Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων & Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Fast Packet Classification on FPGA using RISC-V and binary search acceleration

Διπλωματικη Εργασια

Αρσινόη Γ. Πνευματικού

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής ΕΜΠ

Αθήνα, (Μάρτιος, 2020)

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων & Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Fast Packet Classification on FPGA using RISC-V and binary search acceleration

Διπλωματικη Εργασια

Αρσινόη Γ. Πνευματικού

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την κάτωθι τριμελή επιτροπή την ημερομηνία $23^η$ Μαρτίου 2020.

Δημήτριος Σούντρης	Παναγιώτης Τσανάκας	Διονύσιος Πνευματικάτος
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αρσινόη Γ. Πνευματικού
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

Περίληψη

Η ραγδαία εξέλιξη της τεχνολογίας, οι υψηλές ταχύτητες μετάδοσης της πληροφορίας, οι ανάγκες για υψηλή απόδοση και ο ανταγωνισμός έχουν οδηγήσει στην έρευνα για δημιουργία μοντέρνων επεξεργαστών δικτύου, με σκοπό να ικανοποιήσουν τις αυξημένες απαιτήσεις σε throughput και latency, προσπαθώντας να κρατήσουν όσο γίνεται περισσότερο την ευελιξία μιας CPU γενικού σκοπού. Παράλληλα, η δημιουργία ενός open-source ISA επεξεργαστή, του RISC-V, δίνει τη δυνατότητα σε μεγαλύτερο μέρος της ερευνητικής κοινότητας να ασχοληθεί με θέματα αρχιτεκτονικής και σχεδίασης.

Συνδυάζοντας αυτά τα δύο, στη συγκεκριμένη διπλωματική εργασία θα προσπαθήσουμε να εκτελέσουμε λειτουργίες δικτύου σε σύστημα το οποίο συνδιάζει τον RISC-V με VHDL accelerators ειδικού σκοπού, δημιουργώντας έτσι μία HW/SW co-processing αρχιτεκτονική σχεδίαση. Για το σκοπό αυτό θα προσεγγίσουμε τη λειτουργία ενός Network processor χρησιμοποιώντας μία από τις πιο περίπλοκες και αντιπροσωπευτικές λειτουργίες δικτύου, το Packet Classification, και συγκεκριμένα τον αλγόριθμο HyperSplit, η εκτέλεση της δυαδικής αναζήτησης του οποίου θα είναι και η μετρική της απόδοσης του συστήματός μας. Θα επεκτείνουμε το instruction subset RV64IAC ISA του RISC-V προσθέτοντας μία νέα εντολή που θα αντιπροσωπεύει τη HyperSplit δυαδική αναζήτηση. Για τη σχεδίαση του hardware θα χρησιμοποιήσουμε τον Rocket Chip Generator, δημιουργώντας ένα configuration με έναν RISC-V core και την επιπλέον δυνατότητα επικοινωνίας με hardware accelerators. Θα δημιουργήσουμε έναν HyperSplit search hardware accelerator σε VHDL και θα χρησιμοποιήσουμε το RoCC interface για την επικοινωνία του με τον RISC-V main core.

Υλοποιώντας τον accelerator της HyperSplit δυαδικής αναζήτησης σε ένα Xilinx Ultrascale xcku060 FPGA, αυτός καταναλώνει 5K LUT, 2K DFF, 0 DSP and 570 RAMB (53% του FPGA) πόρους για την αποθήκευση μέχρι και 381K κόμβους του δέντρου. Χρησιμοποιώντας pipeline με P=5 stages ο accelerator πιάνει συχνότητα f_{clk}=227MHz. Αντίστοιχα, ο RISC-V στο ίδιο FPGA καταναλώνει 15K LUT, 7K DFF, 0 DSP και 5 RAMB με f_{clk}=143MHz.

Η σχεδίασή μας τελικά επιτυχαίνει 113 φορές πιο γρήγορο classification από τον RISC-V χωρίς τον accelerator, υποστηρίζοντας με τους υπολογισμούς μας throughput έως και 25.4 εκ. packets/sec (ή ρούτερς με 8.1Gbps traffic).

Λέξεις-κλειδιά: RISC-V, Network Processors, FPGA, HW/SW Co-design, Packet Classification, Soft-Processor, HyperSplit, Hardware Accelerator, Binary Search.

Abstract

Performance demands in communications technology is driving research towards advanced network processors, which are able to handle huge rates of incoming packets via application-specific circuits, however, without sacrificing all of the conventional CPU flexibility. At the same time, the advent of RISC-V, an open-source ISA processor developed at UC Berkeley, is disrupting the industry and academia by opening computer architecture to a broader research community, allowing more researchers to explore architectural and implementation issues.

Combining the above, the current work considers placing dedicated VHDL accelerators next to a RISC-V processor to accommodate network functions via customized HW/SW co-processing. We deal with the most common and challenging network task, that of Packet Classification, using the HyperSplit algorithm for performance measurements in our research. We extend the instruction subset RV64IAC of the RISC-V ISA with a new instruction that corresponds to the HyperSplit binary search tree operation. For our hardware design we use the rocket chip generator, creating a configuration that includes one RV64IAC RISC-V core and the ability for core-accelerator communication. We create a VHDL HyperSplit search hardware accelerator and we connect it to the main rocket chip RISC-V core using the RoCC interface.

For rapid prototyping and design exploration, we implement the binary search of HyperSplit algorithm on an Xilinx Ultrascale xcku060 FPGA. Our VHDL accelerator consumes 5K LUT, 2K DFF, 0 DSP and 570 RAMB (53% of FPGA) for storing data structures with up to 381K nodes. By tuning our pipeline, we achieved f_{clk}= 227MHz for P=5 pipeline stages. Our RISC-V utilizes 15K LUT, 7K DFF, 0 DSP and 5 RAMB and operates at 143MHz.

Adding our new accelerator, the design accelerates Packet Classification task, achieving 113x faster classification than RISC-V alone, sustaining up to 25.4M packets/sec throughput (e.g., supporting routers with 8.1Gbps traffic).

Keywords: RISC-V, Network Processors, FPGA, HW/SW Co-design, Packet Classification, Soft-Processor, HyperSplit, Hardware Accelerator, Binary Search.

Ευχαριστίες

Η εκπόνηση της Διπλωματικής μου εργασίας σηματοδοτεί το τέλος των προπτυχιακών σπουδών μου στη Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου. Πραγματοποιήθηκε στα πλαίσια του εργαστηρίου Μικροϋπολογιστών και Ψηφιακών Συστημάτων VLSI του τομέα Τεχνολογίας Πληροφορικής και Υπολογιστών, με επιβλέποντα καθηγητή τον κ. Δημήτριο Σούντρη, σε συνεργασία με το τμήμα SSD (System Software Development) του RnD της Intracom Telecom, με επιβλέποντα τον κ. Νίκο Κόκκαλη.

Θα ήθελα πρωτίστως να ευχαριστήσω τον καθηγητή μου, Δημήτριο Σούντρη, για την ευκαιρία που μου έδωσε να ασχοληθώ με το συγκεκριμένο επιστημονικό πεδίο, αλλά και τον μέντορά μου στην Intracom Telecom, Νίκο Κόκαλη, για την πρωτοποριακή του ιδέα και την σημαντική βοήθειά του κατά τη διάρκεια της προσπάθειάς μου.

Ιδιαιτέρως θα ήθελα να ευχαριστήσω τον κ. Γιώργο Λεντάρη του εργαστηρίου Μικροϋπολογιστών και Ψηφιακών Συστημάτων VLSI για την πολύτιμη βοήθειά του, τις ιδέες, τις συμβουλές και κατευθύνσεις του, με την βοήθεια του οποίου φτάσαμε στο επιθυμητό αποτέλεσμα και η συγκεκριμένη διπλωματική έγινε πραγματικότητα και προχώρησε σε δημοσίευση.

Θα ήθελα επίσης να ευχαριστήσω τον καθηγητή Παναγιώτη Τσανάκα και τον καθηγητή Διονύσιο Πνευματικάτο που συμπλήρωσαν την τριμελή επιτροπή.

Θα αποτελούσε παράλειψη να μην ευχαριστήσω όλους τους φίλους μου που με την στήριξή τους με βοηθούν όλα αυτά τα χρόνια να πετύχω τους στόχους μου, τόσο συνεργαζόμενη μαζί τους στα πλαίσια της σχολής όσο και σε ψυχολογικό επίπεδο.

Τέλος, ευχαριστώ βαθύτατα τους γονείς μου, καθώς και τον αδερφό μου για την αγάπη, την υπομονή και την στήριξη που μου έχουν προσφέρει όλα αυτά τα χρόνια, χωρίς τους οποίους οι περισσότεροι από τους στόχους μου δε θα γίνονταν πραγματικότητα και τους είμαι πραγματικά ευγνώμων.

Στους γονείς μου, Μαρία και Γιώργο,
Στον αδερφό μου Γιάννη,

Εκτεταμένη περίληψη

Στη συγκεκριμένη εργασία χρησιμοποιούμε μία RISC-V CPU ως soft-core σε ένα FPGA, σε συνδυασμό με ένα hardware design σε αυτό. Το design μας προσανατολίζεται για χρήση σε network processing εφαρμογές και επιτρέπει τη δημιουργία μίας **HW/SW αρχιτεκτονικής σε FPGA**, πράγμα το οποίο οδηγεί σε λύσεις με σχετικά χαμηλό κόστος σχεδίασης σε σχέση με αυτή των custom ASICs. Επίσης, η σχεδίαση αυτή εξετάζει μία αρχιτεκτονική αφήνοντας ανοιχτή την προοπτική για μελλοντικές σχεδιαστικές αλλαγές σε επίπεδο αγοράς.

Η μεγαλύτερη πρόκλησή μας είναι η προσπάθεια επιτάχυνσης λειτουργιών δικτύου, χωρίς να μειώνεται η ευκολία προσαρμοστικότητας ενός ρουτερ σε αλλαγές, και η επίτευξη σημαντικής επιτάχυνσης σε πραγματικές συνθήκες. Κεντρικό σημείο της έρευνάς μας είναι η επέκταση του **RISC-V ISA** με προσανατολισμό εφαρμογές δικτύου και η ανάπτυξη του συστήματος σε FPGA, με τις προδιαγραφές ενός **Network Processor**.

Για να μπορούμε να αγγίξουμε ένα εύρος λειτουργιών δικτύου, επικεντρωνόμαστε στο πρόβλημα του **Packet Classification**, που είναι μία από τις πιο συχνές λειτουργίες σε ένα δίκτυο. Σε κάθε πακέτο του δικτύου αντιστοιχίζονται συγκεκριμένες ενέργειες με βάση ένα σύνολο κανόνων. Έτσι, για κάθε πακέτο γίνεται μία αναζήτηση σε έναν προκαθορισμένο πίνακα από κανόνες, που λέγεται classifier, και καθορίζεται τελικά κάποια ενέργεια σε αυτό. Η διαδικασία αυτής της αντιστοίχισης, άρα και το Packet Classification, αποτελεί το bottleneck πολλών λειτουργειών (filtering, QoS handling κτλ) καθώς η κίνηση και οι εφαρμογές σε ένα δίκτυο αυξάνονται. Στη βιβλιογραφία μπορεί να βρεθεί πληθώρα τεχνικών Packet Classification σε αλγοριθμικό επίπεδο, οδηγώντας τελικά στην επίτευξη λογαριθμικής χρονικής πολυπλοκότητας, χωρίς να προκαλείται μεγάλη αύξηση της χωρικής πολυπλοκότητας. Παρόλο που το μεγαλύτερο ποσοστό της βιβλιογραφίας ασχολείται με την αντιμετώπιση του προβλήματος από την πλευρά του Software, υπάρχουν και Hardware λύσεις, η πλειονότητα, όμως, των οποίων βασίζεται στην χρήση μνημών CAM. Η χρήση CAM έχει σημαντικό κυκλωματικό κόστος, υψηλή κατανάλωση, δυσκολία προσαρμογής σε πολλά είδη αλγορίθμων και χαμηλό scalability. Σε αντίθεση με τα παραπάνω, η λύση που προτείνεται στη συγκεκριμένη έρευνα συνδυάζει SW και HW components, αποφεύγοντας τη χρήση CAM.

Για τους σκοπούς της εργασίας, και έχοντας επιλέξει το πρόβλημα του Packet Classification, επιλέγουμε να χρησιμοποιήσουμε και να βελτιώσουμε τον πολύ αποδοτικό Packet Classification αλγόριθμο **HyperSplit**. Ο αλγόριθμος αυτός είναι ένας decision-tree αλγόριθμος και συνδυάζει με αποδοτικό τρόπο τους αλγορίθμους HSM και HiCuts, που είναι αντιπροσωπευτικοί στις δύο μεγαλύτερες κατηγορίες Packet Classification αλγορίθμων. Ο αλγόριθμος αυτός χωρίζεται σε δύο στάδια. Το πρώτο στάδιο αφορά την κατασκευή ενός δυαδικού δέντρου που δημιουργείται με έξυπνο τρόπο με βάση τον δοσμένο πίνακα των Packet Classification Rules. Αυτό το δέντρο αντικατοπτρίζει το search space του προβλήματος και για κάθε εισερχόμενο πακέτο θα γίνεται μία αναζήτηση σε αυτό, με σκοπό την εύρεση της ενέργειας που αντιστοιχίζεται στο συγκεκριμένο πακέτο. Αυτή η αναζήτηση αποτελεί και το δεύτερο στάδιο του αλγορίθμου.

Με δεδομένα όλα τα παραπάνω, τα στοιχεία συνδυάζονται ως εξής: Χρησιμοποιώντας το open-source RISC-V toolchain, χτίζουμε έναν RISC-V core (επιλέγοντας το RV64IAC subset του ISA), προοριζόμενο για τον προγραμματισμό του πάνω σε FPGA. Το toolchain παράγει τον κώδικα του ζητούμενου RISC-V σε hardware description language. Στον παραγόμενο επεξεργαστή γίνεται και μία βασική εκτέλεση του αλγορίθμου HyperSplit. Ο αλγόριθμος έχει αναπτυχθεί σε γλώσσα C, έχει γίνει compile με τον cross-compiler του RISC-V toolchain και έχει προσομοιωθεί η εκτέλεσή του στον RISC-V τόσο μέσω του Spike simulator, ενός software simulator που περιέχεται στο toolchain, όσο και μέσω του Verilator, ενός clock accurate simulator που χρησιμοποιεί την παραγόμενη verilog του συστήματος για την προσομοίωση. Με τον τρόπο αυτό έχουμε μία βασική μέτρηση της απόδοσης του αλγορίθμου στον RISC-V, συλλέγοντας τα αποτελέσματα βασικών Packet Classification Benchmarks.

Χρησιμοποιώντας τη δυνατότητα του RISC-V για επέκταση του βασικού ISA, και με αυτόν τον τρόπο τη δυνατότητα σύνδεσής του με έναν ή περισσότερους hardware accelerators, εξετάζουμε την πιθανή βελτίωση της απόδοσης του αλγορίθμου HyperSplit, δημιουργώντας έναν hardware accelerator υπεύθυνο να εκτελεί το δεύτερο κομμάτι του αλγορίθμου, αυτό της αναζήτησης. Έτσι, δημιουργούμε μία νέα εντολή στο ISA, η οποία ονομάζεται hypersplit_search, την εισάγουμε στον RISC-V cross-compiler και assembler, αντικαθιστούμε το κομμάτι της αναζήτησης του αλγόρίθμου με αυτή τη νέα εντολή με τα ορίσματα της αναζήτησης και τέλος μεταγλωττίζουμε και παράγουμε το νέο HyperSplit εκτελέσιμο. Από την πλευρά του hardware στον επεξεργαστή, έχουμε επιλέξει τη χρήση του RoCC interface για την επικοινωνία του RISC-V με τον νέο accelerator (ένα έτοιμο ενσωματωμένο interface που περιέχεται και στον κώδικα του toolchain), και παράγουμε ξανά τη νέα Verilog του συστήματος. Επομένως, με αυτόν τον τρόπο έχουμε παράξει όλα τα στοιχεία εκτός από τον ίδιο τον accelerator.

