και το SDK 2018.2.

Processing Units	
Processing System (PS)	Quad-core ARM Cortex-A53
Real-Time Processor	Dual-core ARM Cortex-R5F
Graphics Processing Unit (GPU)	ARM Mali-400 MP2
Programmable Logic (PL)	
System Logic Cells (K)	600
Memory (Mb)	32.1
DSP Slices	2.520
Memory	
DDR4 SODIMM	4GB attached to PS
DDR4 Component	512MB attached to PL

Πίνακας 3.1: Χαρακτηριστικά Xilinx ZCU102

Το εργαστήριο Berkeley Architecture Research (BAR) του UC Berkeley υποστηρίζει την υλοποίηση/ενσωμάτωση του Rocket Chip Generator σε Zynq FPGA[1] αλλά για διαφορετικές FPGA πλατφόρμες από την Xilinx ZCU102. Έπειτα από έρευνα, βρήκαμε λογισμικό ενσωμάτωσης (rc-fpga-zcu)[2] του Rocket Chip στο Xilinx ZCU102 από ερευνητές του Πανεπιστήμιου της Νότιας Ουαλίας η οποία όμως στοχεύει το Vivado 2017.1 και το SDK 2018.2. Πρώτο βήμα ήταν η ενημέρωση του rc-fpga-zcu στην νεότερη έκδοση Vivado 2018.1 και έπειτα η αποσφαλμάτωση του καθώς περιείχε στατικά (hard-coded) στοιχεία τα οποία έπρεπε να γίνουν δυναμικά. Σε επόμενο βήμα έγινε η αυτοματοποίηση της διαδικασίας παραγωγής του bitstream για την φόρτωση του στο Xilinx ZCU102 όπως θα δούμε παρακάτω. Η τελική μορφή με τις αλλαγές που έγιναν στο παραπάνω repository μπορούν να βρεθούν στον σύνδεσμο rc-fpga-zcu[3]. Οι κύριες αλλαγές έγιναν στα TCL scripts και τα configuration files τα οποία χρησιμεύουν στην αυτοματοποιημένη παραγωγή του bitstream, και συνοψίζονται στις παρακάτω.

- Αλλαγή έκδοσης u-boot-xlnx[4] bootloader για την εκκίνηση Linux στο PS (ARM Cores).
- Αναβάθμιση Board Model version της πλακέτας.
- Αναβάθμιση Part Number version του PL.
- Ενημέρωση με παραπάνω αλλαγές των configuration file για το Petalinux, απαραίτητο για το στήσιμο περιβάλλοντος Linux στο PS.
- Αφαίρεση hardcoded paths και δυναμική αρχικοποίηση τους με scripts.

3.2 Ροή ανάπτυξης υλικού

Η ανάπτυξη του υλικού χωρίζεται σε δύο βήματα: Αρχικά μέσω του ακριβή-ανά-κύκλο-επεξεργασίας προσομοιωτή Verilator (cycle-accurate hardware emulator) ελέγχουμε την ορθότητα της σχεδίασης, και έπειτα χρησιμοποιώντας το Xilinx ZCU102 FPGA ελέγχουμε την επίδοσή της σχεδίασης. Οι διαφορές μεταξύ των προσεγγίσεων συνοψίζονται στον πίνακα 3.2.

Feature	Software emulation	FPGA
Compilation Time	Fast	Slow
Simulation Time	Very Slow	Very Fast
Debugging	Easy	Very Hard

Πίνακας 3.2: Hardware emulation vs FPGA

[1] https://github.com/ucb-bar/fpga-zynq
[2] https://github.com/li3tuo4/rc-fpga-zcu
[3] https://github.com/ncppd/rc-fpga-zcu
[4] https://github.com/Xilinx/u-boot-xlnx.git

Αφού μελετήσουμε τα εργαλεία και τις μεθόδους ελέγχου το υλικού στο λογισμικό θα επιστρέψουμε στην διαδικασία στησίματος του bitstream για το Xilinx ZCU102 με το rc-fpga-zcu στην υποενότητα 3.2.3.

3.2.1 Εργαλεία, Compilers και Προσομοιωτές

Πριν προχωρήσουμε πρέπει να σιγουρευτούμε ότι έχουμε έτοιμα τα riscv-tools[5] ακολουθώντας τις οδηγίες στον αντίστοιχο σύνδεσμο. Τα riscv-tools περιέχουν τα απαραίτητα εργαλεία όπως toolchains, RISC-V cross-compilers, riscv-tests, riscv-opcodes καθώς και το Spike το οποίο είναι ένας γρήγορος λειτουργικός προσομοιωτής. Με το Spike τρέχουμε RISC-V binaries ώστε να ελέγξουμε την ορθότητα τους πριν τα φορτώσουμε στο FPGA.

3.2.2 Verilator

Ο Verilator [20] είναι ένα ελεύθερο και ανοιχτού κώδικα εργαλείο το οποίο μετατρέπει την Verilog που παράγει η Chisel σε ένα συμπεριφορικό cycle-accurate C++/SystemC μοντέλο. Εντός του κώδικα Chisel, χρησιμοποιώντας δηλώσεις `assert - printf` διευκολύνεται η αποσφαλμάτωση του σχεδιασμού καθώς ο Verilator υποστηρίζει μηνύματα debug ανά κύκλο μηχανής.

Χρησιμοποιώντας το εκτελέσιμο που παράγει ο Verilator τρέχουμε τα official riscv-tests από τα οποία λαμβάνουμε τα πρώτα αποτελέσματα για την ορθότητα του σχεδιασμού καθώς μπορούμε να ελέγχουμε τι συμβαίνει κύκλο-ανά-κύκλο σε αυτά τα απλά assembly tests. Στήνοντας στοχευμένα assembly tests για τον έλεγχο ορθότητας της σχεδίασης του υλικού, σε συνεργασία με τα μηνύματα αποσφαλμάτωσης, η διαδικασία διόρθωσης του υλικού γίνεται αρκετά γρήγορη και παραγωγική. Επιπλέον στα riscv-tests περιέχονται benchmarks (dhrystone, qsort, spmv μεταξύ άλλων) από τα οποία λαμβάνουμε αρχικά αποτελέσματα για την επίδοση της σχεδίασης. Σημειώνουμε ότι ο Verilator διευκολύνει πολύ το debugging αλλά είναι πολύ αργός[6].

3.2.3 Προετοιμασία bitstream

Αφού ολοκληρωθεί ο έλεγχος της σχεδίασης του υλικού στον Verilator, μπορούμε να προχωρήσουμε στην διαδικασία παραγωγής του bitstream. Πρακτικά το bitstream είναι το πρόγραμμα το οποίο φορτώνεται στο PL. Περιέχει τον προγραμματισμό των **Look-up Tables (LUTs), Flip-Flops (FFs)** του PL καθώς και την δρομολόγηση (routing) μεταξύ των στοιχείων συνδυαστικής λογικής και μνήμης.

Πρώτο βήμα της διαδικασίας είναι χρησιμοποιώντας το πρόγραμμα `git` να κάνουμε `clone` το repository rc-fpga-zcu[7], και `git submodule update --init --recursive` εντός του. Ο φάκελος `zcu102/` περιέχει το script `generate-bitstream.sh` το οποίο παράγει αυτόματα όλα τα αρχεία που θα χρειαστούμε.

[5]https://github.com/li3tuo4/riscv-tools-zcu/tree/434fdeb0f863afd7b083b57bba923da9bc98d9a6
[6]Μία ημέρα για να εκκινήσει linux σε σύγχρονο επεξεργαστή x86-64
[7]https://github.com/ncppd/rc-fpga-zcu.git

Αναλυτικά, το script `generate-bitstream.sh` εκτελεί τα παρακάτω:

1. Μεταφράζει την Chisel του Rocket Chip και του testchipip[8] σε Verilog. Το testchipip περιέχει IP components που διευκολύνουν στο πακετάρισμα του Rocket Chip στο FPGA.

2. Αρχικοποιεί το Vivado Project με την δεδομένη Verilog και βοηθητικό κώδικα Verilog όπως υποστήριξη για ασύγχρονο reset και wrappers του Rocket Chip για το FPGA.

3. Τρέχει αυτοματοποιημένα TCL scripts για τα στάδια των Synthesis, Implementation και τέλος την παραγωγή του bitstream.

4. Αρχικοποιείται ένα Petalinux Project για την διαχείριση του λογισμικού του PS και του στησίματος του PL.

5. Κάνοντας απαραίτητες αλλαγές στα configuration files του Petalinux παράγεται το Linux binary το οποίο θα τρέξει στο PS (ARM Cores) και το BOOT.BIN το οποίο πακετάρει τα:

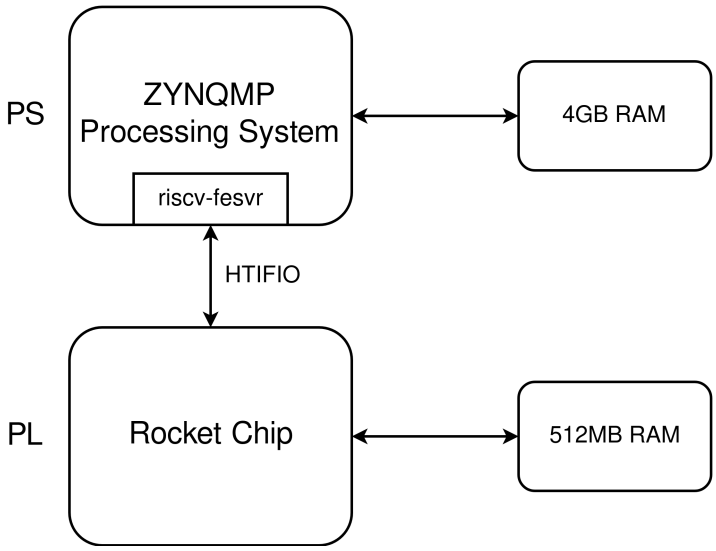
   - First Stage Boot Loader (FSBL): Λογισμικό αρχικοποίησης του FPGA.
   - PMU Firmware: Λογισμικό διαχείρισης πόρων της πλατφόρμας FPGA.
   - U-boot: Bootloader για το Linux binary που θα φορτωθεί στο PS.
   - ARM Trusted Firmware: Λογισμικό ασφάλειας της πλατφόρμας.
   - Το bitstream του Rocket Core το οποίο θα φορτωθεί στο PL.

Αφού ολοκληρωθούν τα Synthesis, Implementation ανοίγοντάς το Vivado μπορούμε να πάρουμε πληροφορίες σχετικά με την σχεδίαση μας και πόσους πόρους δεσμεύει όπως LUTs, FFs και Block RAM. Επίσης μέσω των Timing Constraints που ορίσαμε για την σχεδίαση, δηλαδή τον ελάχιστο χρόνο κύκλου που προσπαθούμε να πετύχουμε, προκύπτει η μετρική Worst Negative Slack. Η μετρική αυτή εάν είναι αρνητική κατά N nanoseconds μας ενημερώνει ότι πρέπει να αυξήσουμε το Timing Constraints κατά τουλάχιστον N nanoseconds για να επιτύχει το Implementation της σχεδίασης, ενώ εάν είναι θετική μας ενημερώνει ότι υπάρχει επιπλέον περιθώριο μείωσης χρονικού κύκλου κατά N nanoseconds. Έτσι μπορούμε εύκολα να υπολογίσουμε το **Critical Path** της σχεδίασής μας, δηλαδή το χειρότερο μονοπάτι κατά την διάρκεια ενεργού παλμού ρολογιού το οποίο καθορίζει τον χρονισμό. Η παραπάνω μέθοδος ονομάζεται Static Timing Analysis [5], μέθοδος με την οποία υπολογίζουμε τον αναμενόμενο χρονισμό ενός κυκλώματος εξερευνώντας τον γράφο κυκλώματος, με ακμές την χρονική καθυστέρηση μεταξύ των κόμβων.

