Master of Science Thesis in Electrical Engineering
Department of Electrical Engineering, Linköping University, 2021

# RISC-V Based Application-Specific Instruction Set Processor for Packet Processing in Mobile Networks

**Oskar Södergren**

Master of Science Thesis in Electrical Engineering

**RISC-V Based Application-Specific Instruction Set Processor for Packet Processing in Mobile Networks**

Oskar Södergren

LiTH-ISY-EX--21/5439--SE

Supervisor: **Matthew Goode**
Nokia, Espoo, Finland

Examiner: **Kent Palmkvist**
ISY, Linköping University

*Division of Computer Engineering*
*Department of Electrical Engineering*
*Linköping University*
*SE-581 83 Linköping, Sweden*

# Abstract

This thesis explores the use of an ASIP for handling O-RAN control data. A model application was constructed, optimized and profiled on a simple RV32-IMC core. The compiled code was analyzed, and the instructions “byte swap”, “pack”, “bitwise extract/deposit” and “bit field place” were implemented. Synthesis of the core, and profiling of the model application, was done with and without each added instruction. Byte swap had the largest impact on performance (14% improvement per section, and 100% per section extension), followed by bitwise extract/deposit (10% improvement per section but no impact on section extensions). Pack and bit field place had no impact on performance. All instructions had negligible impact on core size, except for bitwise extract/deposit, which increased size by 16%. Further studies, with respect to both overall architecture and further evaluation of instructions to implement, would be necessary to design an ideal ASIP for the application.

# Contents

**List of Figures** vii

**List of Tables** viii

**Notation** ix

**1 Introduction** 1
- 1.1 Problem formulation . . . 2
- 1.2 Question formulation . . . 3
- 1.3 Delimitations . . . 3

**2 Theory and related works** 5
- 2.1 LTE and 5G New Radio basics . . . 5
- 2.2 Mobile fronthaul networks and eCPRI . . . 6
- 2.3 eCPRI and O-RAN real-time control data . . . 7
- 2.4 Application specific instruction set processors . . . 9
  - 2.4.1 Design methodology . . . 10
  - 2.4.2 Packet parsing . . . 10
- 2.5 RISC-V . . . 11
  - 2.5.1 The base architecture . . . 11
  - 2.5.2 RISC-V standard extensions . . . 12
  - 2.5.3 Overall structure of architecture . . . 13
  - 2.5.4 Instruction format . . . 13
  - 2.5.5 C extension . . . 14
- 2.6 Bit permutations . . . 15
- 2.7 Extracting and depositing bits in a word . . . 15

**3 Method** 17
- 3.1 Application analysis . . . 18
  - 3.1.1 Software implementation . . . 18
  - 3.1.2 Categories of execution . . . 19
  - 3.1.3 Choice of metrics and calculations thereof . . . 20
  - 3.1.4 Base case hardware implementation . . . 21

3.1.5 Measurements . . . 21
3.1.6 Software optimizations . . . 21
3.2 Instruction set generation . . . 23
3.2.1 Endian swap and generalized reverse . . . 24
3.2.2 Pack . . . 25
3.2.3 Bit Field Place (bfp) . . . 25
3.2.4 Bitwise extract and deposit . . . 26
3.3 Hardware synthesis . . . 29

**4 Result** **33**
4.1 Choice of metrics . . . 33
4.2 Initial application analysis . . . 33
4.3 Instruction set generation . . . 33
4.4 Hardware synthesis . . . 35

**5 Discussion** **37**
5.1 Result . . . 37
5.1.1 Initial application analysis . . . 37
5.1.2 Instruction set generation . . . 38
5.1.3 Synthesis . . . 38
5.2 Methodology . . . 39
5.2.1 Design methodology . . . 39
5.2.2 General process comments . . . 39
5.2.3 Bit field place (bfp) . . . 40
5.2.4 Static bitwise extract and deposit . . . 41
5.2.5 Immediate bit extract . . . 41
5.3 The work in a wider context . . . 41

**6 Conclusion** **43**
6.1 Future work . . . 43

**A Full application analysis results** **47**

**Bibliography** **51**

# List of Figures


2.1 Grid of NR OFDM symbols laid out across time and frequency domains in the case of a subcarrier spacing of $30kHz$. In this case one Resource Element spans $35.7\mu s$ in the time domain and $30kHz$ in the frequency domain. . . . . 6

2.2 Description of eCPRI planes. This paper focuses on the C-plane, or eCPRI type 2. The cloud on top of the lines to M-plane and S-plane signifies that there are a large variaty of transport level protocols that can be used for the M- and S- planes that are not really relevant to this topic. . . . . 8

2.3 Example of C-plane and U-plane messages. . . . . 9

2.4 Very simplified model of a potential parser as explained by [5] . . 12

3.1 Picture from [16] (released under Creative Commons BY 4.0) demonstrating a 5-stage (32 bit) butterfly network, such as used by the `grev` and `bdep`. . . . . 24

3.2 Illustration of the `bext` instruction, simplified to 8 bits . . . . . 26

3.3 Illustration of the `bdep` instruction, simplified to 8 bits . . . . . 27

3.4 Data flow through an inverse butterfly network implementing the `bext` instruction, with control bits in red. The decoding of `rs2` into the control bits is not included in this image. . . . . 28

3.5 An 8 bit example implementation of a `bdep`/`bext` decoder. . . . . 29

3.6 Examples of extracting several packed bit fields at once with the help of the introduced `grev`, `bext` and `bdep` instructions. . . . . 30

4.1 Pie chart showing $t_{section}$ split up into its constituent categories, for the base hardware configuration running the optimized software. 34

4.2 Pie chart showing $t_{extension}$ split up into its constituent categories, for the base hardware configuration running the optimized software. . . . . 34

4.3 Line graph showing the execution time for the 3 main metrics within the data extraction category for different configurations. All values are normalized with respect to $t_{section}$, with the optimized software on base hardware configuration. . . . . 36

# List of Tables

4.1 Normalized synthesis results for area with different combinations of hardware additions. . . . . 35

A.1 The execution time for the different metrics normalized with respect to the total runtime for that specific metric, for the base hardware running with the optimized software. . . . . 47
A.2 The execution time for different categories normalized with respect to $t_{section}$ for the base hardware running with the optimized software. This visualizes the magnitude of each metric to each other. . 48
A.3 Total runtime for model application to run with different configurations. Results are normalized to the respective column for the base hardware running the optimized software. . . . . 48
A.4 Time spent in category *data extraction* for model application in different configurations. Results are normalized to the respective column for the base hardware running the optimized software. . . 48
A.5 Time spent in category *storage management* for model application in different configurations. Results are normalized to the respective column for the base hardware running the optimized software. 49
A.6 Time spent in category *sequential parsing* for model application in different configurations. Results are normalized to the respective column for the base hardware running the optimized software. . . 49

# Notation

### Abbreviations

| Abbreviation | Meaning |
|---|---|
| ASIP | Application Specific Instruction set Processor |
| bdep | binary deposit |
| bext | binary extract |
| bfp | bit field place |
| BPSK | Binary Phase-Shift Keying |
| C-plane | Control plane |
| CPRI/eCPRI | Standards used for fronthaul networks |
| FFT | Fast Fourier Transform |
| grev | generalized reverse |
| iFFT | inverse Fast Fourier Transform |
| ISA | Instruction Set Architecture |
| IQ data | In-phase and Quadrature data |
| LSB | Least Significant Bit/Byte |
| LTE | Long Term Evolution |
| MSB | Most Significant Bit/Byte |
| MTU | Maximum Transmission Unit |
| NR | New Radio |
| OFDM | Orthogonal Frequency Divisision Multiplexing |
| PRB | Physical Resource Block |
| QAM | Quadrature Amplitude Modulation |
| QFSK | Quaternary Frequency Shift Keying |
| RB | Resource Block |
| RE | Radio Equipment |
| RE | Resource Element |
| REC | Radio Equipment Controller |
| RTL | Register Transfer Level |
| RV | RISC-V |
| SCS | Subcarrier Spacing |
| SNR | Signal to Noise Ratio |
| U-plane | User plane |
| VLIW | Very Long Instruction Word |

# 1

# Introduction

Base stations for mobile networks (also known as e-nodeB in LTE or g-nodeB in 5G) are often split into a radio unit (Radio Equipment or RE) and a separate unit handling the higher level functions, and control of the RE, for which we will use the term Radio Equipment Controller (or REC). The REC does most of the digital processing of the baseband signal, and decides on what the radio unit should do, while the radio unit mostly receives/sends samples of the baseband signal and focuses on D/A conversion and analog signal handling. These two units need to be connected with a reliable, high speed, and low latency connection, and that connection is known as a *fronthaul network*. Traditionally, communication over fronthaul networks has been done with a protocol called CPRI.[4]

In 5G, there are several technology changes that cause the amount of data going through the fronthaul network to increase significantly compared to LTE. Partly this is due to wider carrier signals, ie. signals occupying larger parts of the electromagnetic spectrum, but another large contributor is a dramatic increase in the number of antennas used. The large number of antennas allow for using technologies called massive MIMO (mMIMO) and beamforming, which in turn allow us to fit more data into the same amount of spectrum than we would be able to fit without these technologies.

This dramatic increase in data rate between the REC and the RE means that, if CPRI would be used, the data rates would become too high to be manageable (potentially many 100s of Gbps in each direction[4]). Because of this, an alternative protocol, called eCPRI, was developed to allow functionality splits higher up in the network hierarchy. In other words, eCPRI allows moving some of the functionality (such as beamforming and FFT/iFFT) from the baseband unit into the radio unit. Additionally, unlike with CPRI, eCPRI can use standard Ethernet network devices in the fronthaul network. This in turn can generate cost savings for operators, as the networks become cheaper to build.[4]

However, eCPRI also requires higher amounts of, and more complex, control data to be handled in the radio unit, on top of the user plane data. This is a direct consequence of more complex functionality being located in the RE, which in turn needs control data to direct its operations. Since the control data is also complex, and standards keep developing, it can be risky to use a pure hardware solution, as they are not very flexible. At the same time there can be several Gbps of control data, which needs to be handled in real time, which in turn requires a both fast, reliable and efficient solution.

One way to deal with data quickly and efficiently can be to implement specific hardware for it. As SoCs have long development times, and requirements can change over time, this can present a big risk in case there's a change in requirements or even if there would be an oversight in the standard. Add to this that even after deploying a chip in a product, that product has a long lifetime during which even more changes might be desirable, and it is clear that a lack of flexibility is problematic. By extension it can be reasoned that there is a big advantage to the flexibility that is provided by a software solution. However, this flexibility in turn comes with a cost in efficiency, as compared to a pure hardware solution. An application specific instruction set processor, a processor tailored for a particular application, can bring a nice balance between these two, providing both the programmability of a processor and the efficiency of hardware acceleration. This thesis is about finding how the instruction set in such a processor could be chosen.

## 1.1 Problem formulation

The problem is to evaluate the use of an Application Specific Instruction set Processor (ASIP), based on the RISC-V instruction set architecture to handle eCPRI Real-Time Control data in an efficient manner. The problem includes both measuring the performance of the simple single issue, 3 stage pipelined RV32-IMC base core, that is used as the basis for the project, as well as evaluating what additions can be made to the core to improve performance. These improvements should then be evaluated against the relative increases in size and power.

The message format for eCPRI is detailed in section 3.2.3 in the eCPRI specification [4], and the different message types in section 3.2.4 of [4]. What this thesis is interested in is thus eCPRI messages of Message Type 2 (*Real-Time Control Data*), further described in section 3.2.4.3 of [4].

However, the exact format for the *payload* of these messages is not specified in [4], and simply left as "vendor specific". For the purposes of this thesis, the *Open-RAN Fronthaul Control, User and Synchronization Plane Specification, O-RAN-WG4.CUS* [3], will be used to define the payload format for the purposes of implementation and benchmarking. Of particular interest for this thesis is chapter 5 of the O-RAN specification in general and section 5.4 in particular, as this defines the format of control plane messages, what data fields exist and what they are used for.

The O-RAN document[3] is open for anyone to download and read as long as

they agree to the O-RAN adopter license agreement, but because of the confidentiality section of said agreement, the contents can not be restated here. As such this thesis will refer to [3] for exact details on the message format, and not lay them out in the report.

The motivation to have an ASIP with accompanying software implementation for handling "Real-Time Control data", as opposed to a pure hardware solution, is primarily the adaptability, flexibility and, by extension, also lower risk they provide. When put up against a software solution on a general purpose processor (which also has the flexibility advantages of the ASIP), the main advantage of the ASIP solution is better performance in relation to chip area and power consumption.

