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

Implementation and Evaluation of Direct Segments in the Rocket Chip Generator

Διπλωματικη Εργασια

του

ΑΝΔΡΕΑ ΞΥΔΑ

Επιβλέπων: Διονύσιος Πνευματικάτος
Καθηγητής ΕΜΠ

Εργαστηριο Υπολογιστικων Συστηματων
Αθήνα, Νοέμβριος 2023

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

Implementation and Evaluation of Direct Segments in the Rocket Chip Generator

Διπλωματικη Εργασια

του

ΑΝΔΡΕΑ ΞΥΔΑ

Επιβλέπων: Διονύσιος Πνευματικάτος
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 11 Νοεμβρίου 2023.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Διονύσιος Πνευματικάτος	Νεκτάριος Κοζύρης	Γεώργιος Γκούμας
Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ	Αναπληρωτής Καθηγητής ΕΜΠ

Αθήνα, Νοέμβριος 2023

(Υπογραφή)

.....

ΑΝΔΡΕΑΣ ΞΥΔΑΣ

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 ISA αποτελεί μία ανοιχτή Αρχιτεκτονική Συνόλου Εντολών που αναπτύχθηκε στο Πανεπιστήμιο της Καλιφόρνια, Μπέρκλεϋ, βασισμένη στην αρχιτεκτονική RISC λίγων, απλών και γενικών εντολών, σχεδιασμένων να βελτιώσουν την αποδοτικότητα και να μειώσουν την κατανάλωση ισχύς. Μία από τις κύριες υλοποιήσεις του είναι το Rocket Chip Generator, το οποίο είναι ένα ανοιχτού κώδικα System-On-Chip που χρησιμοποιεί τη γλώσσα κατασκευής υλικού Chisel, που προσφέρει υψηλό βαθμό δομικότητας και επαναχρησιμοποίησης. Αποτελεί μία βιβλιοθήκη παραμετροποιήσιμων μερών επεξεργαστών και επιτρέπει την ταχεία ανάπτυξη και προσαρμογή μικρών ενσωματωμένων επεξεργαστών, μέχρι και πολύπλοκων πολυπύρηνων συστημάτων. Η ανοιχτή φύση του RISC-V και η ανοιχτή υλοποίηση του Rocket Chip Generator, αποδείχτηκε σημαντικό στοιχείο για την ευρεία αποδοχή τους από τη βιομηχανία αλλά και για τη χρήση τους σε πολλές ερευνητικές εργασίες. Στην παρούσα εργασία ερευνούμε την υλοποίηση και αξιολόγηση των Direct Segments (DS) στον Rocket Chip Generator. Τα DS αποτελούν μία τεχνική διαχείρισης της μνήμης κατά την οποία μεγάλα συνεχόμενα κομμάτια εικονικής μνήμης χαρτογραφούνται σε μεγάλα συνεχόμενα κομμάτια φυσικής μνήμης. Η ανάπτυξη των DS χωρίστηκε σε δύο μέρη, τη ροή ανάπτυξης υλικού και τη ροή ανάπτυξης λογισμικού. Για την ανάπτυξη κυκλώματος DS χρησιμοποιήθηκαν εργαλεία προσομοίωσης υλικού και συγκεκριμένα ο Verilator, καθώς και το FireSim. Παράλληλα με το TLB που είναι μία μικρή κρυφή μνήμη που κρατάει τις μεταφράσεις από εικονικές σε φυσικές διευθύνσεις μνήμης, υλοποιούμε το DS που δίνει τη μετάφραση εικονικών διευθύνσεων, αν εμπίπτουν μέσα στο Direct Segment. Για τη διαχείριση των DS από το λειτουργικό, τροποποιούμε το MMU του Linux χρησιμοποιώντας τον μηχανισμό CMA, ώστε κάθε διεργασία να μπορεί να δεσμεύσει χώρο στο DS. Η ανάλυση της επίδοσης εξετάζεται χρησιμοποιώντας μετροπρογράμματα της σουίτας SPEC2017 με το FireSim, ο οποίος επιτρέπει την γρήγορη προσωμοίωση του Rocket Chip στο Xilinx Alveo U250 FPGA board. Η υλοποίηση μας εμφανίζει μείωση αστοχιών TLB έως 87%, με συνολική επιτάχυνση έως 28% όσον αφορά τον συνολικό χρόνο εκτέλεσης των μετροπρογραμμάτων.

Λέξεις Κλειδιά

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

Abstract

The RISC-V ISA is an open Instruction Set Architecture developed at the University of California, Berkeley, based on the RISC architecture of few, simple, and general instructions, designed to improve efficiency and reduce power consumption. One of the main implementations is the Rocket Chip Generator, which is an open-source System-On-Chip that uses the Chisel hardware construction language, offering a high degree of modularity and reusability. It is a library of customizable processor components and allows for the rapid development and customization of small embedded processors, up to complex multicore systems. The open nature of RISC-V and the open implementation of the Rocket Chip Generator proved to be a significant factor in their wide acceptance by the industry and their use in many research works. In this work, we investigate the implementation and evaluation of Direct Segments (DS) in the Rocket Chip Generator. DS are a memory management technique where large contiguous pieces of virtual memory are mapped to large contiguous pieces of physical memory. The development of Direct Segments was divided into two parts, the hardware development flow and the software development flow. For the Direct Segment hardware (HW), we used HW simulation tools, specifically Verilator, as well as Firesim for rapid testing. Along with the TLB, which is a small cache that keeps translations from virtual to physical memory addresses, we implement the DS HW that provides the translation of virtual memory to physical if it falls within the Direct Segment range. For the management of DS by the operating system, we modify the MMU of the Linux Kernel using the CMA mechanism, which allows any process to reserve memory in the DS. We examine he performance analysis using benchmarks from the SPEC2017 suite with Firesim, which allows for rapid simulation in the Xilinx Alveo U250 FPGA board. Our implementation shows TLB miss reduction up to 87%, with 28% reduction in total execution time.

Keywords

RISC-V, Hardware Design, Rocket Chip Generator, Chisel, FPGA, Verilator, TLB, Direct Segments, Firesim, Memory Management

στην οικογένεια μου

Ευχαριστίες

Θα ήθελα αρχικά να ευχαριστήσω την οικογένεια μου που στάθηκε δίπλα μου όλα αυτά τα χρόνια και με στήριζε σε κάθε μου απόφαση. Επίσης θα ήθελα να ευχαριστήσω τους φίλους μου με τους οποίους δημιουργήσαμε αναμνήσεις και ζήσαμε αξέχαστες εμπειρίες όντας φοιτητές στην Αθήνα. Τέλος, θα ήθελα να ευχαριστήσω τον καθηγητή κ.Διονύσιο Πνευματικάτο για την επίβλεψη αυτής της διπλωματικής εργασίας και τον ερευνητή Νικόλα Παπαδόπουλο που με καθοδήγησε και με συμβούλευε καθ΄ όλη τη διάρκεια εκπόνησής της.

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

Κεφάλαιο 1

Εισαγωγή

Η Αρχιτεκτονική Υπολογιστών Μειωμένου Συνόλου Εντολών (RISC) αποτελεί ένα θεμελιώδες σχεδιαστικό πρότυπο που αναδύθηκε τη δεκαετία του 1980, υποστηρίζοντας ένα απλοποιημένο σύνολο εντολών που σχεδιάστηκαν για να εκτελούνται με μεγαλύτερη ταχύτητα. Συγκεκριμένα η φιλοσοφία RISC[16] προτάθηκε από ερευνητές του πανεπιστημίου UC Berkeley με το ερευνητικό έργο Berkeley RISC και το Stanford University με το Stanford MIPS, οι οποίοι απέδειξαν ότι ένα πιο απλό σύνολο εντολών θα μπορούσε να προσφέρει ανώτερη απόδοση επιτρέποντας την ταχύτερη εκτέλεση εντολών και τη διαδρομή τους μέσω pipeline.

Ως επέκταση του RISC-I που προτάθηκε το 1980, ξεκίνησε το 2010 η ανάπτυξη της 5ης γενιάς του RISC-based ερευνητικού έργου το οποίο και ονομάστηκε RISC-V. Το RISC-V αποτελεί ένα ανοιχτό πρότυπο αρχιτεκτονικής συνόλου εντολών (ISA), το οποίο σχεδιάστηκε για να είναι κλιμακωσιμό για μια ευρεία γκάμα χρήσεων. Από το 2015, το RISC-V ISA διατηρείται από το RISC-V Foundation το οποίο ιδρύθηκε για την προώθηση του ανοιχτού συνόλου εντολών και τη συλλογική συμβολή στην ανάπτυξή του. Παράλληλα με το ISA αναπτύχθηκε το Rocket Chip Generator για την απόδειξη της ορθής λειτουργίας του, και εξελίχθηκε τα τελευταία χρόνια σε μία αρκετά ευέλικτη γεννήτρια συστημάτων RISC-V System-on-Chip (SoC).

Σημαντικό πυλώνα της αρχιτεκτονικής υπολογιστών αποτελεί επίσης η έννοια της εικονικής μνήμης που χρονολογείται από τις δεκαετίες του 1950, με τον υπολογιστή Atlas να είναι ένας από τους πρώτους που χρησιμοποίησε μια μορφή εικονικής μνήμης για πιο αποδοτική χρήση της κύριας μνήμης του. Με την εξέλιξη αυτής, αναπτύχθηκε επίσης η μονάδα διαχείρισης μνήμης στους μικροεπεξεργαστές η οποία ήταν καθοριστική για την παροχή ανθεκτικών, ασφαλών και αποδοτικών δυνατοτήτων διαχείρισης μνήμης. Από τότε, προτάθηκαν αρκετές τεχνικές διαχείρισης της εικονικής μνήμης με τις κυριότερες να είναι η Σελιδοποίηση (Paging) και η Τμηματοποίηση (Segmentation).

1.1 Αντικείμενο της διπλωματικής

Στην εργασία αυτή θα ασχοληθούμε με την υλοποίηση και αξιολόγηση των Direct Segments στον Rocket Chip Generator. Τα DS αποτελούν μία τεχνική διαχείρισης της μνήμης κατά την οποία μεγάλα συνεχόμενα κομμάτια εικονικής μνήμης χαρτογραφούνται σε μεγάλα συνεχόμενα κομμάτια φυσικής μνήμης. Τα DS απαιτούν υποστήριξη από το λειτουργικό και από το υλικό, για αυτό και η ανάπτυξή τους στην παρούσα εργασία θα χωριστεί σε δύο μέρη, τη ροή ανάπτυξης λογισμικού και ροή ανάπτυξη υλικού. Η ανάπτυξη του υλικού θα επικεντρωθεί κυρίως στο TLB όπου και θα υλοποιηθεί η λογική τους. Παράλληλα με το TLB που είναι μία μικρή κρυφή μνήμη που κρατάει τις μεταφράσεις από εικονικές σε φυσικές διευθύνσεις, υλοποιούμε το DS hardware που δίνει τη μετάφραση εικονικών διευθύνσεων, αν εμπίπτουν μέσα στο Direct Segment range. Το επιπλέον υλικό που προυποθέτει το DS είναι 3 καταχωρητές που αποθηκεύουν την αρχική διεύθυνση του συνεχόμενου χώρου εικονικής μνήμης, την τελική του διέυθυνση, και τη διαφορά της αρχικής διεύθυνσης από αυτή του συνεχόμενου χώρου εικονικής μνήμης. Επιτυχής μετάφραση προκύπτει αν το ζητούμενο vpn εμπίπτει μέσα στο εύρος των 2 πρώτων καταχωρητών, και η μετάφραση δίνεται προσθέτοντας σε αυτό τον τρίτο καταχωρητή. Σε επίπεδο λογισμικού χρειάζεται μία μέθοδος αναγνώρισης των διεργασιών που επιθυμούν να χρησιμοποιήσουν τα Direct Segments και ένας τρόπος διαχείρισης τους στην εικονική μνήμη. Με το που αναγνωριστεί ένα Direct Segment μέσω του CMA δεσμεύεται ο συνεχόμενος χώρος εικονικής μνήμης και δημιουργείται ένα ενιαίο virtual memory area που ονομάζεται Primary Region. Για την δέσμευση μνήμης στο Primary Region χρειάζεται η υλοποίηση ενός custom allocator. Η συγκεκριμένη σχεδίαση προσφέρει βελτιωμένη απόδοση σε συστήματα που χαρακτηρίζονται από μεγάλο memory footprint αφού μπορεί να δώσει μετάφραση σε ένα κύκλο για μεγάλο μέρος εικονικής μνήμης μειώνοντας τις αστοχές TLB. Τέλος, εξετάζουμε την σχεδίαση του προσαρμοσμένου λειτουργικού και υλικού με υποστήριξη Direct Segments έναντι του αρχικού, σε σχέση με την επίδοση τους χρησιμοποιώντας μετροπρογράμματα της σουίτας SPEC2017.

1.2 Οργάνωση του τόμου

Η παρούσα εργασία είναι οργανωμένη στα παρακάτω κεφάλαια.

- Στο κεφάλαιο 2 παρουσιάζουμε το απαραίτητο θεωρητικό υπόβαθρο, δίνοντας περιγραφή των τεχνολογιών που θα χρησιμοποιήθηκαν για την εκπόνηση της παρούσας εργασίας. Οι κυριότερες τεχνολογίες αποτελούν η ανοιχτού συνόλου αρχιτεκτονική RISC-V, η γλώσσα περιγραφής υλικού Chisel, η γεννήτρια μικροεπεξεργαστών Rocket Chip Generator και ο πυρήνας RISC-V Linux Kernel. Στο τέλος του κεφαλαίου παρουσιάζουμε την πλατφόρμα προσομοίωσης υλικού Firesim με την οποία εξετάστηκε η τελική υλοποίηση.
- Στο κεφάλαιο 3 εξετάζουμε αναλυτικά τη σχεδίαση του αρχικού TLB του Rocket Chip Generator βάση της οποίας θα υλοποιήσουμε τη λογική των Direct Segments και παρουσιάζουμε την πρόταση μας σχεδίασης υλικού. Στη συνέχεια αναλύουμε την πρόταση

σχεδίασης του λειτουργικού για υποστήριξη των DS και αναφέρουμε τις διαφορές με την ήδη υπάρχουσα βιβλιογραφία.

- Στο κεφάλαιο 4 παρουσιάζουμε αναλυτικά τα βήματα που ακολουθήσαμε από τη σχεδίαση μέχρι την υλοποίηση των Direct Segments. Αναφέρουμε τις τεχνολογίες Verilator, Proxy Kernel και BBL που χρησιμοποιήθηκαν για την ανάπτυξη υλικού και τις τεχνολογίες QEMU, Open-Sbi και Linux για την ανάπτυξη λογισμικού.

- Στο κεφάλαιο 5 θα παρουσιάσουμε την επίδοση του συστήματος με υποστήριξη DS σε σχέση με το αρχικό σε ότι αφορά τις μετρικές επίδοσης χρησιμοποιώντας την σουίτα μετροπρογραμμάτων SPEC2017. Σε πρώτο στάδιο θα εξεταστεί η επίδοση στον εξομοιωτή QEMU και σε δεύτερο στάδιο στο υλικό με τον DS Rocket Core.

- Τέλος, στο κεφάλαιο 6 μελετούμε διορθώσει που θα βελτίωναν την επίδοση των Direct Segments και πιθανές μελλοντικές προεκτάσεις της εργασίας αυτής.

Κεφάλαιο 2

Θεωρητικό υπόβαθρο

Σε αυτό το κεφάλαιο θα δοθεί αναλυτική περιγραφή των τεχνολογιών που χρησιμοποιήθηκαν για την εκπόνηση της διπλωματικής εργασίας. Αρχικά θα παρουσιαστεί ο πυρήνας της διπλωματικής που είναι η Ανοιχτή Αρχιτεκτονική Συνόλου Εντολών RISC-V και η γεννήτρια μικροεπεξεργαστών Rocket Chip Generator η οποία αποτελεί υλοποίηση της αρχιτεκτονικής RISC-V με χρήση της γλώσσας περιγραφής υλικού Chisel. Στη συνέχεια θα ακολουθήσει περιγραφή του RISC-V Linux Kernel σε συνδυασμό με τον Contiguous Memory Allocator. Τέλος, θα αναλυθεί η τεχνολογία του FireSim μαζί με το FPGA.

2.1 RISC-V Instruction Set Architecture

2.1.1 Ανοιχτή Αρχιτεκτονική Συνόλου Εντολών RISC-V

Η Αρχιτεκτονική Συνόλου Εντολών (ISA) αποτελεί το μοντέλο ενός υπολογιστικού συστήματος το οποίο ορίζει τη διεπαφή μεταξύ λογισμικού και υλικού. Περιλαμβάνει κατά κύριο λόγο τους καταχωρητές, την αρχιτεκτονική μνήμης, το χειρισμό διακοπών και εξαιρέσεων και το σύνολο των μνημονικών εντολών. Μπορεί να θεωρηθεί ως ο οδηγός ενός προγραμματιστή, καθώς ορίζει τον τρόπο με τον οποίο μπορεί να αλληλεπιδράσει με το υλικό σε επίπεδο μηχανής [15].

Το RISC-V ISA είναι βασισμένο, όπως προκύπτει και από το όνομα, στην αρχιτεκτονική RISC, λίγων, απλών και γενικών εντολών, σχεδιασμένων να βελτιώσουν την αποδοτικότητα και να μειώσουν την κατανάλωση ισχύς. Αναπτύχθηκε από το Πανεπιστήμιο της Καλιφόρνιας, Μπέρκλεϋ το 2010 και η ανάπτυξη και διαχείρισή του εκτελείται από το **RISC-V Foundation**[1] το οποίο παρέχει την αρχιτεκτονική υπό μία ελεύθερη, ανοιχτή άδεια χρήσης για όσους επιθυμούν να τη χρησιμοποιήσουν.

Το RISC-V Instruction Set Manual χωρίζεται σε 2 Specifications, το **Volume I: Unprivileged ISA** που επικεντρώνεται στην περιγραφή της αρχιτεκτονικής και των βασικών αρχών του συνόλου εντολών που μπορεί να περιέχει ένα πρόγραμμα που τρέχει σε χώρο χρήστη και το **Volume II: Privileged Architecture** που επικεντρώνεται στις προνομια-

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

κές λειτουργίες και τις επιπρόσθετες επεκτάσεις του συνόλου εντολών σε επίπεδο Machine και Supervisor.

2.1.2 RISC-V User-Level ISA

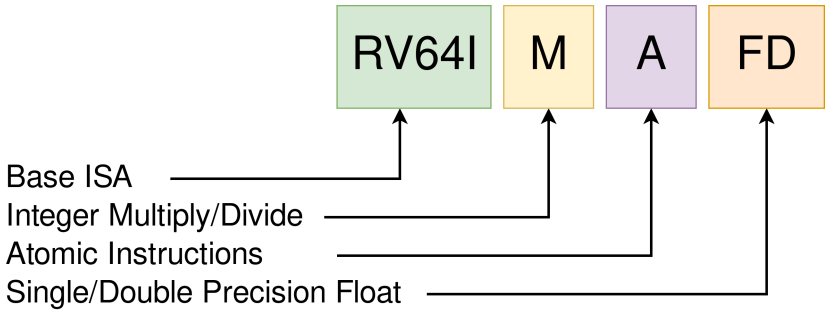
Το User-Level ISA λειτουργεί ως το θεμελιώδες σύνολο εντολών στην αρχιτεκτονική RISC-V το οποίο είναι απαραίτητο για ένα πλήρως λειτουργικό σύστημα. Έχει σχεδιαστεί ώστε να είναι απλό και αποτελεσματικό, καθιστώντας το ιδιαίτερα κατάλληλο για μια ευρεία γκάμα εφαρμογών, από μικρά ενσωματωμένα συστήματα έως μεγάλα κέντρα δεδομένων. Αποτελείται από τα RV32/64/128 Integer Base ISA με εικονικές διευθύνσεις 32, 64 και 128 bits αντίστοιχα. Το βασικό ISA και υποχρεωτικό για όλους τους επεξεργαστές RISC-V είναι το **Integer 'I'**. Παράλληλα, ορίζονται επιπλέον επίσημα Extensions, τα οποία υλοποιούνται και ελέγχονται από το RISC-V Foundation, με σκοπό να προσθέτουν εξειδικευμένες εντολές ή επιπλέον λειτουργικότητα στο σύστημα για συνολική βελτιωμένη επίδοση. Αξίζει να σημειωθεί επίσης, όντας ανοιχτή αρχιτεκτονική, υπάρχει η δυνατότητα υλοποίησης custom Extension στο official ISA για όποιον επιθυμεί να βελτιώσει την επίδοση συγκεκριμένης εφαρμογής. Στον Πίνακα 2.1 φαίνεται το Base ISA, τα Standard και ορισμένα από τα General Extensions όπως προκύπτουν από το Volume I Specification [4]:

Extension	**Description**
	Base ISA
I	Integer
E	Reduced Integer
	Standard Unprivileged Extensions
M	Integer Multiplication and Division
A	Atomic Instructions
F	Single-Precision Floating Point Instructions
D	Double-Precision Floating Point Instructions
	General
Q	Quad-Precision Floating Point Instructions
C	16-bit Compressed Instructions
B	Bit Manipulation
V	Vector Operations

Πίνακας 2.1: Βασικότερα Extensions

Ο συμβολισμός ενός RISC-V Core γίνεται σε σχέση με τα extensions που περιέχει, όπως φαίνεται στο Σχήμα 2.1, και πρέπει να ακολουθεί την κανονική σειρά με την οποία εμφανίζονται στον πίνακα του Volume I Specification (πχ. RV64MAFD είναι επιτρεπτό ενώ RV64AMFD όχι). Στην παρούσα διπλωματική εργασία θα χρησιμοποιηθεί το RV64G ISA, όπου G σημαίνει General-Purpose. Τα Extensions που περιέχει και είναι απαραίτητα για την υποστήριξη ενός Λειτουργικού Συστήματος, για παράδειγμα το Linux, είναι τα **I, M, F, A, D** και τα **Zicsr**

που προσθέτουν Control and Status Registers (CSRs) Access και **Zifencei** για Instruction-Fetch Fence.



Σχήμα 2.1: Συμβολισμός RV64IMAFD Core

2.1.3 RISC-V Privileged Architecture

Στο Privileged Architecture ορίζεται κυρίως η διεπαφή μεταξύ της αρχιτεκτονικής συνόλου εντολών ενός RISC-V hart (hardware thread) και του περιβάλλοντος εκτέλεσης του, καθώς και διαφορετικά privilege levels. Οι σημαντικότεροι λόγοι ύπαρξης επιπρόσθετων επιπέδων εκτέλεσης είναι η διαχείρηση και προστασία κοινών πόρων όπως η μνήμη και οι συσκευές, η υποστήριξη multitasking και η απόκρυψη της υποκείμενης υλοποίησης του υλικού, για μεγαλύτερη ευχέρεια ανάπτυξης υλικού, ασφάλεια και αποτελεσματικότητα.