[8] https://github.com/ucb-bar/testchipip

3.2.4 Επικοινωνία PS-PL

Η εφαρμογή riscv-fesvr επιτρέπει την μεταφορά μηνυμάτων και φόρτωση εκτελέσιμων από το PS στο Rocket Chip το οποίο είναι προγραμματισμένο στο PL. Η διεπαφή που χρησιμοποιείται για την επικοινωνία riscv-fesvr και Rocket Chip ονομάζεται Host-Target Interface (HTIF).



Σχήμα 3.1: Επικοινωνία μεταξύ PS, PL και μνημών με χρήση του RISC-V Front-end Server

3.2.5 Εξερεύνηση Χώρου Σχεδίασης

Όπως αναφέρθηκε στην ενότητα 2.3.2, το Rocket Chip εκθέτει αρκετές παραμέτρους για την ευέλικτη προσαρμογή των πυρήνων, κρυφών μνημών και διασυνδέσεων κατά των σχεδιαστικών αναγκών μας. Αυτοματοποιούμε και διευκολύνουμε περαιτέρω την διαδικασία εξερεύνησης του χώρου σχεδίασης (Design Space Exploration) με το rocket-panel[9], ένα script το οποίο λαμβάνει στην είσοδο μονοδιάστατους πίνακες με τις παραμέτρους που μας ενδιαφέρουν και παράγει μαζικά bitstream για γρήγορο έλεγχο.

3.3 Ροή ανάπτυξης Λογισμικού

Αφού ολοκληρωθούν τα βήματα για την προετοιμασία του υλικού προχωράμε στην προετοιμασία του λογισμικού, δηλαδή στην παραγωγή ενός minimal Linux περιβάλλοντος με βασικές εφαρμογές συστήματος καθώς και benchmarks για την αξιολόγηση της σχεδίασης. Θα βασιστούμε στις παρακάτω τεχνολογίες:

1. **Buildroot**: Εργαλείο παραγωγής απλού αλλά ολοκληρωμένου συστήματος Linux με χρήση cross-compiler για αρχιτεκτονική RISC-V

[9]https://github.com/ncppd/rocket-panel

2. **Linux**: Έκδοση πυρήνα Linux 4.15 για την αρχιτεκτονική RISC-V

3. **Berkeley Boot Loader (BBL)**: Ο bootloader για το RISC-V Linux

3.3.1 Freedom-U-SDK

Για να διευκολυνθεί η διαδικασία παραγωγής του περιβάλλοντος Linux χρησιμοποιούμε το εργαλείο Freedom-U-SDK[10] της εταιρίας Sifive που στοχεύει στην αυτοματοποιημένη παραγωγή Linux περιβάλλοντος με την χρήση των παραπάνω εργαλείων και make scripts. Αφού κάνουμε τις παρακάτω αλλαγές/προσθήκες με χρήση μόνο του Makefile παράγεται έτοιμο εκτελέσιμο για την φόρτωση στο FPGA.

3.3.2 Buildroot

Το Buildroot επιτρέπει την εύκολη προσθήκη στο περιβάλλον του distribution επιπλέον πακέτων που δημιούργησε ο χρήστης. Προσθέτουμε benchmarks και scripts για την συλλογή μετρήσεων από τους performance counters. Στα πλαίσιά της παρούσας εργασίας δημιουργήθηκε το πακέτο tlbtest[11] το οποίο είναι σχεδιασμένο για να ενσωματωθεί στο Buildroot για την ευκολότερη συλλογή μετρήσεων με scripts και μετροπρογράμματα που τεστάρουν το TLB και ελέγχουν την επίδοση του. Ενσωματώνουμε στο Buildroot μετροπρογράμματα της σουίτας SPEC2006 χρησιμοποιώντας το πακέτο tlbtest που σχεδιάσαμε. Έχοντας λοιπόν έτοιμο περιβάλλον ελέγχου και testing και προχωράμε στο στήσιμο του bootloader για τον linux kernel.

3.3.3 Berkeley Boot Loader (BBL)

Ο BBL είναι ο bootloader του RISC-V Linux, τρέχει σε M-mode οπότε έχει πλήρη διαφάνεια στον RISC-V Core, προετοιμάζει το σύστημα, δηλαδή τους απαραίτητους CSRs και το μηχανισμό Physical Memory Protection (PMP) και έπειτα παραδίδει την εκτέλεση στο Linux. Ο BBL περιέχεται στο πακέτο riscv-pk το οποίο μπορούμε να εντοπίσουμε εντός του φακέλου riscv-tools που στήσαμε στην αρχή του κεφαλαίου. Το πακέτο riscv-pk πέρα από τον BBL περιέχει τον RISC-V Proxy Kernel (PK) ο οποίος είναι ένας minimal kernel που χρησιμοποιείται από το Spike για την εκτέλεση προγραμμάτων χώρου χρήστη.

Τροποποιούμε τον BBL προσθέτοντας λειτουργικότητα για την ενημέρωση και το διάβασμα των performace counters για τα γεγονότα που μας ενδιαφέρουν όπως είναι οι αστοχίες στο Data TLB και οι κύκλοι επεξεργασίας. Τέλος, ελέγχουμε ότι είναι ενεργοποιημένη η προώθηση (delegation[12]) των performance counters [17] στο U-mode, ώστε να έχουμε πρόσβαση από τον χώρο χρήστη στους μετρητές. Το τελικό εκτελέσιμο παράγεται με την εξής διαδικασία:

[10] https://github.com/sifive/freedom-u-sdk/tree/linux_u500vc707devkit_config
[11] https://github.com/ncppd/tlbtest/tree/master
[12] Βλέπε καταχωρητές `mcounteren, scounteren`, RISC-V ISA Spec Volume II: Privileged Architecture

1. Από τον φάκελο `work/linux/` του freedom-u-sdk λαμβάνουμε το αρχείο `vmlinux` το οποίο περιέχει το filesystem του buildroot μαζί με τον πυρήνα Linux.

2. Χρησιμοποιώντας το πακέτο riscv-pk το οποίο παρέχεται στον φάκελο των riscv-tools που ετοιμάσαμε στην αρχή του κεφαλαίου, κάνουμε build το τελικό εκτελέσιμο το οποίο θα φορτωθεί στο FPGA. Δεν χρησιμοποιούμε το riscv-pk του freedom-u-sdk καθώς δεν περιέχει κάποιες απαραίτητες αλλαγές που πρέπει να γίνουν στο Host-Target Interface (HTIF) για την σωστή επικοινωνία με τον RISC-V Front-End Server (riscv-fesvr) όπως αναλύθηκε στην ενότητα 3.2.4.

3.3.4 Έλεγχος τελικού εκτελέσιμου

Τελειώνοντας την παραπάνω διαδικασία, το τελικό εκτελέσιμο που παράγουμε θα περιέχει τον πυρήνα linux, το σύστημα αρχείων που δημιουργήσαμε με το Buildroot και τέλος τον bootloader BBL ο οποίος είναι υπεύθυνος για την αρχικοποίηση των CSRs καθώς και για την εκκίνηση του linux.

Για να επιβεβαιώσουμε την ορθή λειτουργία του λειτουργικού συστήματος και των εργαλείων που ετοιμάσαμε, πριν το μεταφέρουμε στο Xilinx ZCU102 για την φόρτωση του στον Rocket Core μέσω του `riscv-fesvr`[13], το ελέγχουμε με τον προσομοιωτή Spike[14]. Το Spike είναι ένας γρήγορος προσομοιωτής (functional simulator) εκτελέσιμων της αρχιτεκτονικής RISC-V. Με το Spike ελέγχουμε την ορθότητα του λογισμικού χωρίς να μπορούμε όμως να δούμε την εσωτερική κατάσταση του επεξεργαστή όπως κάνουμε με τον Verilator. Δεν μπορούμε να δούμε τις τιμές των CSRs, τα σήματα ελέγχου καθώς και το εσωτερικό των κρυφών μνημών όπως για παράδειγμα το TLB. Η διαφορά των δύο προσομοιωτών έγκειται στο γεγονός ότι το Spike χρησιμεύει στον γρήγορο έλεγχο και αποσφαλμάτωση του λογισμικού ενώ ο Verilator στον έλεγχο και την αποσφαλμάτωση του υλικού.

3.4 Σύνοψη μεθοδολογίας

Αφού ολοκληρώσουμε τα παραπάνω βήματα θα έχουμε έτοιμα τα εργαλεία ανάπτυξης υλικού καθώς και το λογισμικό για τον έλεγχο ορθότητας και επίδοσης του υλικού. Στα επόμενα κεφάλαια θα χρησιμοποιήσουμε τα εργαλεία που ετοιμάσαμε για την υλοποίηση και την εκτίμηση της επίδοσης της γεννήτριας TLB στον Rocket Chip Generator.

[13] Βλέπε παράγραφο 3.2.4
[14] https://github.com/riscv/riscv-isa-sim

Κεφάλαιο 4

Ανάλυση και σχεδίαση παραμετροποιήσιμου TLB

Στο κεφάλαιο αυτό παρουσιάζεται η μελέτη λειτουργίας που έγινε στο fully-associative L1 TLB του Rocket Chip Generator και έπειτα η σχεδίαση και υλοποίηση παραμετροποιήσιμου TLB Generator.

4.1 Ανάλυση - περιγραφή αρχιτεκτονικής TLB

Στην ενότητα αυτή παρουσιάζεται η ανάλυση και η αρχιτεκτονική του συστήματος TLB του Rocket Chip Generator, σε σχέση με τα συστήματα που συνδέεται καθώς και της εσωτερικής του οργάνωσης.

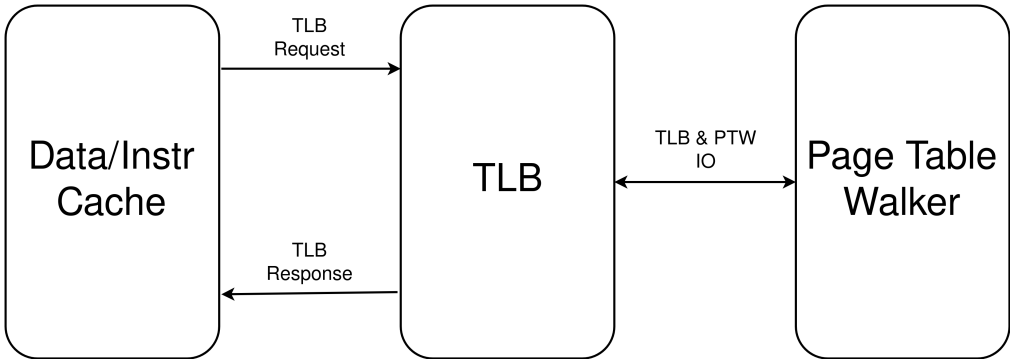
4.1.1 Εξωτερική Διασύνδεση

Τα Data/Instruction TLB είναι δύο διαφορετικές οντότητες ενώ βασίζονται στο ίδιο template του Rocket Chip, και είναι αντίστοιχα συνδεδεμένα με την Data και την Instruction Cache. Επίσης τα Data/Instruction TLB έχουν ιδιωτικό Page Table Walker (PTW) σε περίπτωση αστοχίας TLB, για την εύρεση της φυσικής διεύθυνσης με την προσπέλαση του Page Table. Χωρίς βλάβη της γενικότητας, από εδώ και έπειτα όταν αναφερόμαστε στην έννοια του TLB θα αναφερόμαστε στο Data TLB.