## 1.2 Question formulation

The main question formulations are thus:

- What are the alternatives for enabling faster packet processing for the given subset of packets in a baseline RV32-IMC processor?
- What combination of added instructions gives the best increase in performance in relation to complexity, power and area?

## 1.3 Delimitations

This thesis only deals with ingress eCPRI type 2 data (as defined in [4], page 17), in other words real-time control data (controlling user-plane data flows in both the uplink and the downlink) flowing from the REC to the RE. Within this data, it will focus entirely on ORAN sections of type 1 and extensions of type 1. Section type 1 deals with most normal sections of data, normal here meaning that there is no mixed numerology (ie. that the subcarrier spacing is the same for all subcarriers of the carrier in use). Extension type 1 contains beamforming weights to be used for the beamforming process.

This is motivated by the different section types being very similar in their organization and that it's necessary to keep down the total number of variables to not have to analyze too many different combinations. The parsing software will still check section type and extension type to verify that they are what they should be, so that these execution branches are still measured.

The packet parser assumes that the packet has already been identified as an eCPRI type 2 message and is available in a packet buffer memory for the ASIP in question. This is assumed to be fast memory that can be read in a single cycle. No cache is used. The processor reads and writes words and halfwords in little-endian fashion (ie. least significant bytes are located on the lowest addresses) and can read 32 bit words with any byte alignment without extra delays.

It is expected that the application, for each section extracts a vector containing the extracted fields of interest. One further assumption is that every message contains all the control data for a particular slot, and thus that information does

not have to be synchronized between two or more different messages. The extracted vector for each section should contain all the fields for that section in the packet, except for timing information. These instead decide where in the table the vectors are stored.

The extracted data is to be aligned on byte boundaries and in little endian format. Bit fields longer than 8 bits should be aligned to an even byte address. To address the possibility that some data might need to be modified, two slightly longer data fields (sectionID and beamID, 12 and 15 bits long, respectively) will both be truncated to 10 bits. Furthermore, instead of storing the number of PRBs, the last PRB a section applies to be will be stored, with $lastPRB = firstPRB + numPRB - 1$.

For beamforming weights arriving in section extensions of type 1, their extraction shall be modeled by copying them to a separate table, indexed by their respective beam ID.

To limit the scope, it is assumed that no compression is used for beamforming weights and IQ data, and test data will use 16 bits of precision for beamforming weights. The evaluation of other aspects of processor architecture, such as memory width, pipeline structure, issue width and number of general purpose registers etc. is left for future study. However new functional units may be added, as well as special purpose registers if they would be needed to implement a certain instruction.

# 2

# Theory and related works

## 2.1 LTE and 5G New Radio basics

5G New Radio (5G NR) networks, as well as 4G LTE networks, use a multiplexing scheme called OFDM (Orthogonal Frequency Division Multiplexing). With OFDM, the data on a carrier signal is spread out over several subcarrier signals at different frequencies, spaced out evenly from each other. The spacing between the subcarriers in the frequency domain (called the "Subcarrier Spacing" or "SCS") affects how long each individual piece of information being sent has to be; the larger the subcarrier spacing is, the shorter the smallest unit of time (called the "symbol") can be. [1] [2]

In LTE the subcarrier spacing is fixed to $15kHz$. In 5G NR the spacing of these subcarriers can be changed and mixed with a large degree of freedom. This is called "Mixed Numerology". With a 15kHz subcarrier spacing (what is used in LTE), this means that the shortest logical time unit (called a "symbol") will be about $71.4\mu s$ long, and the individual subcarriers will have frequency offsets of $0kHz$, $15kHz$, $30kHz$, $45kHz$ and so on. [1] [2]

With REs being encoded in both frequency and time domain, they form a grid across both domains, see Figure 2.1. The largest subdivision in the time domain is one frame, and it is $10ms$ long. Each frame in NR is split into 10 subframes of $1ms$ each. Each subframe is in turn split into 1–16 slots, depending on subcarrier spacing (SCS) (1 slot in the case of $15kHz$ SCS, and 16 slots in the case of $240kHz$ SCS). Each slot is finally divided into 12 or 14 symbols, which is the smallest unit in the time domain. In the frequency domain, 12 consecutive subcarriers form a resource block (RB). The total number of RBs on a carrier signal depends on the bandwidth of that carrier. For example a carrier of bandwidth $100MHz$ would span a total of 273 Resource Blocks at $30kHz$ SCS. One subcarrier in the frequency domain, across one symbol in the time domain, forms the smallest

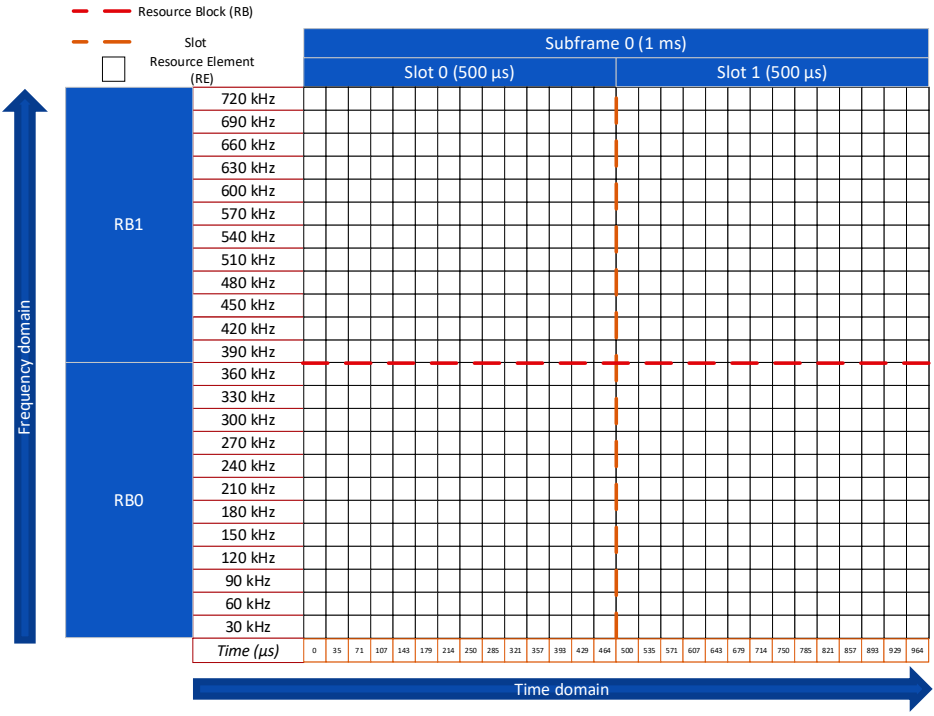


***Figure 2.1:*** *Grid of NR OFDM symbols laid out across time and frequency domains in the case of a subcarrier spacing of $30kHz$. In this case one Resource Element spans $35.7\mu s$ in the time domain and $30kHz$ in the frequency domain.*

element in this grid and is called a Resource Element (RE). Each RE can typically encode 1–8 bits of data depending on the modulation scheme used.[2]

What modulation scheme that in turn can be used depends on how strong the signal is. With a bad Signal to Noise Ratio (SNR), pure frequency shift modulation techniques like BPSK or QFSK can be used, mapping 1, respectively 2 bits per RE. With a better SNR one can pair the frequency shifting with amplitude shifting methods to achieve a keying method called QAM. Some examples are called 16-QAM, 64-QAM or even up to 256-QAM modulation, each in turn mapping 16, 64 or 256 different values in a single RE, in other words providing 4, 6, or 8 bits of data per RE respectively. [1][2]

## 2.2 Mobile fronthaul networks and eCPRI

As also mentioned in the introduction, Section 1, modern mobile base stations are often split up into at least two units, here referred to as the Radio Equipment Controller (REC) and the Radio Equipment (RE), where the latter receives IQ sam-

ples or bit data from the REC together with control information on when and how to send them. This control information can for example be timing information (which frame, subframe, slot and symbol(s) a certain piece of user data belongs to), or frequency information (which carrier signal it belongs to and which resource blocks it spans). The RE then transforms them into radio waves that are transmitted over the air. The Radio Equipment Controller is the one that does most of the logic, coding, transformations and control of what should be transmitted, how and when. Essentially the REC handles everything in OSI layer 2 and above, as well as large chunks of the OSI layer 1 functionality. [4]

With CPRI, typically only a limited set of digital functionality would be done in the radio unit. In general time domain I/Q samples would be sent over the fronthaul network. The radio unit would then implement AD/DA conversion, up/down conversion and other radio circuitry. eCPRI enables putting the protocol split between RE and REC in different places. For example it allows for sending frequency domain I/Q samples, and have beamforming and FFT/iFFT done on the RE side. This significantly reduces the amount of data that has to be sent over the network, and what was previously potentially 100s of Gbps is now in the order of a few 10s of Gbps. However, this also increases the complexity, weight and power usage of the RE. Furthermore, it further complicates the protocol and requires more clever handling of control data. [4]

## 2.3 eCPRI and O-RAN real-time control data

eCPRI is split into a user plane (U-plane), a “control and management” plane (M-plane) and a synchronization plane (S-plane). The eCPRI protocol is only used for what is there called the U-plane, and the only one that is of interest for this paper. The user plane itself does not only contain user data, but also real-time control data. The latter will in this report mostly be referred to simply as the control plane, or C-plane. eCPRI messages are either put directly in an Ethernet packet, or transported inside a UDP/IP frame. See Figure 2.2. [4]

The operation works so that first there will be a C-plane message that describes the user data that will later arrive. The information sent over the C-plane can for example be scheduling information or instructions on how to beamform data that comes over the user plane. They can also contain information regarding the compression of samples in U-plane messages. After this follows a U-plane message that for example contains the data to be sent according to the C-plane message. This is typically I/Q samples of radio data or bit data. See Figure 2.3. [3][4]

As eCPRI does not specify all the details of how C-plane or U-plane messages are formatted, and simply describe the payload as ”vendor specific“, the O-RAN alliance has specified more details on this, so that baseband units and radio units from different vendors can communicate with each other. O-RAN C-plane and U-plane messages divide up user data in sections, spanning a certain number of RBs and symbols. A number of different types of sections are also defined, of which this thesis only deals with sections of type 1, also described as ”normal“

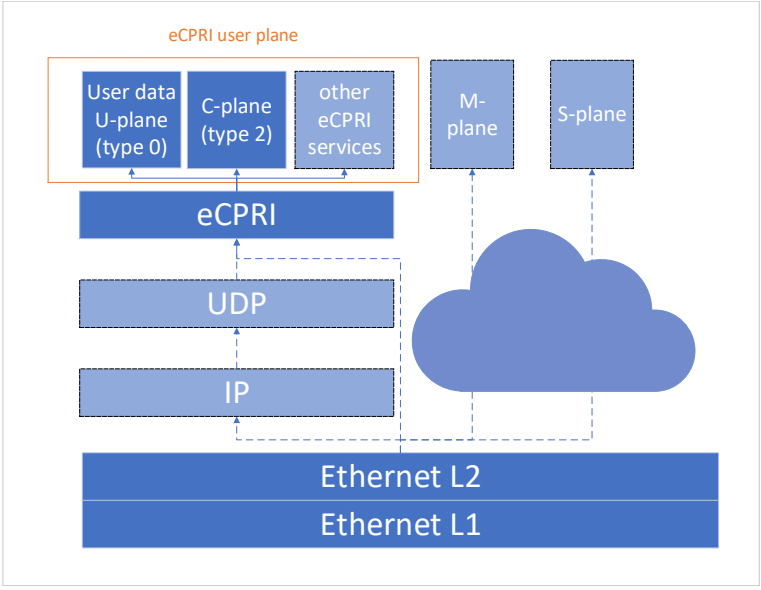


***Figure 2.2:*** *Description of eCPRI planes. This paper focuses on the C-plane, or eCPRI type 2. The cloud on top of the lines to M-plane and S-plane signifies that there are a large variaty of transport level protocols that can be used for the M- and S- planes that are not really relevant to this topic.*

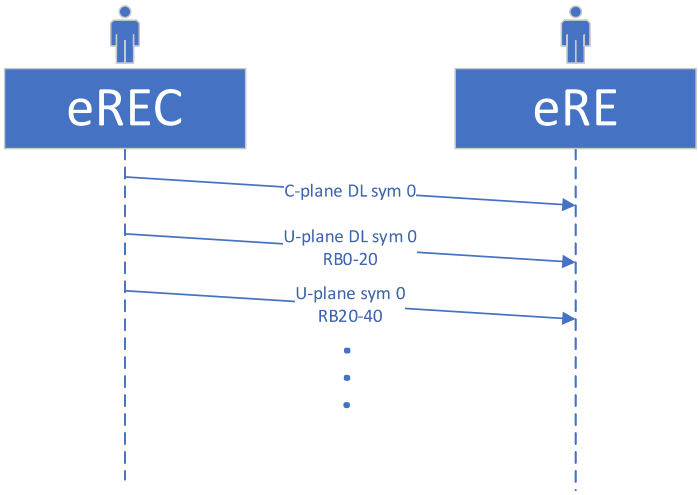


***Figure 2.3:*** *Example of C-plane and U-plane messages.*

data.[3]

A single C-plane packet contains a number of sections, and after each section an arbitrary number of section extensions can also be provided, with additional information relating to that section. There are also several types of section extensions. In this thesis we only deal with section extensions of type 1, which contain information on how beamforming should be done.[3] For details on the formatting of ORAN messages, see [3].