Privilege Levels

Το Privileged Architecture παρέχει τα πιο κάτω Privilege Levels, όπως φαίνονται στον Πίνακα 2.2 :

Επίπεδο	Όνομα	Συντομογραφία
0	User / Application	U-mode
1	Supervisor	S-mode
2	*Reserved*	
3	Machine	M-mode

Πίνακας 2.2: Privilege Levels

Κάθε επίπεδο που προσφέρεται προσθέτει επιπλέον λειτουργικότητα, εντολές και καταχωρητές κατάστασης και ελέγχου, Control and Status Registers (CSRs). Τα πιο προνομιούχα επίπεδα έχουν πρόσβαση στα χαμηλότερα επίπεδα, ενώ το αντίθετο ισχύει σε περιορισμένες περιπτώσεις και υπό προϋποθέσεις για την προώθηση των διακοπών ή την πρόσβαση στους performance counters. Η συγκεκριμένη αρχιτεκτονική, με το συνδυασμό των Privilege Levels, σχεδιάστηκε ώστε να υποστηρίζει τις ανάγκες όλων των εφαρμογών που ποικίλουν σε πολυπλοκότητα και λειτουργικότητα. Οι συνδυασμοί συνοψίζονται στον Πίνακα 2.3:

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: Πιθανοί συνδυασμοί ενός RISC-V συστήματος

Machine-Mode

Το **M-mode** είναι το υψηλότερο επίπεδο της αρχιτεκτονικής RISC-V, παρέχοντας πρόσβαση σε όλους τους πόρους του συστήματος. Προσθέτει βασικούς καταχωρητές για τον έλεγχο, καθώς και επιπλέον καταχωρητές μέτρησης επίδοσης. Παρέχει επίσης μηχανισμούς για τη διαχείρηση της φυσικής μνήμης, σφαλμάτων και γεγονότων και αλληλεπιδρά άμεσα με συσκευές εισόδου/εξόδου. Καλείται σε διάφορα σενάρια όπως:

- **System Initialization**: Η αρχικοποίηση του συστήματος γίνεται σε M-mode, ρυθμίζει το περιβάλλον εκτέλεσης και ανάλογα του συστήματος μεταβαίνει σε χαμηλότερο επίπεδο.
- **Access Faults**: Σφάλματα σελίδας ή ελλιπών δικαιωμάτων πρόσβασης στη μνήμνη.
- **Illegal Instructions**: Εντολές που προσπαθούν να εκτελεστούν με λιγότερα δικαιώματα, ανύπαρκτες εντολές ή εντολές που δεν είναι υλοποιημένες στη μικροαρχιτεκτονική.
- **Breakpoints**: Χρησιμεύουν στην ευκολότερη αποσφαλμάτωση.
- **Environment Calls**: Κλήσεις συστήματος για διαχείριση ενεργειών που απαιτούν περισσότερα δικαιώματα από ανώτερη βαθμίδα.
- **Misaligned address accesses**: Σφάλματα στοίχισης διευθύνσεων, στην αρχιτεκτονική RISC-V δεν επιτρέπονται misaligned διευθύνσεις για την απλοποίηση της μικροαρχιτεκτονικής.

Supervisor-mode

Το **S-mode** έχει σχεδιαστεί κυρίως για την υποστήριξη ενός λειτουργικού συστήματος. Βρίσκεται ενδιάμεσα στο M-mode και U-mode και παρέχει ένα ελεγχόμενο περιβάλλον για τη διαχείριση της εικονικής μνήμης και των κλήσεων συστήματος, διασφαλίζοντας την προστασία της μνήμης και το διαχωρισμό μεταξύ διαφορετικών διεργασιών.

User-mode

Το **U-mode** επιτρέπει την εκτέλεση κώδικα στο επίπεδο του χώρου χρήστη και δεν έχει άμεση πρόσβαση στους πόρους του συστήματος, παρά μόνο με environmental calls όπου και μεταβάινει σε υψηλότερα επίπεδα. Με αυτό τον τρόπο, παρέχεται ένα επίπεδο προστασίας,

καθώς τυχόν κακόβουλος κώδικας απομονώνεται από το υπόλοιπο σύστημα. Το U-mode προσθέτει επίσης το μηχανισμό προστασίας της φυσικής μνήμης, Physical Memory Protection (PMP). Το PMP χωρίζει τη μνήμη του συστήματος σε περιοχές όπου ο κώδικας του χώρου χρήστη έχει συγκεκριμένα δικαιώματα πρόσβασης [11]. Πέρα από το PMP, ορίζεται και ο μηχανισμός Physical Memory Attributes (PMA) ο οποίος οριοθετεί περιοχές μνήμης και ορίζει χαρακτηριστικά τους.

2.2 Rocket Chip Generator

Ο Rocket Chip [1] είναι ένα ανοιχτού κώδικα System-On-Chip (SoC) που αναπτύχθηκε στο UC Berkeley και αποτελεί κύριο στοιχείο του RISC-V ecosystem. Χρησιμοποιεί τη γλώσσα κατασκευής υλικού Chisel, η οποία προσφέρει υψηλό βαθμό δομικότητας και επαναχρησιμοποίησης, για τη σύνθεση πολύπλοκων και παραμετροποιήσιμων γεννητριών για επεξεργαστικούς πυρήνες, κρυφές μνήμες και διασυνδέσεις σε ένα ενσωματωμένο SoC. Ο Rocket Chip επιτρέπει την ταχεία ανάπτυξη και προσαρμογή γενικού σκοπού επεξεργαστών που χρησιμοποιούν το ανοιχτό RISC-V ISA και παρέχει τόσο μία γεννήτρια πυρήνα in-order (Rocket) όσο και μία γεννήτρια πυρήνα out-of-order (BOOM). Υποστηρίζει επίσης την ενσωμάτωση προσαρμοσμένων επιταχυντών με διάφορες μορφές.

Πάρακατω παρουσιάζεται μία σύντομη περιγραφή των πακέτων που μπορούν να βρεθούν στον Rocket Chip Generator και είτε περιέχουν εργαλεία για την παραμετροποίηση των γεννητριών, είτε περιέχουν την Chisel RTL υλοποίηση:

- **Core**: Αποτελεί τον βαθμωτό in-order Rocket core και τον υπερβαθμωτό BOOM out-of-order core generator οι οποίοι υποστηρίζουν προαιρετικό Floating Point Unit (FPU), παραμετροποιήσιμους branch predictors και παραμετροποιήσιμες λειτουργικές μονάδες (functional units).

- **Caches**: Μία συλλογή Generators κρυφών μνημών εντολών και δεδομένων με ρυθμιζόμενο μέγεθος, associativity και πολίτικη αντικατάστασης, καθώς και ένα fully-associative TLB με ρυθμιζόμενο μέγεθος.

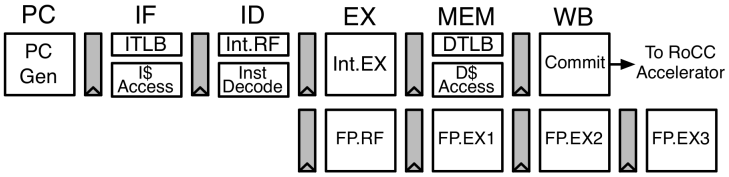
- **RoCC**: Το Rocket Custom Coprocessor interface, ένα template για τη σχεδίαση προσαρμοσμένων μονάδων επιτάχυνσης στον πυρήνα Rocket.

- **Tile**: Ένα Tile-Generator template για το συνδυασμό ενός πυρήνα, των σχετικών του κρυφών μνημών και πιθανώς άλλων στοιχείων όπως οι επιταχυντές RoCC.

- **TileLink**: Ένα πρωτόκολλο διαύλου που διευκολύνει την επικοινωνία μεταξύ διαφορετικών στοιχείων του SoC. Επίσης δίνει τη δυνατότητα ρύθμισης του αριθμού των tiles, των cache-coherency protocols μεταξύ των tiles, την ύπαρξη διαμοιραζόμενου χώρου αποθήκευσης καθώς και την φυσική υλοποίηση των υποκείμενων δικτύων.

Rocket Core

Ο Ροςκετ [10] είναι μια γεννήτρια πυρήνων 5-σταδίων, in-order και scalar, που υλοποιούν τα σύνολα εντολών RV32G και RV64G. Διαθέτει μια Μονάδα Διαχείρισης Μνήμης (MMU) που υποστηρίζει σελιδοποίηση της εικονικής μνήμης, μια non-blocking κρυφή μνήμη δεδομένων και ένα front-end με πρόβλεψη διακλαδώσεων. Υποστηρίζει επίσης τα privilege levels RISC-V machine, supervisor και user. Επιπρόσθετα ο Rocket παρέχει τη δυνατότητα παραμετροποίησης, συμπεριλαμβανομένης της προαιρετικής υποστήριξης ορισμένων επεκτάσεων του ISA (M, A, F, D), του αριθμού των σταδίων της σωλήνωσης κινητής υποδιαστολής, καθώς και των μεγεθών της κρυφής μνήμης και του TLB. Τέλος, μπορεί επίσης να θεωρηθεί ως μια βιβλιοθήκη ξεχωριστών εξαρτημάτων επεξεργαστών, καθώς αρκετές μονάδες όπως οι κρυφές μνήμες, τα TLBs και ο Page Table Walker χρησιμοποιούνται από άλλες υλοποιήσεις όπως ο BOOM. Αυτό συμβάινει διότι ένας Rocket Core σε συνδυασμό με τις L1 κρυφές μνήμες του, δημιουργεί ένα Rocket tile το οποίο είναι επαναχρησιμοποιήσιμο στο SoC.



Σχήμα 2.2: Στάδια διοχέτευσης στον Rocket Core

2.3 Μονάδα Διαχείρισης Εικονικής Μνήμης

Η εικονική μνήμη αποτελεί μία τεχνική διαχείρισης μνήμης που εισήχθη στους σύγχρονους υπολογιστές για τον ασφαλή διαμοιρασμό της φυσικής μνήμης και ασφαλή εκτέλεση ταυτόχρονων διεργασιών. Η εικονική μνήμη χρησιμοποιείται από το λειτουργικό σύστημα για να παρέχει σε κάθε εφαρμογή το δικό της συνεχή χώρο εικονικών διευθύνσεων που χρησιμοποιεί για να προσπελάσει τη μνήμη.

2.3.1 Βασικές Έννοιες

- **Χώρος Διευθύνσεων**: Με την εικονική μνήμη εισάγεται η διάκριση μεταξύ εικονικών και φυσικών διευθύνσεων. Ο εικονικός χώρος διευθύνσεων είναι μοναδικός σε κάθε διεργασία και έχει μέγεθος που εξαρτάται από το μήκος διευθύνσεων σε bit. Ο φυσικός χώρος διευθύνσεων αναφέρεται στη μνήμη PAM και αντιστοιχίζεται σε εικονικές διευθύνσεις με διάφορους μηχανισμούς.

- **Σελιδοποίηση (Paging)**: Η σελιδοποίηση είναι η τεχνική κατά την οποία ο χώρος διευθύνσεων χωρίζεται σε τμήματα συνεχών διευθύνσεων τα οποία ονομάζονται εικονικές σελίδες (virtual pages) και φυσικές σελίδες (physical pages). Στην αρχιτεκτονική RISC-V το ελάχιστο μέγεθος που υποστηρίζεται είναι 4KB.

- **Πίνακας Σελιδών (Page Table)**: Είναι μια δομή δεδομένων που χρησιμοποιείται από ένα σύστημα εικονικής μνήμης για να αποθηκεύσει την αντιστοίχιση μεταξύ εικονικών διευθύνσεων και φυσικών διευθύνσεων.

- **Swapping**: Το swapping είναι η διαδικασία στην οποία το λειτουργικό σύστημα μετακινεί σελίδες από τη RAM, αφού έχει γεμίσει, στο σκληρό δίσκο ή το αντίστροφο όταν ζητηθεί κάποιο τμήμα κώδικα που δε βρίσκεται στη RAM.

- **Σφάλμα Σελίδας (Page Fault)**: Εάν ένα πρόγραμμα προσπαθεί να προσπελάσει μια σελίδα που δεν είναι παρούσα στη RAM, προκαλείται ένα σφάλμα σελίδας. Το λειτουργικό σύστημα μεταφέρει την απαιτούμενη σελίδα από το δίσκο και κάνει σωαπ με μια άλλη σελίδα.

2.3.2 Η Εικονική Μνήμη στο RISC-V ISA

Το σύστημα εικονικής μνήμης βασισμένο στη σελιδοποίηση παρέχεται από το S-mode της RISC-V αρχιτεκτονικής. Σύμφωνα με αυτό, η μνήμη του συστήματος κατακερματίζεται σε σελίδες σταθερού μεγέθους οι οποίες πρέπει να μεταφραστούν από εικονικές σε φυσικές. Για τη μετάφραση αυτών των σελιδών, υπάρχει μία αντιστοίχιση από εικονικές σε φυσικές διευθύνσεις οι οποίες αποθηκεύονται σε μία δενδρική δομή που ονομάζεται Page Table. Για να βρούμε την ακριβή αντιστοίχιση θα πρέπει το σύστημα να φτάσει στους τελευταίους κόμβους του Page Table, οι οποίοι φανερώνουν αν η σελίδα έχει χαρτογραφηθεί σε μια φυσική διεύθυνση και τα δικαιώματα πρόσβασης σε αυτή. Σε περίπτωση που η σελίδα δεν έχει χαρτογραφηθεί (mapped) ή δεν υπάρχουν τα σωστά δικαιώματα οδηγούμαστε σε σφαλμα σελίδας (Page Fault) το οποίο διαχειρίζεται το λειτουργικό.

Το RISC-V ISA ορίζει διάφορα σχήματα εικονικής μνήμης αναλόγως του εικονικού χώρου διευθύνσεων.

- **Sv32**: 4GB virtual address space (2-level Page Table)

- **Sv39**: 512GB virtual address space (3-level Page Table)

- **Sv48**: 256TB virtual address space (4-level Page Table)

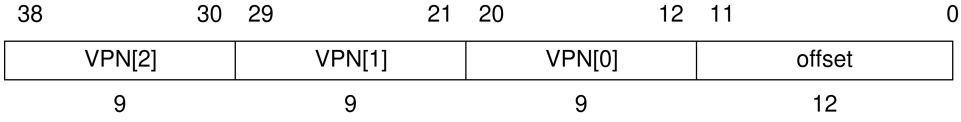
- **Sv57**: 128PB virtual address space (5-level Page Table)

Στην παρούσα εργασία θα ασχοληθούμε με το Sv39 το οποίο είναι και αυτό που χρησιμοποιεί ο πυρήνας RISC-V Linux.

2.3.3 Διευθυνσιοδότηση και Μετάφραση

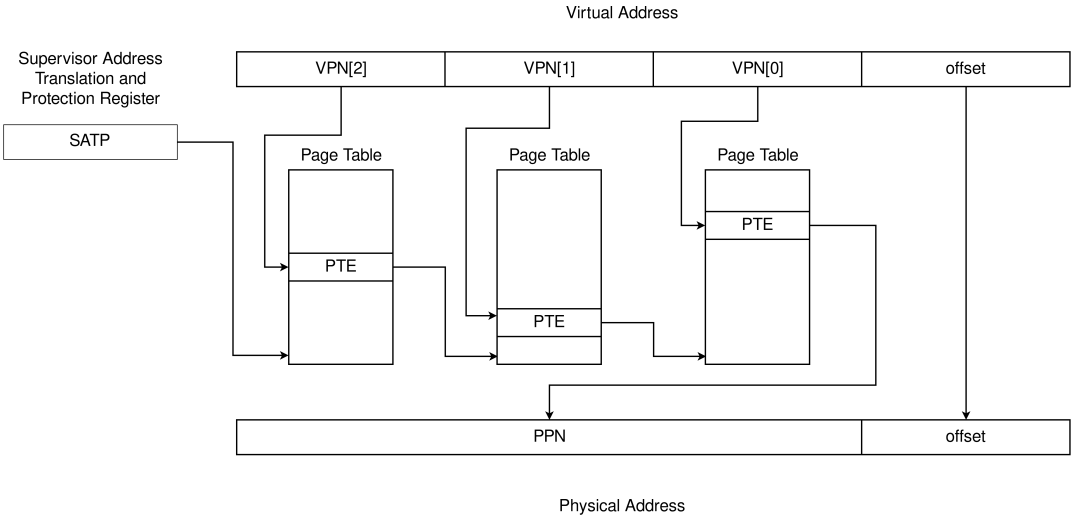
Το σχήμα εικονικής μνήμης που χρησιμοποιεί το κάθε RISC-V σύστημα καταγράφεται στο MODE του καταχωρητή satp (Supervisor Address Translation and Protection), το οποίο σε RV64 Base ISA ορίζεται από τα 4 Most Significant Bits του satp. Στο σχήμα Sv39 με σελίδες μεγέθους 4KB, το οποίο και θα μελετήσουμε, η εικονική διεύθυνση αποτελείται από

39 bits όπως φαίνεται στο Σχήμα 2.3. Τα πρώτα 12 bits ορίζουν το page offset, δηλαδή την ακριβή διεύθυνση μέσα στη σελίδα. Τα επόμενα 27 bits ορίζουν το Virtual Page Number (VPN) το οποίο μεταφράζεται σε Physical Page Number (PPN) και τα υπόλοιπα 25 bits, καθώς δε χρησιμοποιούνται στη διευθυνσιοδότηση, θα πρέπει να είναι ίδια με το MSB της εικονικής διεύθυνσης.



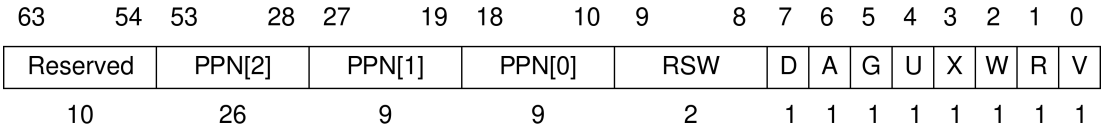
Σχήμα 2.3: Sv39 Εικονική διεύθυνση

Εφόσον στο σχήμα Sv39 έχουμε 3-level Page Table, το VPN της εικονικής διεύθυνσης χωρίζεται σε 3 ξεχωριστά τμήματα των 9 bits. Κάθε τμήμα αναφέρεται στο Page Table Entry (PTE) κάθε επιπέδου του Page Table. Αν το PTE σε υψηλότερα επίπεδα δεν είναι leaf PTE, δηλαδή δεν περιέχει το PPN της φυσικής διεύθυνσης, τότε κάνει point στο χαμηλότερο επίπεδο του Page Table μέχρι να βρεθεί το leaf PTE που αντιστοιχεί στην εικονική διεύθυνση. Η διαδικασία της αντιστοίχισης ενός VPN σε ένα PPN μέσω του 3-level Page Table γίνεται εμφανής με το Σχήμα 2.4:



Σχήμα 2.4: Η οργάνωση του Page Table στο σχήμα sv39 (RV64)

Τα PTEs που βρίσκονται στον Πίνακα Σελιδών έχουν με τη σειρά τους τη δική τους κωδικοποίηση όπως φαίνεται στο Σχήμα 2.5. Το bit 63 είναι reserved για το Svnapot extension όπως και τα bits 62-61 για το Svpbmt extension. Τα bits 60-54 είναι reserved για μελλοντική χρήση και τα bits 53-10 αποτελούν το PPN. Τα υπόλοιπα 10 bits ορίζουν διάφορα χαρακτηριστικά του PTE ¨οπως φαίνεται στον Πίνακα 2.4.



Σχήμα 2.5: Sv39 Page Table Entry

Bit	Description
RSW	Reserved bits, για μελλοντική χρήση από το S-mode
D	Έχει γίνει κάποια εγγραφή στη σελίδα
A	Έχει προσπελαστεί η σελίδα
G	Global Mapping
U	Η σελίδα ανήκει στο U-mode
X	Η σελίδα είναι executable
W	Η σελίδα είναι writable
R	Η σελίδα είναι readable
V	Η σελίδα είναι έγκυρη

Πίνακας 2.4: Τα 10 LSBs ενός PTE

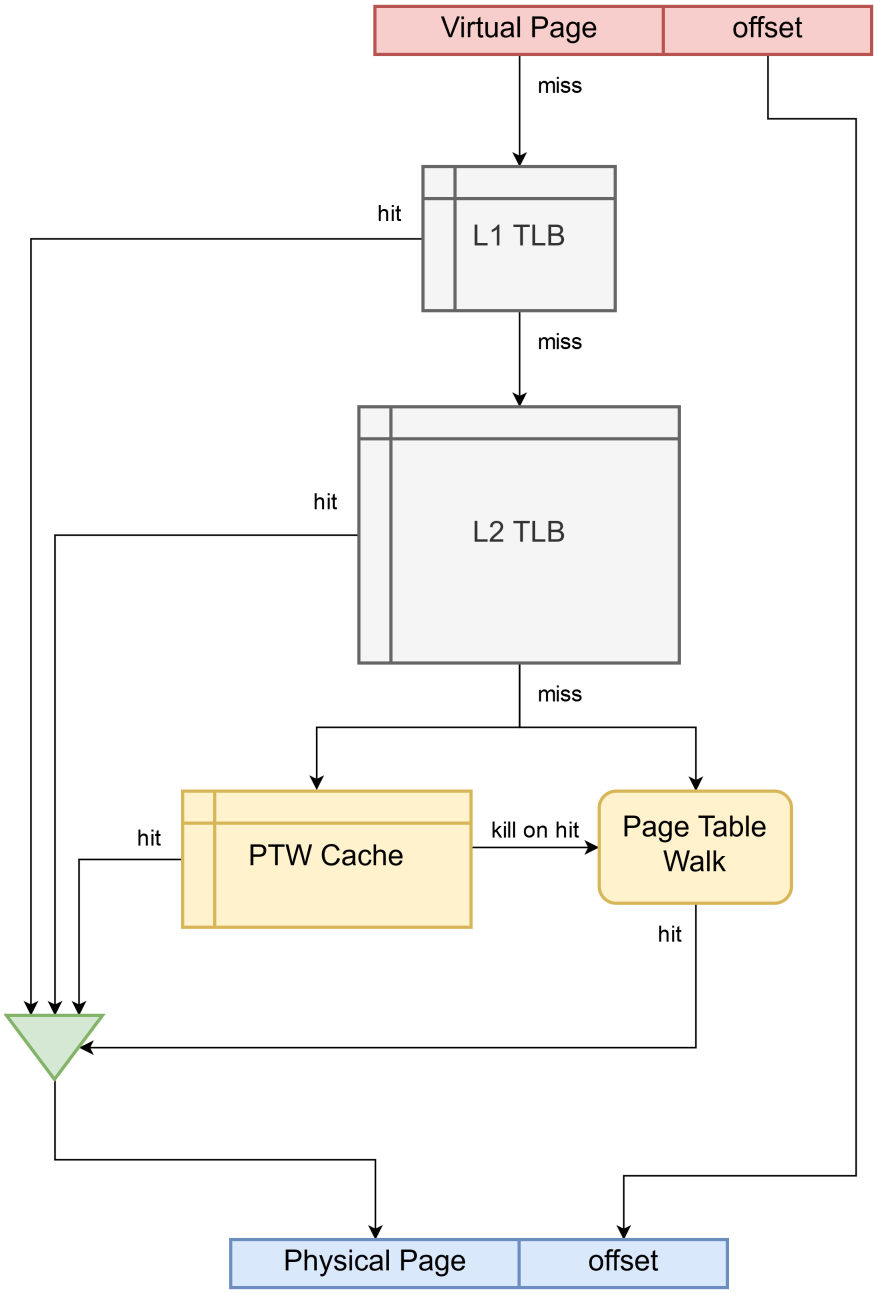
2.3.4 Η Μονάδα Διαχείρισης Μνήμης του Rocket Chip Generator

Το σύστημα διαχείρισης εικονικής μνήμης του Rocket Chip Generator έχει υλοποιημένη μονάδα σε επίπεδο υλικού η οποία διασχίζει το Page Table για την εύρεση των μεταφράσεων από τον εικονικό χώρο διευθύνσεων στο φυσικό. Αυτή η μονάδα ονομάζεται Page Table Walker (PTW) και για την καλύτερη αποδοτικότητα του συστήματος περιλαμβάνει μία κρυφή μνήμη Page Table Walk Cache (PTW Cache) η οποία κρατάει τις ενδιάμεσες μεταφράσεις (non leaf). Επίσης η μονάδα διαχείρισης μνήμης περιλαμβάνει TLB 2 επιπέδων. Στο 1ο επίπεδο υπάρχει ένα Instruction TLB για το στάδιο instruction fetch και ένα Data TLB για το στάδιο data memory access τα οποία είναι μικρά με κόστος hit 1 κύκλο. Σο 2ο επίπεδο υπάρχει ένα κοινό TLB για τα 2 στάδια το οποίο είναι μεγαλύτερο και έχει κόστος hit 2 κύκλων. Η διαδικασία μετάφρασης των εικονικών διευθύνσεων του Rocket Chip περιγράφεται με το Σχήμα 2.6 .