Η εξωτερική διασύνδεση με την Cache και την μονάδα PTW γίνεται μέσω των παρακάτω διεπαφών:

- **TLB Request**: Ένα TLB Request γίνεται issue από την Cache είτε για την εξυπηρέτηση εύρεσης της φυσικής σελίδας είτε ενημέρωσης του TLB. Κύρια δεδομένα που προμηθεύει είναι τα παρακάτω:
  - Virtual Page Number (VPN): Η εικονική διεύθυνση που ζητείται να εξυπηρετηθεί
  - `sfence.vma` Request: Αίτηση για TLB flush γιατί έγινε αλλαγή στο Page Table και πρέπει να ακυρωθούν entries τα οποία δείχνουν πλέον σε λάθος φυσικές σελίδες.

- **TLB Response**: Η απάντηση που επιστρέφεται από το TLB στην Cache, έπειτα από κάποιο TLB Request. Περιέχει τα βασικότερα:
  - Miss/Hit: Ενημέρωση για το εάν βρέθηκε μετάφραση της VPN στο TLB
  - Physical Page Number (PPN): Σε περίπτωση ευστοχίας η PPN επιστρέφεται στην DCache
  - Exceptions: Η DCache ενημερώνεται για τα σφάλματα που μπορεί να προέκυψαν κατά την αναζήτηση στο TLB ή στην μονάδα PTW όπως είναι τα Page Fault, Access Exception, Misaligned superpage
- **Page Table Walk - TLB I/O**: Διεπαφή δύο δρόμων μεταξύ TLB και PTW. Αναλυτικότερα:
  - PTW Request: Αίτηση του TLB με το VPN στο PTW έπειτα από αστοχία TLB
  - PTW Response: Απάντηση του PTW με το PPN στο TLB
  - PMP/PMA Information: Έλεγχος ασφαλείας μηχανισμών Physical Memory Protection και Physical Memory Attributes



Σχήμα 4.1: Διασύνδεση μεταξύ του TLB, της Cache και της μονάδας PTW

4.1.2 Εσωτερική Οργάνωση TLB

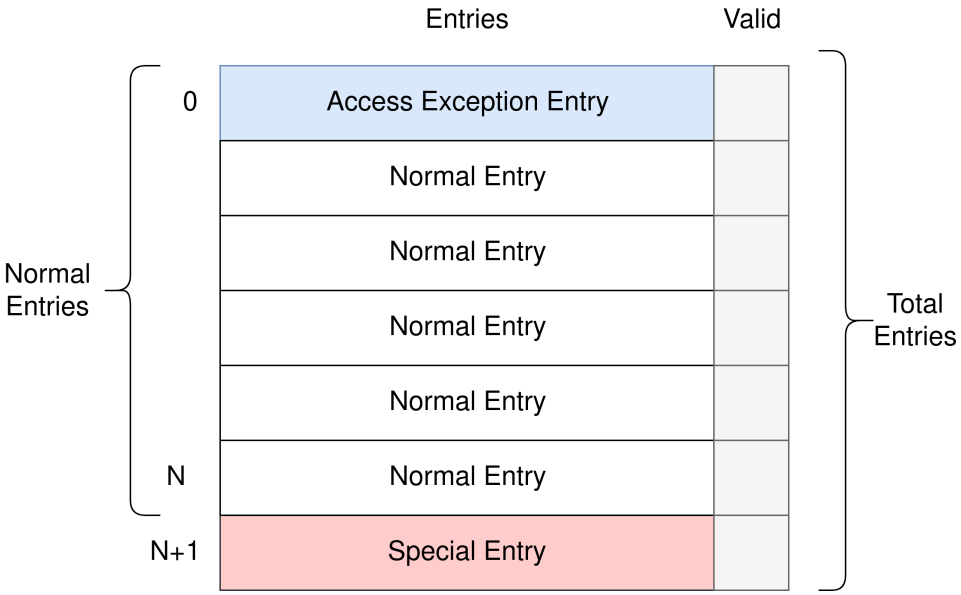
Στο TLB ορίζεται η **κλάση Entry** η οποία κρατάει πληροφορίες κάθε γραμμής του TLB, όπως η **VPN**, η **PPN**, και τα **δικαιώματα πρόσβασης/εκτέλεσης** της κάθε σελίδας. Η παραμετροποίηση που προσφέρει το TLB είναι στον αριθμό Entries που μπορεί να κρατήσει, καθώς από την σχεδίαση του είναι πλήρως συσχετιστικό (fully-associate). Η δεύτερή λειτουργία του TLB πέρα από κρυφή μνήμη εικονικών σελίδων είναι ο έλεγχος των σελίδων έναντι του μηχανισμού Physical Memory Protection (PMP) - Physical Memory Attribute (PMA)[1] δηλαδή του κυκλώματος ελέγχου πρόσβασης δεύτερης βαθμίδας. Μία επιπλέον παράμετρος που μπορεί να αλλάξει για το TLB είναι η επιλογή ενεργοποίησης του

[1]Βλέπε RISC-V Instruction Set Manual, Volume II: Privileged Architecture version 1.10 παράγραφος 3.5

μηχανισμού της εικονικής μνήμης. Ένα εύλογο ερώτημα είναι ποια θα ήταν η χρησιμότητα του TLB σε περίπτωση που δεν χρησιμοποιείται εικονική μνήμη. Όπως αναφέραμε σε προηγούμενο κεφάλαιο η Chisel προσφέρει την δυνατότητα σχεδιασμού Generators, επομένως σε περίπτωση που η εικονική μνήμη είναι απενεργοποιημένη το TLB χρησιμοποιείται σαν μηχανισμός ελέγχου του μηχανισμού PMP/PMA έναντι φυσικών διευθύνσεων.

Η εσωτερική οργάνωση του TLB παρουσιάζεται στο σχήμα 4.2. Αναλυτικότερα, τα είδη των Entries που κρατούνται στο TLB είναι τα παρακάτω:

- **Normal Entry**: Τα Entries τα οποία δεν παρουσίασαν κάποιο Access Exception ούτε απέτυχαν στον έλεγχο PMP και PMA
  - Αποθηκεύονται στις θέσεις από 0 έως και N.
- **Access Exception Entry**: Τα Entries που οδήγησαν είτε σε access exception, page fault, είτε είναι misaligned superpage.
  - Αποθηκεύονται πάντα στην θέση 0
- **Special Entry**: Είναι τα Entries που αναφέρονται σε unmapped μνήμη. Χρησιμοποιούνται για να ελέγξουν τα δικαιώματα πρόσβασης στην φυσική μνήμη έναντι του μηχανισμού PMP/PMA.
  - Αποθηκεύονται πάντα στην θέση N+1



Σχήμα 4.2: Εσωτερική οργάνωση του TLB

Τα Entries παριστάνονται ως διανύσματα καταχωρητών (τύποι `Vec`, `Reg` της Chisel), και συνοδεύονται από ένα διάνυσμα καταχωρητών Valid το οποίο ενημερώνει για τα έγκυρα Entries, όπως φαίνεται στον πίνακα 4.2. Η εγγραφή σε κάποιον καταχωρητή είναι εμφανής τον επόμενο κύκλο στην έξοδο του. Στην πλατφόρμα FPGA, οι καταχωρητές αποθηκεύονται στις

μονάδες **Flip-Flop (FFs)** με συνδυαστική λογική επιλογής διανυσμάτων να υλοποιείται στα **Look-Up Tables (LUTs)** .

Σε περίπτωση flush απλά μηδενίζουμε το εκάστοτε bit στο διάνυσμα Valid. Τα διανύσματα στην Chisel είναι ένας τύπος δεδομένων ο οποίος είναι **indexable**. Το χαρακτηριστικό αυτό είναι ιδιαίτερα χρήσιμο ειδικά για set-associative μνήμες καθώς η επιλογή του set θα γίνεται πολύ εύκολα όπως θα δούμε αργότερα. Η πολιτική αντικατάστασης του TLB είναι η Pseudo Least-Recently-Used (PseudoLRU).

Η εντολή `sfence.vma` γίνεται issue όταν γίνει κάποια αλλαγή στο Page Table και πρέπει να ακυρωθούν τα πλεόν invalid Entries. Αναλόγως την περίπτωση κάνει flush είτε ένα Entry, είτε Entries με συγκεκριμένο Application Space Identifier (το ASID εξυπηρετεί αποδοτικότερα context switches), είτε όλο το TLB.

Το TLB μπορεί να βρίσκεται στις παρακάτω καταστάσεις:

- **Ready** : Έτοιμο να δεχτεί ερώτημα από την Cache είτε για μετάφραση είτε `sfence.vma`
- **Request** : Συνέβη αστοχία TLB και περιμένει για να στείλει ερώτημα στην μονάδα PTW (περιμένει να απαντήσει με μήνυμα Ready η μονάδα PTW). Κατά την διάρκεια της κατάστασης Request το TLB εξυπηρετεί `sfence.vma` με επιστροφή στην κατάσταση Ready
- **Wait** : Η μονάδα PTW αποδέχτηκε το ερώτημα και το TLB βρίσκεται σε κατάσταση αναμονής
- **Wait-Invalidate** : Εάν συμβεί `sfence.vma` κατά την διάρκεια της κατάστασης Wait τότε εξυπηρετείται το flush και αγνοείται η απάντηση του PTW

4.2 Σχεδίαση TLB Generator

Σε περίπτωση που θέλουμε να τρέξουμε στο Rocket Chip μία εφαρμογή η οποία είναι απαιτητική σε μέγεθος μνήμης, θα πρέπει να μεγαλώσουμε το μέγεθος του TLB. Σε αντίθετη περίπτωση το TLB δεν θα μπορεί να εξυπηρετήσει όλες τις σελίδες: Για παράδειγμα, έστω TLB 32 θέσεων και εφαρμογή που διαχειρίζεται κυκλικά 33 σελίδες μνήμης διαβάζοντας μόνο ένα στοιχείο από την κάθε σελίδα. Το TLB δεν πρόκειται να περιέχει ποτέ έγκυρη μετάφραση με αποτέλεσμα να παρατηρείται το φαινόμενο του **TLB thrashing** , από άποψη επίδοσης θα φαίνεται σαν να μην υπάρχει TLB λόγω διαρκών αστοχιών που θα πρέπει να εξυπηρετηθούν από την μονάδα PTW. Αυξάνοντας τις θέσεις του TLB αποφεύγουμε το παραπάνω πρόβλημα και προκύπτει ένα επιπλέον: Λόγω μεγαλύτερου TLB θα μεγαλώσει το κύκλωμα ελέγχου και εύρεσης της μετάφρασης, το οποίο βρίσκεται στο **critical path** του επεξεργαστή. Ειδικά στην περίπτωση πλήρως-συσχετιστικού TLB αναζητείται η μετάφρασή σε όλες τις θέσεις του: Το κύκλωμα αναζήτησης που αποτελείται από πολυπλέκτες και συγκριτές γίνεται μεγαλύτερο με αποτέλεσμα οι επιπλέον πύλες να προσθέτουν επιπλέον καθυστέρηση με αποτέλεσμα την μείωση του χρονισμού του επεξεργαστή.