## 2.4 Application specific instruction set processors

An Application Specific Instruction set Processor, or ASIP, is a processor that uses the specifics of the application they are going to be used for as inputs in the design, such that it becomes specifically tailored for that task. An alternative approach would be to implement a pure hardware solution that does the same thing, and this could potentially be an even more efficient solution, but it is instead set in its functionality, leaving little margin for changes late in the development cycle, or after tape-out. The ASIP solution provides much of the flexibility of a general purpose processor while at the same time being more efficient for that application than a general purpose processor would be. It can therefore balance between the two antipodes of a general purpose processor and a pure hardware implementation in terms of flexibility versus efficiency. [10]

### 2.4.1 Design methodology

Jain et al.[10] explore a range of published design methodologies for ASIP implementations, and break them down into 5 main steps that are mostly shared by the different methodologies. These are

1. Application analysis
2. Architectural design space exploration
3. Instruction set generation
4. Code synthesis
5. Hardware synthesis

Step 1, *application analysis* consists of running a model application software and profile it to get metrics that can be used for the later steps in the process. Step 2, *architectural design space exploration* consists of identifying possible architectures for the processor and evaluate these based on the result of the application analysis. The 3rd step, *instruction set generation*, focuses on what the instruction set of the generated ASIP should be. Step 4, *code synthesis* is generating code for the application with the chosen architecture and instruction set, and step 5, *hardware synthesis* is about generating the hardware implementation.

As further described in [10], not all methodologies they analyzed implement all of these steps, and for example might consider the architecture as fixed when starting the process. For the instruction set generation two main classes of methodologies are identified: *instruction set synthesis* and *instruction selection*. The former starts with the application and from this generates an instruction set based on the required micro operations and their frequencies. The latter starts with a large set of possible instructions and selects among them to fulfill the requirements. However, Jain et al. also point out that there in both categories are flows that start with a basic set of instructions, on top of which the synthesis or selection then identifies application specific instructions to be added on top.

### 2.4.2 Packet parsing

Packets typically consist of several headers, some kind of payload and at times a post-amble. For example, an HTTP message sent over Ethernet, starts with an Ethernet header, then an IP header, followed by a TCP header and so on. One significant potential bottleneck in trying to parse these quickly is that headers can have variable length, and that the type of the next header (and consequently, where the relevant data is within it) depends on the current header. There is thus a sequential dependency from one header to the next to know where the next header will start, and how to find the type of that header. Due to this sequential dependency there is a limit to how much of the parsing that is easily parallelizable.[11]

Kozanitis et al.[11] lay out a method (called Kangaroo) that lets you traverse several of these headers at once using speculative parsing and a Content Addressable Memory (CAM), thus bypassing some of that sequential dependency. This is primarily intended for cases where several small headers appear in quick succession after each other. Gibb et al.[5] lay out design principles for both static and programmable packet parsers, for more use cases than [11], and in more detail. Zolfaghari et al.[18] also lay out a method for a programmable parser that in many regards resembles the programmable parser in [5], but without relying on a CAM.

A common thread, laid out formally in [5], is that a parser can be split up into a *header identification part*, and a *field extraction part*. The main obstacle in parallelizing the parsing is that fields in one header will determine where the next header starts. Thus this header needs to be parsed before parsing of the next header is started, as laid out in the previous paragraph. However, not all fields in a header need to be read to know where the next header will start. Typically there's either some field that describes the length of the current header, a field describing what comes in the next header, or both.

For example, an Ethernet packet might have extra VLAN or MPLS tags attached. If such a tag is attached, the size of the header is bigger, and thus this needs to first be identified to know where the next header will start, and also to know if a IPv4 or IPv6 header follows. However, to identify the location of the next header, an Ethernet switch parsing engine does not need to know source or destination MAC addresses, even if these need to be extracted to be used in other parts of the switch. Because of this, the header identifier will typically contain some sort of state machine that keeps track of the current header, and quickly identify the next one, while this information is fed to the field extractor that will do the work of extracting the fields of interest in this specific header. See Figure 2.4. [5]

## 2.5 RISC-V

The RISC-V instruction set architecture was originally designed primarily for research and educational purposes, but has since expanded its goals to be a free Instruction Set Architecture (henceforth *ISA*) also for the industry. The goal is to be an ISA that is suitable for actual implementations in silicon, not just for simulations. Furthermore, a design goal is to not target any one microarchitectural style or any one technology specifically, and instead be suited for a wide range of technologies. It is also designed to have a simple and small base implementation that then can be expanded upon with both official standard ISA extensions and unofficial custom ISA extensions to suit the needs of the implementer.[14]

### 2.5.1 The base architecture

Fundamentally, RISC-V is not a single ISA, but a family of ISAs, and there are currently four different versions of the base ISA, defined by the width of its integer registers (and by extension its address space), in combination with number

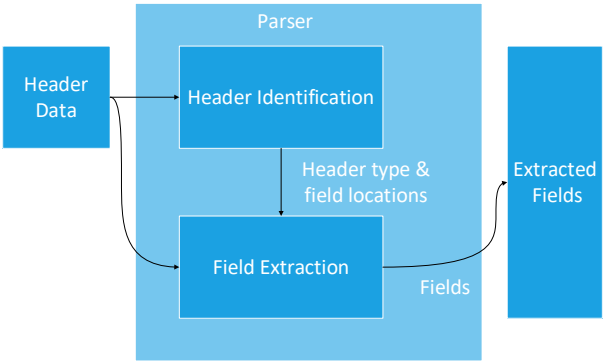


***Figure 2.4:*** *Very simplified model of a potential parser as explained by [5]*

of registers. Most prominent are the RV32I and RV64I sets, which are 32 and 64 bit versions with 32 general purpose integer registers (32 bits and 64 bits wide respectively) each. Moreover there is a RV32E set which also has 32 bit general purpose registers, but only 16 of them. The primary purpose of RV32E is for small embedded applications. On top of this there is room left within the ISA to in the future support a 128 bit version if the need for that should arise. For each of these 4 base instruction sets, there is the unprivileged instruction set [14], but also an optional privileged architecture that is described in its own manual ([15]). [14]

The base instruction set (RV32I, RV64I, RV32E or RV128I), contains instructions for loading/storing, control flow and integer handling. This includes bitwise logical operations, shifting and addition/subtraction, but importantly *not* multiplication or division. [14]

### 2.5.2 RISC-V standard extensions

In addition to the base instruction set, there are a number of so called *standard extensions* in RISC-V. These are groups of related instructions which together are ratified by the RISC-V foundation. The standard extensions should not cause conflicts with each other, and the goal is that it should be possible to pick and choose the extensions needed for a particular application and just add them on top of the base instruction set.[14]

The M extensions importantly adds multiplication and division for integers. The F, D and Q extensions add support for native floating point operations for single, double and quad precision respectively. The C extension adds a compressed subset of instructions to allow for smaller code size, see section 2.5.5. There is also an A extension that covers instructions for atomic read-modify-write mem-

ory operations, thus allowing for synchronising multi-threaded programs sharing the same memory space.[14]

On top of these, there are also many drafts for extensions that haven't yet been ratified. One of these is the B extension for bit manipulation operations that have been used as a basis for many of the instructions implemented here.[16]

The main base architecture that will be discussed in this thesis is the unprivileged RV32I combined with the M and C standard extensions, as well as parts of the draft for the B extension.

### 2.5.3 Overall structure of architecture

RISC-V can be implemented as either big-endian or little-endian, as long as loads and stores are byte-address invariant. This means that writing or reading a byte to a memory address should always behave the same way regardless of endianness used. The privileged architecture also defines a bi-endian mode. Instructions are always stored as 16-bit little endian chunks however. Earlier versions of the specification only had little-endian data in the standard implementation. In those versions, big-endian or bi-endian versions were presented as possible non-standard implementations. Little-endian implementations are the only ones that will be further discussed in this thesis.[14]

The RV32I architecture has 32 general purpose 32 bit integer registers referred to as `x0` to `x31`. Some have dedicated purposes however, but these are only defined by the ABI (Application Binary Interface, a type of protocol defining how different parts of a program communicate with each other). The CPU treats them all the same, with the exception of `x0`, which is hardwired to always read 0. The only other register to be part of the unprivileged architecture is the program counter (`PC`).[14]

This means that no one register is reserved for stack pointer or link register. There is however a standard calling convention, that defines where these should normally be located. This convention specifies x1 as the link register (holding the return address for function calls), and x2 as a stack pointer. There is one place where this makes a difference in hardware however, and this is with the compressed instruction set (see section 2.5.5), where there is an assumption that these conventions are used, allowing for compact function calls and stack operations. As further discussed in section 2.5.5, all instructions defined for the C extension can still be called with 32 bit instruction encodings from the base instruction set.[14]

### 2.5.4 Instruction format

Most computational instructions (`add`, `sub`, `mul`, `or`, etc.) work with two source registers and one destination register, but also have "immediate" versions as well (called `addi`, `subi`, `muli`, `ori` and so on) that can take immediate values up to 12 bits long. For immediate values larger than 12 bits, a `lui` instruction (for "Load Upper Immediate") is available, which takes a 20 bit immediate value, and a destination register `rd`, and loads the immediate value msb aligned in `rd`. This

way a large immediate value can be loaded with 2 instructions. Consequently computational instructions with immediate values only use 1 instruction for values less than or equal to 12 bits, but can take up to 3 instructions for values larger than that (two for loading the value, one for the execution of the desired instruction).[14]

As a load-store architecture, the only memory operations are essentially `load` and `store`, although there are a number of variations on these for different bitwidths and signedness, so in the 32 bit version the loads are `LW`, `LH`, `LHU`, `LB` and `LBU`, for "load word", "load halfword", "load halfword unsigned" and so on. The signed versions sign-extend the loaded value, and the unsigned versions zero-extend them. The same goes for store, but without the unsigned versions as signedness does not matter for the store instructions (as no sign extension is done anyway).[14]

The address to load or store from is acquired by adding the contents of register `rs1` and a 12 bit immediate value. This allows for indexed stack access (up to $2kB$), but also any kind of indexed pointer access in an efficient manner.[14]

The two classes of control transfers that are provided in the RISC-V ISA are *unconditional jumps*, and *conditional branches*. *Unconditional jumps* consist of the two instructions `JAL` ("jump and link"), and `JALR` ("jump and link register"). The former jumps to a target within a 20 bit immediate range and writes the address of the instruction following the jump to rd, typically the link register (which is `x1` according to the calling convention). This can thus be used for function calls within a 1 MiB range from the calling instruction. By using `x0` as the destination register, this can also be used for an unconditional jump without saving a return address. `JALR` instead takes a source register, `rs1`, and a 12 bit immediate added together to calculate the target instruction to jump to. Something like `JALR x0, 0(LR)` can then be used as a return instruction. It also allows for an unconditional jump within the entire 32 bit address space together with `LUI`. [14]

In addition to the unconditional jumps, there is a group of conditional branch instructions. These take two source register and a 12-bit immediate value. The same instruction will then do a comparison ('equal', 'not equal', 'less than', and 'greater than or equal') between the two source registers and either take or not take the branch based on this. As the destination address calculation fits in the same pipeline stage as the comparison, they can be done in the same instruction, unlike architectures where the branch is taken depending on status bits from earlier instructions. The branches are always `PC`-relative. RISC-V does not use delay slots, as this is claimed to make high performance implementations more complicated. [14]

### 2.5.5 C extension

There is a compressed extension of the architecture, the C extension. This is important because large code can also be slow, as it leads to more cache misses. The C extension introduces variable length instruction encodings, which unlike the base implementation which only allows 32 bit instruction words also introduces

a set of compressed 16 bit instructions. The target is to make the most common operations available in the 16 bit space. This manages to stay compatible with the 32 bit instruction encodings since the lower two bits of 32 bit instructions are always set to `11`, with all other values reserved for 16 bit instruction words. Note that all 16 bit compressed instructions all have an equivalent 32 bit instruction doing the same thing, so they both cover the same functionality. According to the RISC-V specification[14], in typical implementations roughly 50%–60% of instructions can be replaced by compressed instructions, giving rise to a reduction of code size by 25%–30%.[14]

## 2.6 Bit permutations

For a given data word of bitwidth $n$ there are up to $n!$ different ways the bits in the same word can be permuted. To do any arbitrary permutation of an $n$-bit word with only basic instructions has a complexity of $O(n)$ instructions.[13] Lee et al. demonstrate in [13] how this can be done in $O(lg(n))$ stages utilizing instructions from a range of different bit permutation networks, among them notably the so called Benes network, which consists of a butterfly network immediately followed by an inverse butterfly network, and the so called Omega Flip network. They also suggest possible CPU instructions for implementing these. For the Benes network they propose a `CROSS` instruction that would implement two stages in the Benes network. The suggested form of the instruction is `CROSS m1,m2,Rs,Rc,Rd`, with `Rs` and `Rc` being source registers and `Rd` being the destination register. `m1` and `m2` are to be constants that define which two stages of the Benes network that are to be performed, each ranging from 0 to $lg(n) - 1$. For the the Omega Flip network they suggest a similar scheme, but as there are only two distinct stages, instead of $lg(n)$ in the case of the Benes network, fewer bits are needed for choosing stage.