Έχοντας παράξει τη Verilog του συστήματός μας και στις δύο περιπτώσεις, έχουμε την απόδοση του αλγορίθμου στον RISC-V και την απόδοση του αλγορίθμου στον, συνδενεμένο με έναν ¨κενό' accelerator μέσω του RoCC interface, RISC-V. Το μόνο που μένει, λοιπόν, για τη μέτρηση της απόδοσης του αλγορίθμου στο νέο συστήμα με τον accelerator, είναι η μέτρηση της απόδοσης της αναζήτησης σε ένα δικής μας σχεδίασης κύκλωμα, το οποίο είναι ο accelerator που θα συνδέσουμε στο προηγουμένως παραγόμενο κύκλωμα. Έτσι, δημιουργούμε σε VHDL το ζητούμενο κύκλωμα, και μετράμε την απόδοση του για την αναζήτηση στα παραγόμενα δέντρα των benchmarks.

Μετά από διάφορες προσομοιώσεις και βελτιστοποιήσεις στο κύκλωμα του accelerator και με δεδομένα τα εξής:

- χρήση του Vivado της Xilinx τόσο για τη σχεδίαση και τις μετρήσεις του κυκλώματος του accelerator, όσο και για την εισαγωγή και τις μετρήσεις της παραγόμενης Verilog του υπόλοιπου συστήματος
- επιλογή του ***xcku060-ffva1156-3-e*** product part της οικογένειας ***Kintex UltraScale***.

μετράμε τα εξής:

- το σύστημα του RISC-V χωρίς τον accelerator στο παραπάνω FPGA πετυχαίνει **συχνότητα ίση με 143MHz**
- το τελικό κύκλωμα του accelerator στο ίδιο FPGA πετυχαίνει **συχνότητα ίση με 227MHz**
- το σύστημα συνολικά χρησιμοποιεί το 5.7% των LUTs του FPGA, το 1.4% των FFs, το 53.2% των BRAMs και 0% DSPs
- ο μέσος χρόνος εντέλεσης των benchmarks στο σύστημα με τον accelerator είναι 113 φορές καλύτερος από τον αντίστοιχο της εκτέλεσης στον RISC-V χωρίς τον accelerator

Τα συμπεράσματα που προκύπτουν είναι:

- είναι εφικτό σενάριο αρχιτεκτονικής για network processing ο συνδυασμός του RISC-V με hardware accelerators ειδικού σκοπού
- ο RISC-V αποτελεί πολύ καλή ερευνητική επιλογή σε πειράματα σε hardware, έχοντας open-source ISA καθώς και εργαλεία για σχεδίαση και υλοποίηση
- η προτεινόμενη αρχιτεκτονική μπορεί να χρησιμοποιηθεί σε ρούτερς μέχρι και 10Gbps. Μπορεί να επιτευχθεί ακόμη μεγαλύτερη επιτάχυνση αν το κύκλωμα υλοποιηθεί σε ASIC.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Network processors, RISC-V and FPGAs

Modern telecommunications and the advent of 5G technologies rely on specialized network processors to meet the increasing demands for throughput and latency optimization. The routers are required to support a wide variety of network applications and forward packets with ever higher efficiency, both in terms of speed and power. The latter becomes paramount in edge and IOT setups, where processing is additionally constrained by energy availability. To this end, the research community focuses on designing novel HW architectures to improve on the performance of the most critical network functions, such as packet filtering, access control, quality of service differentiation & load balancing, policy routing, accounting & billing, traffic rate limiting, traffic shaping and MPLS switching.

When considering general purpose HW, the RISC-V from UC Berkeley [1] is a free and open ISA, gaining ground with a multitude of academic and commercial uses. It enables SW development via free tools for compilation and simulation. Additionally, it allows HW developers to modify the ISA and/or implement only a subset of the CPU functionality, e.g., to omit costly instructions when they are irrelevant to an application field.

A RISC-V CPU can be placed as a soft-core in an FPGA[2] chip, next to any other HDL design. Thus, fine-tuning a RISC-V core for network processing and implementing HW/SW architectures on FPGA allows for rapid prototyping bespoke solutions with relatively limited development costs, i.e., much lower than designing custom ASIC or employing proprietary ISA tools/chips. Moreover, it facilitates exploration/research at architecture level and adaptation to future market changes.

1.2 Thesis Motivation & Contributions

The main challenge that we are trying to overcome in the current work is to provide HW acceleration for network functions without hindering the overall SW flexibility of a router. Our main research considers RISC-V on FPGA with custom ISA extensions for accelerating certain network functions.

To accommodate multiple such functions, we focus on their very common *Packet Classification* task. In most traffic scenarios, in order to decide actions on packets, the router creates and utilizes a table of rules named the *classifier*. For every incoming packet, the router executes a procedure to search the classifier and match the packet's features to a predetermined rule that defines the pending action. This matching operation, i.e., Packet Classification, becomes a bottleneck for many of the functions mentioned above (filtering, QoS handling, etc.) when traffic and network applications grow in numbers. The complexity increases due to the amount of rules added in the table, especially when rules belong to multiple match types, which significantly expand the search space.

Hitherto published works tackle Packet Classification at algorithmic level by already employing a plethora of techniques [3]. Literature begins with linear search having SPACE/-TIME complexity proportional to the amount of rules, while it progresses to decision trees, hash tables, divide & conquer algorithms, or even heuristics, which altogether strive to decrease TIME to logarithmic complexity without increasing SPACE to a highly polynomial complexity. At implementation level, the papers provide either purely SW or purely HW solutions. The vast majority of works is entirely in SW, whereas the vast majority of ASIC/FPGA solutions rely on CAM for matching the data in a single cycle. However, the use of CAM has considerable disadvantages, such as high circuitry cost and power consumption, low suitability to various algorithms/changes (especially when the match type is not *exact*), and limited scalability.

In contrast to the aforementioned, the solution proposed in this work avoids the use of CAM and combines HW with SW components. We adopt a very efficient classification algorithm, namely *HyperSplit* [4], exploiting multiple of the literature's techniques, which we tailor to our embedded heterogeneous platform. More importantly, we develop a low-level VHDL architecture to accelerate the crucial stage of *tree-searching* of HyperSplit. In a HW/SW co-design approach, we combine a RISC-V soft-core with our own VHDL component; we tune RISC-V as a low-area/power unit and we interface our soft-core with our accelerator by introducing our custom RoCC instruction. The resulting architecture allows us to execute any scarcely invoked function on SW, e.g., *tree-building* of HyperSplit or other higher-level OVS[5] functionalities, while we continuously match the incoming packets on HW in a limited number of cycles. Implemented on a Xilinx Kintex xcku060-2 FPGA and tested with real benchmarks, our HW/SW solution achieves 113x higher throughput than the SW-only execution of RISC-V, i.e., it can process up to 25.4M packets/sec.

1.3 Thesis Organization

This document is organised in four main chapters. In chapter 2 we begin by describing the basic concept in practice, giving the reader the ability to follow the logic of our work, and we continue by analysing the main tools and algorithms we use. Chapter 3 is the main chapter of this thesis, in which we describe our work thoroughly, step-by-step. In chapter 4, we present the results on every step of our implementation and finally, in chapter 5 we evaluate the results and the performance of the proposed system, we present our conclusions, and consider scenarios for future development.

Chapter 2

Background

2.1 Network Processors

2.1.1 Definition

A network processor is an integrated circuit which has a feature set specifically targeted at the networking application domain [6]. Network processors are typically special-purpose programmable devices that are commonly used as a network architecture component to construct network systems.

In modern telecommunications networks, information is transferred as packet data, as opposed to older telecommunications networks that carried information as analog signals. The processing of these packets has resulted in the creation of integrated circuits that are optimised to deal with this form of packet data, and so, Network Processors have specific features or architectures that are provided to enhance and optimise packet processing within these networks [6]. They have developed from simple designs to complex ICs with programmable software and a variety of operations and manipulation functions on the data packet. Network processors are employed in the manufacturing of: Routers and network switches, packet inspection, session controllers, firewalls, network monitoring systems, intrusion detection and prevention devices and error detection.

The main tasks and services performed by a Network Processor are:

- **Packet Classification/Filtering** (*Claim/forward/drop decisions, statistics, gathering, firewalling)*
- **Network Address Translation** (*Translate between globally routable and private IP packets. Useful for IP masquerading, virtual web server etc*)
- **IP Packet Forwarding** (*Forward IP packets based on routing information*)
- **TCP connection management** (*Traffic shaping within the network to reduce congestion*)

- **TCP/IP** (*Offload TCP/IP processing from Internet/Web servers*)
- **VPN IP Security** (*Encryption (DES) and Authentication (MD5)*)
- **Duplicate Data Suppression** (*Reduce superfluous duplicate data transmission over high cost links*)
- **Data Transcoding** (*Converting a multimedia data stream from one format to another within the network*)

2.1.2 Common System architecture

The overall architecture of a generic network processor is shown in Figure 2.1 [7]. It shows the main internal components, as well as the external memory and input/output interfaces [7].

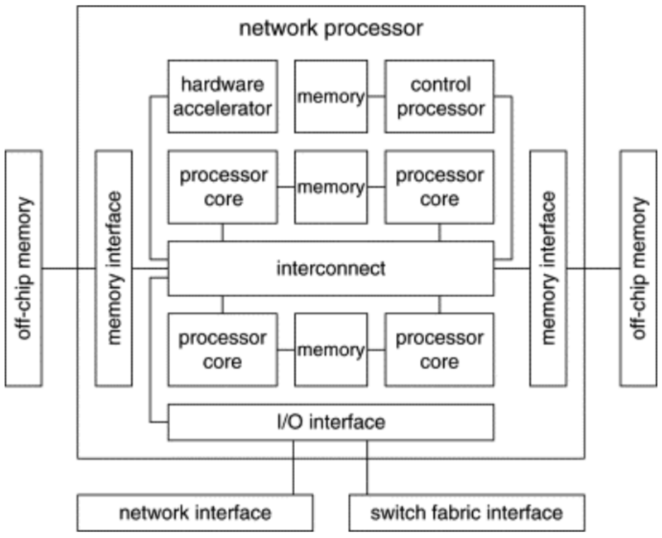


Figure 2.1: System architecture of network processor.

Although different Network Processor models may have specific architecture characteristics, their main components typically include the following [7]:

- ***Multiple processor cores for data path processing***. These processors are used for the processing of network traffic and are typically simple RISC cores, which are usually very simple and not capable of running their own operating system.
- ***Single processor core for control operations***. This processor is used for control operations and slow-path handling of packets. It is often based on an embedded RISC system that is capable enough to run a full-blown embedded operating system.
- ***On-chip memory***. On-chip memory consists of instruction and data memory for data path processors and control processors. In most cases, on-chip memory uses

SRAM technology, as a combination of DRAM and processing logic within a single MPSoC is more difficult to manufacture.

- ***Several interfaces for off-chip memories***. The amount of on-chip memory that can reasonably be included on network processors usually does not provide enough storage for packets that need to be buffered or for programs and program state. To expand the available memory space, off-chip memories are used. Interfaces to access these memories are included in the network processor chip.

- ***High-bandwidth interface for network interface(s)***. The router ports on which network processors are located typically interface with one or more physical links on one side and the router switching fabric on the other side. Because network links use a wide variety of physical layer protocols (e.g., copper wiring, optical fiber), network processors do not connect directly to the physical medium, but send network traffic to separate physical interface components. The same interface is also used to interface with the switching fabric of the router.

- ***High-bandwidth interconnect between internal components***. The various components inside the network processor (data path processor cores, control processor core, memory interfaces, input/output interface) need to be connected to allow for movement of data through the system. The bandwidth of this interconnect needs to be sufficiently high to pass network traffic through at full bandwidth as well as to accommodate memory accesses and other processing-related communication. There are various approaches on how to design such an interconnect.

- ***Specialized hardware accelerators***. Optional but very commonly used components of network processors are hardware accelerators. These blocks implement networking-specific processing tasks in custom logic and achieve much higher performance than typical software implementations. Examples of common hardware accelerators are lookup engines (using specialized logic and/or TCAM), cryptographic coprocessors, content inspection engines, etc.

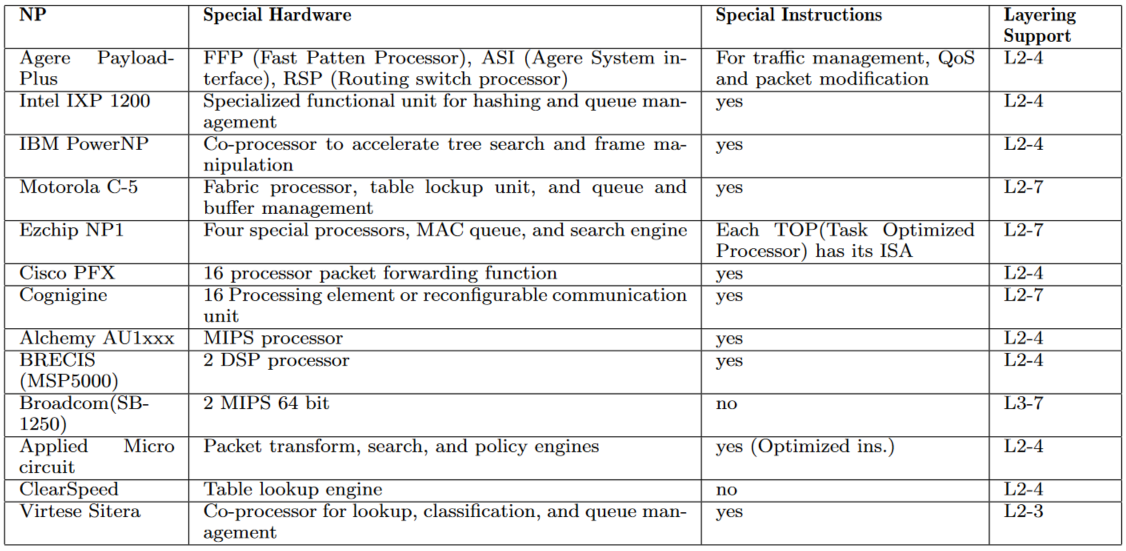
2.1.3 State-of-the-art architectures

The bandwidth growth of modern networks, the introduction of different protocols and the variety of network applications reveal the increasing demand for higher performance and flexibility of Network Processors. Those requirements have made the design of Network Processors an ongoing research and development field.

Although all of the commercial architectures are based on the elements we described in the previous section, the past recent decades, numerous techniques have been proposed, exploiting design alternatives. Some Network Processors are based on the traditional RISC-based architecture and they are trying to solve the bottleneck problems using function portioning, special instructions and cache optimization. Other approaches use special processor architecture techniques, such as modified co-processor and specific functional units to improve performance while others massively parallel architectures with modern RISC [8].

Approaching the design technically, the usage of FPGA is very common for the control plane part of a Network Processor -combined with an ASIC for the data plane part- and seems to be the future trend. Other technical choices include parameterizable hardware for multi-NPs architectures, designing quantitative evaluator software, dedicated operating systems, asynchronous pipelines, small-scale and ad-hoc networks [8].

All of the proposed architectures have a common feature, which is the fact that they use special purpose hardware. Design approaches among different commercial Network Processors are shown in the table 2.2 [8].



NP	**Special Hardware**	**Special Instructions**	**Layering Support**
Agere Payload-Plus	FFP (Fast Patten Processor), ASI (Agere System interface), RSP (Routing switch processor)	For traffic management, QoS and packet modification	L2-4
Intel IXP 1200	Specialized functional unit for hashing and queue management	yes	L2-4
IBM PowerNP	Co-processor to accelerate tree search and frame manipulation	yes	L2-4
Motorola C-5	Fabric processor, table lockup unit, and queue and buffer management	yes	L2-7
Ezchip NP1	Four special processors, MAC queue, and search engine	Each TOP(Task Optimized Processor) has its ISA	L2-7
Cisco PFX	16 processor packet forwarding function	yes	L2-4
Cognigine	16 Processing element or reconfigurable communication unit	yes	L2-7
Alchemy AU1xxx	MIPS processor	yes	L2-4
BRECIS (MSP5000)	2 DSP processor	yes	L2-4
Broadcom(SB-1250)	2 MIPS 64 bit	no	L3-7
Applied Micro circuit	Packet transform, search, and policy engines	yes (Optimized ins.)	L2-4
ClearSpeed	Table lookup engine	no	L2-4
Virtese Sitera	Co-processor for lookup, classification, and queue management	yes	L2-3

Figure 2.2: Network Processor Architectural Comparison

The architecture of a state-of-the-art, high performance Network Processor is shown in figure 2.3 [9]. It performs integrated traffic management, targeting Carrier Ethernet Switches and Routers (CESR) and other Carrier Ethernet platforms that require high performance, flexible packet processing and fine-grained traffic management [9], like Data-Centers.