Η οργάνωση set-associative μειώνει την καθυστέρηση λόγω άμεσης επιλογής του set όπου πρέπει να βρίσκεται η μετάφραση που αναζητούμε, και πλήρως-συσχετιστικής αναζήτησης εντός των (λίγων) θέσεων του. Η προσθήκη ενός Entry σε κάποιο set εξαρτάται από το **index** , ενώ εντός του set μπορεί να μπει σε οποιοδήποτε **way** . Αναλυτικά η διαδικασία αναζήτησης Entry εντός του TLB είναι η εξής (υποθέτουμε μέγεθος VPN ως $39 - 12 = 27$ bits αφού ακολουθούμε το σχήμα σελιδοποίησης Sv39 (RV64)):

1. Έστω ότι έχουμε N sets όπου $N = 2^n$. Το μέγεθος (bitwidth) του index για τα N sets θα είναι $W_{index} = n = \log_2 N$.

2. Από την VPN[26 ... 0] αποκόπτουμε τα τελευταία W_{index} bits. Για παράδειγμα εάν έχουμε 8 sets τότε $W_{index} = \log_2 8 = 3$ (διευθυνσιοδότηση 0-7 sets), επομένως κρατάμε τα 3 τελευταία bits της διεύθυνσης.

3. Έστω ότι τα τελευταία 3 bits της διεύθυνσης είναι $110_2 = 6_{10}$. Γνωρίζουμε ότι εάν υπάρχει, η μετάφραση της VPN θα είναι στο set 6. Το κύκλωμα επιλογής του set είναι πρακτικά ένας πολυπλέκτης ο οποίος επιλέγει το set με χρήση του index.

4. Τέλος κάνουμε πλήρως-συσχετιστική αναζήτηση μέσα στο set για το tag της διεύθυνσης, δηλαδή την VPN χωρίς τα τελευταία W_{index} bits.

Σχήμα 4.3: Χωρισμός της VPN σε index και tag.

Αντίστοιχη διαδικασία ακολουθούμε κατά την αποθήκευση Entry στο TLB. Όταν ληφθεί απάντηση από το PTW χωρίζουμε την VPN σε tag και index, το αποθηκεύουμε στο set που δείχνει το index, στην διεύθυνση που προκύπτει από το κύκλωμα επιλογής διεύθυνσης και αντικατάστασης Entry εντός του set όπως θα δούμε παρακάτω. Αφού περιγράψαμε θεωρητικά την λειτουργία του set-associative TLB προχωράμε στην υλοποίηση του.

4.2.1 Σχεδιαστικές επιλογές

Αντιμετώπιση των ειδικών Entries

Η πρώτη σχεδιαστική ερώτηση που προκύπτει αφορά την αντιμετώπιση των Access Exception, Special Entries για ένα set-associative TLB. Δυνατές λύσεις είναι οι παρακάτω:

1. Δημιουργία πίνακα με όλα τα set και Entries συμπεριλαμβανόμενων των Access Exception/Special Entry.

2. Δημιουργία πίνακα μόνο με τα Normal Entries και αποθήκευση των Access Exception/Special Entries εκτός TLB σε ξεχωριστούς καταχωρητές.

Η πρώτη περίπτωση δεν είναι αποδοτική καθώς δεν θα χρησιμοποιούνται τα επιπλέον Special Entries που θα βρίσκονται εντός του πίνακα σε άλλα set και δεν είναι σωστή αρχιτεκτονικά καθώς ο μηχανισμός απαιτεί μοναδικό Special Entry. Η δεύτερη λύση προσθέτει παραπάνω πολυπλοκότητα στο κύκλωμα του TLB λόγω χτισίματος του πίνακα Entries αλλά είναι αποδοτικότερη σε χώρο και συμβατή με την υπάρχουσα αρχιτεκτονική.

Μονάδα αποθήκευσης TLB Entries

Η δεύτερη σχεδιαστική επιλογή που προκύπτει είναι ο τύπος της μονάδας αποθήκευσης του νέου TLB. Μπορούμε να επιλέξουμε ανάμεσα στον τύπο `Reg` της Chisel που αποθηκεύεται σε FFs στο FPGA και στον τύπο `Mem` ο οποίος αποθηκεύεται σε Block RAMs. Το πρόβλημα του τύπου `Mem` είναι ότι έχει καθυστέρηση ένα κύκλο κατά το διάβασμα οπότε το TLB θα έχει καθυστέρηση απάντησης έναν επιπλέον κύκλο σε σχέση με το αρχικό TLB. Το θετικό του τύπου `Mem` είναι ότι απελευθερώνει FFs τα οποία μπορούν να χρησιμοποιηθούν σε άλλες μονάδες. Σε γενικές γραμμές δεν μας συμφέρει να χρησιμοποιήσουμε `Mem` γιατί θα εισάγουμε επιπλέον καθυστέρηση και θα αυξήσουμε την πολυπλοκότητα στην σχεδίαση. Λαμβάνοντας υπόψιν τα παραπάνω επιλέγουμε να χρησιμοποιήσουμε τον τύπο `Reg`, κατασκευάζοντας διανύσματα καταχωρητών για την αποθήκευση των TLB Entries.

4.2.2 Αρχιτεκτονική συστήματος

Για την ελάχιστη δυνατή αλλαγή της εσωτερικής αρχιτεκτονικής του TLB επιλέγουμε την αποθήκευση των ειδικών Entries ξεχωριστά από τα Normal Entries. Η αρχιτεκτονική του νέου TLB παρουσιάζεται στο σχήμα 4.4.

1. Διαφοροποίηση των Normal Entries από τα Access Exception, Special Entry. Χτίζουμε διάνυσμα καταχωρητών το οποίο έχει μέγεθος $nWays * nSets$, στο οποίο θα αναφερόμαστε από εδώ και έπειτα ως Normal Array. Χρησιμοποιούμε τους τύπους `Reg, Vec` της Chisel. Όπως αναφέραμε προηγουμένως ο τύπος `Vec` είναι indexable οπότε ο κώδικας Chisel για την επιλογή του set είναι αρκετά απλός.

2. Δημιουργούμε μονοδιάστατο διάνυσμα καταχωρητών για το Valid bit του Normal Array. Επιλέγουμε το αντίστοιχο set από το Valid array με χρήση του προηγούμενου index.

3. Για τα Access Exception, Special Entry καθώς και για τα Valid bits, χρησιμοποιούμε `Reg` (καταχωρητές) για την αποθήκευση τους, οπότε πλέον δεν θα συμπεριλαμβάνονται στον μηχανισμό των Normal Entries.

4. Έπειτα από αίτηση της Cache για κάποιο Entry, ακολουθούμε την διαδικασία επιλογής set όπως αναφέρθηκε παραπάνω, χρησιμοποιούμε τα τελευταία bits της VPN και παίρνουμε το σωστό set από το Normal Array.

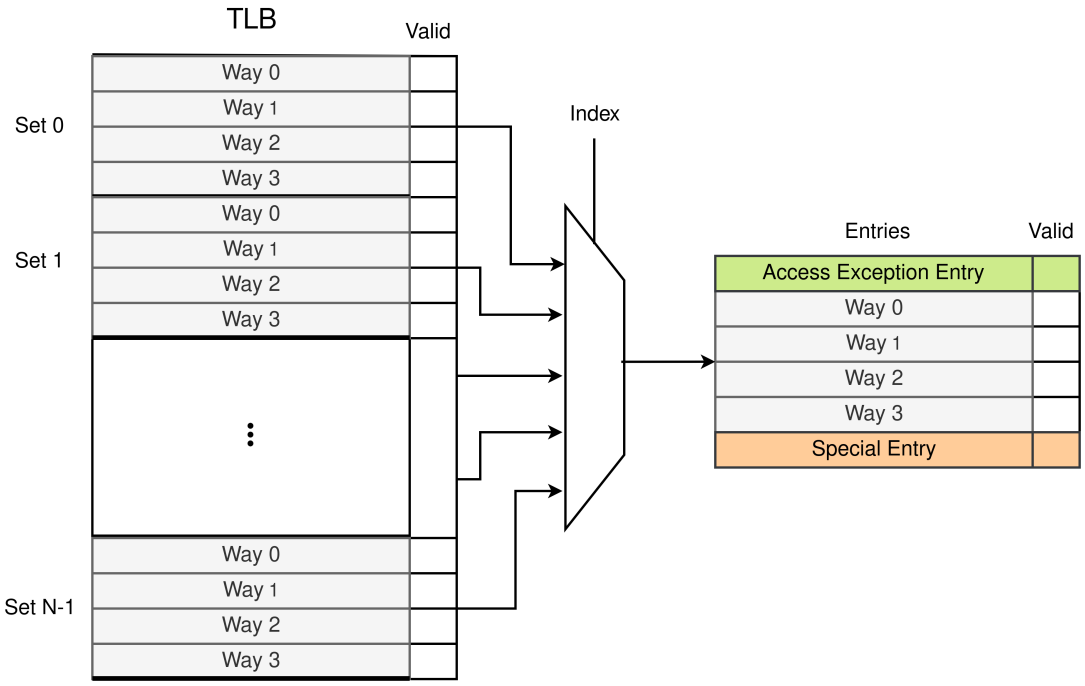
5. Χτίζουμε πίνακα Entries με χρήση τύπου `Wire` της Chisel (χωρίς μνήμη), ο οποίος έχει:

   (α′) Στην θέση 0 το Access Exception Entry

   (β′) Στις θέσεις 1 έως και N το set που επιλέχθηκε από το Normal Array

   (γ′) Στην θέση N + 1 το Special Entry

6. Ο πίνακας Entries είναι αρχιτεκτονικά ίδιος με τον πίνακα του αρχικού set-associative TLB, οπότε τον παραδίδουμε στον μηχανισμό χωρίς περαιτέρω αλλαγές.



Σχήμα 4.4: Σχεδίαση TLB Generator με nWays = 4.

Διαφορές σχεδίασης

Η βασική διαφορά του νέου set-associative TLB είναι ότι περιέχει επιπλέον 2 Entries εκτός των Normal Entries. Κατά την εγγραφή κάποιου νέου Entry ακολουθούμε την παρακάτω διαδικασία:

1. Σε περίπτωση Normal Entry

   (α′) Ο μηχανισμός επιλογής διεύθυνσης εγγραφής εντός του set επιστρέφει μία διεύθυνση, έστω $waddr$

   (β′) Γράφουμε στο TLB το νέο Entry στην διεύθυνση $waddr + (index * nWays)$

   (γ′) Αντίστοιχα, ενημερώνεται το Valid bit array

2. Σε περίπτωση Access Exception το Entry αποθηκεύεται στον καταχωρητή `ae-register` και ενημερώνεται ο καταχωρητής που κρατάει το valid bit.

3. Σε περίπτωση Special Entry αποθηκεύεται στον καταχωρητή `special-register` και ενημερώνεται ο καταχωρητής που κρατάει το valid bit.

Στην περίπτωση που λάβουμε αίτηση `sfence.vma` ακυρώνουμε είτε μοναδική γραμμή είτε ολόκληρο το TLB. Δυνατότητα για ASID flushing θα προσθέσουμε μελλοντικά εφόσον στην εργασία αυτή δεν μελετάμε περιβάλλον που υποστηρίζει πολλαπλά λειτουργικά συστήματα. Όσον αφορά το κύκλωμα αντικατάστασης, αποτελείται από έναν **Priority Encoder** ο οποίος επιστρέφει την πρώτη θέση του set για το οποίο το valid bit είναι μηδέν. Εάν δεν βρεθεί καμία ελεύθερη θέση τότε επιλέγεται μία τυχαία με βάση την πολιτική αντικατάστασης Random Replacement. Υποστήριξη του PseudoLRU για πολλαπλά set θα προσθέσουμε μελλοντικά.