2.4 RISC-V Linux Kernel with CMA Configuration

2.4.1 Fedora Linux

Το Linux αποτελεί λειτουργικό σύστημα τύπου Unix το οποίο βασίζεται στον πυρήνα Linux και είναι open-source. Συνήθως έρχεται packaged ως διανομή η οποία περιλαμβάνει τον πυρήνα, το συνοδευτικό λογισμικό συστήματος και διάφορες βιβλιοθήκες. Ως ένα από τα πρώτα και πιο επιτυχημένα παραδείγματα συνεργασίας ελεύθερου και ανοιχτού κώδικα λογισμικού, χρησιμοποιείται για εφαρμογές από ενσωματομένα συστήματα, μέχρι και ολόκληρους server και υπερυπολογιστές.



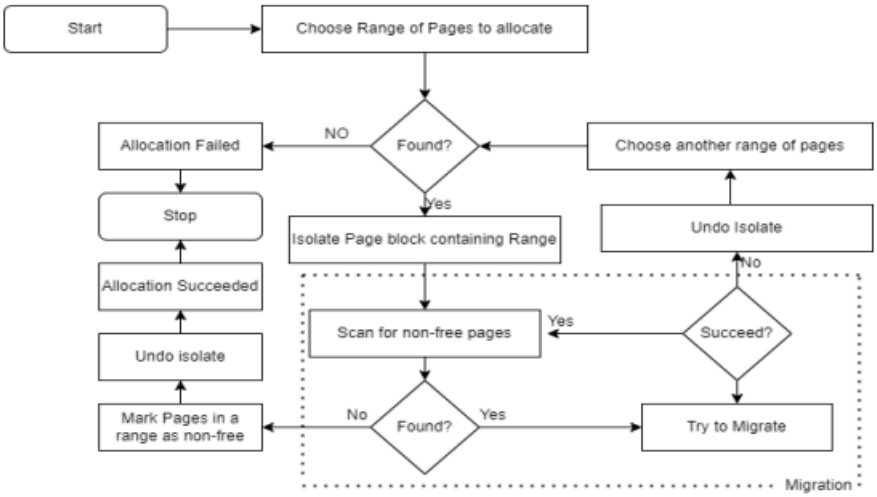
Σχήμα 2.6: Σχήμα μετάφρασης εικονικών διευθύνσεων στον Rocket Chip Generator

Στην παρούσα εργασία χρησιμοποιήσαμε μία από τις πιο γνωστές διανομές, το Fedora Linux distribution. Το Fedora ξεκίνησε το 2003 ως συνέχεια του Red Hat Linux project και από τότε έχει κυκλοφορήσει εκδόσεις για διαφορετικές χρήσεις όπως το Fedora Workstation, Fedora IOT, Fedora Server κτλ. Η ενεργή κοινότητα που συμβάλει στην ανάπτυξή του και το γεγονός ότι σε κάθε έκδοσή του περιλαμβάνει τις τελευταίες εκδόσεις πυρήνα και λογισμικού, καθιστά τη διανομή Fedora ιδανική για χρήση με το RISC-V ISA. Επίσης, το Fedora υποστηρίζει επίσημα την αρχιτεκτονική RISC-V, εξασφαλίζοντας ότι η διανομή είναι βελτιστοποιημένη για την απόδοση και τη συμβατότητα με το υλικό RISC-V.

2.4.2 CMA (Contiguous Memory Allocator)

Το Contiguous Memory Allocator [12] είναι ένας μηχανισμός που σχεδιάστηκε για τη δυναμική κατανομή τμημάτων μνήμης συνεχόμενων φυσικών διευθύνσεων. Η απαιτούμενη μνήμη για χρήση του CMA μπορεί να γίνει reserved κατά το ξεκίνημα του μηχανήματος από το kernel commandline ή από το device tree ή μέσω του linux configuration. Μόλις η μνήμη γίνει reserved, τα CMA regions αποκτούν start και end address. Η κάθε διεργασία στη συνέχεια, μπορεί να ζητήσει μέρος από αυτό το CMA region μέσω DMA API call. Το DMA παρέχει τη δυνατότητα σε υποσυστήματα του υλικού του μηχανήματος για άμεση πρόσβαση, ανάγνωση και εγγραφή στη μνήμη του συστήματος, ανεξάρτητα από το CPU.

Για να αποκτήσει μία εφαρμογή μνήμη από το CMA reserved region πρέπει αρχικά να δηλώσει το μέγεθος της μνήμης που χρειάζεται. Το CMA έπειτα, ξεκινά να επιλέγει συνεχόμενες σελίδες μέχρι να ολοκληρώσει το εύρος που ζητήθηκε στο αίτημα. Αν βρεθεί το απαιτούμενο εύρος σελιδών, απομονώνει αυτό το block και αναζητά σελίδες οι οποίες δεν είναι ελεύθερες. Η επιτυχής κατανομή της μνήμης καθορίζεται από το αν το block που απομονώθηκε αποτελείται από μόνο ελεύθερες σελίδες που δε χρησιμοποιούνται από άλλες διεργασίες ή αν καταφέρει μέσω της μετακίνησης κρατημένων σελιδών να καταστήσει το block ελεύθερο. Η ροή ενός αιτήματος για μνήμη από το CMA φαίνεται στο Σχήμα 2.7.



Σχήμα 2.7: Ροή αιτήματος CMA

2.5 FireSim – FPGA hardware simulation platform

2.5.1 FPGA (Field-Programmable Gate Arrays)

Τα FPGAs [14] είναι ολοκληρωμένα κυκλώματα σχεδιασμένα για να ρυθμίζονται από τον χρήστη μετά την κατασκευή τους, επιτρέποντας custom hardware λύσεις για συγκεκριμένες υπολογιστικές εργασίες. Σε αντίθεση με τους παραδοσιακούς επεξεργαστές που εκτελούν εντολές διαδοχικά, τα FPGAs εκμεταλλεύονται τον χωρικό παραλληλισμό, επιτρέποντας την

ταυτόχρονη εκτέλεση πολλαπλών λειτουργιών. Αυτός ο κατακερματισμός, σε συνδυασμό με τη δυνατότητα αναδιαμόρφωσης του υλικού για συγκεκριμένες εφαρμογές, καθιστά τα FPGAs ιδιαίτερα κατάλληλα για εργασίες που απαιτούν high throughput και low latency, όπως η ψηφιακή επεξεργασία σημάτων, η ανάλυση δεδομένων και το machine learning. Η δημοτικότητα των FPGAs αυξήθηκε τα τέλη της δεκαετίας του 1980 και τις αρχές της δεκαετίας του 1990 με την εμφάνιση πιο σύνθετων εργαλείων σχεδιασμού και την αυξανόμενη ενσωμάτωσή τους σε ενσωματωμένα συστήματα, πλατφόρμες υψηλής απόδοσης υπολογιστών και πιο πρόσφατα, σε cloud datacenters.

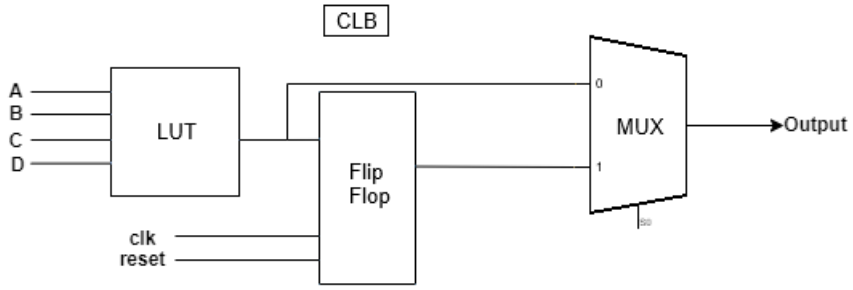
Configurable Logic Blocks (CLBs)

Το CLB χαρακτηρίζεται ως το βασικό component ενός FPGA το οποίο επιτρέπει στο χρήστη να υλοποιήσει σχεδόν οποιαδήποτε λογική λειτουργικότητα εντός του chip. Αποτελείται από πολλαπλά λογικά κελιά τα οποία τυπικά περιλαμβάνουν LUT 4-6 εισόδων, ένα συγκεκριμένο αριθμό Flip-Flop και πολυπλέκτες (Mux) όπως φαίνεται στο Σχήμα 2.8.

- **Look-up Table (LUT)**: Ένα LUT αποθηκεύει μια προκαθορισμένη λίστα εξόδων για κάθε συνδυασμό εισόδων. Τα LUTs παρέχουν έναν γρήγορο τρόπο για να ανακτηθεί η έξοδος μιας λογικής λειτουργίας επειδή τα πιθανά αποτελέσματα είναι αποθηκευμένα και στη συνέχεια αναφέρονται αντί να υπολογίζονται.

- **Flip-Flop**: Ένα κύκλωμα που μπορεί να έχει δύο σταθερές καταστάσεις που αντιπροσωπεύουν ένα μόνο bit. Ένα flip-flop είναι ο μικρότερος αποθηκευτικός πόρος στο FPGA. Κάθε flip-flop σε ένα CLB είναι ένα δυαδικό μητρώο που χρησιμοποιείται για να αποθηκεύσει λογικές καταστάσεις μεταξύ των κύκλων ρολογιού σε ένα κύκλωμα FPGA.

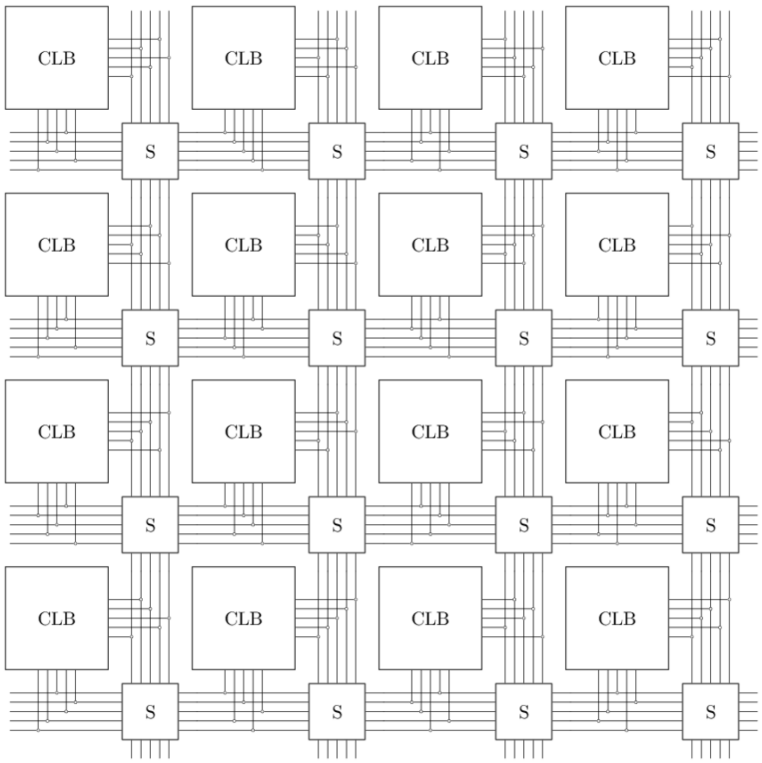
- **Multiplexer**: Ένα κύκλωμα που επιλέγει ανάμεσα σε δύο ή περισσότερες εισόδους και στη συνέχεια επιστρέφει την επιλεγμένη είσοδο.



Σχήμα 2.8: Configurable Logic Block Architecture

Programmable Interconnect Pinpoints (PIPs)

Τα PIPs είναι τα σημεία στα οποία γίνεται η διασύνδεση των CLB και των I/O block εντός ενός FPGA. Η κύρια λειτουργία τους είναι να παρέχουν ευελιξία στη δρομολόγηση των σημάτων σε όλο το FPGA δίνοντας στο χρήστη τη δυναότητα υλοποίησης προσαρμοσμένων κυκλωμάτων. Τα PIPs μπορούν να ενεργοποιηθούν ή να απενεργοποιηθούν με βάση τις συγκεκριμένες απαιτήσεις του σχεδίου. Όταν ένα PIP ενεργοποιείται, δημιουργεί μια σύνδεση μεταξύ δύο σημείων στο δίκτυο διασύνδεσης του FPGA. Αντίστοιχα, όταν απενεργοποιείται, η σύνδεση διακόπτεται. Επίσης, μέσω των PIPs, επιτρέπεται η επαναδρομολόγηση των σημάτων με αποτέλεσμα ένα FPGA να μπορεί να επαναπρογραμματιστεί για διαφορετικές εφαρμογές. Πολλαπλά PIPs αποτελούν ένα Switch Box το οποίο είναι η κεντρική δομή δρομολόγησης και βρίσκεται στις κάθετες και οριζόντιες διασταυρώσεις των καναλιών δρομολόγησης. Η διάταξη των Switch Boxes και των CLBs σε ενα FPGA φαίνεται στο Σχήμα 2.9.



Σχήμα 2.9: Σύνδεση CLB - Switch Box

XILINX U250 FPGA

Το FPGA Xilinix Alveo U250 είναι είναι μια υψηλής απόδοσης επαναπρογραμματιζόμενη υπολογιστική κάρτα σχεδιασμένη για εργασίες σε κέντρα δεδομένων. Βασίζεται στην αρχιτεκτονική UltraScale+, η οποία παρέχει σημαντικές βελτιώσεις απόδοσης σε σχέση με προηγούμενες οικογένειες FPGA της Xilinx, στοιχείο που το καθιστά κατάλληλο για απαιτητικές εφαρμογές όπως το machine learning, video processing και financial computing.

Επίσης επιτρέπει την υλοποίηση σύνθετων σχεδίων καθώς προσφέρει περισσότερα λογικά κελιά, μνήμη και DSP slices. Ακόμα, είναι γνωστό για την ενεργειακή απόδοση του, ιδιαίτερα όταν προσαρμόζεται για συγκεκριμένες εργασίες που μπορούν να οδηγήσουν σε σημαντική εξοικονόμηση ενέργειας. Γενικά, το U250 βρίσκει εφαρμογές σε περιοχές όπου ο συνδυασμός υψηλής απόδοσης, επαναπρογραμματιζόμενης φύσης και ενεργειακής απόδοσης είναι κρίσιμος. Το συγκεκριμένο FPGA θα χρησιμοποιηθεί και για την παρούσα εργασία σε local premises.

2.5.2 Πλατφόρμα Προσομοίωσης FireSim

Το FireSim [6] είναι μία πλατφόρμα προσομοίωσης υλικού ανοιχτού κώδικα για πλήρη συστήματα FPGA-accelerated. Αναπτύχθηκε από την ομάδα Αρχιτεκτονικής του τμήματος Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του πανεπιστημίου UC Berkley και κυκλοφόρησε το 2018. Το project είναι μέρος του ευρύτερου RISC-V οικοσυστήματος καθώς αναπτύχθηκε αρχικά για να προσομοιώσει datacenters συνδυάζοντας RTL σχέδια για επεξεργαστές RISC-V με μια προσαρμοσμένη προσομοίωση δικτύου. Το FireSim αποτελεί πλατφόρμα που χρησιμοποιείται σε πολλούς τομείς όπως την αρχιτεκτονική υπολογιστών, των συστημάτων, των δικτύων, της ασφάλειας και της αυτοματοποιημένης σχεδίασης με contributors από τον ακαδημαϊκό και βιομηχανικό τομέα από όλο τον κόσμο.

Το FireSim μπορεί να λειτουργήσει σε FPGAs on-premises (τοπικα) και σε FPGAs στο public cloud για single-node ή cluster προσομοιώσεις. Δίνει τη δυνατότητα στους χρήστες να αναπτύξουν σχέδια RTL (επεξεργαστές, επιταχυτνές κτλ.) και να τα εκτελέσουν με ταχύτητες που πλησιάζουν τις FPGA-prototype, παίρνοντας αποτελέσματα που μπορούν να συγκριθούν με υλοποιήσεις ASIC του ίδιου σχεδίου. Οι προσομοιώσεις μπορούν να εκτελεστούν σε συχνότητα που ποικίλει από 10δες μέχρι και 100δες MHz, αναλόγως της σχεδίασης υλικού. Επίσης, το FireSim παρέχει στους χρήστες τη δυνατότητα να ενσωματώσουν στο σύστημά τους components βασισμένα σε software models εφόσον δε χρειάζεται ή δεν επιθυμούν να σχεδιάσουν τα RTL. Τα FPGAs που υποστηρίζει μέχρι τώρα το FireSim είναι τα πιο κάτω με κάθε ένα να έχει το δικό του οδηγό έναρξης:

- **AWS EC2 F1**
- **Xilinx Alveo U250**
- **Xilinx Alveo U280**
- **Xilinx VCU118**
- **RHS Research Nitefury II**

2.5.3 FireSim Use Cases and Examples

Σε μόλις 5 χρόνια, από τότε που κυκλοφόρησε το FireSim, πάνω από 40 άρθρα έχουν δημοσιευθεί από τον ακαδημαϊκό και βιομηχανικό τομέα, για τα οποία χρησιμοποιήθηκε το FireSim. Η αξία που προσδίδει στην ανάπτυξη εφαρμογών FPGA-accelerated διαφαίνεται από τα πιο κάτω παραδείγματα.

Ανάπτυξη Επιταχυντών

Το FireSim χρησιμοποιήθηκε για την αξιολόγηση της επίδοσης ενός hardware -assisted Garbage Collector που σχεδιάστηκε από ομάδα ερευνητών του UC Berkeley και δημοσιεύτηκε το 2018. Επίσης, το 2021 δημοσιεύθηκε άρθρο από την ομάδα του UC Berkeley σε συνεργασία με την Google για την ανάπτυξη ενός επιταχυντή για Protocol Buffers [5] (serialization framework). Σε αυτό το άρθρο, το FireSim λειτούργησε ως το εργαλείο για να εκτελέσουν το RTL system και να κάνουν boot το λειτουργικό Linux για να τρέξουν διάφορα hyperscale benchmarks και microbenchmarks.

Αξιολόγηση επεκτάσεων ασφαλείας για CPUs

Σημαντική έρευνα έγινε στα TEEs (Trusted Execution Environments) όπου με τη βοήθεια του FireSim, προσομοιώθηκε και δημοσιεύθυηκε το 2020 το Keystone [8] που αποτελεί το πρώτο ανοιχτού κώδικα framework για τη δημιουργία customized TEEs. Επιπλέον έρευνα έχει γίνει από ομάδα ερευνητών του πανεπιστήμιου Σινγκαπούρης NUS και Ελβετίας ETH Zurich, όπου χρησιμοποιήσαν το FireSim για να προσομοιώσουν το Elasticlave [17], ένα μοντέλο μνήμης που επιτρέπει στα TEEs να μοιράζονται μνήμη με πιο ευέλικτα δικαιώματα.

Κεφάλαιο 3

Σχεδιασμός και Υλοποίηση

Σε αυτό το κεφάλαιο θα γίνει παρουσίαση της μέχρι τώρα μελέτης που έγινε στο TLB σε συνδυασμό με το Page Table Walker και στα Direct Segments. Επισης θα γίνει παρουσίαση της λειτουργίας του TLB του Rocket Chip Generator και έπειτα θα αναλυθεί ο σχεδιασμός και η υλοποίηση των Direct Segments στο TLB ενός Rocket Core, σε επίπεδο υλικού και σε επίπεδο λειτουργικού.

3.1 Λειτουργία του TLB και PTW

Το Rocket Chip διαθέτει ένα Memory Management Unit στο οποίο ορίζεται το TLB και PTW Chisel template [9]. Το TLB template χρησιμοποιείται για να υλοποιηθούν δύο διαφορετικές μονάδες, το Data TLB και Instruction TLB τα οποία είναι συνδεδεμένα με την Data και Instruction Cache αντίστοιχα. Σε περίπτωση αστοχίας μετάφρασης διεύθυνσης, γίνεται προσπέλαση του Page table για την εύρεση της φυσικής διεύθυνσης από ένα Page Table Walker (PTW) το οποίο είναι συνδεδεμένο με το TLB. Στην παρούσα διπλωματική θα ασχοληθούμε με το Data TLB, για αυτό και θα αναφερόμαστε σε αυτό ως TLB, δεδομένου ότι σε ένα memory intensive application οι περισσότερες αστοχίες προκύπτουν σε αυτό και προσδίδουν σε μεγάλο βαθμό ένα overhead στο cycle execution.