Another on-edge thought is that of Smart-NICs (Smart Network Interface Cards). A SmartNIC is a network adapter that accelerates functionality and downloads it from the server (or storage) CPU. In other words, some of the Network Processor's functionality has been integrated into the network card, which can be ASIC based, FPGA based or SoC based. Using its own on-board processor, a SmartNIC may be able to perform any combination of encryption/decryption, firewall, TCP/IP and HTTP processing. SmartNICs are ideally suited for high-traffic Web servers [10].

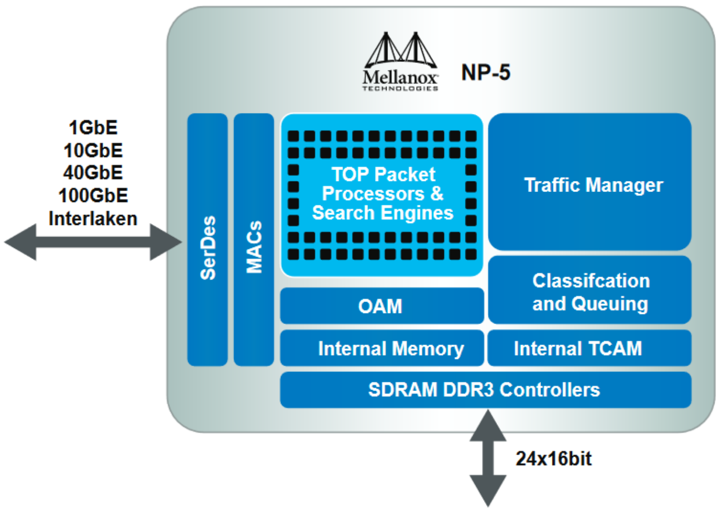


Figure 2.3: Architecture of a state-of-the-art Network Processor for Data Center Applications

In this work, we investigate an FPGA-based architecture for network functionalities, using a RISC soft-processor and adding special purpose hardware accelerators, targeting embedded systems and low power/cost designs. The results of our research can prove whether such an architecture could be the beginning of a new Network Processor design as well as whether it could compete with other existing designs in terms of cost (actual and power), resources and ease of development.

2.2 Packet Classification

2.2.1 Definition

Packet Classification is the main task of a Network Processor. Is the process of categorizing traffic into predefined classes in the network. These classes are basically buckets that map to specific traffic properties such as priority and latency for the traffic involved. Traffic is normally classified as it enters the network, where it is marked for appropriate treatment. Once the traffic has been classified and marked at the edge of the network, the network must be set up to provide differential service to the various traffic flows [11].

More specifically, the Packet Classification process matches an incoming packet to the rules of the classifier and accordingly identifies the type of action to be performed on the packet. Almost every packet in a network encounters classification at one or more stages. For example, elements such as layer-2 (switches) and layer-3 (routers), as well as special-purpose classifiers such as firewalls and load balancers, classify a packet as they forward it from the end host to the web server.

There are a number of network services that require packet classification, such as routing, routing of policy based, limiting rates, controlling access, locating virtual bandwidth, balancing loads, providing differentiated qualities of services, and billing traffics [12] [13]. In each case, it is necessary to determine which flow an arriving packet belongs to. For each arriving packet, it must be determined whether to forward or filter it (Firewall), where to forward it (Router), the class of service it should receive (QoS), or how much should be charged for transporting it (Traffic Billing). The main bottleneck of the above applications is the classification stage. A router classifies the packet to determine where to forward it and determines the QoS it should receive. A load balancer classifies the packet to identify the web server to which it must be forwarded. A firewall then classifies the packet based on its security policies to decide whether to drop it or not, based on the set of rules in the classifier. Therefore, packet classification is one of the most important processes in the design of network devices [14].

2.2.2 Existing Approaches and Algorithms

This section analyzes the difficulties in Packet Classification in practice and presents different approaches to the problem, that have been proposed until now.

Most commonly, the flow is defined through a certain field in the packet header. For example, classification of flow could depend on the IP source address value and IP destination address value, or particular transport port numbers. Otherwise flow could be simply defined by a destination prefix and range of port values. Sometimes, even the protocol type could be used to define a flow [14].

The classifier, also known as a policy database, is a collection of rules or policies. Each rule specifies a class (flow) that the arriving packet may belong to based on some criteria in its header. An action is associated with each rule in the rule set. The packet header has F fields, that could be used in the classification process. Each rule has F components which identify all possible combinations of packet headers that match the rule. Accordingly, a packet will belong to the rule if, and only if, all the fields in that packet belong to the corresponding field in the rule. The rules can have any match type, i.e., *exact, wild card, prefix, range*, and are assigned priorities [15], because a packet might satisfy more than one rules. An example classifier combining various match types is shown in Table 2.1 [16].

Rule	**src IP**	**dst IP**	**src Port**	**dst Port**	**Protcol**	**Priority**	**Action**
R1	*	*	[2,9]	[6,10]	*	1	Allow
R2	1*	0*	[0,15]	[1,4]	10	2	Allow
R3	00*	11*	123	123	*	3	Deny
R4	10*	1*	*	53	*	4	Deny
R5	0*	10*	[100,120]	2	0	5	Allow
R6	001*	1*	*	*	*	6	Deny
R7	*	00*	[0,16]	[2,8]	100	7	Deny

Table 2.1: Representative Rule Table for Packet Classification

The several solutions proposed for Packet Classification can be evaluated based on the rule table lookup time, the system resources used, as well as on the update time (insertion/deletion) of the rule table. They fall into 3 major categories: 1) *software-based*, concerned mostly with the algorithmic complexity of classification, 2) *hardware-based*, using specialized HW for searching, and 3) *hybrid*, that use a combination of the above.

The software-based solutions fall into 3 categories [12]: *Basic data structures*, *Geometry-based*, and *Heuristic*. The first focuses on the way of representing and storing the rule table in memory by using a novel data structure, which is bound to a specific searching algorithm. The Geometry-based solve the problem by constructing a geometric representation of it in a d-dimensional space, where d is the number of the fields of the rule table. The Heuristics prune the search space based on statistics/patterns found on packets or rule table, i.e., by exploiting characteristics/structure real classifiers may have.

Specifically for Geometry-based and Heuristic algorithms, we distinguish between the following approaches [17]:

- **Two-dimensional** (*Set-Pruning tries, Grid-of-Tries*) are efficient on address prefix pairs and have low time complexity but do not extend to multiple field searches.
- **Divide & Conquer** (*RFC, HSM [18], Cross-Producting, BV, ABV*) usually achieve high speed, but via considerable preprocessing time and worst case SPACE complexity.
- **Decision Tree** (*HiCuts [13], HyperCuts, D-Cuts, ExpCuts, HyperSplit*) have improved search speed and use heuristics to reduce the high memory storage requirements, but their preprocessing time is considerable.
- **Tuple space - Hash table** (*Tuple Space search, BSOL, FIS-tree*) have efficient update time, high TIME complexity, but may depend on hash tables and have non-deterministic lookups/updates.
- **Heuristics at bit time** (D^2BS) have higher performance and scalability than HiCuts and HSM, take advantage of a dynamic heuristic partition of ruleset at bit level, but have high requirements in storage and preprocessing.

Purely HW approaches, e.g., TCAMs and Bitmap-Intersection [19] [20], can achieve 1-clock cycle lookup, however via trade-offs in storage, cost and power consumption.

Regrading the hybrid category that combines algorithms and HW implementations, we mention here in summary:

- *Parallel Packet Classification (P2C)* [19] searches each field of a rule in parallel way, constructing "on-the-fly" matching. With the BART scheme this approach achieves high classification speed, having the cost and power trade-offs of TCAM and additional SRAM inclusion.
- *BV-TCAM* [21] splits the multifield packet classification in two. It uses a tree-bitmap implementation of the Bit Vector algorithm for the source and destination port lookup

and a TCAM for the lookup of the remaining header fields. Its success is due to using a TCAM of small size and combining it with an algorithm in SW.

- *DIRPE* (Database Independent Range PreEncoding) and MUD (Multi-match Using Discriminators) Algorithms [22] . These algorithms use TCAMs more efficiently and eliminate some of its weaknesses by using software. DIRPE reduces the worst-case expansion of range rules, while MUD attempts to find multiple matches for a search key.
- *Field-Split parallel Bit Vector (FSBV)* [23] performs splitting similarly to BV-TCAM, but FSBV uses TCAMs for IP classification, CAMs for protocol classification and the Bit Vector Algorithm for the remaining packet header fields. The implementation presented in [23] achieves 4x reduction in power consumption over the BV-TCAM.
- *Decision tree architecture on FPGA* [16] introduces a Packet Classification architecture on FPGA implementing a decision tree based algorithm. It optimizes memory usage and achieves 80Gbps throughput for minimum size packets. Tested with complex rules (with more than 5 fields), it sustained over 40Gbps throughput.

2.2.3 HyperSplit Algorithm

Following the literature survey and as mentioned in introduction, we opted for a HW/SW solution of Packet Classification with a decision-tree based algorithm: HyperSplit. This section describes the algorithm and reveals the reasons for selecting HyperSplit for our research.

The HyperSplit algorithm is a combination of the HSM and HiCuts algorithms in a cost-effective fashion. The algorithm [4] is divided into two procedures/stages: the *tree building* and *tree searching*. Building inputs the Rule Table and outputs a search structure in the form of a sparse binary tree. Recursively, it decomposes the theoretical search space into subspaces, which contain subsets of rules, by selecting the local-optimized rule field at each step via a weighted segmented-balanced strategy. In the end, each node stores a fixed pointer, d_n, that refers to a *dimension* of the rule table, as well as a fixed threshold value, T_n, that will drive the upcoming search towards the leaves of the tree. That is, *searching* becomes a customized binary search of each incoming packet through the created tree.

The procedure inputs a d-tuple describing the packet, e.g., {srcIP, dstIP, srcPORT, dstPORT, protocol}, and successively compares the d values to the tree nodes' data. More specifically, starting from the root, it reads the current node and compares its T_n value to the d_n-th value of the input tuple. Depending on the comparison's output, and only if the current node is internal, searching continues to the left or right child until it reaches a leaf.

By selecting the local-optimized field at each recursion stage for space decomposition and applying weighted segment-balanced strategy, the algorithm results in more efficient searching and achieves higher performance in terms of memory storage and time. In comparison with HSM and HiCuts, HyperSplit improves memory accessing by 70% and preprocessing time by 10–100x, hile it achieves greater throughput.

Given an example 2D rule table, figures 2.4 and 2.5 shows tree built by HSM, HiCuts in comparison with the one of HyperSplit [4]. For further details about the HyperSplit inner functionality advice [4].

Rule	*Priority*	*Field-X*	*Field-Y*
R1	1	[00,01]	[00,00]
R2	2	[00,01]	[00,11]
R3	3	[10,10]	[00,11]
R4	4	[11,11]	[11,11]
R5	5	[11,11]	[00,11]

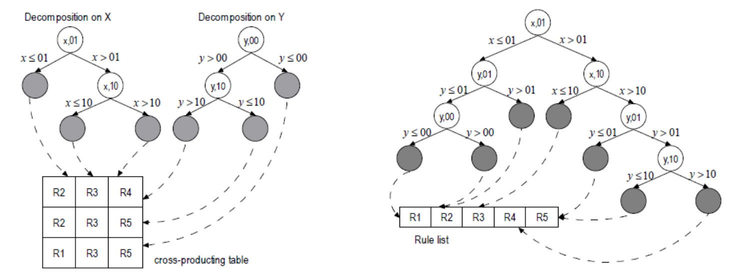


Figure 2.4: Example Rule Table, HSM search tree, HiCuts search tree

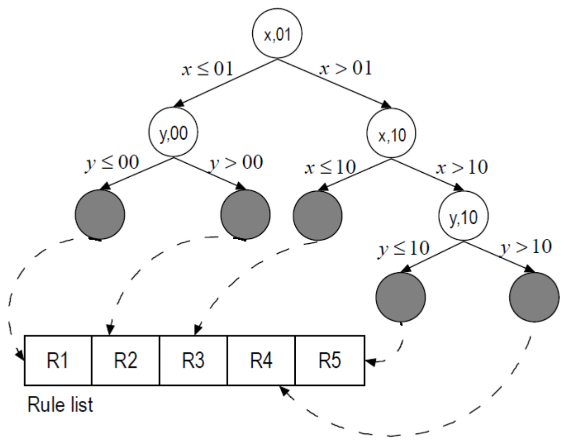


Figure 2.5: HyperSplit Search tree

2.3 RISC-V and Rocket Chip

2.3.1 RISC-V overview

RISC-V is a free and open-source ISA developed at UC Berkeley. The fact that it is open-source allows it to be used both in academic and industrial environment. It is under the governance of the RISC-V Foundation and is intended to become an industry standard [24]. It is designed to be simple and highly extensible and avoids "over-architecting" for a particular microarchitecture style (e.g., mi-crocoded, in-order, decoupled, out-of-order) or implementation technology (e.g., full-custom, ASIC, FPGA), which allows efficient implementation in any of these [1].

The RISC-V ISA consists of the base integer ISA (which must be present in any implementation) and optional extensions to this base ISA. The base integer ISAs are very similar to that of the early RISC processors except with no branch delay slots and with support for optional variable-length instruction encodings. A base is carefully restricted to a minimal set of instructions sufficient to provide a reasonable target for compilers, assemblers, linkers, and operating systems (with additional privileged operations), and so provides a convenient ISA and software toolchain "skeleton" around which more customized processor ISAs can be built [1].

2.3.2 RISC-V ISA description

The base integer instruction sets are RV32I and RV64I (named *"I"*), which provide 32-bit or 64-bit address spaces respectively and contain integer computational instructions, integer loads, integer stores, and control-flow instructions. Each base integer ISA can be extended with one or more of the standard optional instruction-set extensions defined by the Foundation. The extension named *"M"* provides integer multiplication and division, the extension named *"A"* provides standard atomic instructions, that atomically read, modify and write memory for inter-processor synchronization, the extension named *"F"* adds floating-point registers, single-precision computational instructions and single-precision loads and stores, the extension named *"D"* is the standard double-precision floating-point extension and expands the floating-point registers adding double-precision computational instructions, loads and stores and, finally, the extension named *"C"* is the compressed instruction extension that provides 16-bit forms of common instructions. The ensemble of "IMAFD" extension is indicated with "G" and the resulting ISAs are called RV32G for the 32-bit version and RV64G for the 64-bit one.

The base RV32I has 4 basic instruction formats, depending on the kind of the instruction arguments, sources and destination, and they are presented in Figure 2.6.

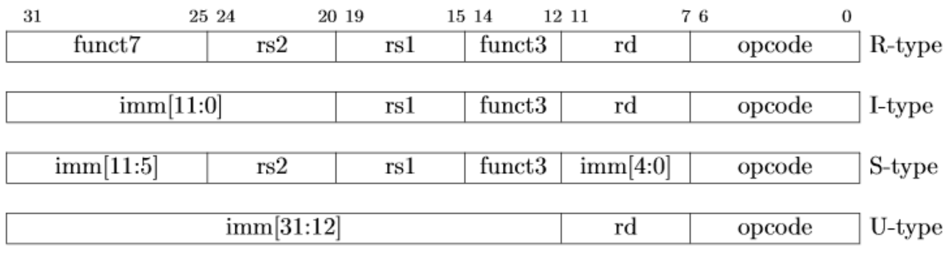


Figure 2.6: RISC-V base instruction formats.

The combination of the opcode and the funct parts of the instruction can describe its functionality exclusively. For example, the opcode dedicated for integer register-register operation is the same for all of the ADD/SLT/SLTU/AND/OR/XOR/LL/SRL instructions (as the opcode denotes the instruction operation "family"), each one of them having their own funct3 for extra specialization and identification.

Figure 2.7 presents the major opcode map for RVG [1]. Major opcodes with 3 or more lower bits set are reserved for instruction lengths greater than 32 bits. Major opcodes marked as custom-0 and custom-1 can be used by custom instruction-set extensions (custom-0 will be used for our extension).

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 ($> 32b$)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	$48b$
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	$64b$
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	$48b$
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	$\geq 80b$

Figure 2.7: RISC-V base opcode map, inst[1:0]=11.