4.3 Συμπεράσματά σχεδίασης παραμετροποιήσιμου TLB

Το TLB που σχεδιάσαμε και υλοποιήσαμε παραμετροποιείται πλήρως αλλάζοντας τις παραμέτρους nSets, nWays. Προφανώς, με **nWays = N, nSets = 1** προκύπτει **fully-associative TLB** και αντίστοιχα με **nWays = 1, nSets = N** προκύπτει **direct-mapped TLB** οι οποίες είναι οι ακραίες περιπτώσεις του set associative TLB. Η υλοποίηση του αρχικού TLB χρησιμοποιούσε κατασκευαστές της Chisel2 ενώ η υλοποίηση μας έγινε στην Chisel3 καθώς είναι βελτιωμένη συντακτικά και έχει περισσότερες δυνατότητες. Εκτός από τα παραπάνω παράγει αποδοτικότερο FIRRTL με αποτέλεσμα να χρησιμοποιεί λιγότερους πόρους του FPGA και να υπάρχει βελτίωση στην επίδοση, όπως θα δούμε στο κεφάλαιο 5. Τέλος, λόγω των μειωμένων tag bits καθώς και της επιλογής πολιτικής αντικατάστασης Random Replacement[2] μειώνονται οι πόροι που χρησιμοποιούνται σε σχέση με το αρχικό TLB.

[2]Η πολιτική αντικατάστασης PseudoLRU που χρησιμοποιεί το αρχικό TLB καταναλώνει πόρους (FFs) για την αποθήκευση της κατάστασης

Κεφάλαιο 5

Ανάλυση επίδοσης TLB

Στο κεφάλαιο αυτό θα αναλύσουμε την επίδοση του αρχικού TLB καθώς και του παραμετροποιήσιμου TLB σε σχέση με μετρικές της πλατφόρμας FPGA και μετροπρογραμμάτων του SPEC2006.

5.1 Μετρικές Ανάλυσης

Οι μετρικές που θα χρησιμοποιηθούν στην παρούσα εργασία χωρίζονται σε τρεις ομάδες, στις μετρικές επιδόσεων, στις μετρικές χώρου (area) που προκύπτουν από την υλοποίηση στο FPGA καθώς και στον συνδυασμός τους.

- **Μετρικές πόρων**: Προκύπτουν από την ανάλυση του Implementation στο Xilinx Vivado 2018.1 και αφορούν τους πόρους της πλατφόρμας FPGA που απαιτεί το κύκλωμα που σχεδιάσαμε. Αποτελούνται από τα παρακάτω:
  - **Look-up-Tables (LUTs)** : Στοιχεία λογικής, συνθέτουν την λειτουργικότητά του κυκλώματος Δεν διαθέτουν μνήμη.
  - **Flip-Flop (FFs)** : Στοιχεία αποθήκευσης πληροφορίας (καταχωρητές), χρησιμεύουν στην διατήρηση πληροφορίας ανά κάθε επεξεργαστικό κύκλο.
- **Μετρικές επίδοσης**: Εξέταση της σουίτας μετροπρογραμμάτων SPEC2006 [9]
  - **Critical Path** : Προκύπτει αφαιρώντας από το Timing Constraint που θέσαμε στην σχεδίαση το Worst Negative Slack (εφόσον είναι Θετικό).
  - **TLB misses** : Οι συνολικές αστοχίες TLB κατά την εκτέλεση ενός προγράμματος.
  - **Total cycles** : Οι συνολικοί κύκλοι επεξεργασίας κατά την εκτέλεση ενός προγράμματος.
  - **Critical Path * Total Cycles** : Από τους συνολικούς κύκλους που προέκυψαν βρίσκουμε τον συνολικό χρόνο που χρειάζεται η σχεδίαση για να ολοκληρώσει το κάθε μετροπρόγραμμα, σύμφωνα με τον χρονισμό του κυκλώματος.

Σημειώνουμε ότι δεν θα ασχοληθούμε με την κατανάλωση πόρων σε Block RAMs καθώς διατηρείται σταθερή. Η σχεδίαση μας δεν έχει στοιχεία αποθήκευσης σε μνήμες SRAM, αλλά μόνο σε καταχωρητές (FFs).

Αρχικά θα εξετάσουμε το αρχικό TLB του Rocket Chip όσον αφορά τον αριθμό των Entries, έπειτα θα ελέγξουμε τις διαφορές μεταξύ του αρχικού TLB και της υλοποίησης μας αναλόγως nSets, nWays και θα σχολιάσουμε τα μεγέθη 32, 128 και 512 Entries για τις δύο υλοποιήσεις του TLB. Να σημειώσουμε ότι θα ασχοληθούμε μόνο με το Data TLB, το Instruction TLB θα είναι σταθερό στα 32 Entries. Θα εξετάσουμε συγκεκριμένα μετροπρογράμματα της σουίτας SPEC2006 καθώς είτε κάποια από αυτά δεν γίνονταν επιτυχώς compile με τα διαθέσιμα toolchains, είτε κάποια από αυτά εμφανίζουν αστοχίες TLB τάξης χιλιάδων.

Την ποσοτική διαφορά των τιμών που εξετάζουμε θα την υπολογίζουμε με την φόρμουλα:

$$PercentIncrease = \frac{NewValue - OriginalValue}{OriginalValue} * 100$$

Η φόρμουλα αυτή μπορεί να πάρει αρνητικές ή θετικές τιμές τις οποίες θα τις αναφέρουμε ώς βελτίωση ή αντίστοιχά επιδείνωσή.

5.2 Ανάλυση αρχικού fully-associative TLB

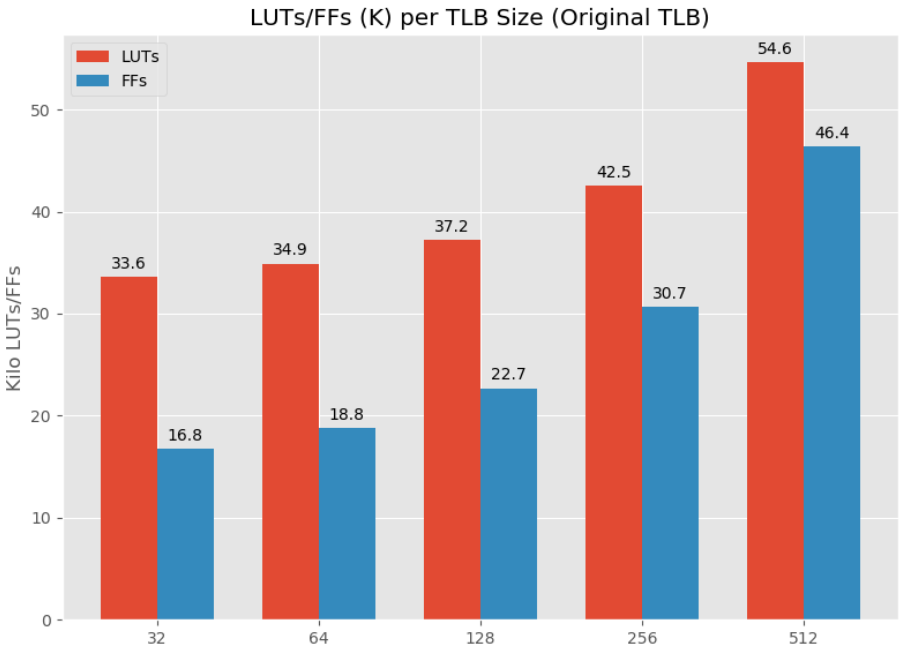
Επιλέγουμε διαφορετικά μεγέθη TLB και ελέγχουμε τα αποτελέσματα που παίρνουμε από την φάση Implementation του Vivado. Όπως είναι αναμενόμενο τα LUTs/FFs αυξάνονται ανά επιλογή μεγέθους όπως βλέπουμε στο σχήμα 5.1, με 62.5% αύξηση LUTs και 176.2% αύξηση FFs από 32 σε 512 Entries. Η πολλαπλάσια αύξηση στα FFs σε σχέση με τα LUTs δικαιολογείται αφού ο τύπος `Reg` της Chisel μεταφράζεται στο FPGA σε FFs, ενώ η αύξηση στα LUTs υπάρχει λόγω του πολυπλοκότερου κυκλώματος επιλογής Entry. Όσον αφορά το critical path (σχήμα 5.2) εμφανίζει αύξηση 102.5% από 32 σε 512 Entries, δηλαδή ο χρονισμός της σχεδίασης πέφτει στο μισό. Για το mcf, οι αστοχίες TLB μειώνονται 76.5% από 32 σε 512 Entries όπως φαίνεται στο σχήμα 5.3.

5.3 Σύγκριση υλοποιήσεων TLB

5.3.1 Ανάλυση πόρων και critical path

Για την σύγκριση μεταξύ των δύο υλοποιήσεων TLB επιλέγουμε να ελέγξουμε την περίπτωση σταθερού μεγέθους 512 Entries. Λόγω μεγάλου μεγέθους TLB θα είναι εμφανείς οι αλλαγές στην κατανάλωση πόρων του FPGA. Οι συνδυασμοί set, ways που επιλέγουμε να ελέγξουμε είναι οι παρακάτω:

- Fully-associative
- 64 Sets, 8 Ways
- 128 Sets, 4 Ways



Σχήμα 5.1: LUTs, FFs ανά διαφορετικά μεγέθη στο αρχικό TLB

- Direct-mapped

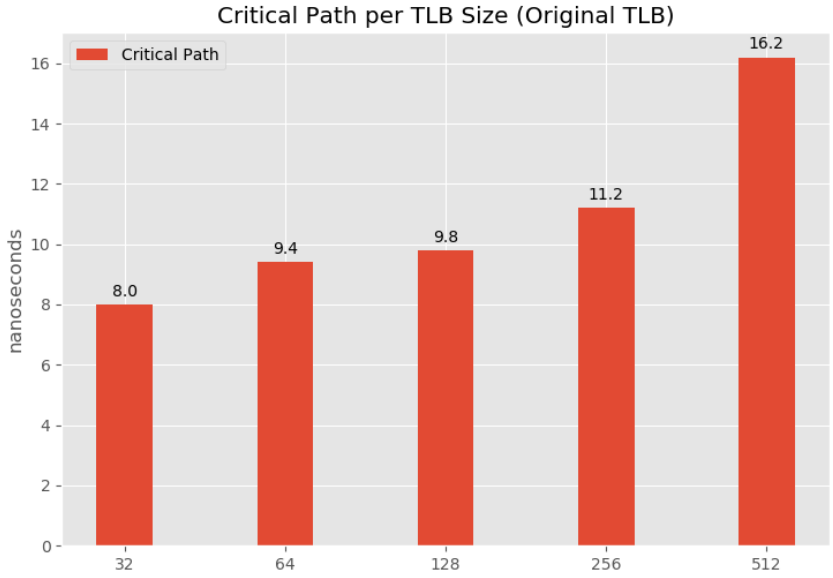
Στο σχήμα 5.4 παρουσιάζουμε των αριθμό LUTs/FFs για τις παραπάνω παραμέτρους. Το καλύτερο configuration σε θέμα LUTs/FFs βλέπουμε ότι είναι το 128 Sets - 4 Ways με μείωση κατά 21.6% στα LUTs και 8.6% στα FFs σε σχέση με το αρχικό TLB. Η μείωση στα FFs υπάρχει λόγω των μειωμένων tag bits. Η μείωση στα LUTs συμβαίνει λόγω απλοποίησης του μηχανισμού, αφού η fully-associative αναζήτηση γίνεται σε set μεγέθους 4. Η χρήση πόρων μεταξύ των οργανώσεων 64 sets / 8 ways, 128 sets / 4 ways και direct-mapped είναι περίπου ίδια.