In [17] they also demonstrate how the Benes network can be split up into a butterfly and and inverse butterfly network, and how each of those halves can be implemented in whole in a single cycle with a critical path shorter than an ALU of the same bitwidth. It is thus possible to do an entire half of the network in a single cycle without impacting the critical path of a CPU. They consequently suggest implementing two instructions: `BFLY` and `IBFLY` that together can perform any arbitrary bit permutation in two cycles. They also demonstrate that in a multi issue architecture (superscalar or VLIW), it's possible to get a throughput of one permutation per cycle as long as they follow the same pattern, as long as a butterfly and a reverse butterfly can be issued in parallel.

Harb and Chavet[6] define a formal method for calculating the control words for any arbitrary permutation.

## 2.7 Extracting and depositing bits in a word

Several architectures have implemented `extract` and `deposit` instructions, such as the HP precision architecture [12] and the Intel Itanium architecture[9], that

can either extract or deposit a continuous bit field from an arbitrary position within a word.

Hilewitz and Lee[7] describe parallel versions of these instructions, `pdep` and `pext`. The `pext` takes bits in inside a word, that do not need to be contiguous (defined by a bit mask) and collect them in the same relative order on the LSB side of the result. The parallel version of the deposit instruction does the opposite, and moves bits from the LSB side into positions defined by a bitmask inside the destination word in the same relative order. They also demonstrate how `pdep` can be implemented with a butterfly network and `pext` with an inverse butterfly network.

In [8] it's also demonstrated how a decoder for generating the control signals for a `pext` instruction from the bitmask can be implemented. This is done in two steps. First a parallel prefix population count is done for the bit mask. This means that for each bit in the mask, a count of the number of 1s that are on the LSB side of that bit, including the bit itself, is made. The second step is using the outputs from the first stage to feed a group of what they call 'left rotate and complement' circuits, that rotate a word leftwards with the input number of steps, but complements the bits as they are wrapped around to the other side. The exact same decoder can be used for both `pdep` and `pext`, but as the stages are in the reverse order, the word order of the control bits need to be reversed as well.[7]

The RISC-V bit manipulation extension lays out 3 main instructions to deal with bit permutations. These are the rotating shift, generalized reverse and the generalized shuffle instructions. [16]

# 3

# Method

As a basis for the evaluation, a software implementation of a parser for O-RAN messages was made. This was implemented to measure the desired metrics (see section 3.1.1). Then the ASIP designer tool suite from Synopsys was used to generate a cycle accurate instruction set simulator based on the base core implementation (see section 3.1.4). This was used to run the model software implementation.

Considering the design methodologies common for ASIP implementations described in section 2.4.1, the first step must be the application analysis, and this is further described in sections 3.1.1 and 3.1.6. As a delimitation of the project is to largely explore things from the base implementation the architectural design space exploration is largely left out. The existing single-issue 32 bit RISC-V core is used as the basis. The next large step is the instruction set generation.

Again as a requirement, the instruction set is based off RISC-V, and thus all the RV32-IMC instructions will be a part of the instruction set. As for the two main categories of methodologies that Jain et al.[10] identify for generating the instruction set, initially the instruction selection method will at least be approached first. Being new to the topic it seems like a reasonable way to further get familiarized with possible ways to approach instruction generation. Also there exists a number of standard extensions, as well as drafts for standard extensions that provide a large superset of instructions to pull from. Especially the draft for the bit manipulation extension [16] provides a number of instructions that should be relevant for this application.

As for the last two major steps identified in [10], code synthesis and hardware synthesis are largely handled by the ASIP Designer tool suite. Once the appropriate infrastructure is in place, with this tool it is relatively easy to run the application (and measure its performance) on a cycle accurate simulator, as well as getting quick feedback on the hardware impact of a single functional unit. As such, the approach chosen was to select one instruction at a time from a set of

possible instructions that might be useful, and evaluate their individual impact on performance in an iterative and agile manner, rather than using a waterfall approach.

## 3.1 Application analysis

### 3.1.1 Software implementation

As a basis for the evaluation a software implementation of a parser for O-RAN messages was used. The application assumes that an eCPRI message has been identified as an eCPRI type 2 message (C-plane) and that this message is located in local memory. The message is assumed to be in a known position in memory, and after boot up the program will begin parsing the message.

Initially in the development process, a lot of time was spent on researching use cases of eCPRI and O-RAN, what could be a reasonable usage in radio products in the future, and what parts of it are most central. This then served as a basis for deciding how it would be reasonable to implement the model application, what fields to extract and how to extract them. Also researched was typical design methodologies for ASIPs, to aid in laying out the approach. Further researched was how packet parsers are typically built in software defined network devices, and whether any of that research could be applied for this scenario.

To model possible behaviour in a radio unit receiving this message, and wanting to extract its data, the relevant fields in the message will be uncompressed and extracted by the program into a separate area of memory. In other words, it will move them to be located along byte boundaries instead of as packed bit fields. The output is stored in little-endian format for the same reasons that RISC-V originally chose this, namely that it is the format that dominates commercially (see [14] p. 9). Also, considering the incoming data is big-endian it makes for a nice worst case scenario to explore. As a single message can not contain information from two different slots, the extracted data will be stored in a buffer big enough to represent information from 32 sections per symbol, for up to 14 symbols.

To account for the possibility that incoming data sometimes might need to be processed slightly before being output, there was also a calculation stage added to the extraction. This is modeled by assuming that instead of outputting the first physical resource block of a section and the number of resource blocks that section contains, we want to calculate the index of the last PRB of that section (still stored together with the index of the first PRB). Moreover we will assume that the section ID and beam ID are truncated to 10 bits. The sections will be stored in a table with homogeneous entries for each section, in symbol order, and within each symbol in the order of which they appear in the incoming messages.

This is modeled by storing a table containing extracted information about sections in memory. To model the processing parts, some of these fields will be slightly modified. Instead of storing the first PRB a section contains, and the number of PRBs, we will store the first and last PRB in that section. Moreover we will assume only the 10 LSB bits of the section ID and beam ID are relevant, and that beam ID is offset by 5, ie. 5 will be added to every beam ID before using it.

This is to model some kind of field data processing. The sections will be stored in symbol order and a separate table will keep track of the first and last line for each symbol, and also has to be managed. Beamforming weights will be stored in yet another table indexed by the beam ID.

To limit the scope, the model software will only actually process section type 1 messages and extensions of type 1 (as per defined in the O-RAN fronthaul specification [3]), but it will still check these values and generate error messages in case they are something else, to make sure that these parts of the processing are actually parsed and not optimized away.

The test software also generates test data for itself and measures its own performance in number of cycles it takes to parse packets of increasing size (increasing the number of sections). Test cases are made for each section containing no extension, every two sections containing an extension and every section containing an extension.

### 3.1.2 Categories of execution

To make it more visible that the application performs several tasks and that these have different requirements, the measurements have been divided up into these three categories:

- Sequential parsing
- Data extraction
- Storage management

To begin with, it has to *parse* the message. In this context this means that for every part of the message that is known to the parser at a given point in time (the current header type), it has to extract the required information from there to know what information follows (ie. the next header type). In the case of a section 1 header in an ORAN C-plane message for example, this would involve checking the `ef` flag to know if the following header is another section 1 header or an extension. If you have an extension, you need to check both extension type and extension length to know what data will follow, and you need to check the `ef` flag to know if another extension follow or if you're back to processing the section you were processing.

The second part the application is designed to do is to *extract* information from the message. I.e. read pieces of information from the message that would be of interest to use for the application. In the case of an IPv4 header this could for example be the source and destination IP addresses. Included in this part is also if there needs to be some for of modifications to the extracted values, such as the above mentioned truncations and offsets, as well as the calculation of the last PRB.

The third part is *storage management* of this information. In the case of this implementation it has to do with keeping tables of the extracted information updated, such that they can be easily searched by later processing stages needing to process this data. This includes calculating what address the data should be

stored at, keeping pointers of this table updated and managing information on how many sections there are within a given symbol.

### 3.1.3 Choice of metrics and calculations thereof

It was hypothesized that the processing time per packet could be modeled as

$$t_{packet} = n \cdot t_{section} + m \cdot t_{extension} + t_{overhead} \tag{3.1}$$

Here $t_{packet}$ is the time to process the entire packet, $t_{section}$ is the time to process one section contained in the message (without extensions), $t_{extension}$ the time to process one extension (of type 1 as this is the only type tested in this application, see section 1.3) and $t_{overhead}$ the overhead for each incoming packet. Further, $n$ is the number of sections the message contains in total, and $m$ the number of these which have an extension attached to them (and thus it's also true that $m \leq n$).

To evaluate whether these are reasonable assumptions to make, test messages were generated containing values in the range from 1 to 12 sections per message in two configurations of either having each message contain no extension or have each section within the message contain one extension. The number 12 was chosen because it was the largest number of sections that, with one extension per section, would fit within the maximum transmission unit (MTU) of an Ethernet frame (1500 bytes). Thus, $t_{section}$ and $t_{overhead}$ can be calculated from the parsing times of the former configuration. If the time to parse one message with $i$ sections and no extensions is $t_i$, and the time to process one message with $i$ sections that all have one extension each, is $t_e i$. They were calculated as follows:

$$t_{section} = \frac{t_{12} - t_1}{12} \tag{3.2}$$

$$t_{overhead} = t_1 - t_{section} \tag{3.3}$$

$$t_{extension} = \frac{t_{e12} - t_{e1}}{12} - t_{section} \tag{3.4}$$

To validate these results values were calculated for what the results should have been for the the packets corresponding to $t_1..t_{12}$ and $t_{e1}..t_{e12}$, using the calculated values fed back into equation 3.1, and for each of these calculate the error, $e$:

$$e = t_{measured} - t_{calculated} \tag{3.5}$$

The worst and average of all the errors were then evaluated. Early testing (see results, section 4.1) showed that this was a very accurate model, and thus the chosen metrics to measure for the profiling were time per section ($t_{section}$, in cycles), time per extension ($t_{extension}$, in cycles) and overhead per packet ($t_{overhead}$, also in cycles).

### 3.1.4 Base case hardware implementation

As a base case for comparison an implementation of a RISC-V 32IMC core in ASIP designer is used. A cycle accurate instruction set simulator was then used to run the model software. The simulator takes into account all internal control signals, hardware conflicts and potentially generated stall signals. No cache memory is used, but the entire memory available to the processor is assumed to be fast memory, and arranged in a Harvard architecture, with separate program and data memories. There are thus no structural hazards due to simultaneous data and program memory access.

### 3.1.5 Measurements

Measuring the chosen metrics was not straight forward, as there was heavy use of inlining. This meant that such performance could not accurately be measured function by function, as the entire parsing ended up as a single function in the compiled code. Removing the inlining to break the application into several functions for profiling would have had significant performance impacts due to register spilling, and would also mean less ability for the compiler to eliminate common subexpressions and do other compile time optimizations. To address this, additional versions of the software were written, where chosen parts of the packet processing were skipped. More specifically, there were runs of 3 versions of the application: one of the full application, one with storage management left out, and one with both storage management and data extraction left out. The time difference between these versions was then measured to calculate how much time the omitted tasks added to the total processing time.