3.1.1 Βασικές λειτουργίες του TLB

Ο κύριος σκοπός του TLB είναι να επιταχύνει τη μετάφραση των εικονικών διευθύνσεων που χρησιμοποιούνται από το λειτουργικό σύστημα σε φυσικές διευθύνσεις. Αυτές οι μεταφράσεις βρίσκονται στο Page Table του οποίου η πρόσβαση μπορεί να είναι κοστοβόρα, και για αυτό το λόγο περιορισμένος αριθμός μεταφράσεων αποθηκεύεται προσωρινά στο TLB, του οποίου η πρόσβαση είναι 1 κύκλος. Το TLB δέχεται λοιπόν στην είσοδο του τη ζητούμενη εικονική σελιδά και σε περίπτωση που βρει την αντίστοιχη μετάφραση στη μνήμη του επιστρέφει τη φυσική σελίδα. Σε αντίθετη περίπτωση, στέλνει αίτημα στον PTW προκειμένου να γίνει διάσχιση του πίνακα σελιδών του προγράμματος που αιτήθηκε την εικονική σελίδα.

Οι βασικές λειτουργίες που υλοποιούνται όπως προκύπτουν από πιο πάνω είναι οι εξής:

- **Lookup**: Η αναζήτηση του Virtual Page Number (VPN) στις αποθηκευμένες καταχωρήσεις του TLB. Σε περίπτωση εύρεσης έγκυρης μετάφρασης, έχουμε TLB Hit και επιστρέφεται το αποθηκευμένο Physical Page Number (PPN).

- **Refill**: Το TLB Refill αναφέρεται στη διαδικασία επαναφόρτωσης μιας καταχώρησης στο TLB μετά από μία αποτυχής αναζήτηση, TLB Miss. Σε αυτές τις περιπτώσεις, το TLB προωθεί το αίτημα στον PTW ο οποίος αναμένεται να απαντήσει με το ζητούμενο PPN ή με κάποιο Exception όπως για παράδειγμα το Page Fault. Μόλις βρεθεί το PPN, η καταχώρηση εικονικής-φυσικής διεύθυνσης φορτώνεται ξανά στο TLB, ώστε να είναι διαθέσιμη για μελλοντικές αναφορές.

- **Invalidate**: Μια ακύρωση του TLB (TLB Invalidate) αναφέρεται στη διαδικασία αφαίρεσης μιας ή περισσότερων καταχωρήσεων από το TLB, εξασφαλίζοντας ότι τυχόν μελλοντικές προσβάσεις στην εν λόγω εικονική διεύθυνση θα προκαλέσουν μια αποτυχία αναζήτησης στο TLB. Αυτό μπορεί να γίνει για διάφορους λόγους, όπως η αλλαγή των δικαιωμάτων πρόσβασης σε μια σελίδα, η αποδέσμευση μιας σελίδας από τη μνήμη ή η αλλαγή της αντιστοίχισης μεταξύ εικονικών και φυσικών διευθύνσεων. Το TLB Invalidate είναι απαραίτητο για την ακρίβεια και την ασφάλεια της λειτουργίας του συστήματος.

3.1.2 Βασικές Λειτουργίες του PTW

Το Page Table Walker είναι μία βασική μονάδα για τα συστήματα που χρησιμοποιούν εικονική μνήμη, καθώς η κύρια λειτουργία του είναι να διασχίζει τον πίνακα σελιδών για την εύρεση της μετάφρασης της εικονικής διεύθυνσης σε φυσική. Ωστόσο, στα σύγχρονα συστήματα όπου υπάρχουν πίνακες σελιδών πολλαπλών επιπέδων, η εύρεση ενός PTE μπορεί να είναι αρκετά κοστοβόρα (από 40-100 κύκλους μηχανής).

Οι βασικές λειτουργίες που υλοποιούνται από ένα PTW είναι οι εξής:

- **Page Table Lookup**: Όταν ζητηθεί μία μετάφραση εικονικής διεύθυνσης η οποία δε βρεθεί στο TLB, ενημερώνεται το PTW και ξεκινάει την αναζήτηση της φυσικής διεύθυνσης στον πίνακα σελιδών. Δηλαδή, ξεκινάει από τη διεύθυνση που κρατά ο CSR Satp, η οποία είναι το root του πίνακα σελιδών, και διασχίζει όλα τα επίπεδα μέχρι να βρει το PTE.

- **Page Table Update**: Σε περίπτωση επιτυχημένης εύρεσης της διεύθυνσης στον πίνακα σελιδών, το PTW είναι υπεύθυνο να ενημερώσει ορισμένα bits στο PTE που συμβολίζουν αν η σελίδα έχει γίνει accessed, αν έχει εγγραφεί ή αν είναι valid για τη συναφεία του συστήματος.

- **Handling Page Fault**: Σε περίπτωση που δε βρεθεί η μετάφραση σε φυσική διεύθυνση της ζητούμενης εικονική διεύθυνσης, το PTW είναι υπεύθυνο να κάνει raise ένα Page Fault exception το οποίο στη συνέχεια θα διαχειριστεί το λειτουργικό.

3.2 Οι διεπαφές και τα states του TLB

Η ορθή λειτουργία ενός συστήματος με εικονική διεύθυνση βασίζεται στη σωστή λειτουργία και επικοινωνία των μονάδων που απαρτίζουν το Memory Management Unit, για παράδειγμα του PTW και του TLB καθώς οι λειτουργίες τους είναι άμεσα συνδεδεμένες. Στην περίπτωση του Rocket Core, το TLB επικοινωνεί με το PTW και με τις κρυφές μνήμες μέσω των πιο κάτω διεπαφών, οι οποίες φαίνονται και στο Σχήμα 3.1:

- **TLB Request**: Αποτελεί το request που έρχεται ως είσοδος όταν χρειάζεται μία μετάφραση εικονικής διεύθυνσης σε φυσική. Τυπικά περιέχει την εικονική διεύθυνση μαζί με κάποιες επιπρόσθετες πληροφορίες, όπως το είδος της προσπέλασης.

- **TLB Response**: Μετά την επεξεργασία ενός ρεχυεστ από το TLB, αυτό απαντάει με ένα response ως έξοδος. Σε περίπτωση TLB hit, η απάντηση περιέχει τη φυσική διεύθυνση που αντιστοιχεί στη μετάφραση. Σε αντίθετη περίπτωση, η απάντηση γνωστοποιεί ότι υπήρξε TLB miss και επιστρέφει επίσης πληροφορίες που δείχνουν αν πρέπει να γίνει raise κάποιο exception.

- **TLB-PTW IO**: Αυτή η διεπαφή υλοποιείται για την αμφίδρομη επικοινωνία του TLB με το PTW. Σε περίπτωση ενός TLB Miss, το αίτημα για τη μετάφραση της εικονικής διεύθυνσης προωθείται από το TLB στον Page Table Walker. Από την πλευρά της εξόδου του TLB, άρα της εισόδου στο PTW, προωθείται ένα PTW Request όπως ακριβώς περιγράφηκε παραπάνω. Ταυτόχρονα ο PTW δέχεται ως είσοδο πληροφορίες σχετικά με την κατάσταση των CSRs και την προστασία περιοχών της εικονικής μνήμης (ποιες λειτουργίες εκ των read/write/execute ορίζονται για κάθε περιοχή). Μετά την διάσχιση του πίνακα σελίδων, η δομή του Page Table Walker απαντάει στο TLB με ένα PTW response, το οποίο περιέχει την καταχώρηση του πίνακα σελίδων που βρέθηκε, το επίπεδό της και αν συνέβη κάποιο access exception.

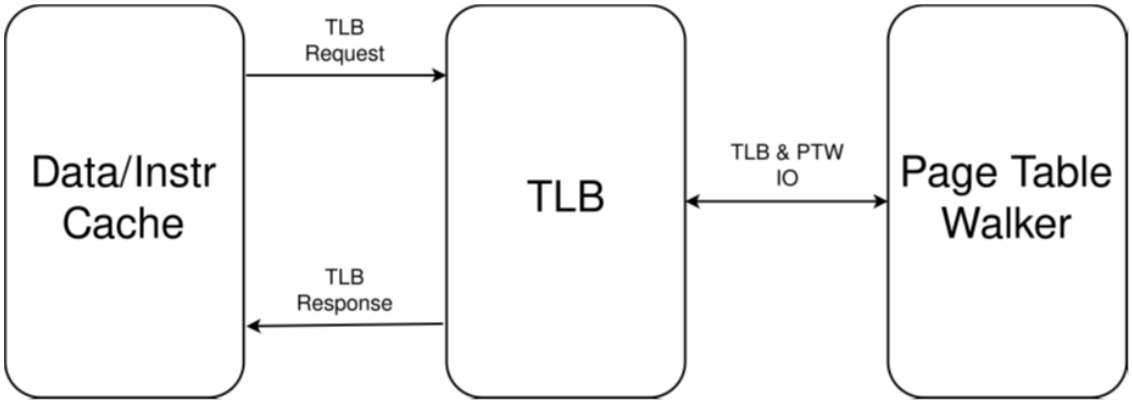
- **Sfence Request**: Το sfence request έρχεται ως είσοδος μετά από την εντολή sfence.vma του λογισμικού, με το οποίο συγχρονίζεται το σύστημα εικονικής μνήμης. Πιο συγκεκριμένα το request χρησιμοποιείται για να γίνουν invalidate μία ή και πολλαπλές καταχωρήσεις. Ο συνδυασμός δύο σημάτων, rs1 και rs2, ορίζουν τον τρόπο που θα γίνει το invalidate, δηλαδή αν θα ακυρωθούν όλες οι καταχωρήσεις του TLB ή καταχωρήσεις που αφορούν συγκεκριμένο χώρο διευθύνσεων.

Η κατάσταση στην οποία βρίσκεται κάθε φορά το TLB ρυθμίζεται από τις πιο πάνω διεπαφές και μπορεί να απεικονιστεί με το διάγραμμα κατάστασης 3.2. Οι 4 καταστάσεις είναι:

- **s_ready**

- **s_request**

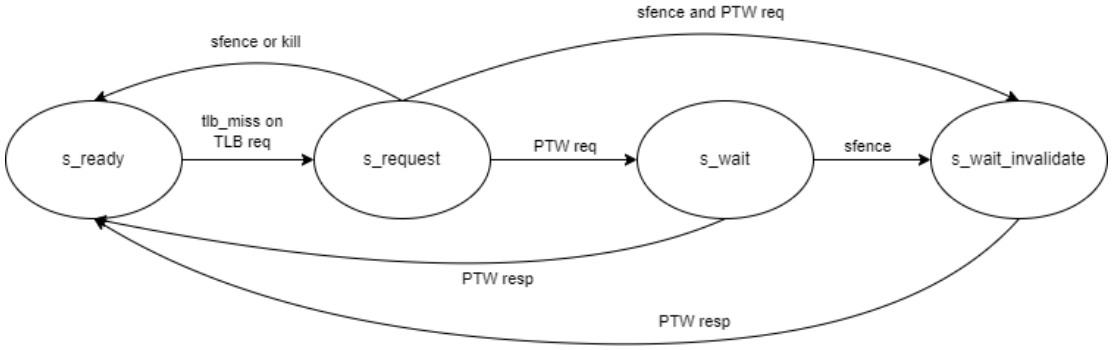
- **s_wait**



Σχήμα 3.1: Οι διεπαφές του TLB με το PTW και Data/Instruction cache

- **s_wait_invalidate**

Η αρχική κατάσταση στην οποία βρίσκεται το TLB, είναι πάντα το s_ready, σύμφωνα με το Rocket Core template. Όταν έρθει ένα TLB Request και δεν βρεθεί πετυχημένα η αντίστοιχη μετάφραση, τότε μεταβαίνει στην κατάσταση s_request. Από εκεί με σήμα kill request από το CPU ή Sfence request μεταβαίνει ξανά στην αρχική κατάσταση, αφού θα γίνει invalidate το TLB entry. Σε περίπτωση όμως που το TLB δημιουργήσει PTW Request για εύρεση της μετάφρασης από το Page Table Walker, τότε μεταβαίνει σε κατάσταση s_wait. Αν παράλληλα έρθει και Sfence request, τότε η κατάσταση του θα είναι s_wait_invalidate ώστε να ακυρώσει και το TLB entry. Από τις 2 τελευταίες καταστάσεις για να μεταβεί ξανά στην αρχική, θα πρέπει να λάβει το PTW Response.



Σχήμα 3.2: Διάγραμμα κατάστασης TLB

3.3 Direct Segments

Στα σύχγρονα συστήματα που χρησιμοποιούν τη συμβατική εικονική μνήμη που βασίζεται στη σελιδοποίηση, παρατηρείται μεγάλος αριθμός κύκλων εκτέλεσης να ξοδεύεται στη διαχείριση των TLB misses. Πιο συγκεκριμένα, εργασίες που χαρακτηρίζονται από τα τεράστια memory footprints και τυχαία μοτίβα πρόσβασης στη μνήμη, συχνά υποφέρουν από υψηλό

ποσοστό αστοχιών ακόμη και όταν χρησιμοποιούνται μεγάλες σελίδες (2MB), με αποτέλεσμα τα TLB miss handling να συμβάλουν σε 5-50% του συνολικού κύκλου εκτελέσεων. Αυτό αποτελεί ένα bottleneck που έγκειται στις ιεραρχικές δομές των Page Table, τα οποία απαιτούν πολλαπλές προσβάσεις στη μνήμη για να επιλύσουν μια μόνο αστοχία του TLB.

Τα Direct Segments [2] εισήχθησαν ως λύση σε αυτήν την πρόκληση. Αποτελεί ένα μηχανισμό στον οποίο περιοχές συνεχόμενων εικονικών διευθύνσεων γίνονται mapped σε περιοχές συνεχόμενων φυσικών διευθύνσεων προσφέροντας μια πιο απλουστευμένη και αποτελεσματική προσέγγιση στη διαχείριση της μνήμης, για εφαρμογές με τεράστιες απαιτήσεις μνήμης. Αυτός ο μηχανισμός υλοποιείται με ελάχιστες προσθήκες υλικού, και μετατροπές της διαχείρισης μνήμης από τον πυρήνα Linux, με αποτέλεσμα τη μείωση της υπερφόρτωσης που σχετίζεται με τις αστοχίες του TLB.

3.3.1 Υποστήριξη υλικού για τα Direct Segments

Ο σκοπός των Direct Segments είναι να επιταχύνει τη μετάφραση εικονικών διευθύνσεων στα σημεία όπου μία διεργασία δεν επωφελείται από την συμβατική εικονική μνήμη που βασίζεται στη σελιδοποίηση. Για αυτό το λόγο, το TLB και PTW διατηρείται ως έχει, και προστίθενται 3 custom καταχωρητές οι οποίοι θα χρησιμοποιούνται για τις εξής λειτουργίες:

- **Base Register**: Σε αυτόν τον καταχωρητή αποθηκεύεται η αρχική διεύθυνση της συνεχόμενης περιοχής εικονικών διευθύνσεων.
- **Limit Register**: Σε αυτόν τον καταχωρητή αποθηκεύεται η τελική διεύθυνση της συνεχόμενης περιοχής εικονικών διευθύνσεων.
- **Offset Register**: Σε αυτόν τον καταχωρητή αποθηκεύεται η αρχική διεύθυνση του συνεχόμενου χώρου φυσικών διευθύνσεων, μείον την αρχική διεύθυνση του συνεχόμενου χώρου εικονικών διευθύνσεων. Δηλαδή τη διαφορά του φυσικού χώρου που αντιστοιχεί στον εικονικό χώρο της διεργασίας.

Η χρήση των καταχωρητών επιτελείται κυρίως στο ΤΛΒ όπου και ορίζεται η λογική του Direct Segment, με το οποίο η μετάφραση εικονικής σε φυσική διεύθυνση διαμορφώνεται ώστε να έχει 2 διαδρομές, μία για το TLB Lookup και μία για το DS Lookup.

Direct Segment Registers

Στο RISC-V Privileged Architecture Specification ορίζεται το encoding space για τους Control and Status Registers το οποίο αποτελείται από 12-bits και μπορεί να υποστηρίξει μέχρι 4096 CSRs. Τα 2 σημαντικότερα bits ([11:10]) δείχνουν αν ο καταχωρητής είναι read-/write όταν έχουν τιμή 00, 01, 10 ή read-only όταν έχουν τιμή 11. Τα 2 επόμενα bits ([9:8)] δείχνουν το λιγότερο προνομιούχο επίπεδο το οποίο έχει πρόσβαση σε αυτούς, με 00 να είναι το User-Level, 01 το Supervisor-Level, 10 το Hypervisor-Level και 11 το Machine-Level. Παράλληλα με το encoding space, ορίζονται οι διευθύνσεις των CSRs που είναι δεσμευμένες για standard και custom χρήσεις.

Το λιγότερο προνομιούχο επίπεδο το οποίο επιθυμούμε να έχει πρόσβαση στους Direct Segment καταχωρητές, είναι το S-mode στο οποίο εκτελείται το λειτουργικό σύστημα. Επίσης οι καταχωρητές θα πρέπει να είναι read/write καθώς σε αυτούς θα αποθηκεύονται και θα διαβάζονται οι αντίστοιχες διευθύνσεις. Σύμφωνα με τα πιο πάνω και το Σχήμα 3.3 για τους Supervisor-Level CSRs, στους Base, Limit, Offset register δόθηκαν οι διευθύνσεις:

- **BASE - 0x5C1**
- **LIMIT - 0x5C2**
- **OFFSET - 0x5C3**

Supervisor-Level CSRs				
00	01	XXXX	0x100-0x1FF	Standard read/write
01	01	0XXX	0x500-0x57F	Standard read/write
01	01	10XX	0x580-0x5BF	Standard read/write
01	01	11XX	0x5C0-0x5FF	Custom read/write
10	01	0XXX	0x900-0x97F	Standard read/write
10	01	10XX	0x980-0x9BF	Standard read/write
10	01	11XX	0x9C0-0x9FF	Custom read/write
11	01	0XXX	0xD00-0xD7F	Standard read-only
11	01	10XX	0xD80-0xDBF	Standard read-only
11	01	11XX	0xDC0-0xDFF	Custom read-only

Σχήμα 3.3: Allocation of RISC-V Supervisor CSR address ranges

Οι Direct Segment καταχωρητές γίνονται aligned με το μέγεθος της σελίδας και έχουν πλάτος ανάλογο του εικονικού χώρου διευθύνσεων. Στην περίπτωση που μελετούμε, η σελίδα είναι 4KB και το σχήμα εικονικής μνήμης είναι το SV39. Επομένος το πλάτος των καταχωρητών είναι 39 - 12 = 27 bits και γίνονται define μέσα στο CSR File του Rocket Core όπως φαίνεται από το πιο κάτω code snippet:

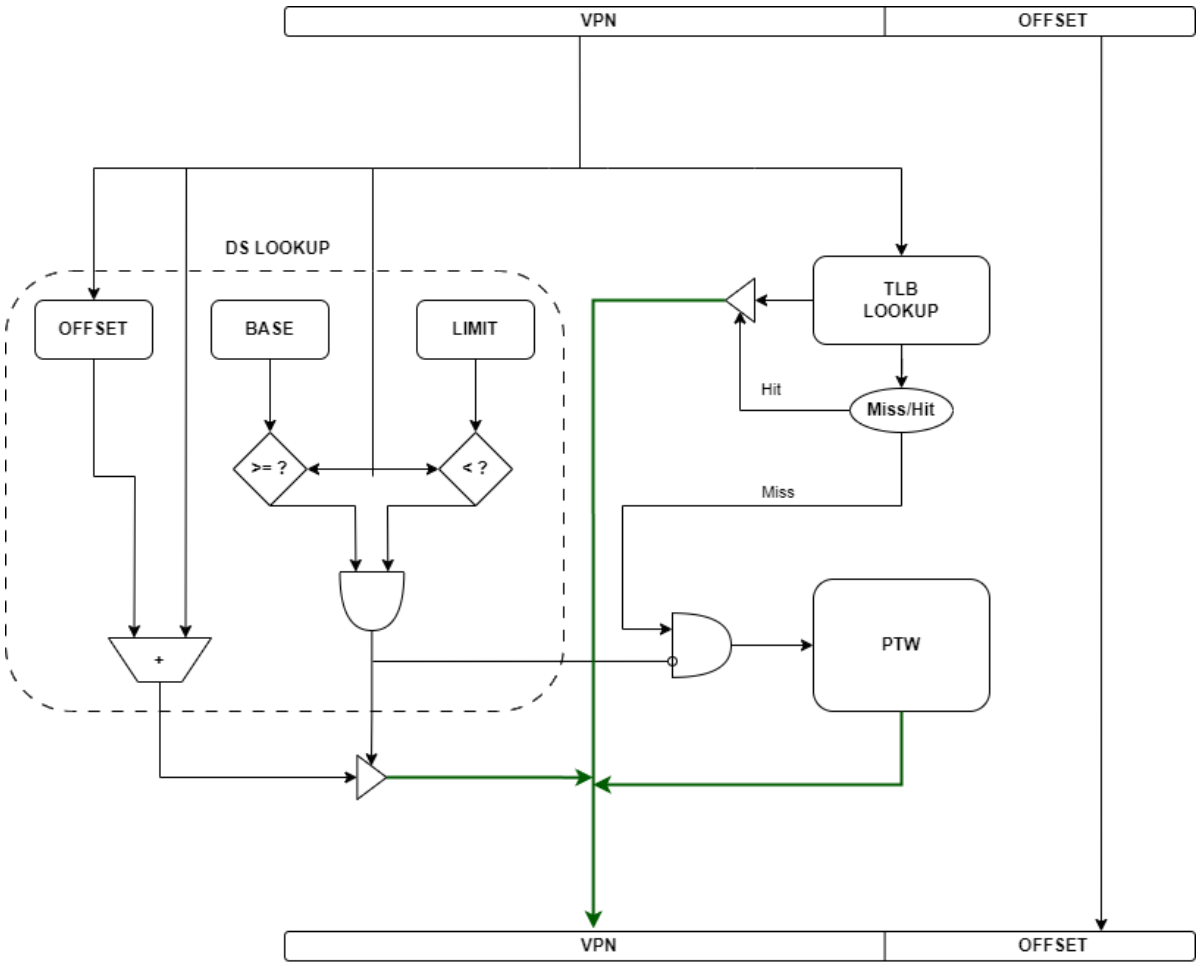
```
class DSB(implicit p: Parameters) extends CoreBundle()(p) {
    val base = UInt(width = vpnBits)
}
class DSL(implicit p: Parameters) extends CoreBundle()(p) {
    val limit = UInt(width = vpnBits)
}
class DSO(implicit p: Parameters) extends CoreBundle()(p) {
    val offset = UInt(width = ppnBits)
}
```

Για τη σωστή σύνδεση και επικοινωνία του λογισμικού με τους καταχωρητές Direct Segment, χρειάστηκε να τους συμπεριλάβουμε μέσα στο datapath. Πιο συγκεκριμένα, τους

προσθέτουμε σαν input στην κλάση DatapathPTWIO όπου γίνονται hooked με το rocket core και στην κλάση TLBPTWIO όπου γίνεται η σύνδεση με το TLB.