Η οργάνωση direct-mapped δεσμεύει τα λιγότερα FFs λόγω των μειωμένων tag bits. Απελευθερώνονται $W_{index} = log_2 512 = 9\,bits$ από το tag, με αποτέλεσμα να χρησιμοποιούμε $9 * 512 \approx 4.5\,Kb$ λιγότερο χώρο σε FFs. Από την άλλη πλευρά, λόγω άμεσης-απεικόνισης ο πολυπλέκτης επιλογής Entry είναι μεγαλύτερος σε σχέση με των περιπτώσεων 64 sets / 8 ways, 128 sets / 4 ways για αυτό παρατηρούμε μικρή αύξηση στα LUTs σε σχέση με αυτές.

Βελτίωση 27.8% παρατηρούμε στο critical path για 64 Sets - 8 Ways, direct-mapped όπως φαίνεται στο σχήμα 5.4. Ενδιαφέρουσα είναι η μείωση του critical path από το αρχικό TLB σε σχέση με την υλοποίηση μας για fully-associative TLB μεγέθους 512. Παρατηρούμε μείωση 14.2% η οποία οφείλεται στα παρακάτω:

1. Χρήση κατασκευαστών της Chisel3 αντί της Chisel2 στην οποία είναι υλοποιημένο το αρχικό TLB οι οποίοι όπως φαίνεται παράγουν πιο αποδοτικό FIRRTL.

2. Αντικατάσταση μηχανισμού fully-associative PseudoLRU ο οποίος αποθηκεύει τις πληροφορίες σε δομή `Reg` που δεσμεύει FFs, με την Random Replacement η οποία δεν



Σχήμα 5.2: Critical Path ανά διαφορετικά μεγέθη στο αρχικό TLB

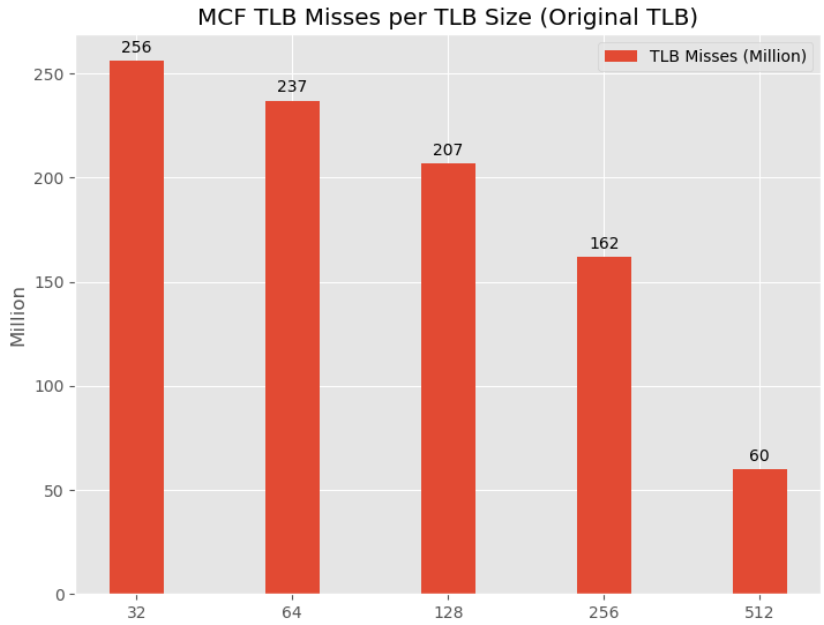
αποθηκεύει πληροφορία, ελευθερώνοντας πόρους αλλά θυσιάζοντας επίδοση σε θέμα ευστοχιών TLB.

5.3.2 Ανάλυση επίδοσης

Σε γενικές γραμμές τα μετροπρογράμματα της σουίτας SPEC2006 εμφανίζουν μείωση στις ευστοχίες TLB της υλοποίησης μας λόγω χρήσης Random Replacement πολιτικής αντικατάστασης, έναντι του PseudoLRU του αρχικού TLB. Όπως έχουμε αναφέρει κατά την διάρκεια της παρούσας εργασίας ο Rocket Chip Generator είναι πρακτικά μία βιβλιοθήκη, οπότε προφέρει διαφορετικές τεχνικές αντικατάστασης. Το πρόβλημα είναι ότι η υλοποίηση PseudoLRU για set-associative δομές (όπως είναι η Cache) χρησιμοποιεί την δομή `Mem`, η οποία απαντάει στο αίτημα read τον επόμενο θετικό παλμό ενώ η δομή `Reg` απαντάει στον ίδιο κύκλο. Το πρόβλημα έγκειται στο ότι η μνήμη `Mem` είναι σύγχρονη, δηλαδή απαντάει σύμφωνα με τον παλμό ρολογιού: Για να γίνεται σύγχρονο διάβασμα, στην έξοδο της μνήμης `Mem` υπάρχει ένας καταχωρητής στον οποίο γίνεται η εγγραφή του αιτήματος read, και τον επόμενο κύκλο βλέπουμε την απάντηση. Λόγω αυτού του προβλήματος δεν μπορούμε να χρησιμοποιήσουμε το PseudoLRU που είναι ήδη υλοποιημένο. Αλλαγή από `Mem` σε `Reg` είναι δυνατή, αλλά κάνουμε την επιλογή να μην την χρησιμοποιήσουμε για λιγότερη κατανάλωση LUTs/FFs, γιατί όπως αναφέραμε η δομή `Reg` αποθηκεύεται σε FFs.

Ανάλυση συνδυασμών TLB

Οι συνδυασμοί μεγεθών και οργανώσεων που επιλέγουμε να αναλύσουμε είναι οι παρακάτω:



Σχήμα 5.3: Αστοχίες TLB ανά διαφορετικά μεγέθη στο αρχικό TLB για το mcf SPEC2006

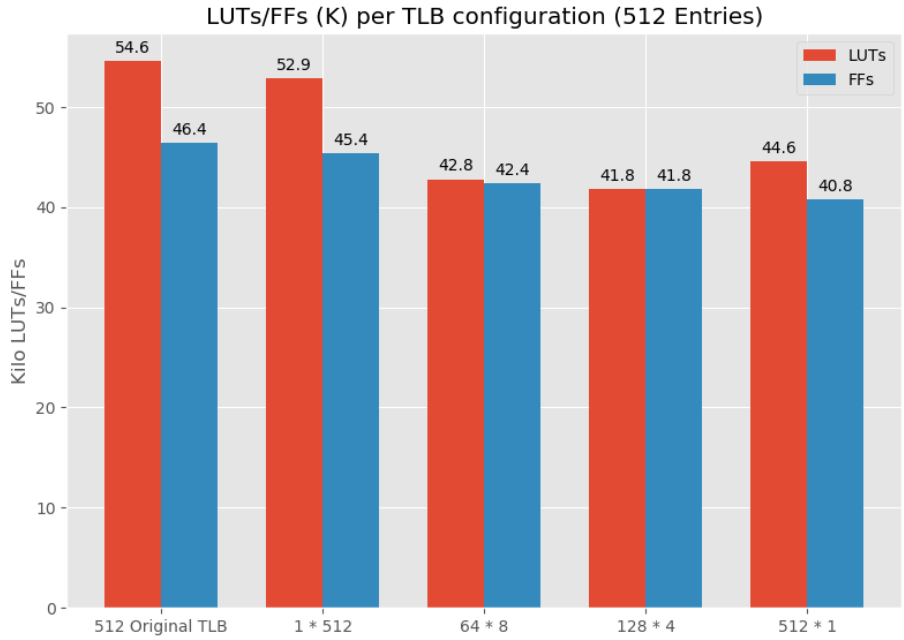
- **32 Entries** : Για τις δύο υλοποιήσεις επιλέγουμε οργάνωση fully-associative, ελέγχοντας την διαφορά στο critical path για τις δύο υλοποιήσεις.

- **128 Entries** : Για τις δύο υλοποιήσεις επιλέγουμε οργάνωση fully-associative με αλλαγή της πολιτικής αντικατάστασης του αρχικού TLB σε Random Replacement. Με αυτό τον τρόπο θα ελέγξουμε την ορθότητα του μηχανισμού που σχεδιάσαμε, αναμένοντας ίδια αποτελέσματα επίδοσης στα μετροπρογράμματα της σουίτας SPEC2006.

- **512 Entries** : Για την υλοποίηση μας επιλέγουμε διαφορετικούς συνδυασμούς οργανώσεων όπως θα δούμε παρακάτω για να ελέγξουμε την επίδοση τους σε σχέση με το αρχικό TLB.

32 Entries

Όσον αφορά fully-associative TLB με 32 Entries, το αρχικό TLB είναι 15% γρηγορότερο από το παραμετροποιήσιμο TLB στο critical path (από 8ns σε 9.2ns). Αυτό συμβαίνει λόγω πολυπλοκότερου τρόπου στησίματος και προετοιμασίας του πίνακα Entries, διαδικασία η οποία μεταφράζεται σε περισσότερες πύλες οπότε και μεγαλύτερη καθυστέρηση. Η εξέταση των ακραίων περιπτώσεων παρουσιάζει την διαφορά όπως θα δούμε αργότερα, εξετάζοντας πολλά Entries παρατηρούμε βελτίωση στο critical path λόγω αποδοτικότερης υλοποίησης και βελτιώσεων της Chisel, ενώ στα 32 Entries φαίνεται η καθυστέρηση λόγω πολυπλοκότερης υλοποίησης του παραμετροποιήσιμου TLB.



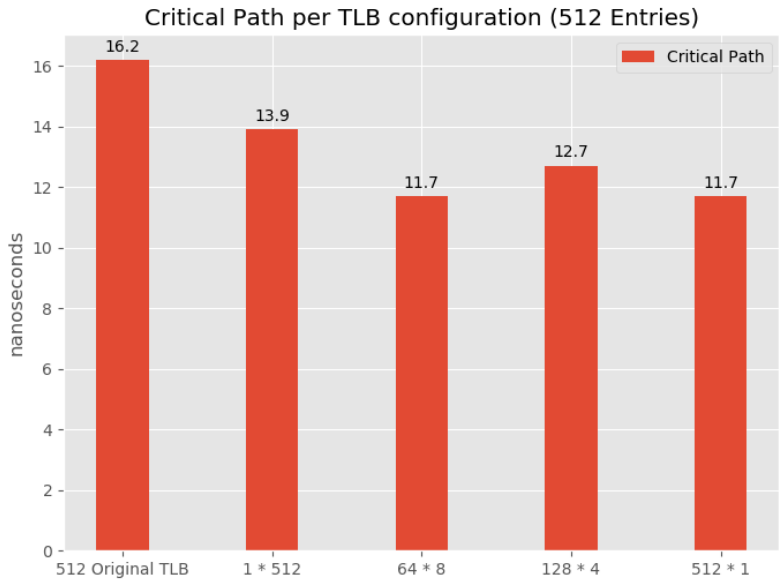
Σχήμα 5.4: LUTs/FFs συγκριτικά με το αρχικό και το παραμετροποιήσιμο TLB

128 Entries

Για τον έλεγχο επίδοσης του αρχικού TLB σε σχέση με την υλοποίηση μας επιλέγουμε μέγεθος 128 Entries fully-associative και πολιτική αντικατάστασης Random Replacement και στους δύο σχεδιασμούς για ελέγξουμε την ορθότητα του μηχανισμού. Επιλέγουμε μετροπρογράμματα που εμφανίζουν περισσότερες αστοχίες TLB για να είναι εμφανείς οι διαφορές. Τα αποτελέσματα από τα μετροπρογράμματα που επιλέξαμε φαίνονται στον πίνακα 5.1. Όπως είναι αναμενόμενο τα αποτελέσματα που λαμβάνουμε διαφέρουν ελάχιστα με διαφορές τάξης μεγέθους 0.01%.