To measure time for storage management, a version of the software saving all vectors to a single static location in memory (instead of saving to a location calculated from the packet contents) was made, and the difference in time of this, compared to the full implementation, was used to determine the extra time caused by storage management. To measure the time of *extraction*, another version of the application was made, where the values of sections are not extracted, and where the weights of extensions are not copied. The time difference between this version and the previous one was then the extraction time. It then also follows that the remaining time in that version was the sequential parsing time.

### 3.1.6 Software optimizations

To make sure that the measurements of the performance increase due to hardware optimization indeed shows performance increases that are caused by actual advantages in hardware, and not the new instructions "masking" a suboptimal software implementation, there was first an initial software optimization step. This way the comparative studies are made between a finetuned software implementation running on the base hardware and a finetuned implementation running on the specialized hardware, so that it should hopefully represent the higher end of what each hardware implementation is capable of.

The application was compiled, and then profiled to determine how many cycles were spent in different parts of the program. Then the disassembled code for the core loops of processing packets was studied and optimizations were done to the code base based on those results. This was done iteratively until there weren't many obvious ways to speed up the performance. Following is an explanation of a large, but not exclusive, part of what was done.

To begin with, an initial step was to make sure that inlining was used where it was possible. Achieving this includes declaring functions to be inlined in header files and making sure that functions that are only needed in a certain file are declared as static. Also a manual check was done that there weren't any obvious large amounts of unnecessary data copies or superfluous initialization of memory. Furthermore, hints were given to the compiler that error handling cases are rarely taken branches to make sure cycles are not wasted preparing for these cases until they actually happen.

It was also realized that the copying of beamforming weights was done in a suboptimal manner. In the initial application, the copying of beamforming weights followed the following process for each weight (with a total of up to 16 weights per message).

```
load real part (16 bits) (2 cycles)
byte shift (5 cycles)
write real part to destination (1 cycle)
load imaginary part (16 bits) (2 cycles)
byte shift (5 cycles)
write real part to destination (1 cycle)
```

This runs a total of 16 cycles. The two cycles for load from memory is due to a data hazard when the loaded value is used by the immediately following instruction. As these 16 bit values are located next to each other in memory this can however be significantly sped up by loading them both at once as a single 32-bit value, byte swapping them in parallel and then writing them back to memory in parallel with a single store word instruction. By also unrolling the loop slightly (ie. each loop iteration works with 2 values at a time, interleaved), this avoids the data hazard in the load instruction. Thus the optimized flow of the weight copying is the following. Normally this would be interleaved with another copy, but that has been left out here for simplicity and clarity. It presumes a source pointer is located in x10, a destination pointer is located in x11 and a mask of 0xFF00FF00 is located in x12:

```
lw x13, 0(x10)
and x14, x13, x12
srli x14, 8
slli x13, 8
and x13, x12
or x13, x14
sw x13, 0(x11)
```

This brings the time to copy a weight pair down to a total of 7 cycles, a reduction by 56% compared to the initial implementation!

## 3.2 Instruction set generation

Based on the results in the initial profiling and optimization stages, hardware modifications were made to facilitate quicker execution of the model software. As an important goal for using an Application specific processor to execute the task in software rather than in hardware, is flexibility in responding to changing specifications or needs, the chosen instructions have been purposefully limited to be somewhat general in their function. Another design constraint used during the evaluation was to try to stay as consistent as possible with the RISC-V instruction set, and try to not modify the overall architecture design of the processor core (ie. change number of pipelining stages, changes in memory architecture and so on).

Based on these constraints, a good source to start from was the proposed RISC-V bit manipulation extension draft.[16] Initially the most obvious step to go was implementing a byte swapping instruction (described below, see section 3.2.1), that would especially help with extracting beamforming weights from extensions, but also with extracting any bit field that crosses a byte boundary within the incoming message.

From the results of the early measurements and software optimizations (see section 4.2), it was apparent that a majority of the time was spent in the data extraction part of the application. A decent portion of the handling of a section is also spent doing storage management, but data extraction was a larger part, and much more easy to find ways of accelerating as well. This is thus where the majority of the instruction set generation efforts were put.

Initially an intention had been to apply a range of techniques used by software defined network parsers such as those suggested by for example [5] or [18], but as these focus on accelerating the sequential parsing portion, which makes up only 13% of section processing and 9% of extension processing on the non-modified base hardware, this did not seem like the most efficient approach.

As the model software is especially focused on dealing with bit fields, further instructions that were evaluated were targeted at constructing, reading and modifying these. Initially the pack instruction was chosen since it should be simple, cheap, have obvious use cases and should bring some performance improvement, even if the performance win is relatively small for each individual use. See section 3.2.2.

In addition to the pack instruction, the bit field place instruction was chosen since a non-trivial part of the cycles spent per loop are spent constructing bit fields. See section 3.2.3.

To complement the bit field place, bitwise extract and deposit were chosen to help with extracting bit fields spread out in a word. These instructions were also seen as useful since the deposit instruction can be efficiently used to construct new words from the data that has been extracted, as long as the bit fields are still arranged in the same order. For more details, see section 3.2.4. Each implemented group of instructions are further detailed in their own section below.

To use the implemented instructions in the parsing software, intrinsic functions were made available for each instruction, that can be called directly from

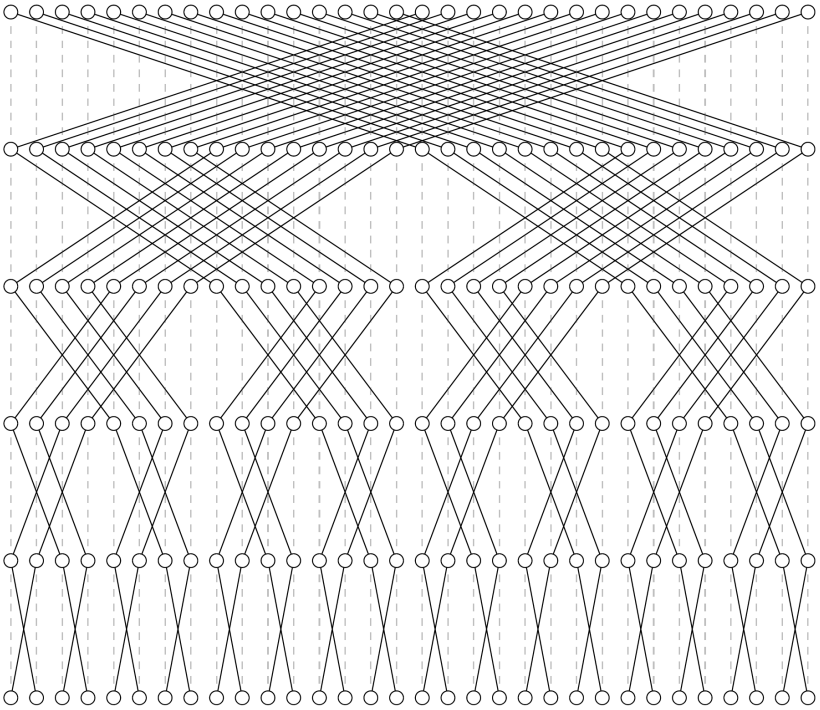

***Figure 3.1:*** *Picture from [16] (released under Creative Commons BY 4.0) demonstrating a 5-stage (32 bit) butterfly network, such as used by the* `grev` *and* `bdep`*.*

the code. This means that the programmer must know of the existence of the instructions to make use of them, and can not rely on the compiler to automatically introduce them into the code.

### 3.2.1 Endian swap and generalized reverse

As laid out in [16], a byte swap can be realized with the help of a butterfly network, that can also be used to implement a generalized reverse instruction. In accordance with [16] this instruction was realized in the form of a butterfly network (see figure 3.1). Two versions of the instruction were implemented, one with the second argument as a register (`grev`), and one immediate version (`grevi`).

The first argument is the source word, that should be altered, and the second argument then works as a mask to determine what stages of the butterfly network where there should be a switch, and which ones where there should not be one. In the case of this instruction each stage either entirely lets all bits through or swaps all of them, and thus only the bottom 5 bits of the second argument are used, even if they can contain 32 bits in the case of `grev` and 12 bits in the case

of `grevi`. Each of these bits indicate whether the corresponding stage of the butterfly network should swap or let through the bits straight for that stage.

A different way of viewing this is to think of each stage of the butterfly network doing an xor operation on one bit in the bit index of the incoming word. For example the first stage (stage 4), can be seen a bit index operation that does an xor on bit 4 of the index. So if $rs2[4] == 1$ this means that bits 16–31 will end up in positions 0–15 and the other way around. Since two bits are always switching position with each other and always remain in the destination word, this is a reversible operation.

With such an instruction, not only can the byte order be reversed in a 4 byte word (with $rs2 == 24$), but for example only swapping the bytes in each halfword of a word is also possible ($rs2 == 16$), which is of use in beamforming weight extraction as described in section 3.1.6. With such an operation beamforming weight copying core loop as described in section 3.1.6 can be further simplified to the following, bringing down the number of instructions per weight to 3. Here the interleaving of two copies to solve the data hazard caused by using a value in the instruction immediately following a load, described further in section 3.1.6, is demonstrated as well, as the total number of operations is so low.

```
lw x13, 0(x10)
lw x14, 4(x10)
grevi x13, x13, 16
grevi x14, x14, 16
sw x13, 0(x11)
sw x14, 4(x11)
```

### 3.2.2 Pack

A `pack` instruction, as well as `packu` and `packh` instructions analog to their definitions in [16] were implemented. `pack` packs the lower half of each of the source registers like this: $rd = rs1[15:0] :: rs2[15:0]$. In the same manner, `packu` packs the upper halfwords: $rd = rs1[31:16] :: rs2[31:16]$, and `packh` packs the lower byte of each register into the lower halfword of the destination register, leaving the upper halfword as 0: $rd = rs1[7:0] :: rs2[7:0]$.

### 3.2.3 Bit Field Place (bfp)

A bit field place instruction was implemented, also in accordance with the definition in [16]. This takes a specially configured control word that packs data up to 16 bits long, an offset and a length, together in a single 32 bit register. The data being located in the lower half of the word and the rest in the upper half of the word means that as long as the upper half is already constructed beforehand, they can be combined quickly with the help of a single pack operation.

The usage of these instructions in the model software were always done together with the pack instruction, as it's relied on to construct the control word. The way this is utilized in the model software is in building the output word.

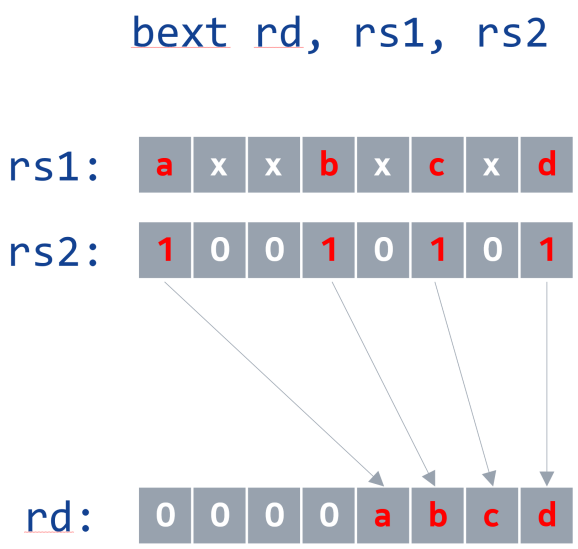


***Figure 3.2:*** *Illustration of the* `bext` *instruction, simplified to 8 bits*

As location and length are known at compile time the upper half of the control word can be preloaded into memory. In the core loop, the following code is used to update a bitfield in an already existing structure. This assumes that the upper half of the control word is located at location `offs` on the stack, the field to be written is located in `x11` and the structure to be updated is located in `x10`:

```
lw x13, offs(SP)
pack x13, x13 x11
bfp x10, x13
```

### 3.2.4 Bitwise extract and deposit

As many small data fields of interest can be located in a single 32 bit data word, but possibly only a few of them being of interest, a bitwise extract functionality can be used to retrieve several of these at once. Paired with a bitwise deposit instruction these can then also be spread to even byte boundaries, or other locations within an output word if that suits the desired output.