Address Translation with Direct Segment Hardware

Με την προσθήκη των καινούριων καταχωρητών, η ροή μετάφρασης της εικονικής διεύθυνσης σε φυσική, όπως κωδικοποιείται στο αρχείο TLB.scala, διαφοροποιείται από τον αρχικό σχεδιασμό. Τα τροποποιημένα σχέδια φαίνονται στο σχηματικό 3.4 στο οποίο προστίθεται το Direct Segment component.



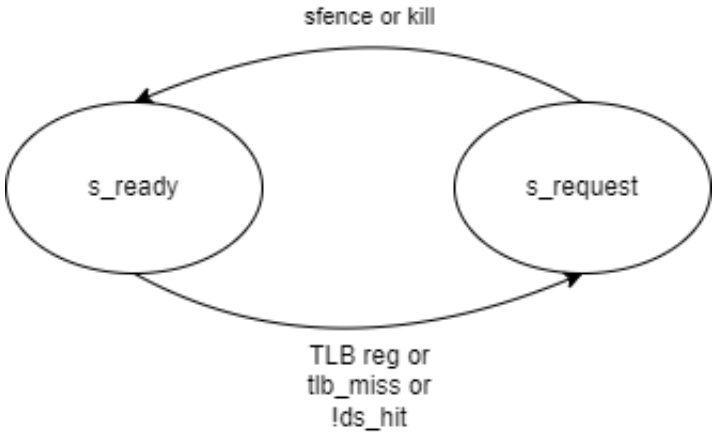
Σχήμα 3.4: Address translation with Direct Segment

Παράλληλα με το TLB Lookup, εκτελείται και το Direct Segment lookup κατά το οποίο συγκρίνεται το Virtual Page Number με την αποθηκευμένη διεύθυνση του καταχωρητή BASE και του καταχωρητή LIMIT. Στην περίπτωση όπου η ζητούμενη μετάφραση βρίσκεται μέσα στο Direct Segment range, τότε προστίθεται η διεύθυνση του καταχωρητή OFFSET ώστε να βρούμε το έγκυρο Physical Page Number. Ο κώδικας για το Direct Segment Lookup φαίνεται πιο κάτω:

```
val ds_base = io.ptw.dsb.base
```

```
val ds_limit = io.ptw.dsl.limit
val ds_offset = io.ptw.dso.offset
val ds_range = Mux((vpn >= ds_base) && (vpn < ds_limit), true.B, false.B)
val ds_hit = ds_range && tlb_miss
ds_ppn := 0.U
if (usingVM) {
   when(ds_range)
      ds_ppn := vpn + ds_offset
}
```

Το πιο πάνω hardware component έχει ως έξοδο ένα καινούριο σήμα, το ds_hit, το οποίο παίρνει τιμή 1 όταν η φυσική διεύθυνση που παράγει το Direct Segment είναι έγκυρη και 0 όταν είναι άκυρη. Αυτό το σήμα χρησιμοποιείται για να οριστούν οι τιμές που θα επιστραφούν στο TLB Response και να ενεργοποιήσει τη μετάβαση του TLB state από s_ready σε s_request όταν το ds_hit έχει τιμή 0 και το tlb_miss έχει τιμή 1. Επομένως, το διάγραμμα κατάστασης διαμορφώνεται όπως φαίνεται στο σχήμα 3.5.



Σχήμα 3.5: Address translation with Direct Segment

3.3.2 Υποστήριξη λογισμικού για τα Direct Segments

Εκτός από την προσθήκη των κατάλληλων καταχωρητών στο υλικό του μηχανήματος, το λειτουργικό σύστημα θα πρέπει να προσαρμοστεί ανάλογα ώστε κάθε διεργασία να χρησιμοποιεί ξεχωριστά το δικό της Direct Segment. Ξεχωρίζουμε λοιπόν, 2 εργασίες για τις οποίες είναι υπεύθυνο το λειτουργικό σύστημα ώστε να εξασφαλίσει τη σωστή λειτουργία:

- **Διάθεση και Διαχείριση του Primary Region**: Το λειτουργικό θα πρέπει να ορίσει για κάθε διεργασία που θα χρησιμοποιήσει τα Direct Segments, μία περιοχή η οποία θα ονομάζεται Primary Region. Αυτή η περιοχή αποτελεί το συνεχόμενο χώρο εικονικών διευθύνσεων οι οποίες θα γίνουν mapped σε συνεχόμενο χώρο φυσικών διευθύνσεων. Επίσης, το σύστημα διαχείρισης της εικονικής μνήμης θα πρέπει να διαθέτει

έναν custom allocator ο οποίος θα δεσμεύει μνήμη από τα Primary Regions και θα μπορεί να ανγνωρίζει πότε πρέπει να γίνει fallback σε σελιδοποίηση.

- **Διαχείριση της φυσική μνήμης και καταχωρητών**: Εκτός από την εικονική μνήμη, το λειτουργικό θα πρέπει να είναι υπεύθυνο, με την έναρξη μιας διεργασίας, να δεσμεύει συνεχόμενο χώρο φυσικής μνήμης μέσω του DMA. Επιπρόσθετα, είναι υπεύθυνο να διαχειρίζεται τους καταχωρητές ώστε σε κάθε context switch, αν η διεργασία διαθέτει primary region, οι Base, Limit και Offset register να έχουν τη σωστή διεύθυνση.

Primary Region

Το Pimary Region αποτελεί μία συνεχή περιοχή εικονικής μνήμης μιας διεργασίας, η οποία έχει read-write access permission και γίνεται mmaped χρησιμοποιώντας τα Direct Segments. Ο κύριος σκοπός του είναι να καλύψει τις περιοχές μνήμης που χαρακτηρίζονται από πολλαπλές προσβάσης, με τρόπο ώστε να βελτιωθεί η απόδοση εφαρμογών που έχουν μεγάλο φόρτο εργασίας μνήμης. Σε αυτή την περιοχή δε χρησιμοποιείται σελιδοποίηση, αλλά χαρτογραφείται ορίζοντας τους καταχωρητές BASE και LIMIT με τη εικονική διεύθυνση που ξεκινά και τελειώνει το Primary Region.

Κατά τη δημιουργία μιας διεργασίας η οποία θα χρησιμοποιήσει Direct Segment, το Λειτουργικό Σύστημα ζητάει ένα μέρος του CMA region μέσω του DMA API αποκλειστικά για τη συγκεκριμένα διεργασία. Το μέγεθος αυτής της περιοχής φυσικής μνήμης που γίνεται reserved είναι και το μέγεθος που θα έχει το Primary Region. Στη συνέχεια, το Λειτουργικό Σύστημα καλείται να βρει ένα virtual memory area (vma) για να κάνει allocate το Primary Region στην εικονική μνήμη. Για τη σωστή διαχείριση του, ορίζονται οι πιο κάτω μεταβλητές στο mm_struct:

- **is_primary**: Έχει τιμή 1 όταν η διεργασία χρησιμοποιεί Direct Segments
- **primary_region_start**: Η εικονική διεύθυνση που αρχίζει το Primary Region
- **primary_region_size**: Το μέγεθος του Primary Region
- **primary_region_mmap_tail**: Η διεύθυνση από την οποία ξεκινάει να δεσμεύει μνήμη το λειτουργικό για τα anonymous memory mappings. Αρχικά, έχει τιμή το τέλος του Primary Region και με κάθε mmap η διεύθυνση μειώνεται.
- **primary_region_heap_head**: Η διεύθυνση η οποία δείχνει το τέλος του heap και χρησιμοποιείται για να δεσμεύει μνήμη το λειτουργικό με το system call "brk". Αρχικά, έχει τιμή την αρχή του Primary Region και με κάθε brk η διεύθυνση αθξάνεται.
- **primary_region_free_mem**: Δείχνει πόση διαθέσιμη μνήμη υπάρχει στο Primary Region για τον έλεγχο του αν μπορεί να δεσμεύσει το ζητούμενο μέγεθος μνήμης ή όχι.

Σύμφωνα με μελέτες, οι περισσότερες προσβάσεις μνήμης, όπου και παρατηρούνται τα περισσότερα TLB misses, είναι οι περιοχές μνήμης που κατανέμονται δυναμικά. Για αυτό το λόγο αποφασίζουμε μέσα στο Primary Region να συμπεριλάβουμαι όλα τα heap allocations και τα anonymous memory allocations, για όσο το υποστηρίζει το μέγεθος του.

Για τη σωστή λειτουργία, αυτά τα δύο κομμάτια επιλέγονται να αυξάνονται προς την αντίθετη κατεύθυνση, με τρόπο ώστε αν πρόκειται το ένα να κάνει overlap με το άλλο, να εκτελείται fallback σε συμβατική σελιδοποίηση. Δηλαδή, το heap έχει αρχική διεύθυνση το primary_region_start και με κάθε allocation το primary_region_heap_head δείχνει σε μεγαλύτερη διεύθυνση. Αντίθετα το anonymous mmap ξεκινάει από το το τέλος του Primary Region και με κάθε allocation το primary_region_mmap_tail δείχνει σε χαμηλότερη διεύθυνση. Με αυτό τον τρόπο, όταν μία διεργασία ζητήσει μέρος εικονικής μνήμης η οποία ξεκινάει από το heap_head και ξεπερνάει το mmap_tail ή το αντίστροφο, απορρίπτεται το allocation στο Primary Region και επσιτρέφει πίσω στη σελιδοποίηση.

Αρχικοποίηση ενός Primary Process

Τα Primary Processes είναι οι διεργασίες οι οποίες χρησιμοποιούν τα Direct Segments και εξασφαλίζουν ένα Primary Region στην εικονική τους μνήμη για να μπορέσει να χαρτογραφηθεί με τους DS registers. Μόλις εντοπιστεί μία τέτοια διεργασία από το λειτουργικό, χρειάζεται να εκτελεστούν ορισμένες διαδικασίες για να διασφαλιστεί η συνεπής λειτουργία της:

- **Reserve CMA region**: Το πρώτο βήμα που εκτελείται μόλις το λειτουργικό αναγνωρίσει ένα primary process, είναι να εξασφαλίσει τη ζητούμενη φυσική μνήμη από το CMA region μέσω του DMA API call "dma_alloc_from_contiguous". Το CMA region αρχικοποιείται κατά το boot του συστήματος.

- **Set up mm_struct**: Αφού βρεθεί ο συνεχόμενος χώρος φυσικών διευθύνσεων, ορίζεται στο mm_struct ότι το συγκεκριμένο process "is_primary", ορίζεται η εικονική διεύθυνση από την οποία θα ξεκινάει το Primary Region και αποθηκεύεται το μέγεθός του.

- **Find available vma**: Στη συνέχεια γίνεται allocation του Primary Region με σκοπό να βρεθεί και να δεσμευτεί συνεχόμενος χώρος διευθύνσεων στην εικονική μνήμη. Με το επιτυχές allocation, το vma παίρνει και αυτό το flag "is_primary".

- **Set the beginning of heap**: Το επόμενο βήμα είναι να οριστεί η αρχή της στοίβας ως η αρχική διεύθυνση του Primary Region και να αποθηκευτεί επίσης στη μεταβλητή primary_region_heao_head ώστε κάθε δυναμική δέσμευση μνήμης brk() να χρησιμοποιεί το Direct Segment.

- **Initialize DS registers**: Τέλος, χρειάζεται να αρχικοποιηθούν οι DS registers BASE, LIMIT, OFFSET με την αρχική διεύθυνση του Primary Region, το μέγεθός του και την αρχική διεύθυνση του φυσικού χώρου συνεχόμενων διευθύνσεων αντίστοιχα.

Επιπρόσθετα, χρειάστηκε να προστεθεί κώδικας, ώστε σε κάθε context switch να γίνεται έλεγχος αν η επόμενη διεργασία είναι Primary Process. Στην αληθή περίπτωση, οι καταχωρητές λαμβάνουν τις σωστές τιμές που είναι αποθηκευμένες στο mm_struct, διαφορετικά ο καταχωρητής LIMIT παίρνει τιμή 0 για την απενεργοποίηση των Direct Segments. Με αυτό τον τρόπο εξασφαλίζεται πως το VPN για το οποίο ζητείται μετάφραση δε θα είναι ποτέ μικρότερο του LIMIT και έτσι το ds_miss signal θα είναι ενεργό.

3.3.3 Διαφορές με την ήδη υπάρχουσα βιβλιογραφία

Η σχεδίαση των Direct Segments σε επίπεδο υλικού που περιγράφηκε πιο πάνω, βασίζεται σε ήδη υπάρχουσα βιβλιογραφία [2]. Αυτή η υλοποίηση έγινε σε αρχιτεκτονική x86-64 και αποτέλεσε τη βάση στην οποία στηρίχτηκε η παρούσα σχεδίαση, έχοντας φυσικά λάβει υπόψη τις θεμελιώδεις διαφορές κάθε αρχιτεκτονικής. Επίσης, υπάρχουσα έρευνα δημοσιεύθηκε [7] το 2018 όπου μελετά την υλοποίηση των Direct Segments σε αρχιτεκτονική RISCV. Ωστόσο, σύμφωνα με τη δημοσίευση, η υλοποίηση περιορίστηκε σε επίπεδο λογισμικού καθώς κύριος σκοπός της ήταν να τροποιήσει τον πυρήνα Linux ώστε να μπορεί να υποστηρίξει το Direct Segment hardware. Αυτή η υλοποίηση έτρεξε μόνο με το εργαλείο προσομοίωσης QEMU και δεν περιείχε αποτελέσματα που να αποδεικνύουν πιθανή βελτίωση της απόδοσης του συστήματος με τη χρήση των Direct Segments. Αξίζει να σημειωθεί πως μέχρι τότε, δεν υπήρχαν προηγμένα εργαλεία που να διευκόλυναν την παραγωγή bitstream και εκτέλεση πυρήνα Linux σε FPGA, καθώς το FireSim κυκλοφόρησε για πρώτη φορά ως ανοιχτού κώδικα εργαλείο το 2018.

Η υλοποίηση που παρουσιάζουμε στην παρούσα διπλωματική αποτελεί ολοκληρωμένη σχεδίαση λογισμικού και υλικού. Περιλαμβάνει τροποποιημένο πυρήνα Linux ώστε να υποστηρίζει τη διαχείριση των Direct Segment καταχωρητών, τη διαχείρηση του Primary Region και τη σωστή δέσμευση μνήμης μέσα σε αυτό. Επίσης περιλαμβάνει τροποποιημένο rocket core στο οποίο προστίθενται 3 καινούριοι καταχωρητές, οι οποίοι γίνονται hooked κατάλληλα και υποστηρίζουν την άμεση μετάφραση μίας εικονικής διέθυνσης η οποία εμπίπτει στο Primary Region. Τέλος, για να αποδειχθεί η βελτιστοποίηση της εικονικής μνήμης με την παρούσα σχεδίαση, παράγεται το αντίστοιχο bitstream το οποίο φορτώνεται σε FPGA, γίνεται boot ο αντίστοιχος πυρήνας πάνω σε αυτό και εκτελούνται διάφορα benchmarks από τα οποία παρουσιάζουμε τα συμπεράσματα για την επίδοση των Direct Segments.

Κεφάλαιο 4

Μεθοδολογία

Σε αυτό το κεφάλαιο θα αναλυθούν όλα τα στάδια που ακολουθήθηκαν, από τη σχεδίαση μέχρι την υλοποίηση, ώστε να φτάσουμε σε ένα ολοκληρωμένο σύστημα που να υποστηρίζει Direct Segments. Η ανάπτυξη των Direct Segments, στην παρούσα διπλωματική, χωρίζεται σε δύο κύριες ενότητες, την ανάπτυξη του υλικού και την ανάπτυξη του λογισμικού. Σε κάθε ενότητα, θα παρουσιαστούν όλες οι πλατφόρμες καθώς και όλα τα εργαλεία που χρησιμοποιήθηκαν για τον έλεγχο της ορθότητας και επίδοσης της υλοποίησης.

4.1 Ροή ανάπτυξης υλικού

Η ανάπτυξη του υλικού έγινε σε 2 στάδια. Το πρώτο στάδιο αφορά στον έλεγχο της ορθότητας της σχεδίασης και έγινε μέσω του ακριβή ανά κύκλο επεξεργασίας προσομοιωτή Verilator (cycle-accurate hardware emulator). Το δεύτερο στάδιο αφορά στην επίδοση της σχεδίασης και έγινε τρέχοντας διάφορα SPEC benchmarks μέσω του Alveo U250 FPGA, χρησιμοποιώντας το FireSim. Οι διαφορές μεταξύ των προσεγγίσεων συνοψίζονται στον πίνακα 4.1. Σε κάθε περίπτωση, χρειάστηκε να τροποποιηθεί ο Rocket του Rocket Chip Generator προσθέτοντας κώδικα ἢisel για την υλοποίηση του Direct Segment hardware.

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

Πίνακας 4.1: Hardware emulation vs FPGA

4.1.1 Στήσιμο RISCV environment

Η πρώτη ενέργεια που χρειάστηκε ώστε να ξεκινήσει η υλοποίηση είναι το στήσιμο του RISCV environment. Για αυτό το βήμα, κατεβάσαμε τα απαραίτητα εργαλεία λογισμικού rocket-tools[1] ακολουθώντας τις οδηγίες στον αντίστοιχο σύνδεσμο, ώστε να υποστηρίξουμε

[1]https://github.com/chipsalliance/rocket-tools

τη σχεδίαση στο Rocket Chip Generator. Τα rocket-tools περιέχουν τα απαραίτητα εργαλεία όπως το riscv-gnu-toolchain που αποτελεί το RISCV C και C++ cross-compiler, τα riscv-tests που αποτελούν βασικό εργαλείο για τον έλεγχο της ορθότητας και τα riscv-opcodes που περιέχουν τους τυπικούς κωδικούς εντολών και CSRs για διάφορα Riscv εργαλεία. Στη συνέχεια κατεβάσαμε το rocket-chip-generator[2] που περιέχει τα απαραίτητα αρχεία για την υλοποίηση του rocket-core, όπως και το hardware emulator Verilator με το οποίο εκτελούνται προσομοιώσεις πάνω στο hardware.

4.1.2 Προσομοίωση υλικού με υποστήριξη λογισμικού

Verilator

Ο Verilator [13] είναι ένα ελεύθερο και ανοιχτού κώδικα εργαλείο το οποίο μετατρέπει τη Verilog, μία γλώσσα περιγραφής υλικού που παράγει η Chisel σε ένα συμπεριφορικό cycle-accurate C++/SystemC μοντέλο. Χρησιμοποιείται κυρίως για προσομοίωση και εκτέλεση εκτεταμένων σουιτών δοκιμών σε σχέδια υλικού, επιτρέποντας υψηλής απόδοσης προσομοίωση και εκτενείς ελέγχους της λογικής του σχεδιασμένου υλικού πριν από την κατασκευή του. Στα πλαίσια του Rocket Chip Generator, το Verilator χρησιμοποιείται για την προσομοίωση του παραγόμενου κώδικα του πυρήνα Rocket και για την αποσφαλμάτωση του σχεδιασμού με δηλώσεις `assert - printf`, καθώς υποστηρίζει μηνύματα debug ανά κύκλο μηχανής.

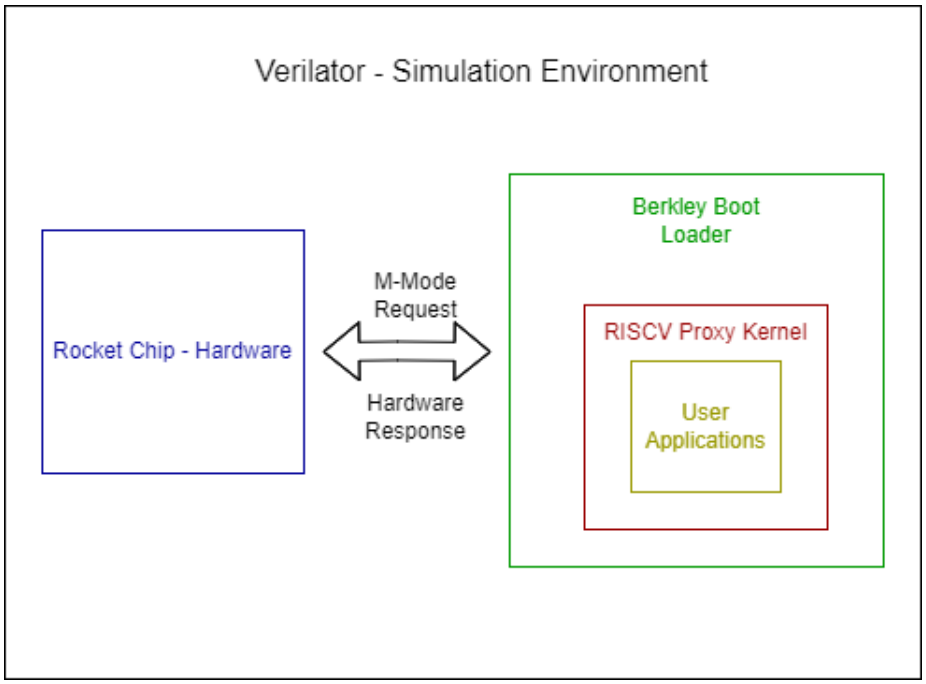
Χρησιμοποιώντας το εκτελέσιμο που παράγει ο Verilator τρέχουμε τα official riscv-tests τα οποία αποτελούνται από assembly tests και διάφορα benchmarks(dhrystone, qsort, spmv μεταξύ άλλων). Από αυτά, λαμβάνουμε τα πρώτα αποτελέσματα για την ορθότητα του σχεδιασμού. Έχοντας λοιπόν προσθέσει το Direct Segment hardware στο critical path, ελέγχουμε αρχικά πως δεν επηρεάζει τη φυσιολογική λειτουργία του rocket core όταν αυτό είναι απενεργοποιημένο. Στη συνέχεια, για τον έλεγχο ορθής λειτουργίας του Direct Segment χρειάζεται να προσθέσουμε υποστήριξη λογισμικού ώστε να διαχειρίζεται τους καταχωρητές ανάλογα με το εκτελέσιμο. Σημειώνουμε ότι ο Verilator διευκολύνει πολύ το debugging αλλά είναι πολύ αργός (μία ημέρα για να εκκινήσει linux σε σύγχρονο επεξεργαστή x86-64). Για αυτό το λόγο επιλέξαμε να εκκινήσουμε το RISCV proxy kernel που εμπεριέχεται μέσα στα rocket-tools.