2.3.3 Rocket Core & Rocket Chip SoC Generator

Rocket is a 5-stage in-order scalar core that implements the RV32G and RV64G ISAs. It has an MMU that supports page-based virtual memory, a non-blocking data cache, and a front-end with branch prediction. Branch prediction is configurable and provided by a branch target buffer (BTB), branch history table (BHT), and a return address stack (RAS). Rocket also supports the RISC-V machine, supervisor, and user privilege levels. A number of parameters are exposed, including the optional support of some ISA extensions (M, A, F, D), the number of floating-point pipeline stages, and the cache and TLB sizes [24]. Rocket can also be thought of as a library of processor components. Several modules originally designed for Rocket are re-used by other designs, including the functional units, caches, TLBs, the page table walker, and the privileged architecture implementation (i.e., the control and status register file). Rocket's pipeline is shown in figure 2.8.

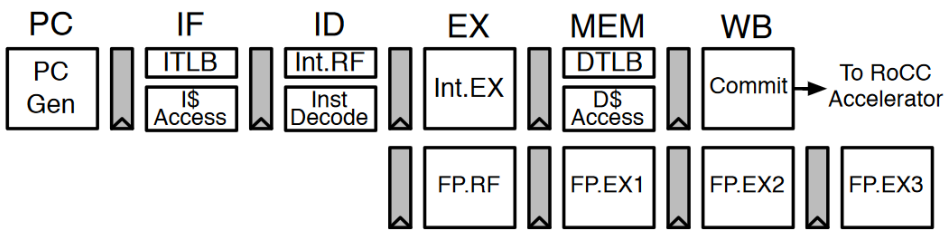


Figure 2.8: The Rocket Core Pipeline.

Rocket Chip is an open-source SoC generator developed at UC Berkeley and is based on the RISC-V ISA. Rather than being a single instance of a SoC design, Rocket Chip is a design generator, capable of producing many design instances from a single high-level source. Its extensive parameterization makes it flexible, enabling easy customization for a particular application. By changing a single configuration, a user can generate SoCs ranging insize from embedded microcontrollers to multi-core server chips [24].

Rocket Chip is implemented in Chisel [25], an open-source hardware construction language embedded in Scala, which generates synthesizable Verilog code, compatible for FPGA and ASIC design tools. Chisel can also generate a fast, cycle-accurate RTL simulator implemented in C++, which is functionally equivalent to but significantly faster than commercial Verilog simulators and can be used to simulate an entire Rocket Chip instance [24].

Figure 2.9 shows an example instance of the chip consisting of common sub-components (core generator, cache generator, RoCC-compatible coprocessor generator, tile generator, tile link generator and peripherals) [24].

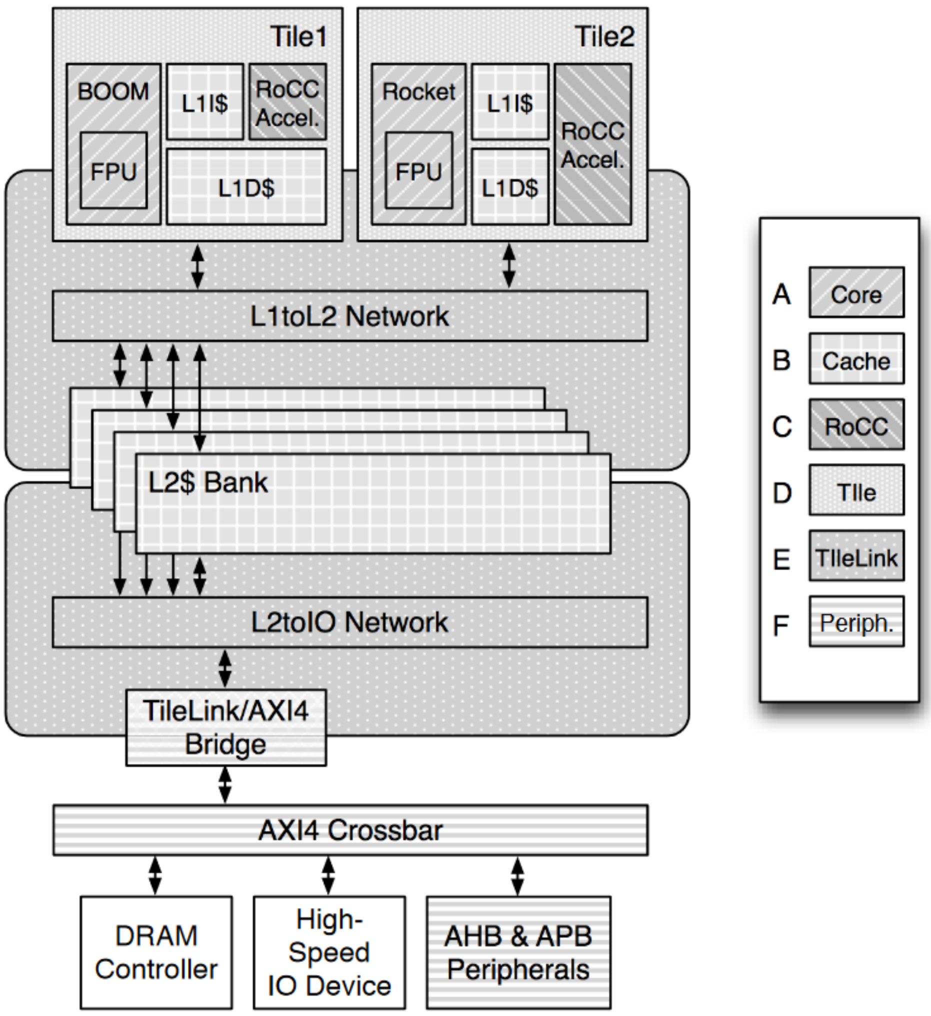


Figure 2.9: Rocket Chip SoC instance example.

2.3.4 The RoCC Co-Processor Interface

The Rocket Custom Co-processor Interface (RoCC) facilitates decoupled communication between a Rocket processor and attached co-processors. The RoCC interface accepts co-processor commands generated by committed instructions executed by the Rocket Core. The commands include the instruction word and the values in up to two integer registers, and commands may write an integer register in response [24].

The RoCC interface also allows the attached co-processor to share the Rocket core's data cache and page table walker, provides a facility for the co-processor to interrupt the core and be able to connect to the outer memory system directly over the TileLink interconnect [24]. A simplified view of the interface is shown in figure 2.10 [26].

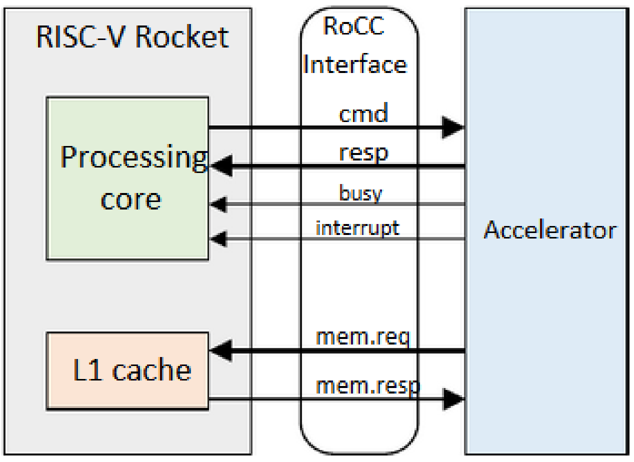


Figure 2.10: A simplified view of the RoCC interface.

In general, 32-bit RoCC instructions extend the RISC-V ISA and are formatted as shown in figure 2.11.

31 25	24 20	19 15	14	13	12	11 7	6 0
funct7	rs2	rs1	xd	xs1	xs2	rd	opcode
7	5	5	1	1	1	5	7

Figure 2.11: The RoCC instruction encoding.

The RoCC instruction is of the R-Type instruction format shown in 2.6. The fields are:

- ***opcode***, can be one of the custom opcodes not used for the base RISC-V instructions
- ***rs1 & rs2***, the two source registers (0-31)
- ***rd***, the destination register (0-31)
- ***xd***, flag indicating if a value need to be written back to the destination register rd
- ***xs1 & xs2***, flags indicating if the current RoCC instruction needs source values from the registers rs1 and rs2 respectively
- ***funct7***, extra specialization for the same opcode.

As mentioned above, when a RoCC type instruction reaches the write-back stage of the pipeline, rocket uses the RoCC interface to send a command to a dedicated co-processor. The signals include the RoCC instruction itself, the source register values (if existed) and several control (value/ready/interrupt) signals. The signals we used for our implementation will be discussed in Chapter 3.

Chapter 3

The proposed system

3.1 HW/SW architecture overview

The main purpose of our system is to ultimately improve the performance of packet classification. Once we have selected the HyperSplit algorithm for our research, our first step is to discover the **critical part** of the algorithm, whose execution on a dedicated hardware accelerator would lead to a significant performance improvement.

As explained in the previous chapter, the algorithm is divided into two stages, that of the building of the search tree -creating the search space using the given rule table- and that of the tree searching -search needed for every incoming packet to be classified-. Building the HyperSplit tree is a complicated and time consuming process and is needed when the rule table changes (rule insertion/deletion etc). Obviously, the searching process is much more frequent than the building one (since every incoming packet needs to be classified) and also offers space for additional improvements, such as potential parallelization of the process for many packets (as it does not modify the tree elements).

Therefore, we will create a hardware accelerator on which, having the tree in memory, the **search for an incoming packet** will be performed. Thus, the accelerator will receive "one packet", which will actually be a combination of the source IP address, destination IP address, source port, destination port and protocol number, search the tree and return the result location. The result will be a pointer to the final rules that the current packet belongs to.

The search arguments can fit into two 64-bit registers, and therefore could be used as arguments in a new RoCC command (software-side), as well as be send to the accelerator using the two 64-bit channels of the RoCC Interface (hardware-side). Accordingly, the result will be send back to the main core using the 64-bit return channel.

Our general system layout is shown in figure 3.1, which also includes the RoCC interface's channels for direct accelerator-cache communication. Those will not be used in our final implementation, as basic channels for RoCC instruction arguments are enough for the data needed by the accelerator, which will be proved later in the chapter, as well as the internal accelerator functionality.

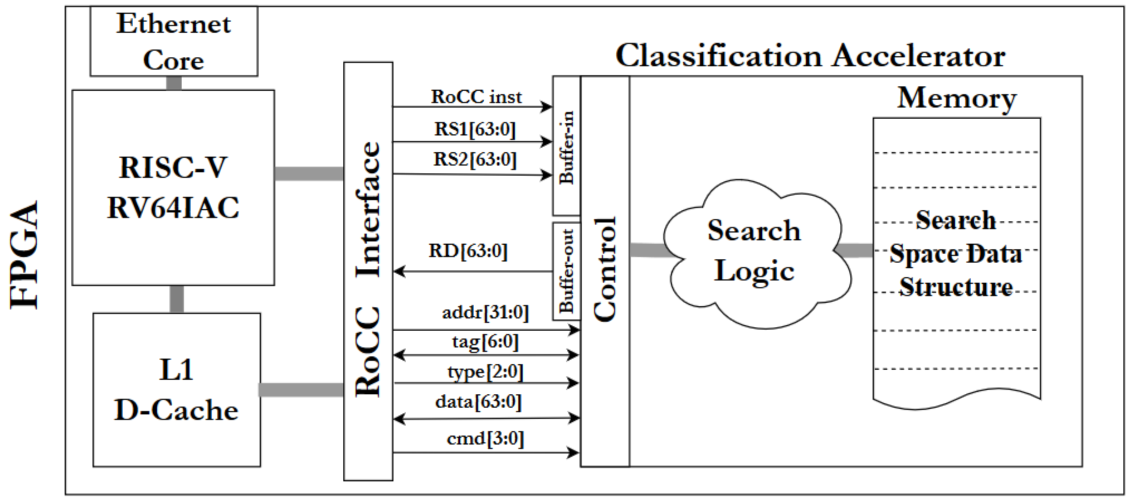


Figure 3.1: Proposed high-level system architecture

3.2 Processor Customization

In this section we describe how we use the software & hardware tools available to customize, build and run the RISC-V core and the Rocket Chip Generator.

In practice, our system combines all the elements and methods described in the previous chapter applying the following design decisions, towards an embedded and low cost solution:

1. From the RISC-V family, we select the instruction subset **RV64IAC**, since the targeted applications (packet management and processing) do not require complex operations (multipliers/divisions/floating-point number manipulation) and therefore a simple integer hardware can be used with no loss in capabilities.

2. We configure the RISC-V cache sizes (for both software and hardware simulation) to **L1 D-Cache** of 256-sets, 4-ways and 64-Byte blocks (total 64KB) and **L1 I-Cache** of 64-sets, 4-ways and 64-Byte blocks (total 16KB). We choose relatively big cache sizes for an embedded system. The reason is that the RISC-V performance will be the base performance, to which we will later compare our accelerator's one. So, we choose upper limit cases to control the safety of our final performance factor.

3. We add a new R-type instruction definition in the RISC-V ISA (that is, a new opcode), also adding it to the compiler and assembler in order to be able to use it in our C code.

4. We will use the following channels of the RoCC Inteface: the channel that will transmit the RoCC instruction itself, two channels for the values of RS1 and RS2, the channel for the return value of RD and the control signals valid and ready.

5. All of the above will be described in Chisel language of the Rocket Chip Generator, which will finally generate a synthesizable verilog code of the system.

We use the fpga-zynq git repository [27], which contains everything needed to port a Rocket Chip on a zynq family fpga. The *zynq-fpga* repository contains the *Rocket Chip Generator* git submodule [28], which contains the *RISC-V toolchain* git submodule [29]. These are the two basic repositories our development requires. Figure 3.2 shows the organisation of the basic repositories used. Figure 3.3 shows the action flow for generating and running the HyperSplit binary (our customization applies to the boxes with a star next to them). The rest of this chapter follows a top-down description of our development steps.

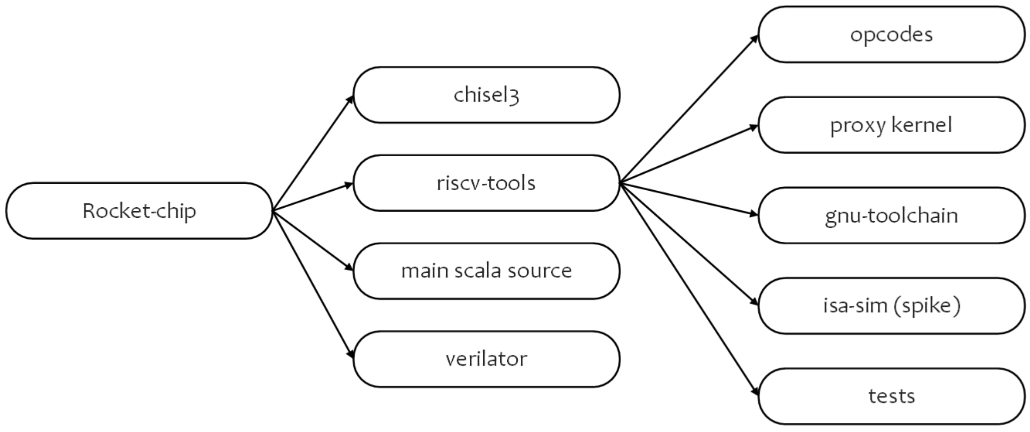
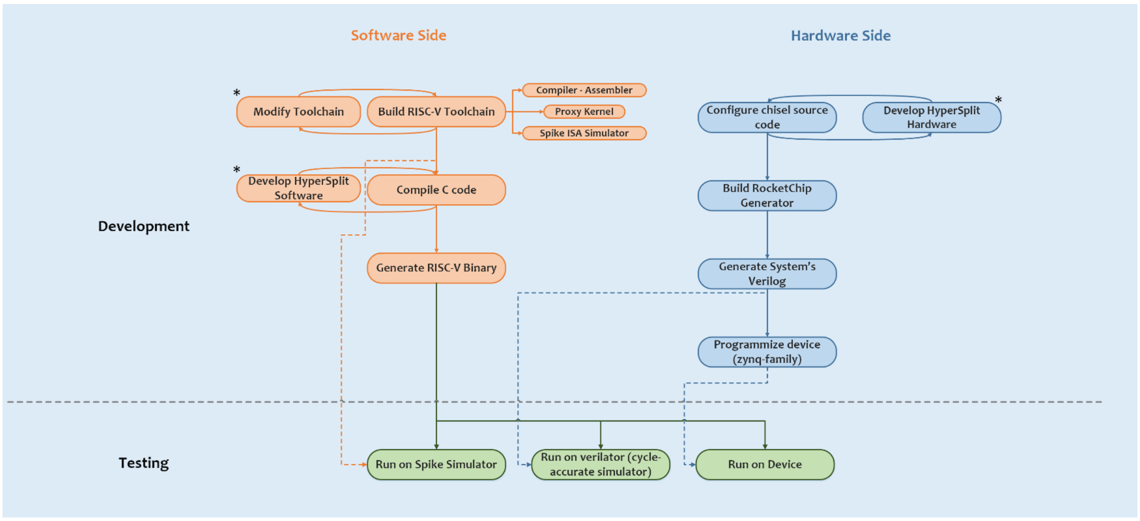


Figure 3.2: Rocket Chip Repository organisation



Software Side
Hardware Side
Development
Testing
Modify Toolchain
Build RISC-V Toolchain
Compiler - Assembler
Proxy Kernel
Spike ISA Simulator
Develop HyperSplit Software
Compile C code
Generate RISC-V Binary
Configure chisel source code
Develop HyperSplit Hardware
Build RocketChip Generator
Generate System's Verilog
Programmize device (zynq-family)
Run on Spike Simulator
Run on verilator (cycle-accurate simulator)
Run on Device

Figure 3.3: Proposed HW/SW System development and testing (flow chart). Starred boxes apply to the main effort (customization, design & coding) of the current work.