Bitwise extract and deposit functions were implemented, in accordance with the ones suggested in [16]. Thus the bitwise extract was implemented as

```
bdep rd, rs1, rs2
```

where the source material is located in `rs1`. Then the bitmask in `rs2` indicates which bits in `rs1` to be selected and extracted, and these are collected in the same order but LSB justified, see figure 3.2. The bitwise deposit is the inverse function of the bitwise extract, so

```
bext rd, rs1, rs2
```

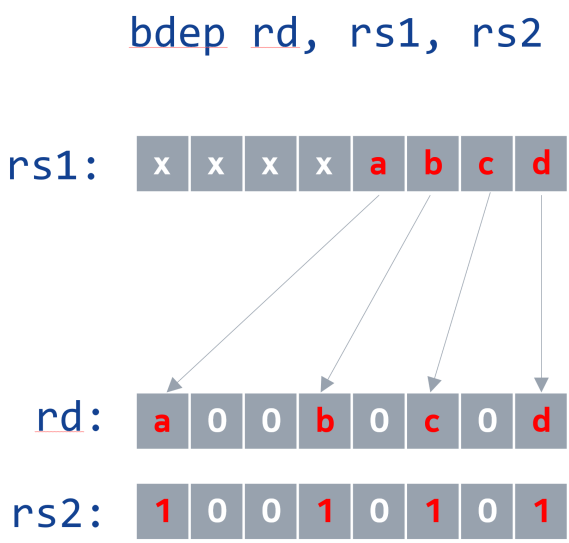


***Figure 3.3:*** *Illustration of the* `bdep` *instruction, simplified to 8 bits*

takes a number of LSB justified bits from `rs1` corresponding to the number of set bits in the bitmask in `rs2`, and spreads them out to the respective location of those set bits (still in the same order as they appeared in the source word), see figure 3.3.

They were realized with the help of a butterfly network (in the case of `bdep`) and an inverse butterfly network (in the case of `bext`), as laid out by Hilewitz and Lee in [7]. Both are 32 bits wide, and consequently have 5 stages each. See also figure 3.1 on page 24 for an example of a full 32 bit butterfly network. There is also an additional bitwise `and` step with the bitmask, to mask out the data bits that are not of interest (ie. those for which their corresponding bit in the bitmask is `0`).

For a simplified 8 bit example of the data flow through the inverse butterfly network during the execution of a `bext` instruction, see figure 3.4. Red paths represent the path of selected bits, dashed lines bits that are not swapped in that stage (i.e. their corresponding control bit is `1`) and unbroken lines are used for bits that are (with their corresponding control bit being `0`). Note how in the first stage, each pair of bits has the option to be swapped a single step over, in the second stage 2 steps over, in the third stage 4 steps and so on. Note how a network that is $n$ bits wide will have $\frac{n}{2}\log_2(n)$ control bits, i.e. $\frac{n}{2}$ bits per butterfly stage. For our 32 bit network there are thus a total of 80 control bits needed.

The decoder is constructed according to the method laid out in [8]. This method consists of two parts, a so called parallel prefix population count and a collection of "left rotate and complement" (LROTC) circuits. The "parallel prefix" population count counts the number of set bits in the bitmask on the LSB side of the bit in question, including that bit itself, for each of the indexes in the bitmask.

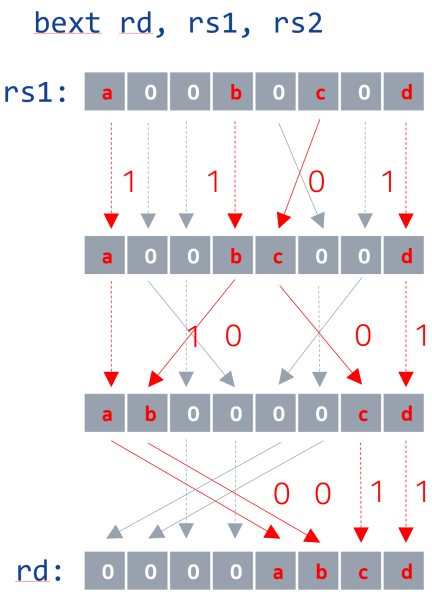


***Figure 3.4:*** *Data flow through an inverse butterfly network implementing the `bext` instruction, with control bits in red. The decoding of `rs2` into the control bits is not included in this image.*

These individual values are then fed into the LROTC blocks. These start with all `0`s and rotates the value left by the amount coming from the prefix population count, but upon wraparound each bit is inverted. They are of different bitwidths for the different stages of the networks. For the first stage of the butterfly network (where one pair of 16 bits each can swap between each other), it is 16 bits wide. In the second step, where two pairs of 8 bits each, there are two LROTC blocks of 8 bits each and so one. For the last step, the LROTC block would be 1 bit, but this is functionally equivalent to just taking the LSB of the incoming value, so the LSBs are just concatenated straight off, bypassing the LROTC stage for that stage.

An example of how this process looks like in the earlier 8 bit examples is presented in figure 3.5. As the bdep and bext operations are essentially inverse operations of each other, the same decoder can be used for both instructions, although of course the order in which each of the control words are fed to the corresponding butterfly networks has to be reversed. For a more thorough explanation of why this works, how to select which prefix population count value for which LROTC block, and other implementation details, see [8].

The circuit is pipelined in two stages. In the first stage the control words are calculated and in the second one these are used to pass the source word through the butterfly network.

These instructions were used when there was a need to extract several bit fields from the same 32 bit chunk in parallel. This was done by first loading the entire 32 bit chunk into a register with a normal load instruction. Since incoming

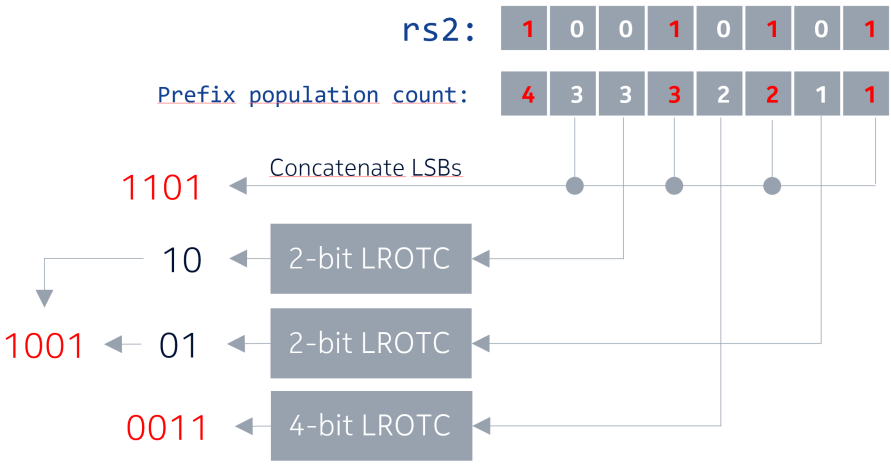


***Figure 3.5:*** *An 8 bit example implementation of a* `bdep/bext` *decoder.*

data is big endian, but the processor used is little endian, initially the data would have to be flipped with the help of a `grevi` instruction (see section 3.2.1 on page 24) to make sure that bits are in the right order relative to each other. This would be followed by a `bext` instruction to LSB justify the parts of interest to be extracted. When this is done they can then be aligned to byte boundaries with the help of a single `bdep` instruction and a well chosen bitmask.

Examples of extraction of 4 packed bit fields can be found in figure 3.6. Here 4 data fields of lengths 4, 10, 5, and 13 bits respectively exist in the incoming packet. To fit everything in one extraction, and to fit each value into a byte each, fields 2 and 4 are truncated to 8 bits in the first example. If we would want all values at full length on byte boundaries, more than 32 bits would be needed to store the output, and thus two more instructions would be needed (in addition to an extra store instruction of course). If say, field 2 is not interesting for a particular application however, then fields 1, 3, and 4 can be extracted, field 4 occupying the first two bytes, then followed by 3 and 4 occupying one byte each. This scenario is also demonstrated in figure 3.6.

## 3.3 Hardware synthesis

Finally setting up a synthesis flow for the generated synthesizable RTL code was necessary for comparing increases in area. This was done with the help of Synopsys Design Compiler and a $7nm$ technology library. A sweep of runs were made with frequency increments in steps of $50MHz$, until it failed to meet the desired target, to find the top speed for the respective designs. This was done for a range of hardware configurations. For each version, it was noted whether the speed

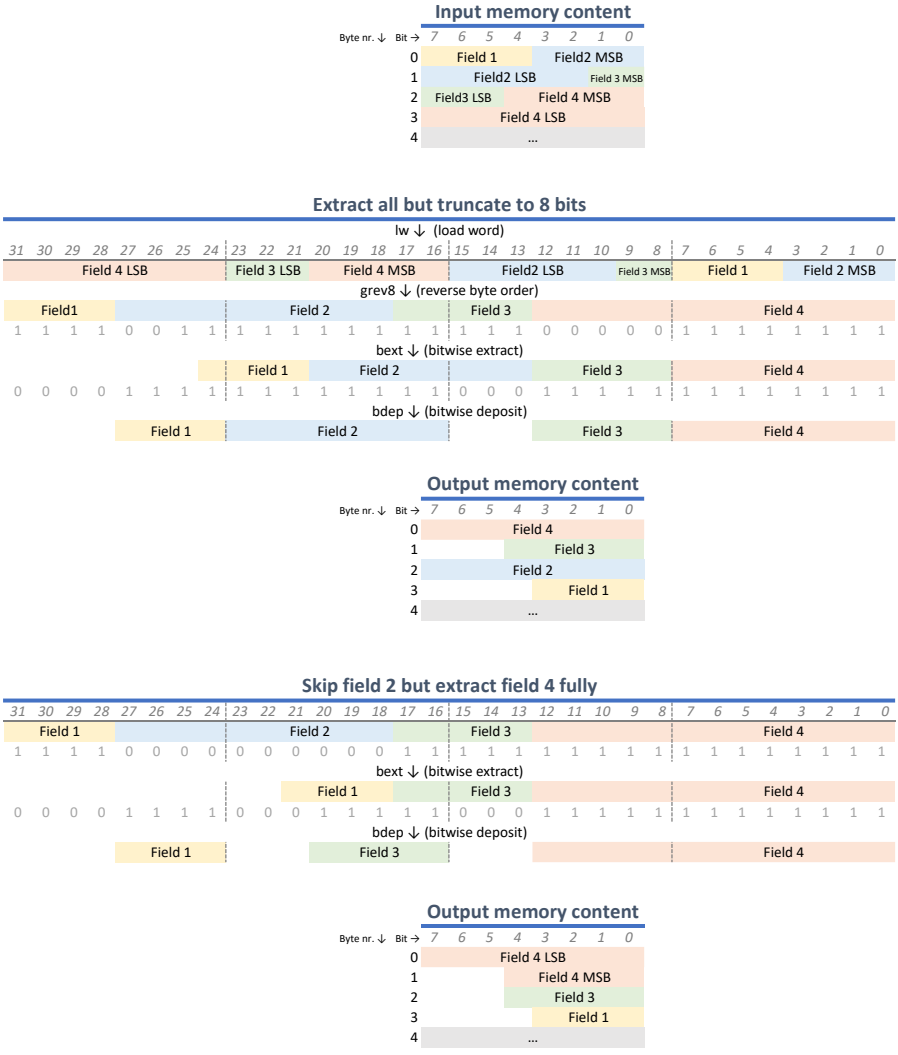


***Figure 3.6:*** *Examples of extracting several packed bit fields at once with the help of the introduced* `grev`*,* `bext` *and* `bdep` *instructions.*

that they met was lower than the base core, and finally area was compared when compiled for the highest speed that they all met.

Synthesis was done only on the core, not including memories, as this simplified the process, and went significantly faster. This was seen as reasonable since all the work was done on the core itself and the memory interfaces stayed the same, and none of the added instructions deal directly with memory.

# 4

# Result

## 4.1 Choice of metrics

In section 3.1.3 the choice of metrics is described. This is motivated by early measurements where the average and max error that this method generates for up to 12 sections per message was measured.

In all of the measurements done the maximum absolute error (measured as described in Section 3.1.3) was 0, and the average error was 0, so it was determined that these were valid metrics to use for messages of any size.

## 4.2 Initial application analysis

Detailed tables for the application analysis can be found in Appendix A (page 47). Table A.1 shows the breakdown of each metric into the three measured categories, all values normalized with respect to the total run time for that metric. A pie chart for the division of $t_{section}$ is also available in Figure 4.1 and the same for $t_{extension}$ can be seen in Figure 4.2.

Table A.2 instead shows the relative sizes of the different metrics, by showing $t_{extraction}$ and $t_{overhead}$ normalized with respect to $t_{section}$ for each category of execution, as well as total run time.