Proxy Kernel and Berkeley Boot Loader

Το RISCV proxy kernel, γνωστό και ως pk, είναι ένα ελαφρύ λειτουργικό σύστημα που λειτουργεί ως ενδιάμεσος (proxy) για την εκτέλεση εφαρμογών χρήστη μονής διεργασίας σε περιβάλλοντα όπου δεν υπάρχει πλήρες λειτουργικό σύστημα. Το proxy kernel παρέχει μια απλοποιημένη διεπαφή συστήματος για εφαρμογές χρήστη και χειρίζεται κλήσεις συστήματος όπως η διαχείριση μνήμης, είσοδος/έξοδος αρχείων, και άλλες βασικές λειτουργίες, καθιστώντας το ιδιαίτερα χρήσιμο για την αποσφαλμάτωση, την ανάλυση και την επαλήθευση των σχεδιασμών hardware σε ένα ελεγχόμενο περιβάλλον. Μαζί με το pk έρχεται και ο Berkley Boot Loader(BBL), ο οποίος αποτελεί φορτωτή εκκίνησης που χρησιμοποιείται για την

[2]https://github.com/chipsalliance/rocket-chip

εκκίνηση του RISC-V πυρήνα. Οι κύριες λειτουργίες του BBL είναι να φορτώνει τον πυρήνα στη μνήμη, να αρχικοποιεί τον υλικοτεχνικό εξοπλισμό και να περνά τον έλεγχο στον πυρήνα για να ξεκινήσει η εκτέλεση του λειτουργικού συστήματος. Στην περίπτωση που μελετάται, ο bootloader είναι υπεύθυνος να εκκινήσει το pk στο προσομοιωμένο περιβάλλον υλικού που παρέχει ο Verilator. Η διαδικασία εκκίνησης και αλληλεπίδρασης του Verilator, BBL και pk φαίνεται στο σχηματικό 4.1.



Σχήμα 4.1: Verilator, BBL and pk interaction

Τροποποιήσεις κώδικα

Αρχικά ορίσαμε τις διευθύνσεις στο αρχείο encoding.h του machine directory στις οποίες κωδικοποιούνται οι Direct Segment καταχωρητές ώστε να υπάρχει επικοινωνία υλικού-λογισμικού. Έπειτα, τροποποιήσαμε το αρχείο μμαπ.ς ώστε να υποστηρίζει το Primary region από το οποίο θα δεσμεύουν μνήμη οι κλήσεις συστήματος mmap και brk. Πιο συγκεκριμένα, κατά την αρχικοποίηση της εικονικής μνήμης μίας διεργασίας, ορίζεται το μέγεθος της κύριας μνήμης να είναι ίσο με 2GB και η διεύθυνση της πρώτης φυσικής σελίδας. Τον συνεχόμενο χώρο φυσικών διευθύνσεων στον οποίο θα χαρτογραφηθεί ο συνεχόμενος χώρος εικονικών διευθύνσεων, αποφασίσαμε αυθαίρετα να τον τοποθετήσουμε 100 σελίδες μετά την πρώτη διαθέσιμη και ορίσαμε το μέγεθός του να είναι ίσο με 10MB:

```
ds_first_free_paddr = first_free_paddr + 100 * RISCV_PGSIZE; //DS added
printk("mmap.c - ds_first_free_paddr is:  %lx", ds_first_free_paddr);
ds_current_free_paddr = ds_first_free_paddr;
ds_size = 1048576; // 10MB
```

Στη συνέχεια, για κάθε κλήση συστήματος η οποία θέλει να δεσμεύσει δυναμικά μνήμη με

δικαιώματα PROT_READ — PROT_WRITE και δεν γίνεται backed από κάποιο file, καλεί τη συνάρτηση set_direct_segment ούτως ώστε να χρησιμοποιήσει τους DS καταχωρητές. Την πρώτη φορά που θα γίνει κλήση της συγκεκριμένης συνάρτησης θα αρχικοποιηθεί ο καταχωρητής BASE, LIMIT, OFFSET και για κάθε επόμενη φορά θα αλλάζει μόνο ο καταχωρητής LIMIT. Σημειώνουμε ότι γίνεται έλεγχος του μεγέθους που πρέπει να δεσμευτεί ώστε να μην ξεπεράσει το συνολικό μέγεθος Direct Segment στη φυσική μνήμη.

```
static int __set_direct_segment(uintptr_t addr, size_t npage) {
   printk("Entered set_ds with addr %p and npage %d", addr, npage);
   if (npage * RISCV_PGSIZE <= (ds_size -
         (ds_current_free_paddr - ds_first_free_paddr))) {
      ds_current_free_paddr = ds_current_free_paddr + npage * RISCV_PGSIZE;
      printk("ds_current_free_paddr is %p", ds_current_free_paddr);
      if (!ds_set) {
         ds_limit = addr + npage * RISCV_PGSIZE;
         write_csr(sdsb, addr >> RISCV_PGSHIFT);
         write_csr(sdso, (ds_first_free_paddr - addr) >> RISCV_PGSHIFT);
         write_csr(sdsl, ds_limit >> RISCV_PGSHIFT);
         printk("DS_Set");
      } else {
         ds_limit = ds_limit + npage * RISCV_PGSIZE;
         write_csr(sdsl, ds_limit >> RISCV_PGSHIFT);
      }
      if (!ds_set) {
         ds_set = 1;
      }
      printk("SDSL set");
      return 0;
   }
   printk("SDSL NOT set");
   return 1;
}
```

Ο πιο πάνω κώδικας χρησιμοποιηθήκε με σκοπό την ανάλυση της ορθής λειτουργίας του Direct Segment hardware για αυτό και γράφτηκαν custom microbenchmarks που λαμβάνουν υπόψη την πιο πάνω διαχείριση των Direct Segments από το λειτουργικό. Αυτά τα benchmarks έκαναν χρήση της κλήσης malloc() σε επίπεδο χρήστη, με σκοπό να δεσμεύσουν συνεχόμενη εικονική μνήμη η οποία μεταφράζεται σε συνεχόμενη φυσική μνήμη μέσω των αντίστοιχων καταχωρητών. Επιπρόσθετα, γίνεται ανάγνωση και εγγραφή των εικονικών διευθύνσεων που δεσμεύονται δυναμικά ώστε να γίνει έλεγχος ορθής χρήσης του εισαχθέντος υλικού στο Memory Management Unit του rocket-core. Η αποσφαλμάτωση έγινε με τη χρήση της κλήσης printf() σε επίπεδο χρήστη και machine και printk() σε επίπεδο supervisor για τη διόρθωση και επανασχεδίαση του υλικού μέχρι να πετύχουμε το απαιτούμενο αποτέλεσμα. Ορισμένα

θέματα που εμφανίστηκαν κατά τη διάρκεια ανάπτυξης του υλικού ήταν ο συγχρονισμός, δηλαδή η επιτυχής μετάφραση μία διεύθυνσης σε 1 κύκλο, η σωστή ανάγνωση/εγγραφή των καταχωρητών στον ίδιο κύκλο και η σωστή προώθηση των σημάτων για την αλλαγή των states.

4.1.3 Στήσιμο του FireSim

Για να στήσουμε το περιβάλλον του FireSim ακολουθήσαμε το σχετικό Documentation[3] για την έκδοση 1.17. Αρχικά στήθηκε το Run Farm Machine που είναι το μηχάνημα στο οποίο εκτελούνται οι προσομοιώσεις και είναι συνδεδεμένο το FPGA. Στη συνέχεια στήθηκε το Manager Machine το οποίο αποτελεί το κύριο μηχάνημα που εκκινεί τα builds και τις προσομοιώσεις. Σε πρώτο στάδιο κάναμε clone το Firesim repo και checkout στην έκδοση 1.17. Σε δεύτερο στάδιο, εγκαταστήσαμε τον software package manager Miniforge Conda με τον οποίο δημιουργείται το περιβάλλον λογισμικού firesim και εκτελούμε την εντολή setup που κάνει install τα RISC-V tools και dependencies. Τέλος, παράξαμε τα αρχικά configuration files με τα οποία ορίζονται οι παραμέτροι για το bitstream build και την εκτέλεση προσομοιώσεων.

4.1.4 Προετοιμασία bitstream

Το επόμενο στάδιο, αφού ολοκληρλωθηκε ο έλεγχος της ορθής λειτουργίας του προσαρμοσμένου υλικού στον Verilator και στήθηκε το περιβάλλον του FireSim, είναι η διαδικασία παραγωγής του bitstream. Το bitstream είναι ένα binary αρχείο το οποίο φορτώνεται στο FPGA και καθορίζει πώς τα προγραμματιζόμενα λογικά στοιχεία και οι διασυνδέσεις πρέπει να διαμορφωθούν για να υλοποιήσουν το σχεδιασμένο σύστημα. Περιέχει δηλαδή τον προγραμματισμό των **Look-up Tables (LUTs), Flip-Flops (FFs)** του PL καθώς και την δρομολόγηση (routing) μεταξύ των στοιχείων συνδυαστικής λογικής και μνήμης.

Για την παραγωγή του bitstream χρειάζεται να τροποποιήσουμε το config_build αρχείο του FireSim. Ορίζουμε το default_build_dir που είναι ο φάκελος στον οποίο θέλουμε να αποθηκευτεί το bitstream και το build_farm_hosts που είναι το μηχάνημα το οποίο θα αναλάβει τη διαδικασία της παραγωγής. Επίσης ορίζουμε το builds_to_run που αποτελεί το configuration του FPGA στο οποίο θα αναφέρεται το bitstream, στην περίπτωσή μας είναι το μονού πυρήνα alveo_u250_firesim_rocket_singlecore_no_nic. Στη συνέχεια τρέχουμε την εντολή firesim buildbitstream με την οποία πραγματοποιείται η διαδικασία κατασκευής παίρνοντας το RTL Chisel και παράγοντας ένα bitstream που λειτουργεί στο FPGA Xilinx Alveo U250, ακολουθώντας τα πιο κάτω βήματα:

1. Δημιουργία του RTL (Register-Transfer Level σχεδίου από τη γλώσσα περιγραφής υλικού Chisel στην οποία είναι γραμμένος ο rocket core και οι υπόλοιπες μονάδες του Rocket Chip Generator.

[3]https://docs.fires.im/en/stable/Getting-Started-Guides/On-Premises-FPGA-Getting-Started/Initial-Setup/Xilinx-Alveo-U250.html

2. Σύνθεση της υψηλού επιπέδου περιγραφής RTL του υλικού σε ένα χαμηλού επιπέδου netlist το οποίο περιγράφει το σχέδιο σε όρους πυλών λογικής, flip-flops και άλλων υλικών πρωτογενών στοιχείων.

3. Τοποθέτηση και δρομολόγηση των λογικών στοιχείων σε συγκεκριμένες θέσεις στο FPGA.

4. Παραγωγή του bitstream που περιέχει τον προγραμματισμό και τη διασύνδεση των λογικών στοιχείων.

5. Δημιουργία του output directory που περιέχει όλες τις παραγόμενες εξόδους

4.2 Ροή ανάπτυξης Λογισμικού

Αφού ολοκληρωθούν τα βήματα για την προετοιμασία του υλικού προχωράμε στην προετοιμασία του λογισμικού, δηλαδή στην παραγωγή του βασικού Linux περιβάλλοντος. Τα πρώτα στάδια αξιολόγησης του λογισμικού για τη διαχείρηση του Direct Segment hardware έγιναν πιο πάνω χρησιμοποιώντας τον minimal Linux kernel pk σε συνδυασμό με το Berkeley Boot Loader. Ωστόσο, για ένα ολοκληρωμένο περιβάλλον Linux που υποστηρίζει πολλαπλές διεργασίες, το λογισμικό που χρησιμοποιήθηκε δεν είναι αρκετό. Για αυτό για την ανάπτυξη λογισμικού σωστής διαχείρισης των Direct Segments θα βασιστούμε στις παρακάτω τεχνολογίες:

- **QEMU**: Ανοιχτού κώδικα εργαλείο εξομοίωσης επεξεργαστή υπολογιστικού συστήματος με τη δυνατότητα υποστήριξης πολλαπλών instruction set.
- **Open-Sbi**: Ανοιχτού κώδικα υλοποίηση firmware για αρχιτεκτονική RISC-V που χρησιμοποιείται για άμεση εκκίνηση του πυρήνα Linux.
- **Linux**: Ανοιχτού κώδικα λειτουργικό σύστημα τύπου Unix.

4.2.1 QEMU

Το QEMU αποτελεί ανοιχτού κώδικα εργαλείο που δίνει τη δυνατότητα εξομοίωσης πλήρως υλικού εξοπλισμού ενός υπολογιστή, επιτρέποντας την εκτέλεση λειτουργικών συστημάτων και εφαρμογών σε διαφορετικές αρχιτεκτονικές. Στην παρούσα διπλωματική χρησιμοποιηθήκε η έκδοση 8.0.0 και κατασκευάστηκε το εκτελέσιμο binary για αρχιτεκτονική RISC-V με την εντολή ./configure –target-list=riscv64-softmmu.

Για την υποστήριξη των Direct Segments χρειάστηκε να γίνουν ορισμένες τροποποιήσεις στα αρχεία του κώδικα:

1. cpu_helper.c
2. cpu_bits.h
3. cpu.h

4. cpu.c

5. csr.c

Πιο συγκεκριμένα η λογική των Direct Segments υλοποιήθηκε στο αρχείο cpu_helper.h στη μέθοδο get_physical_address() η οποία εκτελεί τη μετάφραση απο εικονική σε φυσική διεύθυνση. Στον κώδικα που προσθέσαμε διαβάζουμε τους DS registers πριν εκτελεσθεί το page table walk και αν η εικονική διεύθυνση βρίσκεται μέσα στο Primary region επιστρέφουμε τη σωστή φυσική διεύθυνση προσθέτοντας της τον καταχωρητή offset, με δικαιώματα PAGE_READ και PAGE_WRITE:

```
// Use Direct Segment registers to get the physical address
hwaddr offset;
hwaddr sdsb = (hwaddr)get_field(env->sdsb, SATP64_PPN) << PGSHIFT;
hwaddr sdsl = (hwaddr)get_field(env->sdsl, SATP64_PPN) << PGSHIFT;
if (addr >= sdsb && addr < sdsl) {
   offset = (hwaddr)get_field(env->sdso, SATP64_PPN) << PGSHIFT;
   *physical = addr + offset;
   qemu_printf("SDSB: %lx , SDSL: %lx , SDSO: %lx , virtual:  %lx , physical:
%lx", sdsb, sdsl, offset, addr, *physical);
   *ret_prot = PAGE_READ | PAGE_WRITE;
   return TRANSLATE_SUCCESS;
}
```

Στα υπόλοιπα αρχεία έγιναν τα definitions των καταχωρητών, ορίστηκαν τα δικαιώματα ανάγνωσής τους και συμπεριληφθήκαν μέσα στο cpu.

4.2.2 Open-Sbi

Το Open-Sbi αποτελεί μία ανοιχτού κώδικα υλοποίηση του RISC-V Supervisor Binary Interface και καθορίζει τη διεπαφή μεταξύ του supervisor execution environment (SEE) και του λογισμικού που θα εκτελεστεί σε αυτό, στην περίπτωση μας ο πυρήνας Linux. Το Open-Sbi τρέχει σε M-mode οπότε έχει έχει πλήρη διαφάνεια στον RISC-V Core, προετοιμάζει το σύστημα, δηλαδή τους απαραίτητους CSRs και το μηχανισμό Physical Memory Protection (PMP). Αφού ολοκληρώσει την προετοιμασία του περιβάλλοντος, παραδίδει την εκτέλεση στο Linux. Ωστόσο, εκτός από την αρχικοποίηση του μηχανήματος, το Open-Sbi λειτουργεί και ως firmware διευκολύνοντας την επικοινωνία του λειτουργικού με το υλικό. Είναι υπεύθυνο για να εκτελέσει αιτήματα του λειτουργικού που χρειάζονται M-mode privilege, όπως τη ρύθμιση των timers και αποστολή διακοπών διεργασίας (IPIs. Επίσης είναι υπεύθυνο για το χειρισμό των traps και interrupts, καθώς οι διακοπές έχουν ως αποτέλεσμα τοη μεταφορά του ελέγχου σε αυτό. Η έκδοση του Open-Sbi που χρησιμοποιήσαμε είναι η 1.2 και έρχεται μαζί με το Firesim.

4.2.3 RISC-V Linux

Για το στήσιμο των Direct Segments σε επίπεδο λογισμικού χρησιμοποιήσαμε τον πυρήνα Linux έκδοσης 6.2 της διανομής Fedora που έρχεται μαζί με το Firesim. Για τη σωστή υποστήριξη των Direct Segments από τον πυρήνα χρειάστηκε να ολοκληρώσουμε τα πιο κάτω βήματα:

1. Δέσμευση συνεχόμενου χώρου φυσικών διευθύνσεων μέσω του CMA
2. Αναγνώριση των διεργασιών που θα κάνουν χρήση των Direct Segments
3. Στήσιμο του Primary Region και του memory management struct της διεργασίας
4. Υλοποίηση custom mmap μεθόδου που θα διαχειρίζεται τη μνήμη από το Primary region

CMA Region

Για το configuration του CMA υπάρχουν 3 διαφορετικοί τρόποι, από το kernel configuration, το kernel cmdline ή το device tree. Εμείς επιλέξαμε να το κάνουμε configure μέσω του linux configuration στο οποίο ορίσαμε τις πιο κάτω τιμές:

- **CONFIG_CMA=y**: Ενεργοποιεί τον Contiguous Memory Allocator που διατηρεί περιοχή φυσικής μνήμης από την οποία θα δεσμευτούν συνεχόμενα τμήματα
- **CONFIG_DMA_CMA=y**: Εξασφαλίζει ότι οι λειτουργίες Direct Memory Access (DMA) μπορούν να χρησιμοποιήσουν συνεχείς περιοχές μνήμης που παρέχονται από το CMA
- **CONFIG_CMA_SIZE_SEL_MBYTES=y**: Δηλώνει ότι το μέγεθος της περιοχής CMA ορίζεται σε MB
- **CONFIG_CMA_SIZE_MBYTES=1024**: Ορίζει το μέγεθος της περιοχής CMA. Στην περίπτωση που μελετάμε διατηρεί 1024 MB
- **CONFIG_CMA_DEBUG=y**: Ενεργοποιεί την υποστήριξη αποσφαλμάτωσης για το CMA. Χρησιμοποιείται κατά τη διάρκεια ανάπτυξης του λογισμικού.

Αναγνώριση Primary process

Για την αναγνώριση των διεργασιών που θα χρησιμοποιήσουν τα Direct Segments επιλέξαμε να βασιστούμε στο όνομα του executable. Κατά την αρχικοποίηση της διεργασίας γίνεται ένας έλεγχος εντός του function begin_new_exec() του αρχείου exec.c και αν το όνομα του εκτελέσιμου τελειώνει με το suffix "_ds" τότε αναγνωρίζεται ως primary process και θέτουμε το flag is_primary του mm_struct στην τιμή 1.

Στήσιμο Primary Region

Το στήσιμο του Primary Region αποφασίσαμε να εκτελείται κατά τη διάρκεια του set_brk() στο αρχείο binfmt_elf.c. Αρχικά, μέσω του dma_alloc_from_contiguous() δεσμεύεται συνεχόμενος χώρος διευθύνσεων από το CMA Region και επιστρέφεται η αρχική του διεύθυνση. Στη συνέχεια, εκτελείται το allocate_primary_region() που αποτελεί custom μέθοδο με σκοπό τη δέσμευση του συνεχόμενου χώρου διευθύνσεων στην εικονική μνήμη, Αυτή η μέθοδος καλείται με αρχική διεύθυνση την διέυθυνση από την οποία ξεκινάει το heap και μέγεθος το συνεχόμενο χώρο που δεσμεύσαμε από το CMA region. Η σωστή εκτέλεσή της προϋποθέτει να βρεθεί ένα unmapped area το οποίο θα δεσμευτεί με δικαιώματα read και write και να δημιουργηθεί το virtual memory area (vma) στην εικονική μνήμη με το ζητούμενο μέγεθος και τη ζητούμενη αρχική διεύθυνση. Η εκτέλεση της πιο πάνω μεθόδου χωρίς κάποιο σφάλμα συνεπάγεται σωστή δέσμευση του Primary Region.

Διαχείριση Primary Region

Το τελευταίο στάδιο για να πετύχουμε ολοκληρομένη υποστήριξη των Direct Segments από το λειτουργικό είναι η διαχείριση της μνήμης που δεσμεύτηκε για το Primary Region. Η διαχείρισή της θα γίνεται με 2 τρόπους, με το system call brk() και το system call mmap(). Στη πρώτη περίπτωση, εντός του brk() ελέγχουμε αν η διεργασία αποτελεί primary process. Στη συνέχεια ελέγχουμε αν το primary region έχει τη ζητούμενη διαθέσιμη μνήμη, και αν ισχύει, τότε μειώνεται το primary_region_free_mem, αυξάνεται η διεύθυνση του primary_region_heap_head και επιστρέφεται η ζητούμενη διεύθυνση. Στη δεύτερη περίπτωση, με την κλήση του system call mmap ελέγχουμε εάν η διεργασία είναι primary και αν πρόκειται για anonymous mappings ώστε να προχωρήσουμε στην εκτέλεση της custom μεθόδου do_mmap_primary_region(). Η συγκεκριμένη μέθοδος, με παρόμοιο τρόπο ελέγχει τη διαθέσιμη μνήμη εντός του primary region, και αν υπάρχει επιστρέφει τη ζητούμενη διεύθυνση μειώνοντας το primary_region_mmap_tail.

4.2.4 Έλεγχος του τελικού εκτελέσιμου με το QEMU

Για να ελέγξουμε την ορθή υποστήριξη των Direct Segments από το λειτουργικό σύστημα χρειάστηκε να γράψουμε tests τα οποία κάνουν malloc() μεγάλο χώρο μνήμης και διαβάζουν/γράφουν από το χώρο που δεσμεύτηκε. Τα συγκεκριμένα microbenchmarks έγιναν compile με το binary riscv64-unknown-elf-gcc το οποίο δημιουργείται με την κατασκευή του riscv-gnu-toolchain, ώστε να είναι compatible με την αρχιτεκτονική RISC-V. Στη συνέχεια, τα αρχεία τα οποία χρειάζεται να εκτελεστούν εντός του εξομοιωμένου μηχανήματος, τα συμπεριλάβαμε εντός του overlay directory. Το overlay χρησιμοποιείται για filesystem customization, δηλαδή κατά το build του Linux image, ό,τι βρίσκεται μέσα σε αυτό το φάκελο τοποθετείται πάνω από το base filesystem. Τέλος, με την εντολή marshal -v build fedora-base.json, που παρέχεται από το FireMarshal, κατασκευάσαμε το firmware binary και το Linux kernel image τα οποία θα φορτωθούν στο εξομοιωμένο μηχάνημα.