Benchmark	Original TLB misses (Million)	TLB Generator misses (Million)
cactusADM	1,315	1,314
sjeng	11,41	11,42
astar	138,61	138,62
h264	23,855	23,853
sphinx3	5,038	5,039
mcf	212,68	212,68

Πίνακας 5.1: Αποτελέσματα SPEC2006 για 128 Entries fully-associative TLB και στις δύο υλοποιήσεις, με μηχανισμό αντικατάστασης Random Replacement



Σχήμα 5.5: Critical Path συγκριτικά με το αρχικό και το παραμετροποιήσιμο TLB

512 Entries

Στην περίπτωση των 512 Entries η πολιτική αντικατάστασης της υλοποίησης μας είναι Random Replacement ενώ για το αρχικό TLB διατηρούμε την PseudoLRU.

Η σημαντικότερη παρατήρηση είναι η μείωση αστοχιών TLB 16.7% στο mcf στην σχεδίαση μας, η οποία πιθανότατα οφείλεται στην αφαίρεση του Access Exception Entry από το TLB ελευθερώνοντας την θέση 0. Προσθέσαμε μία επιπλέον θέση στο TLB οπότε πλέον δεν επηρεάζει ο μηχανισμός του AE Entry τα Normal Entries στο TLB για την συγκεκριμένη περίπτωση. Επειδή το AE Entry αποθηκεύεται πάντα στην θέση 0 του TLB στην αρχική υλοποίηση, υπάρχει μεγάλη πιθανότητα να ακυρώνει κάποιο έγκυρο Normal Entry το οποίο βρίσκεται στην θέση 0. Ως αποτέλεσμα, ενώ το TLB έχει ελεύθερες θέσεις το AE Entry μπορεί να ακυρώνει χρήσιμα Entries τα οποία πιθανότατα θα οδηγήσουν σε μελλοντικές αστοχίες TLB. Η αφαίρεση του AE Entry από τον μηχανισμό των Normal Entries βελτιώνει επομένως την επίδοση του TLB. Ακόμα και αν χρησιμοποιούμε την λιγότερο αποδοτική πολιτική Random Replacement η υλοποίηση μας εμφανίζει λιγότερες αστοχίες TLB στην περίπτωση του mcf.

Για όλα τα μετροπρογράμματα που εξετάζουμε παρατηρούμε αστοχίες TLB που κυμαίνονται από -16.7% (λιγότερες αστοχίες) έως 20.6% (περισσότερες αστοχίες) ποσοτικής μείωσης/αύξησης σε σχέση με το αρχικό TLB. Ερμηνεύουμε το αποτέλεσμα ως απόρροιά της αφαίρεσης του AE Entry από το TLB (βελτίωση) καθώς και της χρήσης πολιτικής αντικατάστασης Random Replacement (επιδείνωσή) η οποία είναι λιγότερο αποδοτική έναντι της PseudoLRU. Το ποσοστό βελτίωσης/επιδείνωσης των αστοχιών TLB της υλοποίησης μας έναντι του αρχικού TLB φαίνεται στον πίνακα 5.2. Συνδυάζοντας τα αποτελέσματα του πίνακα 5.2 με τα αποτελέσματα του πίνακα 5.1 παρατηρούμε ότι εάν η υλοποίηση μας χρησιμοποιούσε

πολιτική αντικατάστασης PseudoLRU θα εμφάνιζε σαφή βελτίωση στις συνολικές αστοχίες TLB λόγω αποδοτικότερης αντιμετώπισης του AE Entry.

Να σημειώσουμε ότι γίνεται εμφανής η βελτίωση στις αστοχίες TLB σε κάποιες περιπτώσεις (xalancbmk, mcf) στην σχεδίαση μας λόγω μεγαλύτερου μεγέθους TLB. Η διαφορά αυτή δεν είναι εμφανής στην περίπτωση των 128 θέσεων για τον λόγο ότι τα TLB δεν είναι αρκετά μεγάλα ώστε να φανούν οι αρχιτεκτονικές διαφορές τους: Κάποιες αστοχίες είναι υποχρεωτικές λόγω μικρού μεγέθους TLB.

Benchmark	TLB Miss Percentage difference (SPEC2006)
cactusADM	19.1%
sjeng	16.3%
astar	20.6%
xalancbmk	-4.2%
h264	-0.1%
sphinx3	11,7%
mcf	-16.7%

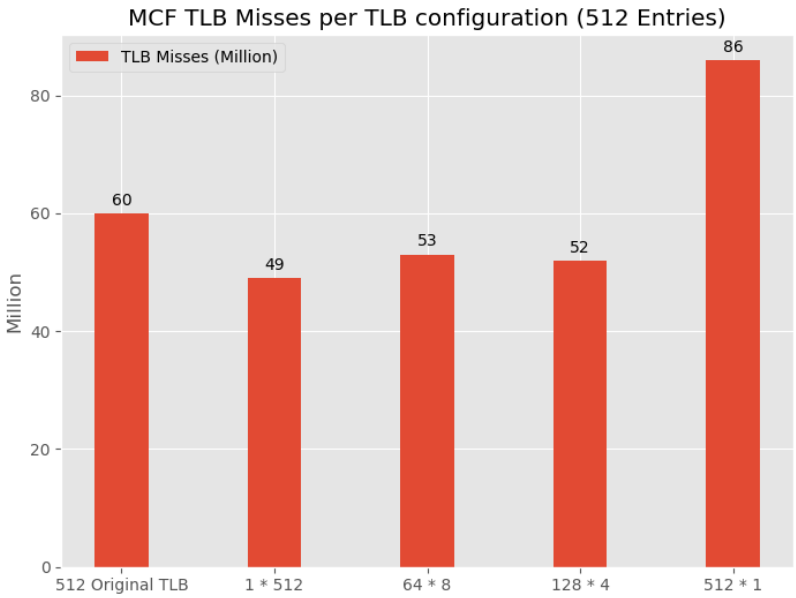
Πίνακας 5.2: Επιδείνωσή/βελτίωση αστοχιών παραμετροποιήσιμου TLB σε μετροπρογράμματα του SPEC2006 σε σχέση με το αρχικό TLB. Αρνητικό πρόσημο σημαίνει βελτίωση στην υλοποίηση μας (λιγότερες αστοχίες)

Πολλαπλασιάζοντας το critical path με τους συνολικούς κύκλους ολοκλήρωσης του προγράμματος προκύπτει ο χρόνος ολοκλήρωσης προγράμματος όπως βλέπουμε στο σχήμα 5.7 για το mcf. Το critical path είναι ο καθοριστικός παράγοντας χρονισμού του επεξεργαστικού πυρήνα[1] και τελικά της επίδοσης του, γιατί με σταθερό αριθμό των επεξεργαστικών κύκλων ο παράγοντας που ρυθμίζει την επίδοση είναι το critical path. Συνδυάζοντας το μειωμένο critical path με την ύπαρξη λιγότερων αστοχιών TLB που οδηγούν σε λιγότερους κύκλους προγράμματος βλέπουμε **μείωση χρόνου ολοκλήρωσης 28.5% από το αρχικό TLB στο παραμετροποιήσιμο με 64 sets - 8 ways**. Η δεύτερη καλύτερη επίδοση χρόνου ολοκλήρωσης αποτελεί **η περίπτωση του direct-mapped TLB με 26% καλύτερο χρόνο σε σχέση με το αρχικό TLB, αν και εμφανίζει 43.3% περισσότερες αστοχίες TLB**. Στον πίνακα 5.3 συνοψίζονται οι διαφορές στην βελτίωση χρόνου περάτωσης του mcf σε σχέση με το αρχικό TLB.

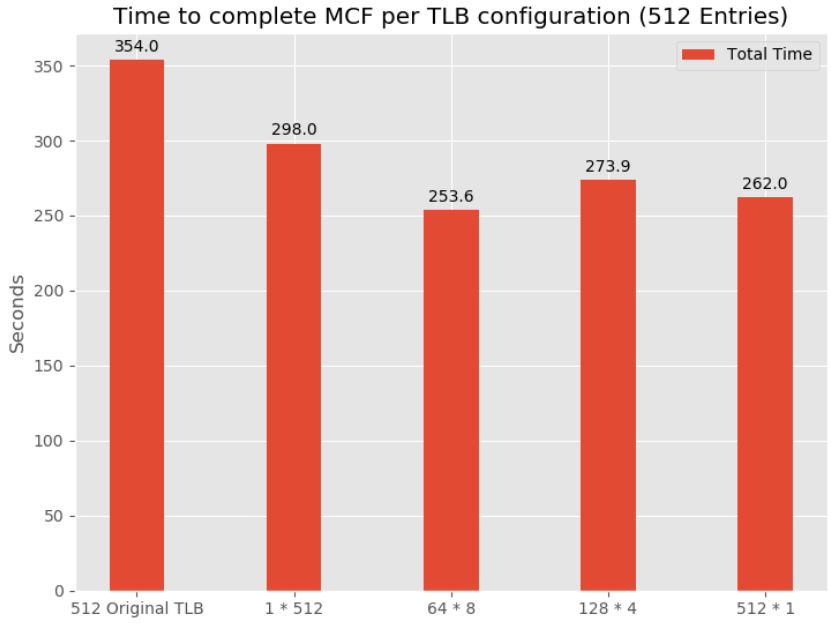
5.4 Σύνοψη αποτελεσμάτων

Παρατηρούμε άμεση βελτίωση στο critical path όπως στοχεύσαμε στην αρχική υπόθεση μας, καθώς και μείωση στην κατανάλωση πόρων του FPGA. Οι πόροι που ελευθερώνονται

[1] Στην πραγματικότητα ο χρονισμός στο FPGA είναι 20ns για όλες τις περιπτώσεις. Το critical path προκύπτει όπως αναφέραμε από την σχέση $CriticalPath = TimingConstraint - WorstNegativeSlack$, οπού το $TimingConstraint$ είναι ο χρονισμός που προσπαθούμε να επιτύχουμε και το $WorstNegativeSlack$ είναι ο επιπλέον δυνατός χρόνος που μπορεί να αφαιρεθεί, ώστε να πετύχουμε τον υψηλότερο δυνατό χρονισμό του κυκλώματος για το mapping στο FPGA(προκύπτει από το Timing Report του Vivado)



Σχήμα 5.6: TLB Misses στο mcf SPEC2006 συγκριτικά με το αρχικό και το παραμετροποιήσιμο TLB



Σχήμα 5.7: Χρόνος να ολοκληρωθεί το mcf SPEC2006 συγκριτικά με αρχικό και το παραμετροποιήσιμο TLB

μπορούν να χρησιμοποιηθούν για να προσθέσουμε παραδείγματος χάρη επιπλέον επεξεργαστικούς πυρήνες σε περίπτωση που θέλουμε να τεστάρουμε κάποια multicore σχεδίαση. Ένα από τα σημαντικότερα αποτελέσματα είναι η δυνατότητα δημιουργίας TLB με πολλά Entries

Παράμετροι TLB	Βελτίωση χρόνου περάτωσης mcf
Fully-Associative	15.8%
64 Sets, 8 Ways	28.5%
128 Sets, 4 Ways	22.9%
Direct-Mapped	26%

Πίνακας 5.3: Βελτίωση χρόνου περάτωσης του mcf σε σύγκριση με το αρχικό TLB

χωρίς να επιβαρύνεται η σχεδίαση μας λόγω της πολυπλοκότητας του fully-associative TLB. Παρατηρούμε ότι για μεγαλύτερά μεγέθη η σχεδίαση μας είναι αποδοτικότερη σε σχέση με μικρότερα λόγω πολυπλοκότερου μηχανισμού προετοιμασίας του set.