3.2.1 Software Side

The *RISC-V toolchain* git repository includes all the components needed to generate a RISC-V binary. They are highlighted in figure 3.4.

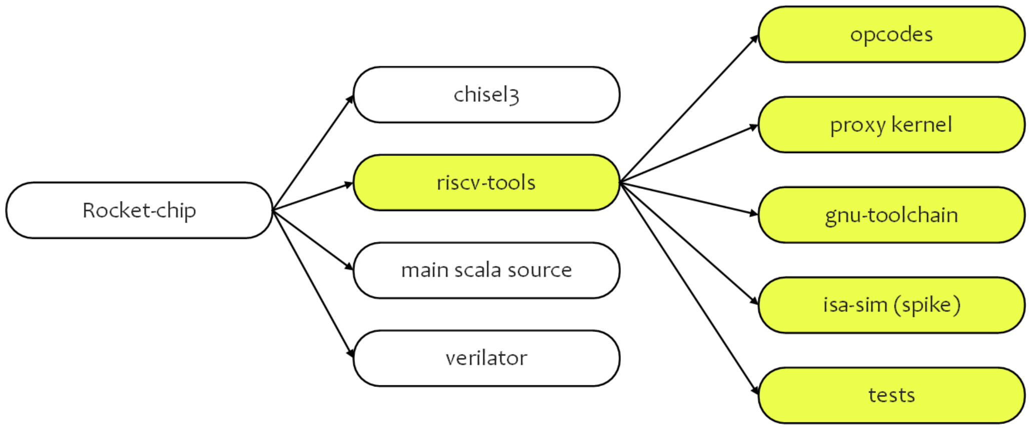


Figure 3.4: Software related parts

It contains the following submodules [29]:

- **riscv-opcodes**: enumerates standard RISC-V instruction opcodes and control and status registers.
- **riscv-gnu-toolchain**: is the RISC-V C and C++ cross-compiler
- **riscv-pk**: includes the RISC-V Proxy Kernel (a lightweight application execution environment that can host statically-linked RISC-V ELF binaries) and the Berkeley Boot Loader (bbl).
- **riscv-isa-sim**: builds Spike, the RISC-V ISA simulator.
- **riscv-tests**: hosts unit tests for RISC-V processors.

We clone all of the above, configure and build the cross-compiler, the proxy kernel and spike and, after we cross-compile the unit tests, we run them using spike and pk to test the mechanism as a whole. We then clone the HyperSplit C code [30] and add it to the riscv-tests. We cross-compile it and run it with Spike too, for a first performance evaluation.

The HyperSplit repository includes a test folder, that contains the common ClassBench classification benchmarks to test the algorithm [31]. ClassBench includes a Filter Set Generator that produces synthetic and with variable size filter sets that accurately model the characteristics of real filter sets. The filter sets range in size from 68 to 4557 entries and utilize one of the following formats [31]:

1. Access Control List (ACL) - standard format for security, VPN, and NAT filters for firewalls and routers (enterprise, edge, and backbone).
2. Firewall (FW) - proprietary format for specifying security filters for firewalls.
3. IP Chain (IPC) - decision tree format for security, VPN, and NAT filters for software-based systems.

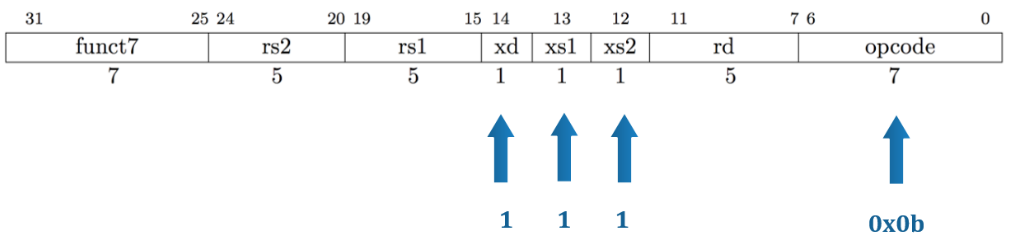
It also includes a Trace Generator that produces a sequence of packet headers to exercise the synthetic filter set [31]. We can now test HyperSplit using Spike simulator and ClassBench benchmarks.

Spike performs "ISA simulation", which means that it performs the binary's disassembly and uses an object oriented way of simulating the system, keeping control and status register counters, such as the number of instructions executed for a program. So, that is a first way of measuring the base performance of the algorithm on a RISC-V core. Detailed measurements will be presented in the next chapter.

Having configured and used all the RISC-V tools and the HyperSplit codes, we can now intervene to the toolchain. Our goal is to introduce a new instruction and use it instead of the binary search in the HyperSplit C code.

- **Step 1**

Considering the RoCC instruction encoding, we added a new instruction description in the riscv-opcodes repository. We use one of the custom instructions defined and we name our new instruction **hypersplit_search**. In practice, we used a new instruction of RoCC-type, having the bits 12, 13 and 14 set to 1 (our instruction will need two source registers and a destination register) and the unused opcode 0x0b.



- **Step 2**

Having our RoCC instruction definition, we included a description of it in the C cross-compiler, assembler and debugger. Now we are able to use the new instruction and generate a RISC-V executable.

- **Step 3**

The last step is to introduce the new instruction to the Spike simulator. So, we add all the definitions to Spike's code in order for it to be able to disassembly the instruction and also add a software description of performing the binary search on a HyperSplit search tree (to be executed when the new instruction is called). For the purpose of the simulation at this point, we send as arguments a pointer to the input packet and a pointer to the HyperSplit tree.

To measure the performance of HyperSplit using our new instruction, we need to design the accelerator hardware and measure its latency, including the interface overhead. Spike simulation is not representative in this case, as it only counts the instructions executed. In any case, the aforementioned steps have introduced a way for the software to be able to "call" our accelerator. We can now focus on the hardware design and development.

3.2.2 Hardware Side

This section describes our approach and steps of the system's hardware generation. The parts of the git tools we use in this stage are highlighted in figure 3.5.

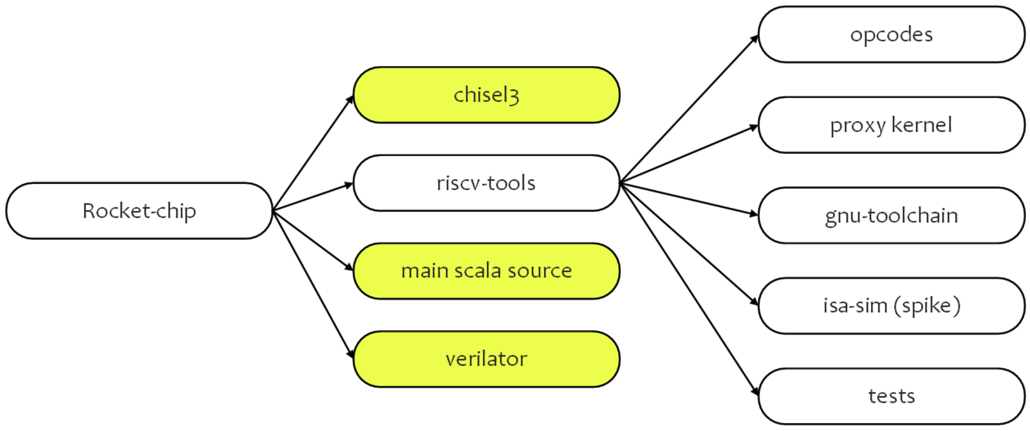


Figure 3.5: Rocket Chip Repository's Hardware related parts

Our goal is, after we choose an fpga/board to use for our design, to generate the verilog code for our Rocket Chip as shown in hardware steps in 3.3, reconfigure it to include our accelerator and end up having conclusions about:

1. the maximum clock frequency our final design can reach
2. the final number of clock cycles our accelerator needs for the HyperSplit search
3. the fpga resources our whole design requires

3.2.2.1 HyperSplit performance verification on verilator

We begin by exploring the Rocket Chip chisel source code. Chisel uses scala object oriented language and "transforms" it into a high level hardware description language. At first, we build the main source code using the configurations we described in this chapter's introductory section (RISC-V subset, cache sizes etc) before adding RoCC interface and accelerator to hardware description. Using the generated verilog code and the cycle-accurate hardware simulator in the directory, we run the default generated HyperSplit binary in order to verify the time we calculated using Spike (because we used the hypothesis of 40 clock cycles for memory accesses). The measurements are really close to that of table 4.3. So, we can now start modifying the hardware.

3.2.2.2 Chisel Rocket Chip configuration extension

The repository itself contains some classes for the implementation and use of RoCC-type accelerators. Using the SmallConfig (that includes only one Rocket Core), configuring the cache sizes as we want and removing mult/div hardware, we add a new configuration class that uses the given RoCC interface hardware description and connects a new accelerator to the main core using that interface. To do that, we also need to add the description of the instruction (new opcode of the RoCC instruction) that will reach the core's pipeline. For extra details about the chisel sources and the modifications please advice Appendixes.

At first, we describe a "dummy" accelerator that only performs the sum of two integers (given to RS1 and RS2 source registers) and returns the result to the core (to the RD destination register) after as many clock cycles as the value of RS1.

Applying all of the above, we now:

- Build our code and generate the new system's verilog.
- Create and compile (with the RISC-V cross-compiler) a C program which uses an instruction of our new type.
- Run the generated binary using the verilator.

As the main body of our "dummy" accelerator is actually an adder, we measure the clock cycles of a program that has only add commands. We can measure the clock cycles of our program with the RoCC command by using different input values to the first instruction argument. This way we can calculate the exact extra latency clock cycles that the RoCC interface adds, needed for the communication with the core. This procedure is shown in figure 3.6.

Using different input examples and for-loops in order to extract efficiently the extra time added, we concluded that the core-accelerator communication latency is **5 clock cycles**.

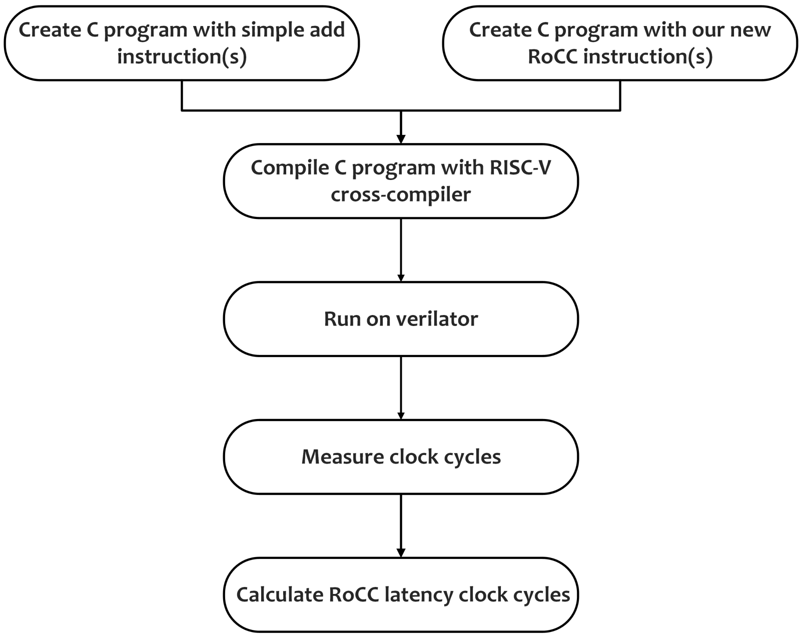
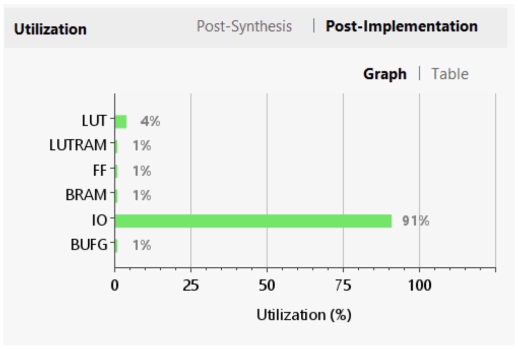

Create C program with simple add instruction(s)
Create C program with our new RoCC instruction(s)
Compile C program with RISC-V cross-compiler
Run on verilator
Measure clock cycles
Calculate RoCC latency clock cycles

Figure 3.6: Testing method of measuring RoCC interface latency.

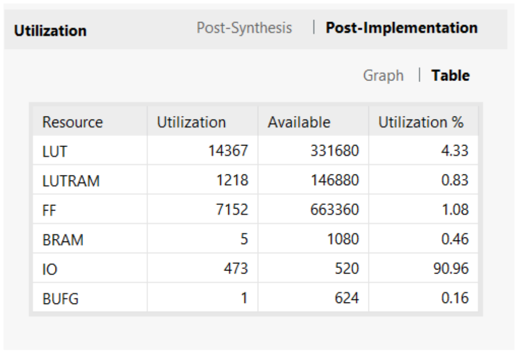
3.2.2.3 Rocket Chip on Vivado

The last external (before the addition of the accelerator) step is to gather the resources and frequency of the Rocket Chip with our "dummy" accelerator on the FPGA. We create a vivado project, in which we include the verilog sources generated so far. For our implementation we use the ***xcku060-ffva1156-3-e*** product part of the ***Kintex UltraScale*** product family.

Figures 3.7, 3.8 and 3.9 show the FPGA resources, power and max clock frequency of the system respectively. The implementation includes only the core (without any wrappers, and that's the reason for the increased IO resources).



(a) Utilization graph

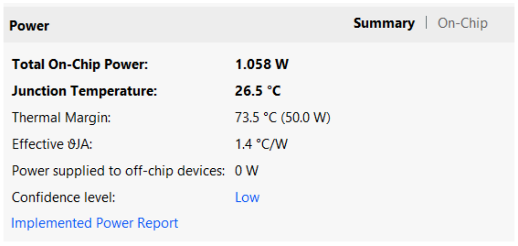


Utilization | Post-Synthesis | Post-Implementation | Graph | Table

Resource	Utilization	Available	Utilization %
LUT	14367	331680	4.33
LUTRAM	1218	146880	0.83
FF	7152	663360	1.08
BRAM	5	1080	0.46
IO	473	520	90.96
BUFG	1	624	0.16

(b) Utilization table

Figure 3.7: Rocket Chip Resources on Vivado

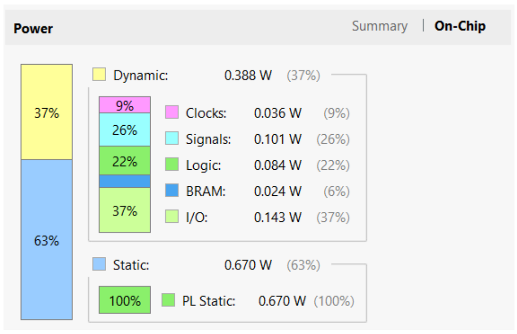


Power | **Summary** | On-Chip

Total On-Chip Power:	**1.058 W**
Junction Temperature:	**26.5 °C**
Thermal Margin:	73.5 °C (50.0 W)
Effective ϑJA:	1.4 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Implemented Power Report

(a) Power Summary



Power | Summary | **On-Chip**

37%	Dynamic:	0.388 W	(37%)
9%	Clocks:	0.036 W	(9%)
26%	Signals:	0.101 W	(26%)
22%	Logic:	0.084 W	(22%)
	BRAM:	0.024 W	(6%)
37%	I/O:	0.143 W	(37%)
63%	Static:	0.670 W	(63%)
100%	PL Static:	0.670 W	(100%)

(b) Power On-Chip

Figure 3.8: Rocket Chip Power on Vivado

when:

```
-----------------------------------------------------------------------------------------------
| Clock Summary
| -------------
-----------------------------------------------------------------------------------------------

Clock  Waveform(ns)       Period(ns)      Frequency(MHz)
-----  ------------       ----------      --------------
clock  {0.000 3.500}      7.000           142.857
```

Figure 3.9: RocketChip achieved on fpga frequency

3.2.2.4 Accelerator development

At this point, the only missing part is the performance of the actual HyperSplit search accelerator. We will develop the accelerator in VHDL separately and we will measure its performance using vivado.