Το FireMarshal αποτελεί εργαλείο ενσωματομένο στο Firesim, το οποίο είναι σχεδιασμένο για να αυτοματοποιεί και να απλοποιεί τη διαδικασία κατασκευής και δοκιμής λογισμικού για πλατφόρμες RISC-V. Προσφέρει διάφορες εντολές που αυτοματοποιούν ροές εργασίας, όπως την κατασκευή ενός Linux kernel image. Επίσης επιτρέπει τη διαμόρφωση λογισμικού χρησιμοποιώντας αρχεία JSON που καθορίζουν την έκδοση του πυρήνα Λινυξ, το σύστημα αρχείων ροοτ και οποιαδήποτε επιπλέον πακέτα λογισμικού. Ακόμα αυτοματοποιεί τη διαδικασία εκκίνησης του πυρήνα Linux που διευκολύνει την αποσφαλμάτωση και διασφάλιση σωστής λειτουργίας του λειτουργικού.

Χρησιμοποιώντας το πιο πάνω εργαλείο και εντάσσοντας το προσαρμοσμένο binary qemu-system-riscv64 στο PATH variable, εκτελούμε την εντολή marshal launch fedora-base.json με την οποία αρχικοποιείται το περιβάλλον προσομοίωσης και εκκινείται ο προσαρμοσμένος πυρήνας Linux χρησιμοποιώντας τον εξομοιωτή QEMU. Για να επιβεβαιώσουμε την ορθή λειτουργία του λειτουργικού συστήματος πριν το μεταφέρουμε στο Xilinx Alveo U250 για τη φόρτωσή του στο Rocket Core, τρέχουμε τα microbenchmarks που σχεδιάστηκαν να εξετάσουν τη διαχείριση των Direct Segments σε επίπεδο λογισμικού.

4.3 Μεταφορά πυρήνα Linux στο FPGA

Το τελευταίο στάδιο της υλοποίησης των Direct Segments είναι ο έλεγχος της ορθής λειτουργίας του custom πυρήνα Linux πάνω στον custom Rocket Core. Για αυτό το βήμα, χρειάστηκε να τροποποιήσουμε το config_runtime.yaml ορίζοντας τις πιο κάτω τιμές:

- default_simaulation_dir: Αποτελεί το directory όπου το Firesim θα αποθηκεύσει τα αποτελέσματα και τα τελικά αρχεία της προσομοίωσης, όπως το uartlog και τα waveforms
- default_hw_config: Αποτελεί το hardware configuration που θα χρησιμοποιηθεί για την προσομοίωση, στην περίπτωσή μας το alveo_u250_firesim_rocket_singlecore_no_nic
- workload_name: Αποτελεί το JSON αρχείο που περιέχει το workload που θα εκτελεστεί στην προσομοίωση, στην περίπτωσή μας το fedora-base.json

Επίσης προσθέσαμε στο config_hw.yaml, κάτω από το hardware configuration που χρησιμοποιούμε, το path όπου βρίσκεται το bitstream που κατασκευάσαμε στη ροή ανάπτυξης υλικού.

Τέλος, για να ετοιμάσουμε το infrastructure της προσομοίωσης τρέχουμε το firesim infrasetup που κάνει deploy όλα τα απαρίτητα στοιχεία για την εκτέλεση του workload στο Run Farm Machine και με την εντολή firesim runworkload εκκινεί η προσομοίωση. Έχοντας ολοκληρωθεί επιτυχώς το boot του πυρήνα Linux στο υλικό με υποστήριξη Direct Segments hardware εκτελούμε τα microbenchmarks για τον έλεγχο της ορθής λειτουργίας του συστήματος.

Κεφάλαιο 5

Ανάλυση επίδοσης των Direct Segments

Στο κεφάλαιο αυτό θα αναλύσουμε την επίδοση του προσαρμοσμένου Rocket Core με υποστήριξη Direct Segment σε σχέση με μετρικές των μετροπρογραμμάτων του SPEC2017. Η ανάλυση θα χωριστεί σε δύο ενότητες, αρχικά θα παρουσιαστούν τα αποτελέσματα των benchmarks στον εξομοιωτή QEMU και έπειτα τα αποτελέσματα από την εκτέλεσή τους στο FPGA.

5.1 Εργαλεία ανάλυσης επίδοσης και Performance Counters

Στο RISC-V Privileged ISA αναφέρεται το Hardware Performance Monitor Unit που αποτελείται από 31 καταχωρητές, τον mcycle που μετράει τους κύκλους, τον minstret που μετράει τις εντολές και 29 επιπλέον προγραμματίσιμους event counters. Η διαθεσιμότητα των καταχωρητών από το M-mode στο S-mode ελέγχεται από τον 32-bit καταχωρητή mcounteren και από το S-mode στο U-mode από τον 32 bit scounteren. Επίσης η ενεργοποίηση των Performance counters, ώστε να αυξάνονται κανονικά, ελέγχεται από τον καταχωρητή mcountinhibit. Για τη διάθεση των Performance Counters στον Rocket Core αρκεί να δηλώσουμε στις παραμέτρους του την τιμή nPerfCounters: Int = 29.

Η υποστήριξη των Performance Counters από το λογισμικό εκτελείται σε M-mode από το OpenSBI pmu extension[1] το οποίο δίνει τη δυνατότητα στο λειτουργικό που βρίσκεται σε S-mode να τους κάνει configure και να τους σταματά ή ξεκινά οποιαδήποτε στιγμή. Γι΄ αυτό χρειάστηκε να τροποποιήσουμε το αρχείο sbi_hart.c του firmware και το riscv_pmu_sbi.c από το directory drivers/perf. Επίσης εγκαταστήσαμε στο Linux το εργαλείο perf το οποίο παρέχει υποστήριξη για διάφορα hardware και software events όταν ετκελείται με το QEMU. Όταν εκτελείται με το προσαρμοσμένο υλικό, παρέχει αποτελέσματα μόνο για κύκλους και εντολές και έτσι προγραμματίσαμε επιπλέον τον mhpmcounter3 να μετράει τα TLB misses.

[1] https://github.com/riscv-software-src/opensbi/blob/master/docs/pmu_support.md

5.2 Μετρικές Επίδοσης

Οι μετρικές που θα χρησιμοποιηθούν στην παρούσα εργασία αφορούν την εξέταση της σουίτας μετροπρογραμμάτων SPEC2017[3] και είναι οι πιο κάτω:

- **TLB misses** : Οι συνολικές αστοχίες TLB κατά την εκτέλεση ενός προγράμματος.
- **Total cycles** : Οι συνολικοί κύκλοι επεξεργασίας κατά την εκτέλεση ενός προγράμματος.
- **Total instructions** : Οι συνολικές εντολές που εκτελέστηκαν μέχρι την ολοκλήρωση ενός προγράμματος.

Αρχικά θα εξετάσουμε την επίδοση του Rocket Core του Rocket Chip όσον αφορά την εκτέλεση των SPEC benchmarks, χωρίς όμως να χρησιμοποιήθεί το Direct Segment hardware που είναι ενσωματομένο στο υλικό. Στη συνέχεια θα επαναλάβουμε την ίδια διαδικασία, θέτωντας στο όνομα των μετροπρογραμμάτων το suffix "_ds", ώστε να χρησιμοποιήσουν τα Direct Segments. Έπειτα, θα συγκρίνουμε τις διαφορές στις μετρικές μεταξύ της εκτέλεσης χωρίς Direct Segment support και της εκτέλεσης με Direct Segment support και θα σχολιάσουμε τα αποτελέσματα. Να σημειώσουμε ότι τα TLB misses θα αναφέρονται μόνο στο Data TLB, καθώς το Instruction TLB δεν παρουσιάζει μεγάλα ποσοστά αστοχιών που να συμβάλλουν σημαντικά στην επίδοση ενός συστήματος. Θα εξετάσουμε συγκεκριμένα μετροπρογράμματα της σουίτας SPEC2017 καθώς είτε κάποια από αυτά δεν γίνονταν επιτυχώς compile με τα διαθέσιμα toolchains, είτε κάποια από αυτά εμφανίζουν αστοχίες TLB τάξης χιλιάδων.

Σύμφωνα με τα αποτελέσματα που θα πάρουμε, θα αξιολογήσουμε την επίδοση χρησιμοποιώντας τις πιο κάτω φόρμουλες:

$$Speedup = \frac{TotalCycles(DS) - TotalCycles(noDS)}{TotalCycles(noDS)} * 100$$

Αυτή η φόρμουλα παίρνει αρνητικές και θετικές τιμές, οι οποίες μεταφράζονται σε μείωση ή αύξηση των κύκλων εκτέλεσης αντίστοιχα.

$$TLBMissesImprovement = \frac{TLBMisses(DS) - TLBMisses(noDS)}{TLBMisses(noDS)} * 100$$

Αυτή η φόρμουλα παίρνει αρνητικές και θετικές τιμές, οι οποίες μεταφράζονται σε μείωση ή αύξηση στις TLB αστοχίες αντίστοιχα.

$$MPKI = \frac{TLBMisses}{TotalInstructions} * 1000$$

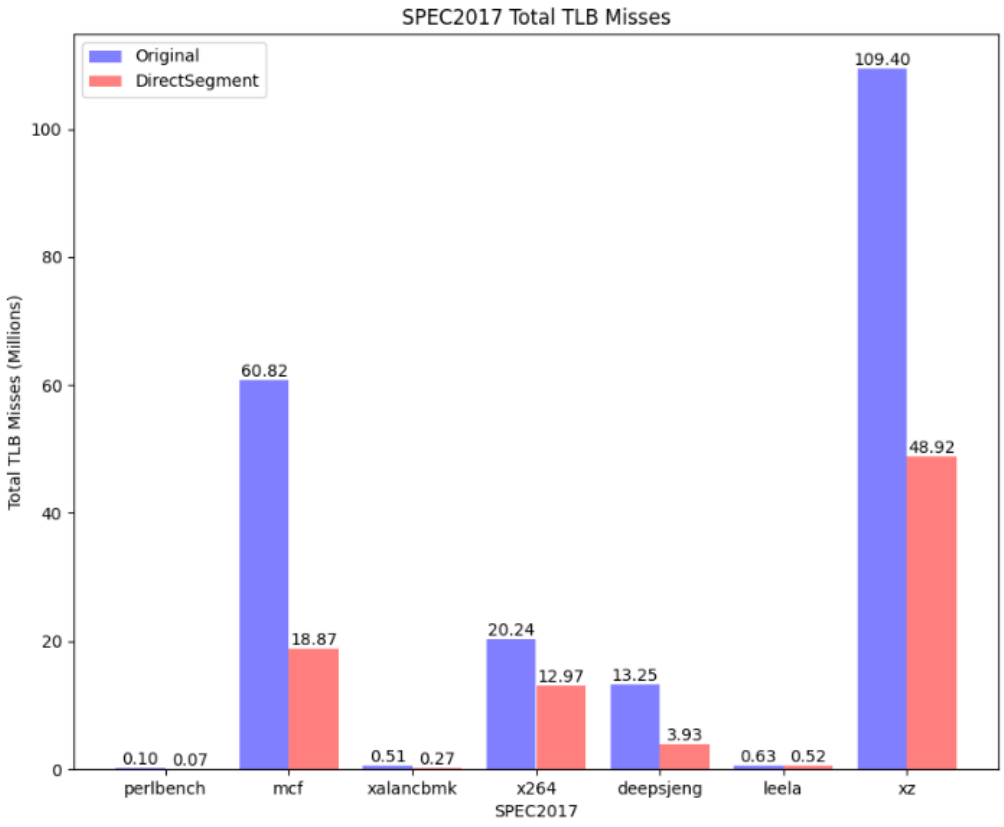
Αυτή η φόρμουλα παίρνει θετικές τιμές, με τη μικρότερη τιμή να σημαίνει και καλύτερη επίδοση (Misses per Kilo Instructions).

Σε πρώτη φάση θα παρουσιάσουμε και θα σχολιάσουμε τα αποτελέσματα των Direct Segments στον εξομοιωτή QEMU και στη συνέχεια θα παρουσιάσουμε τα αποτελέσματα από το Xilinx Alveo U250.

5.3 Ανάλυση επίδοσης στον εξομοιωτή QEMU

Τα αποτελέσματα από την εκτέλεση των μετροπρογραμμάτων πάρθηκαν με το εργαλείο perf ορίζοντας τα events cycles, instructions, dTLB-load-misses και dTLB-store-misses. Ωστόσο, το συγκεκριμένο εργαλείο παρουσιάζει προβλήματα μέτρησης των εντολών εκτέλεσης και γι΄ αυτό η ανάλυση της επίδοσης θα βασιστεί στους συνολικούς κύκλους εκτέλεσης και στο σύνολο των TLB misses.

Τα αποτελέσματα των συνολικών TLB misses μας δείχνουν ποια μετροπρογράμματα αποτελούν περισσότερο memory intensive τα οποία και αναμένουμε να έχουν το καλύτερο speedup. Από τη γραφική 5.1 φαίνεται πως το mcf και το xz έχουν τα περισσότερα tlb misses και η διαφορά μεταξύ του DS και non-DS Rocket Core είναι εμφανής.



Σχήμα 5.1: Σύγκριση των TLB misses για κάθε benchmark με και χωρίς υποστήριξη DS

Ωστόσο, βλέπουμε πως όλα τα benchmarks εμφανίζουν TLB misses improvement όπως παρατηρείται και από τον πίνακα 5.1. Μεγαλύτερη μείωση TLB αστοχιών έχουμε στο deepsjeng με τιμή $-70,3\%$ και στο mcf με -69%. Η μικρότερη μείωση παρατηρείται στο leela με τιμή $-17,5\%$. Αυτή η βελτίωση παρατηρείται διότι οποιοδήποτε malloc() γίνεται σε επίπεδο χρήστη, χαρτογραφείται στο Primary Region και η πρόσβαση σε αυτές τις διευθύνσεις γίνεται σε 1 κύκλο χωρίς να υπάρξει αστοχία.

Όσον αφορά στα αποτελέσματα των συνολικών κύκλων εκτέλεσης των SPEC2017 bench-

Benchmark	TLB Misses Improvement
perlbench	-30%
mcf	-69%
xalancbmk	-47%
x264	-35,9%
deepsjeng	-70,3%
leela	-17,5%
xz	-55,3%

Πίνακας 5.1: Το ποσοστό μείωσης των αστοχιών TLB μεταξύ DS και non-DS support

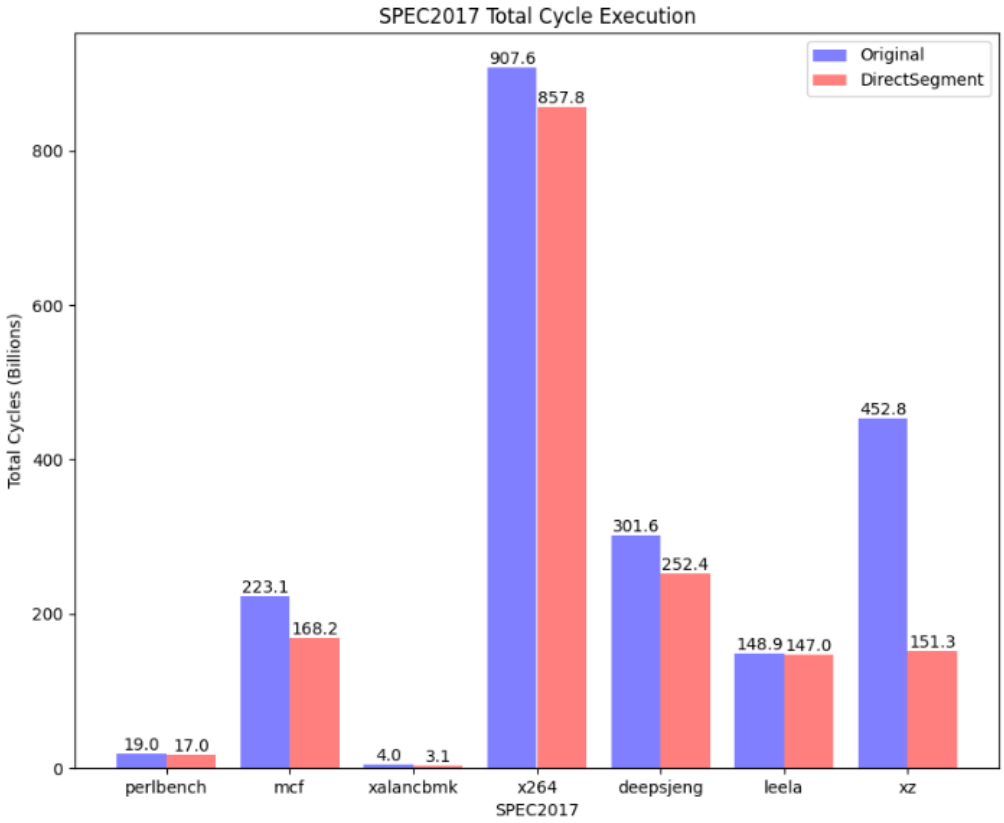
marks, σύμφωνα με τη γραφική 5.2 παρατηρούμε πως υπάρχει διαφορά μεταξύ DS και non-DS support, αλλά γίνεται εμφανής στα benchmarks που είναι πιο memory intensive, όπως αναφέραμε πιο πάνω, δηλαδή το mcf και xz. Αυτό συμβαίνει διότι οι αστοχίες TLB προσδίδουν overhead στα προγράμματα που χαρακτηρίζονται από πολλά memory accesses, και ενώ μπορεί να υπήρχε μεγάλο TLB Misses Improvement, αυτό να μην αντικατοπτρίζεται στους συνολικούς κύκλους εκτέλεσης.

Παρατηρούμε πως όλα τα benchmarks εμφανίζουν speedup με υποστήριξη DS, ακόμα και αν αυτό είναι μικρό σε ορισμένες περιπτώσεις, όπως φαίνεται από τον πίνακα 5.2. Πολύ σημαντικό speedup έχουμε στο xz benchmark το οποίο εκτελείται σχεδόν 3 φορές γρηγορότερα με μείωση $66,59\%$ στους κύκλους εκτέλεσης. Το αμέσως επόμενο καλύτερο είναι το mcf με speedup $-24,61\%$ και το deepsjeng με $-16,31\%$. Όσο για το x264, ενώ παρατηρήσαμε διαφορά σε TLB αστοχίες της τάξης των εκατομμυρίων, το speedup είναι ελάχιστο διότι από τους συνολικούς κύκλους εκτέλεσης του προγράμματος, φαίνεται πως το workload δεν είναι memory intensive και οι αστοχίες δεν πρόσδιδαν μεγάλο overhead.

Benchmark	Speedup
perlbench	-10,53%
mcf	-24,61%
xalancbmk	-22,5%
x264	-5,49%
deepsjeng	-16,31%
leela	-1,28%
xz	-66,59%

Πίνακας 5.2: Η αυξημένη επίδοση σε θέμα κύκλων εκτέλεσης μεταξύ DS και non-DS support

Σε αυτό το σημείο αξίζει να σημειώσουμε πως το mcf αναμενόταν να είχε μεγαλύτερο speedup. Ωστόσο, το συγκεκριμένο πρόγραμμα έκανε χρήση του system call mremap() το οποίο συρρικνώνει ή επεκτείνει ένα υφιστάμενο memory mapping, με μεγάλη πιθανότητα να το μετακινήσει σε καινούρια διεύθυνση. Από τη στιγμή όμως που το Primary Region αποτελεί ένα ενιαίο vma, αυτό το system call δεν ήταν δυνατόν να πραγματοποιηθεί επιτυχώς. Έτσι



Σχήμα 5.2: Σύγκριση των συνολικών κύκλων εκτέλεσης για κάθε benchmark με και χωρίς υποστήριξη DS

επιλέξαμε να μην επιτρέψουμε το system call mmap() για anonymous pages κατά τη διάρκεια εκτέλεσης του mcf για να μπορέσει να ολοκληρωθεί επιτυχώς. Επομένως μόνο το system call brk() δέσμευε μνήμη από το Primary Region, αλλά και αυτό ήταν αρκετό για να φανεί ένα σημαντικό speedup.

5.4 Ανάλυση επίδοσης στο Xilinx Alveo U250 FPGA

Ο Rocket Core στον οποίο εξετάσαμε την εκτέλεση των benchmarks έχει τα πιο κάτω χαρακτηριστικά:

- **TLB**: Full associative - 32 entries
- **L2 TLB**: 0 entries
- **DCache**: 32 KB
- **ICache**: 32 KB

Για τα αποτελέσματα της εκτέλεσης των μετροπρογραμμάτων στο υλικό, το εργαλείο perf φάνηκε να αδυνατεί να μετρήσει τα TLB misses ή να δώσει σωστές μετρήσεις για τα instructions και έτσι αποφασίσαμε να πάρουμε τα αποτελέσματα απευθείας από τους Performance

Counters. Συγκεκριμένα γράψαμε ένα script το οποίο μετρούσε τους κύκλους, εντολές και αστοχίες πριν και μετά την εκτέλεση του μετροπρογράμματος και τύπωνε τη διαφορά αυτών των τιμών.

Κατά την πρώτη εκτέλεση των SPEC2017 benchmarks στο υλικό παρουσιάστηκε ένα πρόβλημα. Το system call brk() αποτύγχανε με BADADDRESS error, γεγονός που δε μας επέτρεπε να χρησιμοποιήσουμε το heap για το Primary Region. Ως λύση σε αυτό το πρόβλημα, αποφασίσαμε να επιτρέψουμε μόνο το system call mmap() για anonymous pages να δεσμεύει μνήμη από το Primary Region, καθώς διαφορετικά δε θα μπορούσαμε να αναλύσουμε τη σωστή λειτουργία και επίδοση των Direct Segments στο Rocket Core του Rocket Chip Generator. Ωστόσο, τα microbenchmarks που γραφτήκαν και έγιναν compile με το toolchain που είναι συμβατό με το version του firesim δεν παρουσίασαν κάποιο πρόβλημα ως προς τη χρήση του system call brk(). Ένα πιθανό σενάριο για την αιτία του προβλήματος είναι ότι το toolchain με το οποίο έγιναν compile τα SPEC2017 benchmarks έχει διαφορές με αυτό που χρησιμοποιεί η τελευταία έκδοση του firesim. Ακόμη ένα σενάριο είναι ότι το heap allocation μπορεί να είναι architecture specific σε επίπεδο λογισμικού και έτσι να ήθελε περισσότερη υποστήριξη από το λειτουργικό.