## 4.3 Instruction set generation

For evaluation of hardware optimization effects on execution time, the results for each hardware configuration, and each category are displayed in tables A.3–A.6. For the base hardware, results are shown for both the initial software implementation before optimizations, as well as the software optimized version (also running

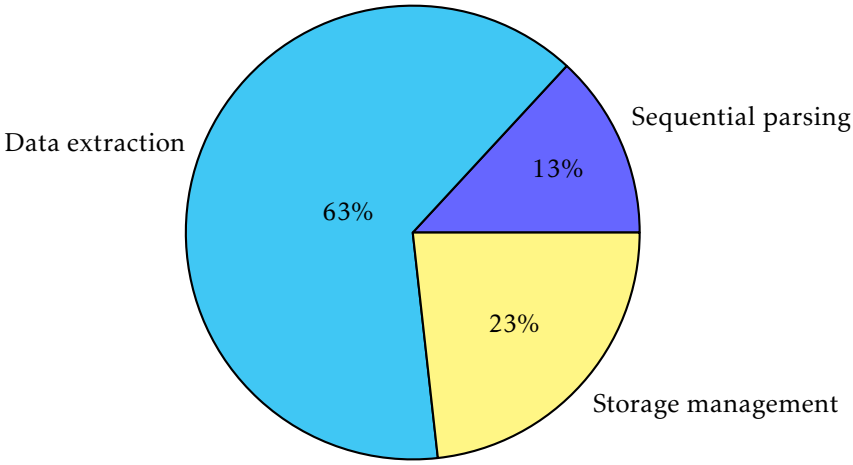


***Figure 4.1:*** *Pie chart showing $t_{section}$ split up into its constituent categories, for the base hardware configuration running the optimized software.*

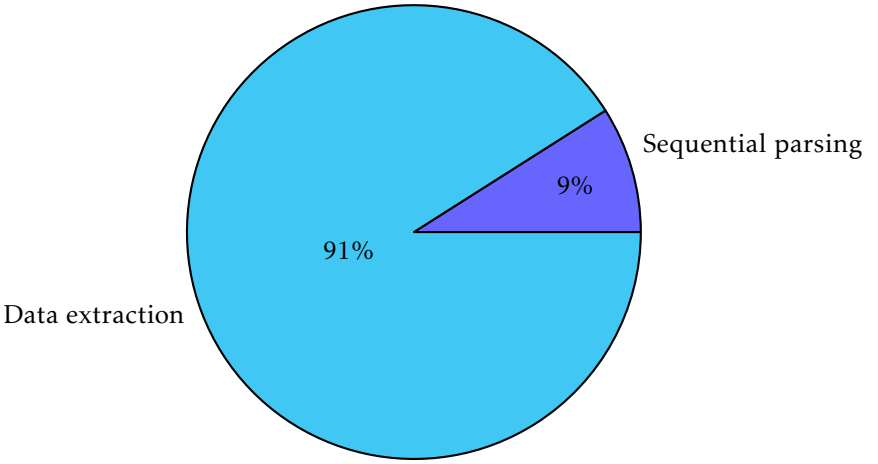


***Figure 4.2:*** *Pie chart showing $t_{extension}$ split up into its constituent categories, for the base hardware configuration running the optimized software.*

on the base hardware). Table A.3 displays the total running time, and tables A.4–A.6 are for each of the categories measured, "sequential parsing", "data extraction" and "storage management". These results are normalized for each column relative to the same column for the configuration running optimized software on the base hardware.

The results related to *data extraction* are also summarized in Figure 4.3, showing how time extracting data per section and extension changes with different configurations.

## 4.4 Hardware synthesis

Results for area are shown in table 4.1.

| *Configuration* | *area* |
|---|---|
| **Base hardware** | 1.00 |
| With `pack` | 0.98 |
| With `grev` | 1.01 |
| With `bdep/bext` | 1.16 |
| with all extensions | 1.17 |

***Table 4.1:*** *Normalized synthesis results for area with different combinations of hardware additions.*

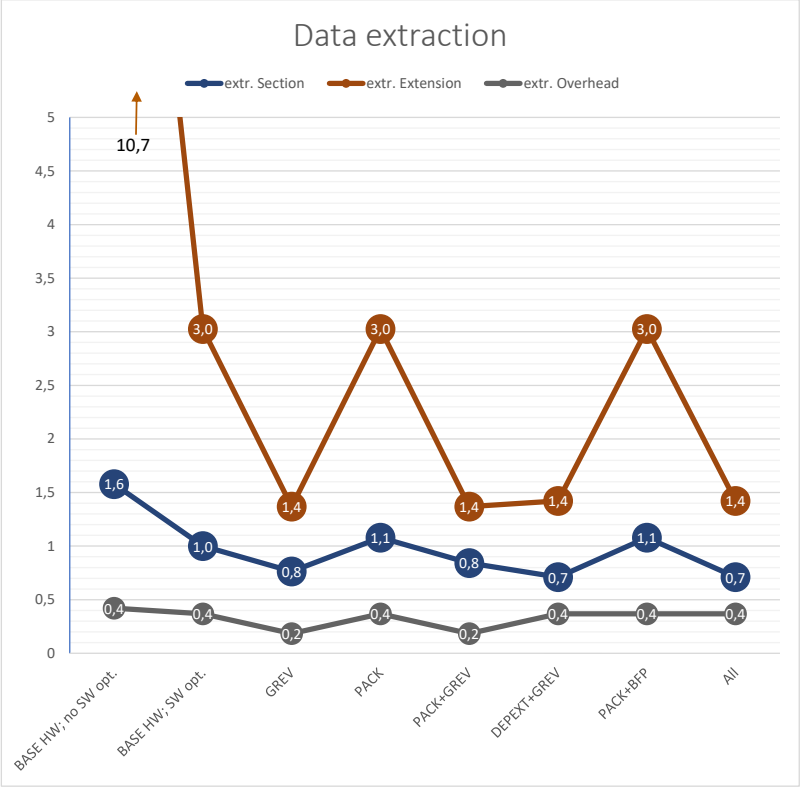


***Figure 4.3:*** *Line graph showing the execution time for the 3 main metrics within the data extraction category for different configurations. All values are normalized with respect to $t_{section}$, with the optimized software on base hardware configuration.*

# 5

# Discussion

## 5.1 Result

#### Choice of metrics

At first glance it might be slightly surprising that the maximum error (between the model and measured values) is 0, or in other words that the model presented in section 3.1.3 would be perfect in relation to all measured sizes. Upon inspection of the generated code, it is however not that remarkable, as every section and every extension processed have mutually exclusive parts of code that run. For every package, there will be some overhead processing and then it will loop through the same sets of instructions for every section. For each section it then in turn checks for extensions, and optionally execute the code to handle that extension. The same goes for extensions.

Some important aspects to also note is that it is a very simple core, with single issue, in order execution. As the memory model also does not account for arbitration in the memory (ie. there is no waiting for memory reads), and there is no cache, the same sequence of instructions will always take the same number of cycles to run. As the core used also supports unaligned memory reads with the same delay as aligned reads this further ensures that packets located differently in memory take the same amount of time to process.

### 5.1.1 Initial application analysis

As can be seen from tables A.1, A.2 and A.5 (pages 47–49), the time spent on storage management when processing extensions is 0. This happens as a direct consequence of the chosen implementation. Section extractions in this implementation are complex, in the sense that a given symbol (the smallest time unit) can contain an arbitrary number of sections (spanning different parts of the frequency spec-

trum). These need to be kept easily searchable, which in turn introduces a large overhead of essentially maintaining a data structure (table) in which they are stored.

Conversely, beamforming weights (the data contained in the extensions), is stored entirely indexed on the included beamID value, and no maintenance of any data structure is thus required. Instead the values are simply copied to an address offset defined by the beamID (multiplied with the size of a set of beamforming weights of course).

### 5.1.2 Instruction set generation

One can note, that except for the initial software optimization, not much difference in time is seen for storage management and sequential parsing. But as also pointed out in section 3.2 the hardware optimization effort was focused on decreasing the extraction time, so this is not surprising at all.

It is notable however that the addition of `pack` reduces time in storage management (see table A.5), but it's also followed by an increase in data extraction time (see table A.4). This essentially boils down to additional store instructions in the storage management section being replaced by pack instructions in the data extraction, and as can be seen from table A.3, the total execution time is essentially the same with and without `pack`.

Analyzing this further, this is not specific to our software implementation, but valid for any software on our chosen base hardware, with single instruction issue and single cycle store instructions. On this architecture, if you first pack two halfwords into a word and store this to memory, this takes the same amount of time (2 cycles) as just storing each halfword individually with `SH` (store halfword) instructions. On a multi-issue architecture however, there is a potential advantage to having the `pack` instruction, at least where memory accesses are a bottleneck in the execution, as the `pack` instructions can then replace `store` instructions on the congested memory interface.

### 5.1.3 Synthesis

It can be clearly seen that both `pack` and `grev` had an impact on size that is essentially insignificant and within the margin that random changes from the synthesis tool have bigger impacts, as for example can be seen from the fact that the version with only the `pack` instruction added actually is slightly smaller than the basecore, despite having extra hardware added to it.

In the case of these two instructions, there's not much reason not to add them. Especially in the grev case, there are very noticeable performance advantages to a trivial amount of size increase. As for pack, as mentioned above, there is not much of a case for using it on this simple single-issue architecture, but immediately should be of interest in the case of any type of multi-issue architecture, since it again adds a trivial amount of hardware.

The `bdep` and `bext` instructions are harder to come to any solid conclusions around. For this specific application the performance benefits of these instruc-

tions, in addition to `grev`, are modest (around 10%, see Table A.3 on page 48), with a roughly 15% increase in area required. At face value this might seem as if it doesn't pay off, but here it is important to take the entire system into account. In addition to the core, an ASIP needs other circuitry as well, most importantly memory, which can be of considerable size. It's entirely out of the scope of this report to speculate on how much memory would be needed, but it's not hard to imagine a scenario where data and program memory combined are several times as large as the size of the core. If that is the case, then it could be argued that the extra hardware is worth it, if the size of the entire system increases with noticeably less than 10% in size.

## 5.2 Methodology

### 5.2.1 Design methodology

As mentioned in Section 3 on page 17, architectural exploration was left out of the design process, and this was a choice made early to simply limit the scope of the project. While this probably was a necessary limitation to make (as there is only so much time), it is in some ways unfortunate, as this could potentially have been one of the most effective ways to speed up execution.

As described earlier, data extraction is a large part of the total processing time, and by its nature extraction of independent data fields should have a fairly large degree of instruction level parallelism. This parallelism could for example be exploited by using some form of multi-issue architecture, and the simplest would likely be some form of VLIW implementation. Of course this would require extensive testing and exploration to figure out the optimal issue width of such an implementation, as well as what instructions to allocate to which execution slots etc.

Another aspect to explore would be the width of the memory interfaces and registers. Data extraction, if it can be parallelized, should get more parallelism the wider chunks of data that are simultaneously processed. One alternative would be to explore something like RV64, ie. the 64 bit version of RISC-V. At least this should be beneficial for extraction schemes using the `bdep` and `bext` instructions (as described in section 3.2.4), as they can simply work on wider words and more data at a time. Another approach would be to use wider memory channels loaded to special purpose registers and then run some form of SIMD vector instructions on them.

### 5.2.2 General process comments

As for more general comments on the project as a whole, one thing that would have been beneficial would have been to more early on try to delimit the work. A lot of time was spent trying to find information on what good use cases to test against would be. The problem is that eCPRI and ORAN are quite new technologies, and finding usage statistics — or even harder, openly publishable usage statistics — turned out to be a task I did not manage to overcome. In the end I

had to define a fairly general scenario and simply measure performance per data unit and measure relative improvement, instead of having a certain rate to compare this against. It would have saved a lot of time, and probably have been a wise decision, to come to this conclusion earlier.

A lot of time was also spent manually doing calculations and collecting metrics, that I eventually automated with scripts, which saved considerable amounts of time. It would be prudent to try to automate as much as possible of such work as early as it becomes clear that it will be repeated many times over.

Another thing to note is that the improvements achieved due to software optimizations were much bigger than the ones achieved through the addition of new instructions. This highlights the need to first make some software optimization efforts for the results to have much meaning at all, as otherwise a change in results could just as well be attributed to changes made in the program code to make use of the new instructions. However, this brings its own set of problems, as it's hard to know how "optimal" a solution actually is, and this is largely up to the skill and discretion of the implementer. It's hard to find something objective to measure against.