The main elements that our accelerator needs are:

- the HyperSplit tree **memory space**. Having loaded the whole HyperSplit search tree in the accelerator after the building phase will save us all the time needed for memory acceses and cache misses. This design choice is by itself a huge performance improvement factor.
- a **comparator** component. In every stage of the binary search the input needs to be compared with the value of the current tree node, in order the next node in the searching process to be defined.
- the HyperSplit **search logic**. We need a logic circuit that will be able to extract the needed part of the input header and control the connections and the synchronization of the accelerator.

The accelerator's input is d-tuple describing the packet, e.g., {srcIP, dstIP, srcPORT, dstPORT, protocol}. Every tree node contains a threshold value T and the corresponding dimension d of the tuple (that it refers to and needs to be compared with d-th value of the input tuple).

So, we create a **ROM component** (with 380K entries), every entry of which will contain:

1. the current node's value dimension d_n (5-bit field in our case)
2. the current node's threshold value T_n (32-bit field in our case)
3. the address of the left child in the tree A_n (19-bit field in our case). The address of the right child will be the $A_n + 1$.

We, also, create a **32-bit comparator component** that will be responsible for the comparison of the d-th value of the input tuple and the T_n value at every stage of the searching process.

The extra assistant components are a **dto1 Multiplexer**, responsible for selecting the d-th value of the input tuple and forward it to the comparator, and a **19-bit adder**, responsible for generating the next address at every stage, considering the comparator's result.

The modular schematic diagram of the accelerator is shown in figure 3.10.

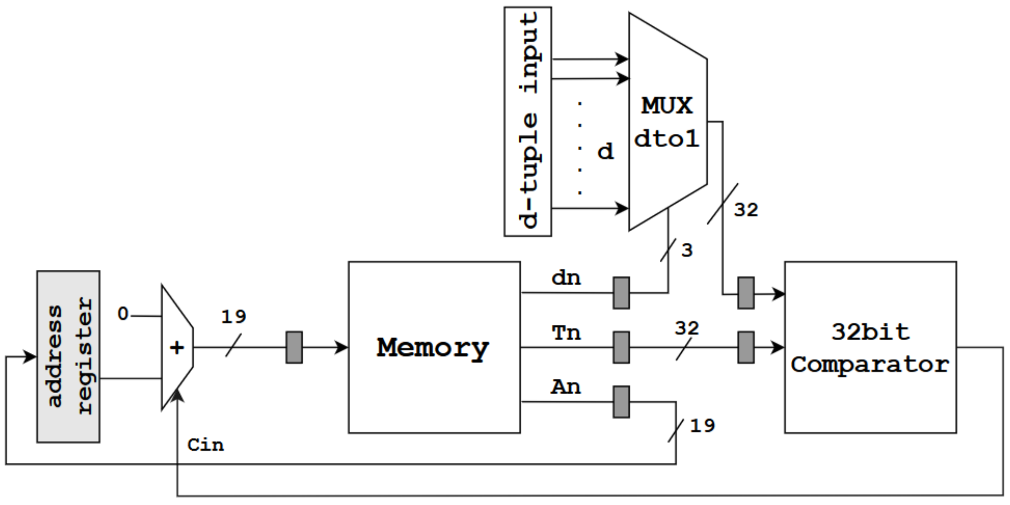


Figure 3.10: Proposed HyperSplit Search accelerator schematic diagram

Circuit functionality

At every stage of the searching procedure, the current memory element is the main control factor. The d_n value is used as the MUX input in order to extract the needed part of the input tuple, which will be the first input of the comparator. The T_n threshold value is used as the second input of the comparator. Depending on the comparator output, the next memory address will be generated and be used as the next stage's memory input. This procedure continues repeatedly until a tree leaf is reached (tree leaves use the value of zero as child address).

Synchronization

First of all, we create all the components to be asynchronous, in order to be able to manage the circuit synchronization externally, using registers where needed. That does not apply to the memory one, as we use a synchronous custom IP ROM component, which means 1 clock cycle latency for a read operation. As the MUX component needs the memory output and the comparator needs both the memory and the MUX output we add one extra delay register driving the threshold signal. We also need to add an extra register to the generated address signal and one to the memory input address. These registers are shown in figure 3.10 in grey boxes, including the internal memory ones. Our design can achieve at least $12,05$x performance improvement until now. Exact timing results will be shown in the next chapter.

3.3 Hardware Accelerator Optimizations

This section introduces some improvements to the body of the accelerator, as well as to the system design, in order to improve performance.

3.3.1 Optimization 1: *Pipelining & Packet Coalescing*

Observing the accelerator's design, we can easily conclude that there are idle cycles, as each component needs the result of other components, and "waits" until it's ready. That's the reason we added the extra registers. Our first thought is to use the pipeline technique to make use of these registers, by sending 3 input packets to the accelerator instead of one. More specifically, at each time instant, in a round-robin fashion, each pipeline stage will be dedicated to a distinct packet: e.g., while a node related to packet P_a is being processed by the comparison stage, a new node related to P_b is being fetched from RAMB. Therefore, in the long term, no empty cycles appear in our pipeline.

The only missing part of the implementation of this idea is the input RoCC instructions and arguments. If we create, for example, P pipeline stages on our accelerator, we should have P input packets ready to be searched. We can overcome this problem by coalescing P packets. Packet coalescing is the grouping of packets, already in use in network systems, as a way of limiting the number of receive interrupts and, as a result, lowering the amount of processing required [32]. Having the packets grouped that way, we can send them to the accelerator using sequent HyperSplit RoCC instructions.

So, now the accelerator will be able to search 3 packets almost simultaneously, and the new accelerator schematic is shown in figure 3.11.

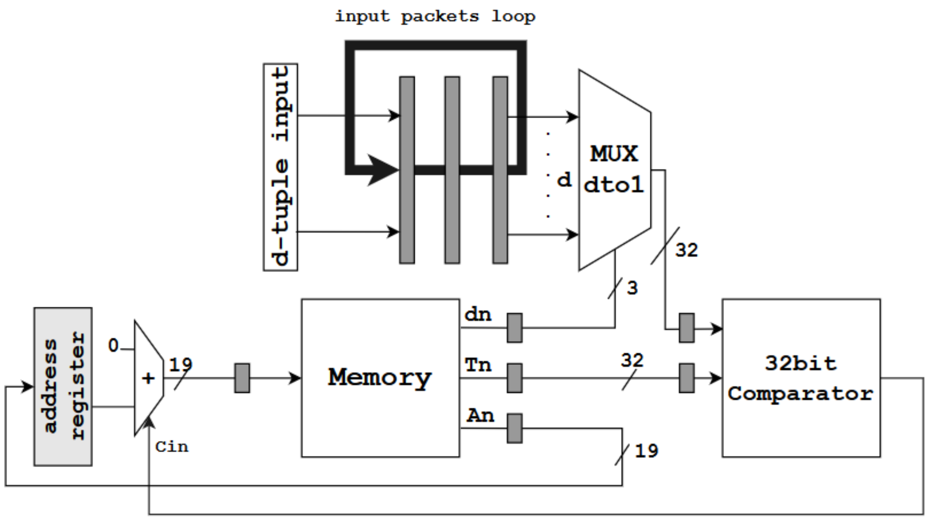


Figure 3.11: Proposed HyperSplit Search accelerator schematic diagram with P=3

3.3.2 Optimization 2: *Extra parallelization using Dual Data-path*

Another idea is to take advantage of the ROM memory IP block ability for dual-porting. This means that we can access two memory addresses at the same time. Using a dual-port memory will actually double the accelerator data-path, duplicating all the components (comparator, mux, registers) but **not** the memory. This optimization does not add any significant development overhead, but automatically doubles the number of packets that can be searched at the same time and therefore doubles the overall throughput of the accelerator.

3.3.3 Optimization 3: *Continuous stream processing without IO stalls*

All our measurements until now include the accelereator latency and the 5 clock cycles overhead of the core-accelerator communication for every packet (that is every RoCC instruction). In this section we consider the possibility of a non-blocking HyperSplit RoCC instruction, that triggers the search execution of the accelerator, enabling the main core to continue its normal execution without waiting for the accelerator result.

RoCC interface makes this idea possible including an interrupt signal to the core. So, we proceed to the following changes:

1. We modify our new RoCC-type HyperSplit instruction, having the xd flag bit to zero. That means that the instruction will not wait for a result to the RD register at the write-back pipeline stage of RISC-V core and the execution will continue normally.

2. We add two buffers to the accelerator, one input buffer and one output buffer, which will collect N (multiple of 10) packets by packet coalescing, and, when the results for all of them are ready, an interrupt will be sent to the core.

3. An interrupt handler will stop the core's execution and handle the results for the sent packets all together.

Notes:

- The input buffer is responsible for sending a new packet to the accelerator when an empty slot appears in the pipeline. The output buffer will be responsible for sending an interrupt signal to the core, as soon as there are N packets ready.
- Every packet has a unique id, to keep the input-output order of them, because different packets need different amount of searching iterations in the searching tree.
- The number of packets that control when the accelerator interrupts the core, as well as the buffer size, can be found by fine-tuning and depend on the network (e.g., throughput, interfaces), the device (e.g., frequencies) and the dataset (e.g., complexity). So, we can assume safely that this mechanism inserts no stalls to the implementation.

The rolling IO will actually "mask" the communication overhead but add the interrupt overhead. Using the buffers this overhead will be really minor, as it will apply to the N packets in total, a number really greater than 10.

Chapter 4

Performance Evaluation

4.1 Time and Resources per implementation step

This section presents the results of our implementation, in terms of time and FPGA resources. The main performance metrics are the clock cycles, the achieved clock frequency, the FPGA resources and of the HyperSplit binary search.

4.1.1 Baseline performance on RISC-V (Software-only execution)

The performance of the search on RISC-V (i.e, the needed execution clock cycles) is the base performance and we measure it using Spike simulator. The RISC-V frequency on our selected platform achieved $f_{clk} = 143MHz$ as described in the previous chapter. Adding some performance counters to the HyperSplit code, Spike results are shown in table 4.2. Table 4.1 shows some measurements regarding the HyperSplit's inputs and building tree. Table 4.3 shows the performance of HyperSplit search, ignoring the building phase. The last column represents the clock cycles needed for the search and was calculated using the following equation:

$$clock\ cycles_{per\_packet} = \frac{Instructions\ for\ Searching + DCache\ accesses + 40 * DCache\ misses}{Number\ of\ trace\ packets} \tag{4.1}$$

Equation interpretation: When simulating with Spike, we assume that the calculated number of instructions is equal to the clock cycles needed, as in RISC-V every instruction needs one clock cycle. That is not accurate for the instructions that perform memory accesses, as there is some extra overhead for fetching the data to the registers. That overhead is really big when data are not in the cache memory. So, assuming that a data memory access would take one clock cycle in the case of a D-Cache hit and 40 cycles on average in the case of a D-Cache miss (average time for accessing a DDR3 memory, common case), the run time calculation take the form of equation 4.1.

Benchmark	**Rules**	**Tree nodes**	**Worst depth**	**Trace packets**
acl1	753	7931	19	8140
acl1_5K	4415	55997	24	45600
acl1_10K	9603	121313	24	97000
fw1	270	75157	24	2830
fw1_100	93	7887	20	920
ipc1	1550	381069	26	17020
ipc1_100	100	3439	18	990
ipc1_1K	938	191059	26	9380

Table 4.1: Binary Trees built by HyperSplit for 8 test cases

Rule File	**Instructions for Building (us)**	**Instructions for Searching (us)**	**D-Cache accesses**	**D-Cache miss rate (%)**
acl1	95994458	2352326	53743507	1.67
acl1_5K	1068626590	15889789	464331366	7.396
acl1_10K	3981180850	35737971	1669860057	15.107
fw1	463509968	894466	175877797	1.173
fw1_100	47434372	243735	27280783	2.547
fw1_1K	3214346262	2924736	1148416473	0.941
ipc1	3109391562	6131943	1118471003	1.061
ipc1_100	16033666	236745	16464167	3.619
ipc1_1K	1146605751	3175587	427486317	1.143

Table 4.2: HyperSplit base performance on Spike

Rule File	**Instructions for Searching (us)**	**D-Cache accesses**	**D-Cache miss rate (%)**	**Clock cycles**
acl1	2352326	474127	6.39	496.14
acl1_5K	15889789	3437143	12.12	789.26
acl1_10K	35737971	7775447	13.84	892.26
fw1	894466	153736	11.17	613.13
fw1_100	243735	10951	20.98	376.70
fw1_1K	2924736	598052	13.44	837.05
ipc1	6131943	1301909	13.43	847.61
ipc1_100	236745	9457	17.20	314.43
ipc1_1K	3175587	653407	13.31	778.98

Table 4.3: HyperSplit search performance on Spike

4.1.2 Accelerator performance (HW/SW co-processing)

Now, we can calculate the clock cycles of the HyperSplit binary search on the accelerator, for every step of our optimization. We measure this performance using the tree measurements, our VHDL accelerator design and the interface and other extra overhead that may exist.

- **Default accelerator**

Our accelerator design was described in the previous chapter and is shown in figure 3.10. Every searching stage needs 3 clock cycles to generate the next memory address. So, the total clock cycles needed for an incoming packet will depend on the tree depth multiplied by 3, adding the 5-cycles RoCC communication overhead. Considering the average depth of the generated tree for every benchmark, new clock cycle measurements are now shown in 4.4. Our design can achieve at least **12.05x** performance improvement until now, operating on the RISC-V frequency (**143MHz**).

Benchmark	**Worst tree depth**	**Average tree depth**	**Clock cycles/packet**
acl1	19	15	50
acl1_5K	24	18	59
acl1_10K	24	19	62
fw1	24	16	53
fw1_100	20	13	44
ipc1	26	18	59
ipc1_100	18	12	41
ipc1_1K	26	17	56

Table 4.4: Proposed accelerator's performance

- **Optimization 1: Pipelining & Packet Coalescing**

Creating a loop in the accelerator input and using the packet coalescing technique as described in the previous chapter (figure 3.11), we manage to achieve further performance improvement. The improvement factor goes now from 12.2x to at least **30.4x** (operating on the RISC-V frequency). The renewed clock measurements are shown in table 4.5.

Benchmark	**Worst tree depth**	**Average tree depth**	**Clock cycles/3 packets**
acl1	19	15	60
acl1_5K	24	18	69
acl1_10K	24	19	72
fw1	24	16	63
fw1_100	20	13	54
ipc1	26	18	69
ipc1_100	18	12	51
ipc1_1K	26	17	66

Table 4.5: Accelerator performance with P=3 pipeline stages

Pipeline will not only solve the problem of idle clock cycles, but also will improve the clock frequency of the accelerator. So, we will try adding extra registers to the path in order to decide the ideal number of pipeline stages that gives the maximum **frequency/stage latency** factor to the circuit of the accelerator.

After several tries on Vivado, we find **P=5** (for P>5 pipeline stages we get no extra improvement in frequency and only add extra latency to the path). New clock measurements are shown in table 4.6 and new accelerator schematic in figure 4.1. For P=5 pipeline stages, the circuit of the accelerator can operate at $f_{clk} = 227$MHz (for P=3 we measure $f_{clk} = 190$MHz).