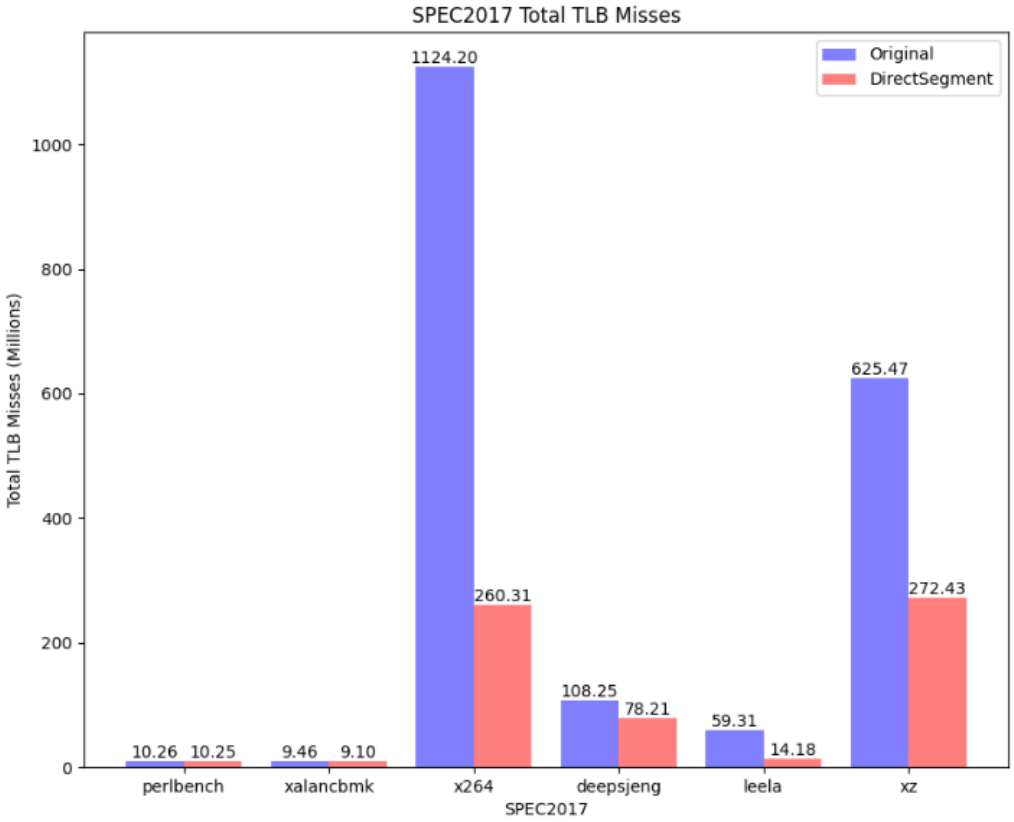
Το γεγονός ότι η εκτέλεση των μετροπρογραμμάτων έγινε χωρίς το brk() να δεσμεύει μνήμη από το Primary Region φαίνεται και από τα αποτελέσματα των TLB Misses στη γραφική 5.3.

Παρατηρούμε ότι η διαφορά των αστοχιών μεταξύ DS και non-DS Rocket Core είναι μικρότερη σε σύγκριση με τα αποτελέσματα που πήραμε από τον εξομοιωτή QEMU. Ωστόσο υπάρχει σταθερά μία μικρή αύξηση στην επίδοση του TLB ακόμα και στα μετροπρογράμματα που δεν έχουν πολλά memory accesses. Αυτό φαίνεται και στον πίνακα 5.3 όπου η μεγαλύτερη μείωση αστοχιών παρατηρείται στο x264 και leela με τιμή $-76,8\%$ και $-76,6\%$ αντίστοιχα. Στα benchmarks perlbench και xalancbmk δεν παρατηρείται ιδιαίτερη μείωση στις αστοχίες σε σχέση με τα νούμερα που είχαμε στον εξομοιωτή QEMU, γεγονός που δείχνει πως τα συγκεκριμένα μετροπρογράμματα δέσμευαν μνήμη στο Primary Region με την κλήση brk(), κάτι το οποίο στο υλικό δεν υποστηρίζεται.

Benchmark	TLB Misses Improvement
perlbench	-0.1%
xalancbmk	-3,8%
x264	-76,8%
deepsjeng	-27,8%
leela	-76,6%
xz	-56,4%

Πίνακας 5.3: Το ποσοστό μείωσης των αστοχιών TLB μεταξύ DS και non-DS Rocket Core (FPGA)

Όσον αφορά στους συνολικούς κύκλους εκτέλεσης παρατηρούμε πως τα μετροπρογράμματα που δεν είχαν ιδιαίτερη μείωση στις αστοχίες TLB παρουσιάζουν πολύ μικρές διαφορές

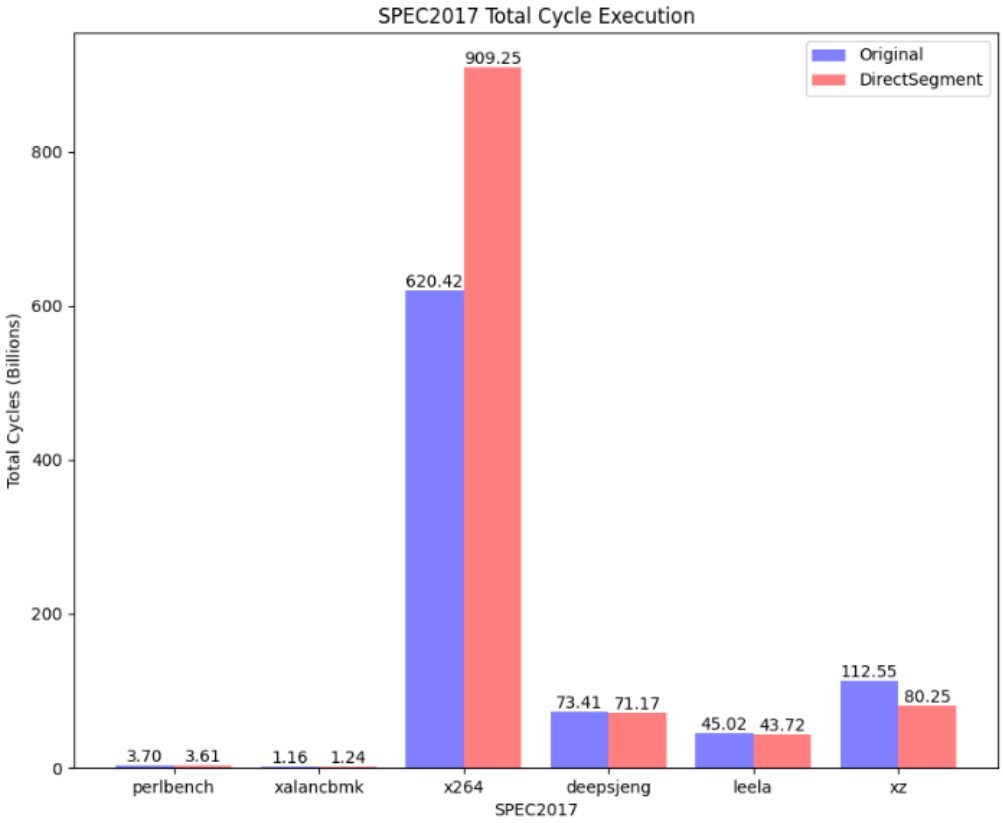


Σχήμα 5.3: Σύγκριση των TLB misses για κάθε benchmark σε DS και non-DS Rocket Core (FPGA)

μεταξύ DS και non-DS Rocket Core. Αυτό που φαίνεται να επωφελείται σημαντικά είναι το xz που παρουσιάζει και την πιο εμφανή μείωση σε συνολικούς κύκλους εκτέλεσης. Στην περίπτωση του x264 παρατηρούμε μία ασυνήθιστη συμπεριφορά καθώς οι κύκλοι εκτέλεσης αυξάνονται κατά πολύ, γεγονός που υποδεικνύει πως η διαχείριση του Primary Region από το λειτουργικό δε γίνεται σωστά.

Το μεγαλύτερο speedup το παρατηρούμε στο μετροπρόγραμμα xz το οποίο έχει μείωση $28,70\%$ και αντικατοπτρίζει τη βελτίωση που μπορεί να έχει μία memory intensive εφαρμογή με τη χρήση των Direct Segments. Τα υπόλοιπα μετροπρογράμματα δείχνουν μία μικρή αύξηση εκτός του xalancbmk το οποίο φαίνεται να μην επωφελείται από τα Direct Segments και το x264 που φαίνεται πως η επίδοσή του έπεσε σημαντικά με τη χρήση των DS. Θα αναμέναμε το mcf που επίσης αποτελεί memory intensive εφαρμογή να επιδύκνυε παρόμοια συμπεριφορά με το xz, το γεγονός όμως ότι δεν υποστηρίζεται το system call brk() στο υλικό, δεν επέτρεψε να εξετάσουμε την επίδοσή του.

Στο υλικό καταφέραμε να μετρήσουμε με επιτυχία τις συνολικές εντολές εκτέλεσης των μετροπρογραμμάτων με τον Performance Counter minstret (retired instructions). Όπως είναι αναμενόμενο οι εντολές μεταξύ DS και non-DS Rocket Core δεν είναι μεγάλες. Με τα αποτελέσματα των εντολών θα μετρήσουμε το MPKI (Misses per Kilo Instructions) για



Σχήμα 5.4: Σύγκριση των συνολικών κύκλων εκτέλεσης για κάθε benchmark σε DS και non-DS Rocket Core (FPGA)

να αξιολογήσουμε την επίδοση του σχεδιασμένου υλικού ως προς την εκτέλεση των μετροπρογραμμάτων και θα υπολογίσουμε την ποσοστιαία διαφορά, με αρνητική τιμή να σημαίνει βελτίωση. Παρατηρούμε, σύμφωνα με τον πίνακα 5.5 πως όλα τα benchmarks έχουν βελτιωμένη επίδοση σε αστοχίες TLB για κάθε 1000 εντολές εκτέλεσης. Το μόνο benchmark μου φαίνεται να έχει ελαφρώς χειρότερη επίδοση είναι το perlbench που προφανώς να μην χρησιμοποίησε σε μεγάλο βαθμό το Direct Segment.

5.5 Σύνοψη αποτελεσμάτων

Σύμφωνα με τα αποτελέσματα που είχαμε από τον εξομοιωτή QEMU παρατηρούμε πως χρησιμοποιώντας τα Direct Segments επιτυγχάνουμε σημαντική βελτίωση στο ποσοστό μείωσης των TLB αστοχιών. Συγκεκριμένα το ποσοστό κυμαίνεται από $-17,5\%$ μέχρι και $-70,3\%$. Η μείωση αυτή ωφείλεται στο ότι πλέον το brk() και mmap() δεσμεύει μνήμη από το Primary Region και η πρόσβαση στη μνήμη που δεσμεύεται δυναμικά από τα μετροπρογράμματα γίνεται σε 1 κύκλο χωρίς να υπάρχει η πιθανότητα αστοχίας. Επίσης παρατηρούμε speedup που κυμαίνεται από $-1,28\%$ εώς και $-66,59\%$ με τον Rocket Core να γίνεται αποδοτικότερος χρησιμοποιώντας τα Direct Segments, καθώς το Direct Segment hardware

Benchmark	Speedup
perlbench	-2,43%
xalancbmk	6,89%
x264	46,55%
deepsjeng	-3,051%
leela	-2,89%
xz	-28,70%

Πίνακας 5.4: Η αυξημένη επίδοση σε θέμα κύκλων εκτέλεσης μεταξύ DS και non-DS Rocket Core (FPGA)

Benchmark	MPKI Improvement
perlbench	0,77%
xalancbmk	-8,28%
x264	-87,28%
deepsjeng	-25,81%
leela	-76,17%
xz	-8,45%

Πίνακας 5.5: Η ποσοστιαία επίδοση του MPKI μεταξύ DS και non-DS Rocket Core (FPGA)

προσφέρει συνεχόμενο χώρο εικονικής μνήμης που δεν υπάρχει πιθανότητα αστοχίας. Τα μετροπρογράμματα που εμφανίζουν το μεγαλύτερο speedup είναι αυτά που αποτελούν περισσότερο memory intensive, όπως το xz και το mcf.

Στην περίπτωση εκτέλεσης των μετροπρογραμμάτων στο υλικό επιλέξαμε να μην επιτρέψουμε το system call brk() να δεσμεύει μνήμη από το Primary Region ώστε να ελέγξουμε την ορθή λειτουργία τους και έτσι τα αποτελέσματα δεν είναι τόσο αντιπροσοπευτικά όπως αυτά του εξομοιωτή. Συγκεκριμένα το εύρος του ποσοστού μείωσης των TLB αστοχιών κυμαίνεται από $-0,1\%$ μέχρι $-76,8\%$. Σε ότι αφορά το speedup, τα benchmarks που έδειξαν βελτίωση κυμαίνονταν από τιμες $-2,43\%$ μέχρι $-28,70\%$, με το καλύτερο να είναι το xz, όπως έδειξαν και τα αποτελέσματα του εξομοιωτή QEMU. Όσον αφορά στη μετρική MPKI παρατηρούμε πως όλα τα μετροπρογράμματα εκτός του perlbench δείχνουν σημαντική μείωση στις αστοχίες TLB ανά 1000 εντολές με τιμές που κυμαίνονται από $-8,28\%$ μέχρι $-87,28\%$.

Ωστόσο, η εκτέλεση των μετροπρογραμμάτων στο υλικό φανέρωσε πιθανά προβλήματα στη διαχείριση των Direct Segments από το λειτουργικό, καθώς το brk() system call οδηγούσε σε σφάλμα με αποτέλεσμα να μην μπορεί να χρησιμοποιηθεί. Επίσης, πιθανό πρόβλημα στον Chisel κώδικα του TLB μπορεί να αποτέλεσε την αιτία που το x264 παρουσίασε χειρότερη επίδοση στον DS Rocket Core. Είναι σημαντικό να σημειωθεί όμως, πως το συγκεκριμένο benchmark ήθελε 5-6 ώρες να ολοκληρωθεί στο υλικό, γεγονός που το καθιστά δύσκολο στην αποσφαλμάτωσή του.

Κεφάλαιο 6

Μελλοντικές Επεκτάσεις και Διορθώσεις

Στο κεφάλαιο αυτό θα παρουσιάσουμε τις διορθώσεις που χρειάζονται να γίνουν ώστε τα Direct Segments στον Rocket Chip Generator να γίνουν αποδοτικότερα, όπως και μελλοντικές επεκτάσεις που μπορούν να προσφέρουν καινούριες δυνατότητες.

6.1 Πλήρης υποστήριξη SPEC2017 στο υλικό

Η εκτέλεση των benchmarks που δίνει η σουίτα SPEC2017, εξετάστηκε αρχικά στον εξομοιωτή QEMU, όπου και λειτουργούσαν κανονικά με πλήρη υποστήριξη από τα Direct Segments. Αυτό σημαίνει ότι το λειτουργικό μπορούσε να δεσμεύσει επιτυχώς μνήμη με το system call brk και mmap. Όμως, κατά τη διάρκεια της εκτέλεσης των μετροπρογραμμάτων στο υλικό με DS Rocket Core, εμφανίστηκε σφάλμα πρόσβασης στη μνήμη που δεσμευόταν με το brk, με αποτέλεσμα τα μετροπρογράμματα να εξεταστούν με το Primary Region να δέχεται μνήμη που δεσμεύεται μόνο από το mmap.

Πιθανές διορθώσεις για την εξάλειψη αυτού του προβλήματος θα ήταν αρχικά τα SPEC Benchmarks να γίνουν recompile με τις τελευταίες βιβλιοθήκες, καθώς οι τεχνολογίες που χρησιμοποιήθηκαν στην παρούσα διπλωματική είναι νεότερες και μπορεί να υπάρχουν incompatibility issues. Επίσης η καλύτερη διαχείριση των Primary Region από το λειτουργικό, προσαρμοσμένη στο memory layout και memory attributes του υλικού που χρησιμοποιήσαμε θα ήταν μία πιθανή λύση, καθώς το πρόβλημα μπορεί να έγκειται σε μεθόδους του λογισμικού που είναι architecture specific.

Η πλήρης υποστήριξη και εκτέλεση των SPEC2017 Benchmarks στο υλικό θα μας έδινε μία καλύτερη εικόνα της επίδοσης του DS Rocket Core, πολύ πιο κοντά σε αυτή που ανέδειξε ο εξομοιωτής QEMU.

6.2 Αποσφαλμάτωση του x264 benchmark

Το γεγονός ότι το x264 μετροπρόγραμμα είχε την καλύτερη επίδοση σε TLB Misses Improvement αλλά τη χειρότερη επίδοση σε speedup, χρειάζοντας περισσότερους κύκλους για να εκτελεστεί σε DS Rocket Core μπορεί να εμφανίσει πιθανά προβλήματα σε edge cases της σχεδίασης σε επίπεδο λειτουργικού - υλικού. Η μέγαλη διάρκεια όμως εκτέλεσης του στο υλικό (5-6 ώρες) καθιστά ιδιαίτερα χρονοβόρα την αποσφαλμάτωση του στο χρονικό διάστημα περάτωσης της παρούσας διπλωματικής.

Σε μελλοντική δουλειά, θα ήταν ιδιαίτερα σημαντικό να αναλυθεί η εκτέλεση του συγκεκριμένου benchmark, πιθανώς με περισσότερα εργαλεία και σε υλικό που μπορεί να πετύχει μεγαλύτερες ταχύτητες ώστε να προκύψουν προβλήματα που μπορεί να εμπεριέχονται στην ανάπτυξη λογισμικού ή υλικού.

6.3 Εξερεύνηση χώρου σχεδίασης

Η παρούσα διπλωματική εξερεύνησε τη χρήση των Direct Segments μόνο στο DTLB, καθώς εκεί είναι που γίνονται οι περισσότερες αστοχίες σε ένα σύστημα. Ωστόσο καλό θα ήταν η εξερεύνηση του πώς θα μπορούσε να βελτιώσει την επίδοση ενός συστήματος η χρήση των Direct Segments και για το ITLB.

Επίσης, η επέκταση των Direct Segments ώστε να λειτουργούν αποτελεσματικά σε περιβάλλον πολλαπλών επεξεργαστών θα μπορούσε να βελτιώσει σημαντικά την επίδοση ενός συστήματος. Η έρευνα θα μπορούσε να επικεντρωθεί στον τρόπο με τον οποίο τα Direct Segments μπορούν να μοιραστούν ή να διανεμηθούν μεταξύ πολλών επεξεργαστών και πώς να διαχειριστεί η συνέπεια και ο συγχρονισμός.

Αυτό οδηγεί και στην εξερεύνηση πολλαπλών set DS καταχωρητών οι οποίοι θα αντιστοιχούν σε διαφορετικά κομμάτια μνήμης όπου θα μπορούσε μία διεργασία ή και πολλαπλές διεργασίες να χρησιμοποιούν για να δεσμεύουν συνεχόμενα κομμάτια εικονικής μνήμης σε συνεχόμενα κομμάτια φυσικής μνήμης.

Όσον αφορά στην ανάπτυξη λογισμικού, θα μπορούσαν να υλοποιηθούν προσαρμοστικοί αλγορίθμοι που θα έδιναν τη δυνατότητα σε ένα σύστημα να αποφασίζει δυναμικά πότε θα χρησιμοποιηθούν τα Direct Segments. Αυτή η απόφαση θα γινόταν βάσει μοτίβων φόρτου εργασίας και θα μπορούσε να βελτιστοποιήσει τη χρήση τους, αφού η μνήμη θα δεσμευόταν στο Primary Region μόνο όταν θα οδηγούσε σε καλύτερη επίδοση.

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] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D Hill και Michael M Swift. Efficient virtual memory for big memory servers. *ACM SIGARCH Computer Architecture News*, 41(3):237–248, 2013.

[3] James Bucek, Klaus Dieter Lange και Jóakimv. Kistowski. Spec cpu2017: Next-generation compute benchmark. Στο *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, σελίδες 41–42, 2018.

[4] Andrew Waterman and Krste Asanovic and SiFive Inc. The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA, Document Version 20191213. RISC-V Foundation, 2019.

[5] Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolic, Krste Asanovic και Parthasarathy Ranganathan. A hardware accelerator for protocol buffers. Στο *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, σελίδες 462–478, 2021.

[6] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra και others. Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud. Στο *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, σελίδες 29–42. IEEE, 2018.

[7] Nikhita Kunati και Michael M Swift. Implementation of direct segments on a risc-v processor. Στο *Proceedings of Second Workshop on Computer Architecture Research with RISC-V*, 2018.

[8] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović και Dawn Song. Keystone: An open framework for architecting trusted execution environments. Στο *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[9] Nikolaos Charalampos Papadopoulos, Vasileios Karakostas, Konstantinos Nikas, Nectarios Koziris και Dionisios N. Pnevmatikatos. A configurable tlb hierarchy for the risc-v architecture. Στο *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, σελίδες 85–90, 2020.

[10] Berkeley Architecture Research. Chipyard Documentation, 2019.

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

[12] AS Suryavanshi και Sanjeevkumar Sharma. An approach towards improvement of contiguous memory allocation linux kernel: A review. *Indones. J. Electr. Eng. Comput. Sci*, 25:1607, 2022.

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

[14] Wikipedia contributors. Field-programmable gate array — Wikipedia, the free encyclopedia, 2023.

[15] Wikipedia contributors. Instruction set architecture — Wikipedia, the free encyclopedia, 2023.

[16] Wikipedia contributors. Reduced instruction set computer — Wikipedia, the free encyclopedia, 2023.

[17] Jason Zhijingcheng Yu, Shweta Shinde, Trevor E Carlson και Prateek Saxena. Elasticlave: An efficient memory model for enclaves. Στο *31st USENIX Security Symposium (USENIX Security 22)*, σελίδες 4111–4128, 2022.

Συντομογραφίες - Αρκτικόλεξα - - Ακρωνύμια

BBL	Berkeley Boot Loader
CLB	Configurable Logic Block
CMA	Contiguous Memory Allocator
DS	Direct Segments
FF	Flip-Flop
FPGA	Field Programmable Gate Array
HLS	High Level Synthesis
HTIF	Host-Target Interface
ISA	Instruction Set Architecture
LUT	Lookup Table
MMU	Memory Management Unit
pk	Proxy Kernel
PL	Programmable Logic
PMA	Physical Memory Attributes
PMP	Physical Memory Protection
PPN	Physical Page Number
PTE	Page Table Entry
PTW	Page Table Walk
RAM	Random Access Memory
RISCV	Reduced Instruction Set Computer
RTL	Register Transfer Level
SoC	System-on-Chip
TLB	Translation Lookaside Buffer
VPN	Virtual Page Number