Η μέγιστη βελτίωση 27.8% στο critical path για την περίπτωση των 512 Entries δείχνει την αποδοτικότητα του set-associative μηχανισμού στην απλοποίηση του κυκλώματος εύρεσης μετάφρασης χωρίς να αυξάνεται ο αριθμός των αστοχιών (εκτός της περίπτωσης direct-mapped TLB). Ακόμα και αν αυξάνονται 43.3% οι αστοχίες στην περίπτωση του direct-mapped TLB ο καθοριστικός παράγοντας της επίδοσης οφείλεται τελικά στο critical path [10]. Ο συνδυασμός των δύο παραγόντων όμως τελικά θα καθορίσει την μέγιστη επίδοση όπως βλέπουμε στην περίπτωση των 64 sets / 8 ways η οποία συνδυάζει την βελτίωση στο critical path με τις μειωμένες αστοχίες TLB.

Κλείνοντας, αναφέρουμε ότι τα παραπάνω προκύπτουν από την σχεδίαση στην πλατφόρμα FPGA, υλοποίηση σε ASIC με λεπτομερή ανάλυση και σχεδίαση των στοιχείων του κυκλώματος περιμένουμε ότι θα προκύψει περαιτέρω βελτίωση στο critical path. Η αρχιτεκτονική της πλατφόρμας FPGA περιέχει LUTs σταθερού μεγέθους (πρακτικά πολυπλέκτες) τα οποία υλοποιούν την λογική του κυκλώματος και είναι εύλογο ότι θα εμφανίζουν χειρότερη επίδοση σε σχέση με μία fine-grained σχεδίαση σε ASIC.

Κεφάλαιο 6

Μελλοντικές Επεκτάσεις

Στο κεφάλαιο αυτό θα παρουσιάσουμε τις πιθανές μελλοντικές επεκτάσεις για το παραμετροποιήσιμο TLB, καθώς και το Rocket Chip Generator.

6.1 Αναβάθμιση σε νεότερη έκδοση του Rocket Chip

Από το έτος 2018 που ξεκίνησε η παρούσα εργασία, ο Rocket Chip Generator έχει αναβαθμιστεί αρκετά (επιπλέον 1500 commits στο Github, περίπου 3 την μέρα), διαφορές οι οποίες όπως είναι φυσικό δεν μπορούσαν προστεθούν όλες λόγω συμβατότητας. Η έκδοση που χρησιμοποιήθηκε είναι ¨παγωμένη¨, οπότε θα είχε νόημα η προσθήκη των αλλαγών σε νεότερη έκδοση του Rocket Chip που υποστηρίζει περισσότερα features. Επίσης μία μελέτη πάνω στο παραμετροποιήσιμο TLB θα μπορούσε να βελτιώσει ακόμα περισσότερο την επίδοση του, καθώς βελτιώνεται η Chisel και ο FIRRTL.

Μία μελλοντική επέκταση η οποία θα βελτίωνε την επίδοση του παραμετροποιήσιμου TLB θα ήταν η υλοποίηση αποδοτικότερων πολιτικών αντικατάστασης όπως είναι η Least Recently Used (PseudoLRU) για set-associative TLB.

6.2 Εξερεύνηση χώρου σχεδίασης

Καθώς μιλάμε για επαναχρησιμοποιήσιμα μέρη επεξεργαστών, το παραμετροποιήσιμο L1 TLB μπορεί να χρησιμοποιηθεί από επεξεργαστές υψηλότερων επιδόσεων όπως τον BOOM, με δυνατό ένα Design Space Exploration με βάση πιο απαιτητικά workloads. Επίσης λόγω της δυνατότητας του Rocket Chip Generator να παράγει πολυπύρηνα συστήματα θα ήταν εύλογη η έρευνα αποδοτικών συνδυασμών ετερογενών συστημάτων, με απλούς, αργούς καθώς και πολύπλοκους, γρήγορους επεξεργαστές.

Στην παρούσα εργασία εξετάσαμε την επίδοση μόνο του Data TLB, μία μελλοντική έρευνα θα μπορούσε να ασχοληθεί με την επίδοση του Instruction TLB σε σχέση με τις παραμέτρους sets, ways για μετροπρογράμματα της σουίτας SPEC2006.

6.3 Επιπλέον παραμετροποίηση

Χώρος για επιπλέον παραμετροποίηση υπάρχει, καθώς το L2 TLB και η PTW Cache έχουν μοναδική παράμετρο τα Entries. Το L2 TLB επίσης είναι υλοποιημένο με χρήση του τύπου δεδομένων `Mem` της Chisel. Ο τύπος `Mem` είναι μία σύγχρονη addressable μνήμη η οποία υλοποιείται στο FPGA σε Block RAMs. Για να υποστηρίζει σύγχρονο γράψιμο/διάβασμα εισάγει καθυστέρηση ενός κύκλου και για αυτό το L2 TLB έχει κόστος hit 2 κύκλους. Θα μπορούσαμε να υλοποιήσουμε το L2 TLB ώστε να χρησιμοποιεί διανύσματα καταχωρητών στα οποία η εγγραφή εμφανίζεται στην έξοδο τον επόμενο κύκλο. Επιπλέον θα μπορούσαμε να χτίσουμε ένα L2 TLB Generator, γιατί η direct-mapped οργάνωση οδηγεί σε χειρότερη επίδοση ευστοχιών TLB χωρίς να παρέχεται δυνατότητα παραμετροποίησης. Αντίστοιχα, θα μπορούσαμε να υλοποιήσουμε παραμετροποιήσιμη Page Table Walk Cache διαφορετικών αρχιτεκτονικών [3] εφόσον υπάρχει πεδίο έρευνας και ανάλυσης πάνω στο θέμα αυτό.

6.4 Προηγμένα σχήματα διαχείρισης εικονικής μνήμης

Πέρα από την κλασσική μορφή του συστήματος διαχείρισης εικονικής μνήμης θα είχε ενδιαφέρον να υλοποιηθούν πιο προηγμένα σχήματα διαχείρισης εικονικής μνήμης για την αύξηση της επίδοσης ενός επεξεργαστή σε πιο σύνθετα workloads όπως είναι τα Coalesced Large-Reach TLBs [14], τα Redundant Memory Mappings [11] και να εξεταστούν έναντι σε υπάρχοντα, παραδείγματος χάρη τα Direct Segments [12] που έχουν ήδη υλοποιηθεί στον Rocket Chip Generator. Η εκτίμηση των αποτελεσμάτων επίδοσης, της μελέτης κατανάλωσης πόρων και της επίδοσης του critical path έχει ιδιαίτερο ερευνητικό ενδιαφέρον στα πιο προηγμένα σχήματα. Παραπέμπουμε τον αναγνώστη να ανατρέξει στις αντίστοιχες δημοσιεύσεις.

Βιβλιογραφία

[1] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo και Andrew Waterman. The Rocket Chip Generator. Τεχνική Αναφορά υπ. αρίθμ. ΥCB/EECS-2016-17, EECS Department, University of California, Berkeley, 2016.

[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek και Krste Asanović. Chisel: Constructing hardware in a scala embedded language. Στο *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, σελίδες 1216–1225, New York, NY, USA, 2012. ACM.

[3] Thomas Barr, Alan Cox και Scott Rixner. Translation caching: Skip, don't walk (the page table). τόμος 38, σελίδες 48–59, 2010.

[4] Christopher Celio, David A. Patterson και Krste Asanović. The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor. Τεχνική Αναφορά υπ. αρίθμ. ΥCB/EECS-2015-167, EECS Department, University of California, Berkeley, 2015.

[5] Wikipedia contributors. Static timing analysis — Wikipedia, the free encyclopedia, 2019.

[6] Andrew Waterman and Krste Asanovic. The RISC-V Instruction Set Manual, Volume 1: User-Level ISA, Document Version 2.2. RISC-V Foundation, 2017.

[7] Daniel D. Gajski και Loganath Ramachandran. Introduction to High-Level Synthesis. *IEEE Des. Test*, 11(4):44–54, 1994.

[8] Neel Gala, Arjun Menon, Rahul Bodduna, G. S. Madhusudan και V. Kamakoti. Shakti processors: An open-source hardware initiative. Στο *Proceedings of the 2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*, VLSID '16, σελίδες 7–8, Washington, DC, USA, 2016. IEEE Computer Society.

[9] John L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

[10] Mark D. Hill. A Case for Direct-Mapped Caches. *Computer*, 21(12):25–40, 1988.

[11] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift και Osman Ünsal. Redundant memory mappings for fast access to large memories. Στο *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, σελίδες 66–78, New York, NY, USA, 2015. ACM.

[12] Nikhita Kunati και Michael M. Swift. Implementation of Direct Segments on a RISC-V Processor. Στο *In Proceedings of Second Workshop on Computer Architecture Research with RISC-V*, CARRV 2018, χ.χ.

[13] David Patterson και Andrew Waterman. *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon, 2017.

[14] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel και Abhishek Bhattacharjee. Colt: Coalesced large-reach tlbs. Στο *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, σελίδες 258–269, Washington, DC, USA, 2012. IEEE Computer Society.

[15] D. Rossi, F. Conti, A. Marongiu, A. Pullini, I. Loi, M. Gautschi, G. Tagliavini, A. Capotondi, P. Flatresse και L. Benini. Pulp: A parallel ultra low power platform for next generation iot applications. Στο *2015 IEEE Hot Chips 27 Symposium (HCS)*, σελίδες 1–39, 2015.

[16] Sifive. RISC-V Core IP Products, 2017.

[17] Sifive. U54-MC Manual, 2019.

[18] Wikipedia contributors. Berkeley risc — Wikipedia, the free encyclopedia, 2019.

[19] Wikipedia contributors. Instruction set architecture — Wikipedia, the free encyclopedia, 2019.

[20] Wikipedia contributors. Verilator — Wikipedia, the free encyclopedia, 2019.

Συντομογραφίες - Αρκτικόλεξα - - Ακρωνύμια

ASIC	Application Specific Integrated Circuit
BBL	Berkeley Boot Loader
FF	Flip-Flop
FIRRTL	Flexible Intermediate Representation for RTL
FPGA	Field Programmable Gate Array
HLS	High Level Synthesis
HTIF	Host-Target Interface
ISA	Instruction Set Architecture
LUT	Lookup Table
MMU	Memory Management Unit
PL	Programmable Logic
PMA	Physical Memory Attributes
PMP	Physical Memory Protection
PPN	Physical Page Number
PS	Processing System
PTW	Page Table Walk
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
SoC	System-on-Chip
TLB	Translation Lookaside Buffer
VPN	Virtual Page Number
riscv-fesrv	RISC-V Front-End Server