Benchmark	Worst tree depth	Average tree depth	Clock cycles/5 packets
acl1	19	15	100
acl1_5K	24	18	115
acl1_10K	24	19	120
fw1	24	16	105
fw1_100	20	13	90
ipc1	26	18	115
ipc1_100	18	12	85
ipc1_1K	26	17	110

Table 4.6: Accelerator performance with P=5 pipeline stages

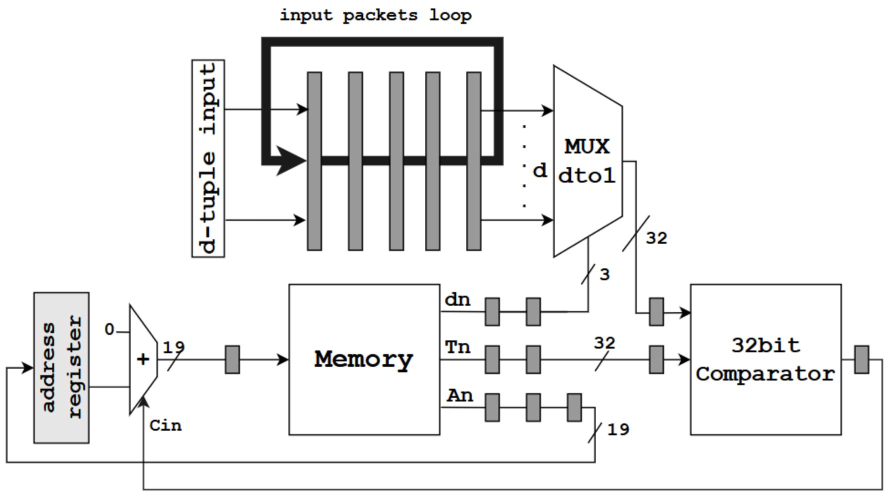


Figure 4.1: Proposed HyperSplit Search accelerator schematic diagram with P=5

We notice that clock cycles/packet are increased compared with the ones when P=3, but the new execution supports higher clock frequency ($f = 227MHz$ than $f = 190MHz$) and so the performance is actually increased.

- **Optimization 2: Extra parallelization using Dual Data-path**

As described in the previous chapter, we double the circuit data-path by taking advantage of the ROM memory IP block ability for dual-porting. This way we can automatically double the number of packets that can be searched at the same time and therefore doubles the overall throughput of the accelerator. So, the new measurements are shown in table 4.7. Considering only the clock cycles and the communication overhead (not taking into account the frequency increase) the improvement factor becomes **49.12x**.

Benchmark	Worst tree depth	Average tree depth	Clock cycles/10 packets
acl1	19	15	125
acl1_5K	24	18	140
acl1_10K	24	19	145
fw1	24	16	130
fw1_100	20	13	115
ipc1	26	18	140
ipc1_100	18	12	110
ipc1_1K	26	17	135

Table 4.7: Accelerator performance with P=5 pipeline stages and dual-port memory

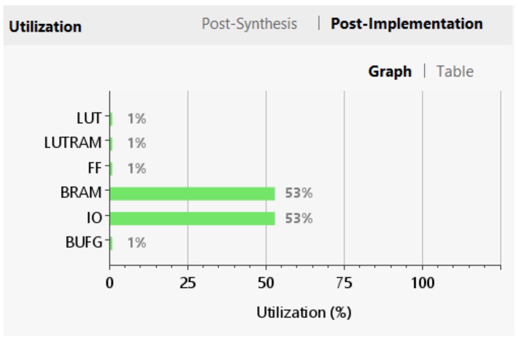
- **Optimization 3: Continuous stream processing without IO stalls**

The continuous IO we described in the previous chapter will actually "mask" the communication overhead and add the interrupt overhead, which will refer to the N packets in total (max number of ready packets -found by fine-tuning-, after which the buffer will send an interrupt signal to the main core). We can achieve **at least 70x** performance improvement, only in clock cycle level.

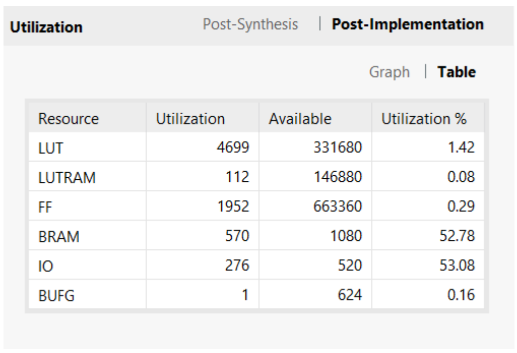
Table 4.8 shows the final calculated clock cycles measurements. Figures 4.2, 4.3 and 4.4 show the implementation resources, power and clock results of the final circuit of the accelerator on Vivado.

Benchmark	Worst tree depth	Average tree depth	Clock cycles/10 packets
acl1	19	15	85
acl1_5K	24	18	100
acl1_10K	24	19	105
fw1	24	16	90
fw1_100	20	13	75
ipc1	26	18	100
ipc1_100	18	12	70
ipc1_1K	26	17	95

Table 4.8: Accelerator's performance with P=5, dual-port memory and continuous IO streaming



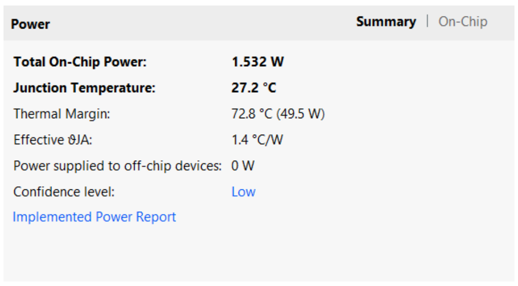
(a) Utilization Graph



Resource	Utilization	Available	Utilization %
LUT	4699	331680	1.42
LUTRAM	112	146880	0.08
FF	1952	663360	0.29
BRAM	570	1080	52.78
IO	276	520	53.08
BUFG	1	624	0.16

(b) Utilization Table

Figure 4.2: Proposed Accelerator's Resources on Vivado

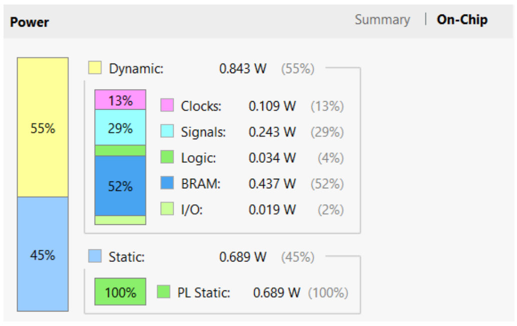


Power | **Summary** | On-Chip

Total On-Chip Power:	**1.532 W**
Junction Temperature:	**27.2 °C**
Thermal Margin:	72.8 °C (49.5 W)
Effective ϑJA:	1.4 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Implemented Power Report

(a) Power Summary

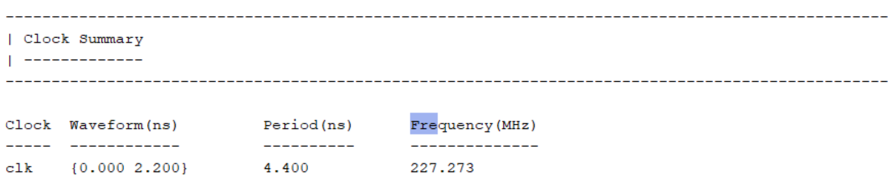


Power | Summary | **On-Chip**

🟨 Dynamic:	0.843 W	(55%)	55%
🟪 Clocks:	0.109 W	(13%)	13%
🟦 Signals:	0.243 W	(29%)	29%
🟩 Logic:	0.034 W	(4%)	
🟦 BRAM:	0.437 W	(52%)	52%
🟩 I/O:	0.019 W	(2%)	
🟦 Static:	0.689 W	(45%)	45%
🟩 PL Static:	0.689 W	(100%)	100%

(b) Power On-Chip

Figure 4.3: Proposed Accelerator's Power on Vivado



```
------------------------------------------------------------------------------------------------
| Clock Summary
| -------------
------------------------------------------------------------------------------------------------

Clock  Waveform(ns)        Period(ns)      Frequency(MHz)
-----  ------------        ----------      --------------
clk    {0.000 2.200}       4.400           227.273
```

Figure 4.4: Proposed Accelerator's Frequency on the FPGA

4.2 Final Implementation Results

Table 4.9 shows the final calculated clock cycles needed for the searching of 10 packets on the accelerator in comparison with the ones on RISC-V. Table 4.10 shows the Xilinx Ultrascale xcku060-3 FPGA resources of the implementation of the proposed system as a whole.

Rule File	**tot. RISC-V instructions**	**D-Cache (SW total) accesses**	**D-Cache miss rate (%)**	**Cycles per 10 packets**	
				RISC-V	**proposed**
acl1	2352326	474127	6,40	4961,39	85
acl1_5K	15889789	3437143	12,12	7892,64	100
acl1_10K	35737971	7775447	13,84	8922,56	105
fw1	894466	153736	11,17	6131,31	90
fw1_100	243735	10951	20,98	3767,02	75
ipc1	6131943	1301909	13,43	8476,08	100
ipc1_100	236745	9457	17,04	3144,26	70
ipc1_1K	3175587	653407	14,31	7789,80	95

Table 4.9: HyperSplit Searching on RISC-V or proposed VHDL accelerator (+I/F cycles)

	Available on FPGA	**Rocket Chip**	**HyperSplit Accelerator**	**Total Utilization**	**Total Utilization (%)**
LUTS	331680	14367	4699	19066	5.748
FFs	663360	7152	1952	9104	1.372
BRAMs	1080	5	570	575	53.24
DSPs	2760	0	0	0	0

Table 4.10: FPGA utilization of the proposed HW/SW system (Kintex Ultrascale xcku060-3)

Our final implementation gets up to **113x** faster processing, on average, than the initial RISC-V execution. Table 4.11 shows the actual achieved speedup at every step of our implementation. Graph 4.5 shows the final improvement we get, both in terms of needed clock cycles and achieved clock frequency.

	RISC-V	**Default accelerator**	**P=3**	**P=5**	**Dual data-path**	**Final**
Frequency	143	143	190	227	227	227
Period	7	7	5.26	4.4	4.4	4.4
Cycles/10 packets (avg)	6385.6	530	210	210	130	90
Time (ns, avg)	44699.2	3710	1104.6	924	572	396
Speedup	1	12.05	40.47	48.38	78.15	112.88

Table 4.11: Achieved speedup on every step of the accelerator implementation

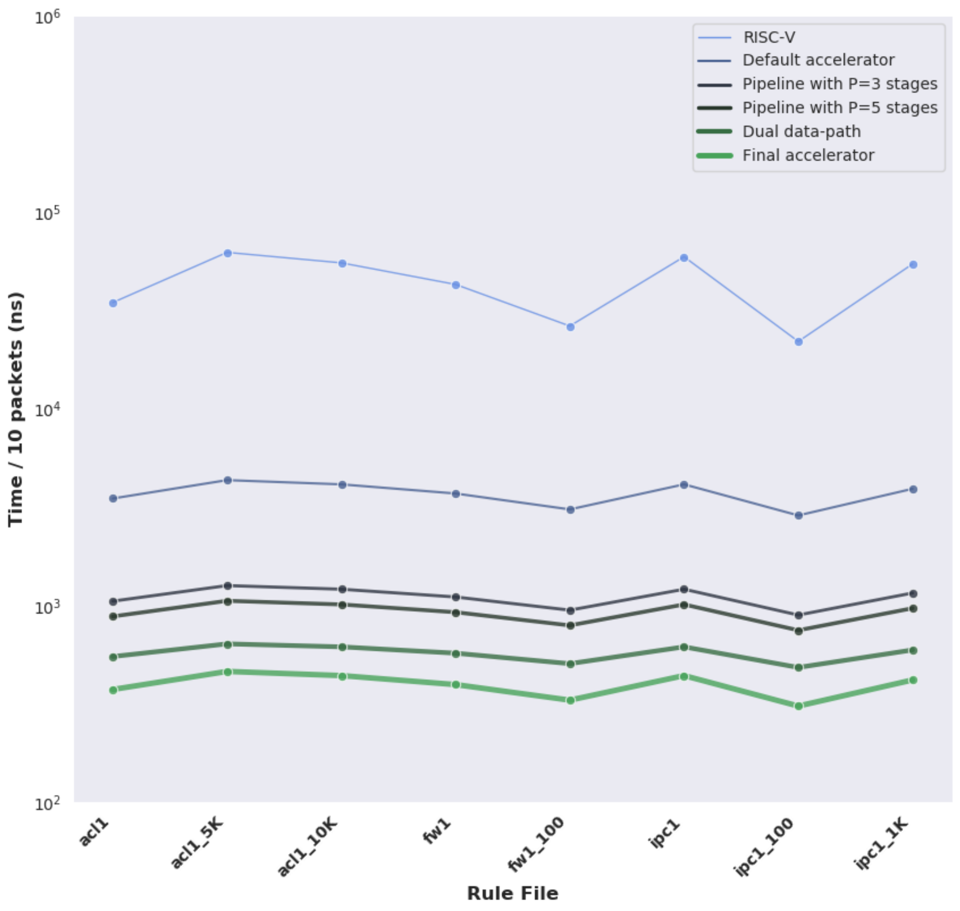


Figure 4.5: HyperSplit search performance comparison during our implementation

Chapter 5

Conclusions & Future work

This work examined the possibility of accelerating Packet Classification, using a RISC-V core combined with dedicated hardware accelerators. We placed a HyperSplit binary search VHDL accelerator next to a RV64IAC RISC-V core, using the RoCC interface for the core-accelerator communication. The proposed HW/SW co-design and presented results led us to the following conclusions:

- Placing dedicated hardware accelerators next to the RISC-V is a feasible scenario for a network processing architectural design.
- Functionalities that look inherently sequential can actually be accelerated on FPGA by improving the memory accesses bottleneck (when there are big data structures and searches) and clock frequency of a general purpose CPU.
- We tested our implementation using a really common network functionality (Packet Classification) and managed to get two orders of magnitude higher performance (up to 113x assuming correctly tuned buffering, e.g., without interrupt overload).
- The current work can be used even in 10Gbps routers. We can get even higher acceleration by implementing the design on ASIC, considering that the main acceleration factor is the memory access rate improvement.
- RISC-V is a great choice for experimenting on hardware design, having open-source ISA, simulation and implementation tools.

Future work can take into consideration the following actions:

- We can take advantage of high performance FPGA resources, such as the Ultra-Ram, which is a perfect choice for storing big data structures, without the need of sophisticated configuration (default configuration is sufficient).
- The actual system integration using a real network (with 10/25G Ethernet MAC subsystem and optimal feeding of ethernet packets to RISCV).

- The acceleration and testing of other common network functionalities on our system, such as data bitfield manipulation and queue management.
- Given the design testing on a real system, we could proceed to an ASIC implementation, in order to gain one more order of magnitude.

Appendix A

Acronyms

CPU	Central Processing Unit
VHDL	Very Large Scale Integration
HW	Hardware
SW	Software
ISA	Instruction Set Architecture
FPGA	Field Programmable Gate Array
IOT	Internet Of Things
MPLS	Multiprotocol Label Switching
HDL	Hardware Description Language
ASIC	Application-Specific Integrated Circuit
QoS	Quality of Service
CAM	Content-Addressable Memory
TCAM	Ternary Content-Addressable Memory
IC	Integrated Circuits
IP	Internet Protocol
TCP	Transmission Control Protocol
TCP	Virtual Private Network
RISC	Reduced Instruction Set Computer
SoC	System-on-Chip
MPSoC	Multi-Processor System-on-Chip
HTTP	HyperText Transfer Protocol
MMU	Memory Management Unit
TLB	Translation Lookaside Buffer
RTL	Register-Transfer Level
NAT	Network Address Translation
VPN	Virtual Private Network
IO	Input-Output
OVS	Open Virtual Switch

Appendix B

Toolchain Building steps

B.1 riscv-gnu-toolchain

In order to add the new instruction to the RISC-V ISA, we firstly need to add a new opcode decleration to the opcode files and then add the new instruction to the RISC-V cross compiler. So, we add the following to the corresponing file(s):

1. riscv-tools/riscv-opcodes/opcodes

```
hypersplit_search rd rs1 rs2 31..25=0  14..12=7 6..2=0x02 1..0=3
```

2. riscv-tools/riscv-gnu-toolchain/riscv-binutils-gdb/include/opcode/riscv-opc.h

```
#define MATCH_HYPERSPLIT  0x0000700b
#define MASK_HYPERSPLIT   0xfe00707f

DECLARE_INSN(hypersplit_search, MATCH_HYPERSPLIT, MASK_HYPERSPLIT)
```

3. riscv-tools/riscv-gnu-toolchain/riscv-binutils-gdb/opcodes/riscv-opc.c

```
riscv_opcodes[] = {
 .
 .
 .
 {"hypersplit_search", "I", "d,s,t", MATCH_HYPERSPLIT,
MASK_HYPERSPLIT, match_opcode, 0 }
 .
 .
 .
 }
```

Next, we describe the steps for building the tool-chain, in order to generate the cross-compiler and be able to use it.