I do see this as problematic, but don't have any immediate solutions to suggest either. In an application which is better explored, or has a greater number of accessible existing implementations (say if you were for example implementing an IP stack or similar), it is easier. Then one of these, presumably already well optimized solutions can be selected to compare against, but in a case like this where I also have to implement my own parser, it is tricky to make a comparative analysis.

### 5.2.3 Bit field place (bfp)

The bit field place instruction (`bfp`) also did not really provide much improvement in terms of performance. Bit field place patterns can be expensive to execute without specialized instructions, because bit masks need to be generated, and then on top of this, a shift, an `and` and an `or` instruction need to be done, and compared to this the `bfp` instruction can save many cycles! However, in the construction of bit fields in this application, there are only a few bit field locations and lengths, specific to a few data structures, that are all known at compile time. Thus the masks can be calculated beforehand and saved in memory. Also, all the bit fields created in this application are built from scratch (ie. we start with all zeros), and thus the need to clear values from the corresponding area in the field with a separate and instruction (and managing the corresponding mask) is not needed. Consequently almost all of the expensive parts of updating bit fields aren't actually done in a well optimized software implementation of this application.

In a case where locations and lenghts of bit fields vary a lot, and/or aren't known at compile time there is much more potential for a `bfp` instruction to save time. It would be more efficient if the offset and length of the field could be given as immediate values, so that the preparatory setup steps aren't needed. However, that would require 10 bits of instruction encoding space for those immediate

values (5 bit offset + 5 bit length), a register for the source field (5 bits), a register for the source value (5 bits) and a destination register (5 bits), which would be too much to fit into a normal 32 bit RISC-V instruction (or at the very least would take up an entire major op-code slot). One potential solution to this would be to do this in place, and use the same register for both the source field and the destination, in which it would only take up a minor op-code slot (similar to for example ALU instructions with immediate values, or memory operations).

### 5.2.4 Static bitwise extract and deposit

The `bdep` and `bext` instructions as implemented here (see Section 3.2.4 for details), calculate the control words for the butterfly networks on every execution of either operation. As the calculation of the control words is the most complicated part of the process[7], this seems somewhat wasteful. Instead of this implementation, it would be possible to have a separate instruction for preparing the control word in a special purpose register, and then use this for the actual bit permuting instructions. Hilewitz and Lee[7] also point out that in applications that mostly deal with uniform looking bitfields, this is likely the best compromise.

### 5.2.5 Immediate bit extract

One common pattern in the generated code was that of extracting a bit field from some location within a 32 bit word. This is essentially two instructions, a right-shift followed by an `andi` operation to mask out higher bits. If the field is wider than 11 bits, an immediate value can not be used as a mask, and another operation or two might be needed for making the value available in a register.

However, most bitfields are smaller than this so it could be questioned how useful an operation replacing only two instructions is. However, it is such a common pattern, and such a cheap operation to implement (the already existing shift unit can be reused) that it's worth exploring.

## 5.3 The work in a wider context

As global warming keeps becoming a greater concern in all parts of the world, power usage in most industries also continues to be of interest, including the telecommunications industry. As such, the power savings that ASIPs can provide, compared to for example a general purpose processor implementation, or an FPGA implementation, can be considered as positive influences.

# 6

# Conclusion

One important conclusion to draw from this work is that delimitations are necessary but will also strongly influence the nature of the results. In this case this can be most clearly seen in the delimitation to stick to the same overall processor architecture and only focus on which instructions to use.

Within the set of delimitations chosen, and the set of instructions that were added, the `grev` instruction for byte swapping was clearly the most efficient addition. For the other instructions evaluated only `bdep`/`bext` had any noticeable impact on performance. Within the set delimitations there seems to be little advantage of adding `pack` and `bfp` instructions.

## 6.1 Future work

For further work on the topic, I can see three main tracks that would be interesting to explore. The first one that should be very simple, straight forward and cheap would be a direct extract instruction, as discussed in section 5.2.5.

The second one would be to look closer at the instruction level parallelism of the chosen applications, and explore what benefits a multi-issue architecture could bring to this application. Likely a VLIW implementation would be the easiest way to explore this to avoid having to also implement a more complex decoder.

Thirdly, a wider memory interface with some form of vector instructions would be an interesting case as well. Especially for something like the beamforming weights as they contain a lot of uniform data copying, but also `bdep` and `bext` could make efficient use of working with wider data words.

# Appendix

# A

# Full application analysis results

The full results from the application analysis are presented here. For a closer description of these, see section 4.2.

| *Configuration* | $t_{section}$ | $t_{extension}$ | $t_{overhead}$ |
|---|---|---|---|
| Total time | 1.00 | 1.00 | 1.00 |
| Sequential parsing | 0.13 | 0.09 | 0.19 |
| Data extraction | 0.63 | 0.91 | 0.11 |
| Storage management | 0.23 | 0 | 0.69 |

***Table A.1:*** *The execution time for the different metrics normalized with respect to the total runtime for that specific metric, for the base hardware running with the optimized software.*

| *Configuration* | $t_{section}$ | $t_{extension}$ | $t_{overhead}$ |
|---|---|---|---|
| Total time | 1.00 | 2.12 | 2.07 |
| Sequential parsing | 1.00 | 1.50 | 3.00 |
| Data extraction | 1.00 | 3.03 | 0.37 |
| Storage management | 1.00 | 0 | 6.14 |

***Table A.2:*** *The execution time for different categories normalized with respect to $t_{section}$ for the base hardware running with the optimized software. This visualizes the magnitude of each metric to each other.*

| *Configuration* | $t_{section}$ | $t_{extension}$ | $t_{overhead}$ |
|---|---|---|---|
| BASE HW; no SW optimizations | 2.73 | 3.41 | 0.77 |
| BASE HW with SW optimizations | 1.00 | 1.00 | 1.00 |
| GREV | 0.88 | 0.50 | 0.98 |
| PACK | 0.98 | 1.00 | 1.00 |
| PACK+GREV | 0.88 | 0.50 | 0.98 |
| DEPEXT+GREV | 0.80 | 0.51 | 0.98 |
| All | 0.80 | 0.51 | 0.98 |

***Table A.3:*** *Total runtime for model application to run with different configurations. Results are normalized to the respective column for the base hardware running the optimized software.*

| *Configuration* | $t_{section}$ | $t_{extension}$ | $t_{overhead}$ |
|---|---|---|---|
| BASE HW; no SW optimizations | 1.58 | 3.52 | 1.14 |
| BASE HW with SW optimizations | 1.00 | 1.00 | 1.00 |
| GREV | 0.76 | 0.45 | 0.50 |
| PACK | 1.08 | 1.00 | 1.00 |
| PACK+GREV | 0.84 | 0.45 | 0.50 |
| DEPEXT+GREV | 0.71 | 0.47 | 1.00 |
| All | 0.71 | 0.47 | 1.00 |

***Table A.4:*** *Time spent in category* data extraction *for model application in different configurations. Results are normalized to the respective column for the base hardware running the optimized software.*

| *Configuration* | $t_{section}$ | $t_{extension}$ | $t_{overhead}$ |
|---|---|---|---|
| BASE HW; no SW optimizations | 6.64 | 0 | 0.48 |
| BASE HW with SW optimizations | 1.00 | 0 | 1.00 |
| GREV | 1.14 | 0 | 1.06 |
| PACK | 0.71 | 0 | 1.00 |
| PACK+GREV | 0.93 | 0 | 1.05 |
| DEPEXT+GREV | 0.93 | 0 | 0.98 |
| All | 0.93 | 0 | 0.98 |

***Table A.5:*** *Time spent in category* storage management *for model application in different configurations. Results are normalized to the respective column for the base hardware running the optimized software.*

| *Configuration* | $t_{section}$ | $t_{extension}$ | $t_{overhead}$ |
|---|---|---|---|
| BASE HW; no SW optimizations | 1.38 | 2.33 | 1.63 |
| BASE HW with SW optimizations | 1.00 | 1.00 | 1.00 |
| GREV | 1.00 | 1.00 | 1.00 |
| PACK | 1.00 | 1.00 | 1.00 |
| PACK+GREV | 1.00 | 1.00 | 1.00 |
| DEPEXT+GREV | 1.00 | 1.00 | 1.00 |
| All | 1.00 | 1.00 | 1.00 |

***Table A.6:*** *Time spent in category* sequential parsing *for model application in different configurations. Results are normalized to the respective column for the base hardware running the optimized software.*

# Bibliography


[1] 3GPP. Technical specification group radio access network; evolved universal terrestial radio access (e-utra); physcial channels and modulation (release 16). Technical Specification (TS) 36.211, 3rd Generation Partnership Project (3GPP), 12 2019. URL https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=2425. Version 16.0.0.

[2] 3GPP. Technical specification group radio access network; nr; physcial channels and modulation. Technical Specification (TS) 38.211, 3rd Generation Partnership Project (3GPP), 12 2019. URL https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3213. Version 16.0.0.

[3] O-RAN Alliance. O-ran fronthaul working group. control, user and synchronization plane specification. Technical Report O-RAN-WG4.CUS.0-v03.00, 2019. URL https://www.o-ran.org/specifications. Version 16.0.0.

[4] *eCPRI Specification*. Ericsson AB, Huawei Technologies Co. Ltd, NEC Corporation and Nokia, 05 2019. URL http://www.cpri.info/downloads/eCPRI_v_2.0_2019_05_10c.pdf. V2.0.

[5] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design principles for packet parsers. *Architectures for Networking and Communications Systems, Architectures for Networking and Communications Systems (ANCS), 2013 ACM/IEEE Symposium on*, pages 13 – 24, 2013. ISSN 978-1-4799-1640-5.

[6] H. Harb and C. Chavet. Back-to-back butterfly network, an adaptive permutation network for new communication standards. In *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1688–1692, 2020.

[7] Y. Hilewitz and R. B. Lee. Fast bit compression and expansion with parallel extract and parallel deposit instructions. In *IEEE 17th International*

*Conference on Application-specific Systems, Architectures and Processors (ASAP'06)*, pages 65–72, 2006.

[8] Y. Hilewitz, Z. J. Shi, and R. B. Lee. Comparing fast implementations of bit permutation instructions. In *Conference Record of the Thirty-Eighth Asilomar Conference on Signals, Systems and Computers, 2004.*, volume 2, pages 1856–1863 Vol.2, 2004.

[9] *Intel Itanium Architecture Software Developer's Manual, Volume3 : Intel Itanium Instruction Set Reference*. Intel, 05 2010. URL `https://www.intel.com/content/dam/www/public/us/en/documents/manuals/itanium-architecture-vol-3-manual.pdf`. Revision 2.3.

[10] M. K. Jain, M. Balakrishnan, and A. Kumar. Asip design methodologies: survey and issues. In *VLSI Design 2001. Fourteenth International Conference on VLSI Design*, pages 76–81, 2001.

[11] C. Kozanitis, J. Huber, S. Singh, and G. Varghese. Leaping multiple headers in a single bound: Wire-speed parsing using the kangaroo system. *2010 Proceedings IEEE INFOCOM, INFOCOM, 2010 Proceedings IEEE*, pages 1 – 9, 2010. ISSN 978-1-4244-5836-3.

[12] R. B. Lee. Hp precision: a spectrum architecture. In *[1989] Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences. Volume 1: Architecture Track*, volume 1, pages 242–251 vol.1, 1989. doi: 10.1109/HICSS.1989.47164.

[13] R. B. Lee, Zhijie Shi, and Xiiao Yang. Efficient permutation instructions for fast software cryptography. *IEEE Micro*, 21(6):56–69, 2001.

[14] *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. RISC-V Foundation, 12 2019. URL `https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf`. Editors Andrew Waterman and Asanović Krste.

[15] *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified*. RISC-V Foundation, 06 2019. URL `https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMFDQC-and-Priv-v1.11/riscv-privileged-20190608.pdf`. Editors Andrew Waterman and Asanović Krste.

[16] *RISC-V Bitmanip extension, Document Version 0.92*. RISC-V Foundation, 11 2019. URL `https://github.com/riscv/riscv-bitmanip/blob/master/bitmanip-0.92.pdf`. Editor Clifford Wolf.

[17] Z. Shi, X. Yang, and R. B. Lee. Arbitrary bit permutations in one or two cycles. In *Proceedings IEEE International Conference on Application-Specific Systems, Architectures, and Processors. ASAP 2003*, pages 237–247, 2003.

[18] H. ( 1 ) Zolfaghari, J. ( 1 ) Nurmi, and D. ( 2 ) Rossi. A custom processor for protocol-independent packet parsing. *Microprocessors and Microsystems*, 72, 2020. ISSN 01419331.