1. Init and update all git submodules.

```
$ git submodule update --init --recursive
```

2. Install dependency packages.

```
$ sudo apt-get install autoconf automake autotools-dev curl device
-tree-compiler libmpc-dev libmpfr-dev libgmp-dev gawk build-
essential bison flex texinfo gperf
```

3. Select installation path

```
$ export TOP=/home/apnev/fpga-zynq/rocket-chip
$ export RISCV=$TOP/riscv
$ export PATH=$PATH:$RISCV/bin
```

4. In gnu-toolchain directory:

```
    $ ./configure --prefix=$RISCV &  make)
```

5. In fesvr directory:

```
    $ mkdir build
    $ cd build
    $ ../configure --prefix=$RISCV
    $ make install
```

6. In pk directory:

```
    $ mkdir build
    $ cd build
    $ ../configure --prefix=$RISCV --host=riscv64-unknown-elf
    $ make
    $ make install)
```

7. In isa-sim derectory:

```
    $ apt-get install device-tree-compiler
    $ mkdir build
    $ cd build
    $ ../configure --prefix=$RISCV --with-fesvr=$RISCV
    $ make
    $ [sudo] make install)
```

8. In tests directory:

```
    $ git submodule update --init --recursive
    $ autoconf
    $ ./configure --prefix=$RISCV/target
    $ make
    $ make install
```

Having cloned the HyperSplit C code into the riscv-tests directory, we cross-compile the code and then we execute it using spike and pk. An example HyperSplit usage is:

```
$ spike pk hypersplit -r test/rules/fw1 -t test/traces/fw1_trace
```

B.2 rocket-chip

After we configure our main rocket-chip directory ($ROCKETCHIP), we build the DefaultConfig and run the given tests by:

```
    $ cd $ROCKETCHIP/emulator
    $ make -jN run
    or
    $ make -jN run-asm-tests
```

The next step is to add the RoCC HyperSplit "dummy" module (the accelerator circuit) to the rocket-chip code using the RoCC interface. So, we add our custom descriptions to the following files:

- **Custom Instruction related parts**

    1. In /home/apnev/fpga-zynq/rocket-chip/src/main/scala/rocket/Instructions.scala:

    ```
        def HYPERSPLIT = BitPat("b?????????????????111?????0001011")
    ```

    2. In /home/apnev/fpga-zynq/rocket-chip/src/main/scala/rocket/IDecode.scala:
       At class RoCCDecode:

    ```
        .
        .
        HYPERSPLIT-> List(Y,N,Y,N,N,N,Y,Y,A2_ZERO,A1_RS1,IMM_X,DW_XPR,
       FN_ADD,N,M_X,MT_X,N,N,N,N,N,N,Y,CSR.N,N,N,N,N,N),
        .
        .
    ```

- **Configuration related**

    1. In src/main/scala/tile/LazyRoCC.scala:

    ```
    class  HyperSplitSearch(implicit p: Parameters) extends LazyRoCC {
        override lazy val module = new HyperSplitSearchModule(this) }
    ```

    2. In src/main/scala/system/Configs.scala:

    ```
    class HypersplitRoccConfig extends Config(new WithHypersplitRocc
        ++ new DefaultSmallConfig)
    ```

       where:

    ```
    class DefaultSmallConfig extends Config(new WithNSmallCores(1) ++
        new BaseConfig)
    ```

    3. In src/main/scala/subsystem/Configs.scala:

    ```
    class WithNSmallCores(n: Int) extends Config((site, here, up) => {
      case RocketTilesKey => {
        val small = RocketTileParams(
          core = RocketCoreParams(useVM = false, fpu = None),
          btb = None,
          dcache = Some(DCacheParams(
            rowBits = site(SystemBusKey).beatBits,
            nSets = 256,
            nWays = 4,
            nTLBEntries = 4,
            nMSHRs = 0,
            blockBytes = 64)),
    ```

```
      icache = Some(ICacheParams(
        rowBits = site(SystemBusKey).beatBits,
        nSets = 64,
        nWays = 4,
        nTLBEntries = 4,
        blockBytes = site(CacheBlockBytes))))
    List.tabulate(n)(i => small.copy(hartId = i))
  }
})
```

and

```
class WithHypersplitRocc extends Config((site, here, up) => {
  case RocketTilesKey => up(RocketTilesKey, site) map { r =>
    r.copy(rocc =
      Seq(
        RoCCParams(
          opcodes = OpcodeSet.hypersplit,
          generator = (p: Parameters) => {
            val hypersplit_search = LazyModule(new
    HyperSplitSearch()(p))
            hypersplit_search})
        ))
    }
})
```

- **Accelerator Related**

  1. In /src/main/scala/tile/LazyRoCC.scala, we add the class af the "dummy" accelerator:

```
class HyperSplitSearchModule(outer: HyperSplitSearch, n: Int = 4)(
    implicit p: Parameters) extends LazyRoCCModule(outer)
  with HasCoreParameters {
    val count = Reg(UInt(width = xLen))
    val wdata = Reg(UInt(width = xLen))
    val adder1 = Reg(UInt(width = xLen))
    val adder2 = Reg(UInt(width = xLen))
    val resp_rd = Reg(io.resp.bits.rd)

    val s_idle :: s_wait :: s_resp :: Nil = Enum(Bits(), 3)
    val state = Reg(init = s_idle)

    val finished = Reg(Bool())

    io.cmd.ready := (state === s_idle)
    io.resp.valid := (state === s_resp)
    io.resp.bits.rd := resp_rd
    io.resp.bits.data := wdata
    when (io.cmd.fire()) {
      adder1 := io.cmd.bits.rs1
      adder2 := io.cmd.bits.rs2
      wdata := io.cmd.bits.rs1 + io.cmd.bits.rs2
      resp_rd := io.cmd.bits.inst.rd
      count := UInt(0)
      finished := Bool(false)
      state := s_wait
    }


    when (state === s_wait) {
      when (!finished) {
        count := count + UInt(1)
      }
      when (count === adder1) { finished := Bool(true) }
      when (count === UInt(3)) {
        io.interrupt := Bool(true)
        state := s_idle
      }
      state := Mux(finished, s_resp, s_wait)
    }

    when (io.resp.fire()) { state := s_idle }

    io.busy := (state =/= s_idle)
    io.interrupt := Bool(false)
}
```

Finally, we can see the system's configurations of the beginning of the Rocket Chip building output. The following instance uses the configuration including the Dummy HyperSplit RoCC Accelerator (instruction that waits for rd result).

```
Interrupt map (1 harts 2 interrupts):
  [1, 2] => dut

/dts-v1/;

/ {
  #address-cells = <1>;
  #size-cells = <1>;
  compatible = "freechips,rocketchip-unknown-dev";
  model = "freechips,rocketchip-unknown";
  L13: cpus {
    #address-cells = <1>;
    #size-cells = <0>;
    L5: cpu@0 {
      clock-frequency = <0>;
      compatible = "sifive,rocket0", "riscv";
      d-cache-block-size = <64>;
      d-cache-sets = <256>;
      d-cache-size = <65536>;
      device_type = "cpu";
      i-cache-block-size = <64>;
      i-cache-sets = <64>;
      i-cache-size = <16384>;
      next-level-cache = <&L7>;
      reg = <0>;
      riscv,isa = "rv64imac";
      status = "okay";
      timebase-frequency = <1000000>;
      L3: interrupt-controller {
        #interrupt-cells = <1>;
        compatible = "riscv,cpu-intc";
        interrupt-controller;
      };
    };
  };
  L7: memory@80000000 {
    device_type = "memory";
    reg = <0x80000000 0x10000000>;
  };
  L12: soc {
    #address-cells = <1>;
    #size-cells = <1>;
    compatible = "freechips,rocketchip-unknown-soc", "simple-bus";
    ranges;
    L1: clint@2000000 {
      compatible = "riscv,clint0";
      interrupts-extended = <&L3 3 &L3 7>;
      reg = <0x2000000 0x10000>;
      reg-names = "control";
    };
    L2: debug-controller@0 {
      compatible = "sifive,debug-013", "riscv,debug-013";
      interrupts-extended = <&L3 65535>;
      reg = <0x0 0x1000>;
      reg-names = "control";
    };
    L10: error-device@3000 {
      compatible = "sifive,error0";
      reg = <0x3000 0x1000>;
      reg-names = "mem";
    };
    L6: external-interrupts {
      interrupt-parent = <&L0>;
      interrupts = <1 2>;
    };
    L0: interrupt-controller@c000000 {
      #interrupt-cells = <1>;
      compatible = "riscv,plic0";
      interrupt-controller;
      interrupts-extended = <&L3 11>;
      reg = <0xc000000 0x4000000>;
      reg-names = "control";
      riscv,max-priority = <7>;
      riscv,ndev = <2>;
    };
    L8: mmio-port-axi4@60000000 {
      #address-cells = <1>;
      #size-cells = <1>;
      compatible = "simple-bus";
      ranges = <0x60000000 0x60000000 0x20000000>;
    };
    L9: rom@10000 {
      compatible = "sifive,rom0";
      reg = <0x10000 0x10000>;
      reg-names = "mem";
    };
  };
};

Generated Address Map
         0 -      1000 ARWX  debug-controller@0
      3000 -      4000 ARWX  error-device@3000
     10000 -     20000  R XC rom@10000
   2000000 -   2010000 ARW   clint@2000000
   c000000 -  10000000 ARW   interrupt-controller@c000000
  60000000 -  80000000  RWX  mmio-port-axi4@60000000
  80000000 -  90000000  RWXC memory@80000000

-----------------------
```

We can generate system's verilog in the verilator subdirectory using:

```
$ cd $ROCKETCHIP/emulator
$ make -jN CONFIG=HypersplitRoccConfig
```

where N = number of threads used for builing.

Bibliography

[1] *RISC-V Foundation: Instruction Set Architecture (ISA)*, 2019. [Online]. Available: https://riscv.org/.

[2] 2019. [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html.

[3] D. Taylor, "Survey Taxonomy of Packet Classification Techniques", *ACM Computing Surveys*, vol. 37, Sep. 2004. DOI: 10.1145/1108956.1108958.

[4] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li, "Packet Classification Algorithms: From Theory to Practice", in *IEEE INFOCOM 2009*, Apr. 2009, pp. 648–656. DOI: 10.1109/INFCOM.2009.5061972.

[5] 2019. [Online]. Available: https://www.openvswitch.org/.

[6] 2019. [Online]. Available: https://en.wikipedia.org/wiki/Network_processor.

[7] D. Serpanos and T. Wolf, "Chapter 11 - Specialized hardware components", in *Architecture of Network Systems*, ser. The Morgan Kaufmann Series in Computer Architecture and Design, D. Serpanos and T. Wolf, Eds., Boston: Morgan Kaufmann, 2011, pp. 211–227. DOI: https://doi.org/10.1016/B978-0-12-374494-4.00011-6. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780123744944000116.

[8] M. Ahmadi and S. Wong, "Network processors: Challenges and trends", in *Proc. of the 17th Annual Workshop on Circuits, Systems and Signal Processing, ProRisc 2006*, Citeseer, 2006, pp. 223–232.

[9] 2019. [Online]. Available: https://www.mellanox.com/sites/default/files/related-docs/prod_npu/PB_NP-5.pdf.

[10] 2019. [Online]. Available: https://blog.mellanox.com/2018/08/defining-smartnic/.

[11] D. Medhi and K. Ramasamy, "Chapter 15 - IP Packet Filtering and Classification", in *Network Routing (Second Edition)*, ser. The Morgan Kaufmann Series in Networking, D. Medhi and K. Ramasamy, Eds., Second Edition, Boston: Morgan Kaufmann, 2018, pp. 500–547. DOI: https://doi.org/10.1016/B978-0-12-800737-2.00018-1.

[Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780128007372000181.

[12] P. Gupta and N. McKeown, “Algorithms for packet classification”, *IEEE Network*, vol. 15, Mar. 2001. DOI: 10.1109/65.912717.

[13] P. Gupta and N. McKeown, “Packet Classification using Hierarchical Intelligent Cuttings”, *Proc. Hot Interconnects*, vol. 20, Jan. 2000.

[14] O. Ahmed, “Towards Efficient Packet Classification Algorithms and Architectures”, PhD thesis, University of Guelph, 2013.

[15] V. Sahasranaman and M. M. Buddhikot, “Comparative evaluation of software implementations of layer-4 packet classification schemes”, pp. 220–228, Nov. 2001. DOI: 10.1109/ICNP.2001.992902.

[16] W. Jiang and V. K. Prasanna, “Scalable Packet Classification on FPGA”, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, Sep. 2012.

[17] E. Asha and S. Kavatha, “Packet Classification Algorithms: A Survey”, *International Journal of Research in Advent Technology, Vol.2, No.12, December2014 E-ISSN: 2321-9637*, Dec. 2014.

[18] Bo Xu, Dongyi Jiang, and Jun Li, “HSM: a fast packet classification algorithm”, in *19th International Conference on Advanced Information Networking and Applications (AINA'05) Volume 1 (AINA papers)*, vol. 1, Mar. 2005, 987–992 vol.1. DOI: 10.1109/AINA.2005.200.

[19] J. van Lunteren and T. Engbersen, “Fast and scalable packet classification”, *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 4, pp. 560–571, May 2003. DOI: 10.1109/JSAC.2003.810527.

[20] [Online]. Available: http://conferences.sigcomm.org/sigcomm/2005/slides-LakRan.pdf.

[21] H. Song and J. W. Lockwood, “Efficient Packet Classification for Network Intrusion Detection Using FPGA”, in *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*, ser. FPGA '05, Monterey, California, USA: ACM, 2005, pp. 238–245. DOI: 10.1145/1046192.1046223. [Online]. Available: http://doi.acm.org/10.1145/1046192.1046223.

[22] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, “Algorithms for Advanced Packet Classification with Ternary CAMs”, *SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 4, pp. 193–204, Aug. 2005. DOI: 10.1145/1090191.1080115. [Online]. Available: http://doi.acm.org/10.1145/1090191.1080115.

[23] W. Jiang and V. K. Prasanna, “Field-split Parallel Architecture for High Performance Multi-match Packet Classification Using FPGAs”, in *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA ’09, Calgary, AB, Canada: ACM, 2009, pp. 188–196. DOI: 10.1145/1583991.1584044. [Online]. Available: http://doi.acm.org/10.1145/1583991.1584044.

[24] [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.pdf.

[25] *Chisel/FIRRTL Hardware Compiler Framework*. [Online]. Available: https://www.chisel-lang.org/.

[26] S. Savas, Z. Ul-Abdin, and T. Nordström, “Designing Domain-Specific Heterogeneous Architectures from Dataflow Programs”, *Computers*, vol. 7, p. 27, Apr. 2018. DOI: 10.3390/computers7020027.

[27] *Support for Rocket Chip on Zynq FPGAs*. [Online]. Available: https://github.com/ucb-bar/fpga-zynq.

[28] *Rocket Chip Generator*. [Online]. Available: https://github.com/chipsalliance/rocket-chip.

[29] *RISC-V Tools (ISA Simulator and Tests)*. [Online]. Available: https://github.com/riscv/riscv-tools.

[30] *A functional HyperSplit algorithm implementation*. [Online]. Available: https://github.com/fooozhe/HyperSplit.

[31] D. E. Taylor and J. S. Turner, “ClassBench: A Packet Classification Benchmark”, *IEEE/ACM Transactions on Networking*, vol. 15, no. 3, pp. 499–511, Jun. 2007. DOI: 10.1109/TNET.2007.893156.

[32] 2019. [Online]. Available: https://whatis.techtarget.com/definition/packet-coalescing.