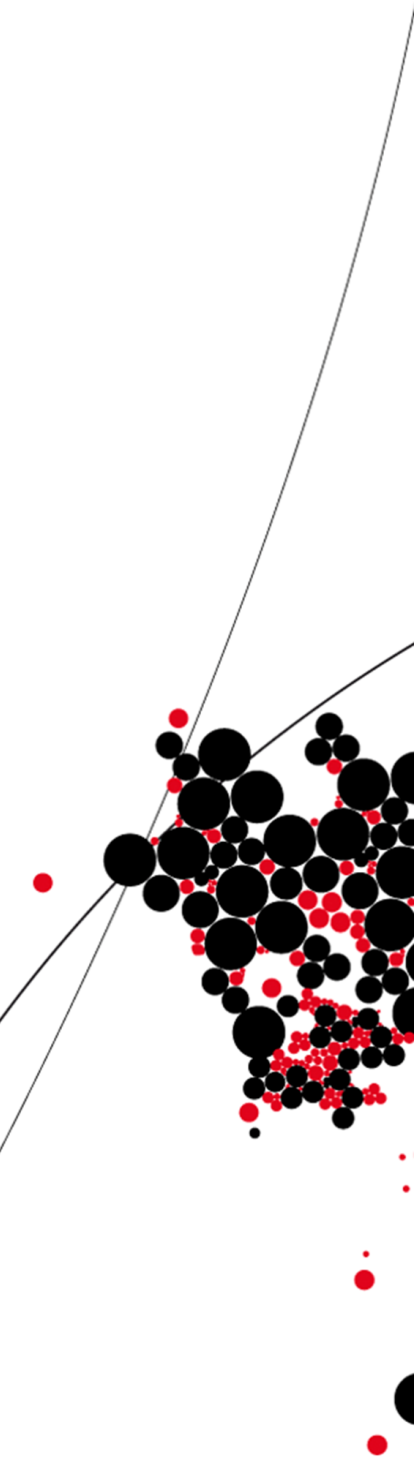
UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics & Computer Science



ASIP design and algorithm implementation for beamforming in an antenna array

J.M. Oedzes

M.Sc. Thesis
August 2020

Supervisors:
dr.ir. S.H. Gerez
dr.ir. A.B.J. Kokkeler
dr.ir. M.S. Oude Alink
Masoud Abbasi Alaei M.Sc.

Computer Architecture and Embedded Systems Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

Beamforming is a technique that can be used in antenna arrays to electronically steer wireless signals in a specific direction for directional signal transmission or reception. The technique is widely used in a multitude of communication standards. Beamforming requires coefficients to be chosen for each of the antennas in the array, which proves to be a computationally intensive task. This work investigates the implementation of a 4-antenna beamforming algorithm for a hybrid beamforming system onto an application-specific instruction set processor (ASIP), which is a type of processor that can be extended with custom instructions to suit its specific application. The Tzscale processor was used as the base for the ASIP and is build upon the RISC-V instruction-set. An implementation of the Tzscale processor from the ASIP Designer Environment by Synopsys was used. Extensive complexity analysis is performed on the algorithm before implementation in C. After software optimizations, the required number of clock cycles already reduced by 44%. The ASIP was then extended with native fixed-point support for multiplication and division, as profiling results indicated that this would be the most beneficial for the implementation. The native fixed-point support reduced the required number of clock cycles for running the optimized software version by 61%, while only increasing the required area of the processor by 8%. Power numbers based on a 10% toggle rate show that the implementation also requires only 40% percent of the energy to run an algorithm cycle.

Contents

List of acronyms

ADC	analog to digital converter
ALU	arithmetic logic unit
APN	analog preprocessing network
ASIC	application-specific integrated circuit
ASIP	application-specific instruction set processor
CAU	complex arithmetic unit
CGRA	coarse-grained reconfigurable architecture
CPU	central processing unit
FPGA	field programmable gate array
HDL	hardware description language
HLS	high-level synthesis
ISA	instruction set architecture
ISS	instruction set simulator
LSB	least significant bit
MIMO	multiple-input multiple-output
MRB	matrix register banks
MSB	most significant bit
PDG	primitives definition and generation
QMP	quantized matching pursuit
RF	radio frequency
RISC	reduced instruction set computer
RTL	register transfer level
SDK	software development kit
SIMD	single instruction multiple data

SISO single-input single-output

SVD singular value decomposition

TTA transport triggered architecture

VHDL very high speed integrated circuit hardware description language

Chapter 1

Introduction

In multiple-input multiple-output (MIMO) systems multiple antenna elements are deployed at both the transmitter and the receiver end, in contrast to single-input single-output (SISO) systems where only a single antenna is used. This increased number of antennas allows for multiple data streams to be transmitted simultaneously, which can be used to increase the reliability of the communication channel as well as increase the data rate. The multiple antennas also allow for a technique called beamforming: a signal processing technique utilizing multiple antennas for directional signal transmission or reception, making beamforming a specific case of MIMO. Beamforming is achieved by transmitting the same signal through multiple antennas, while varying the phase and magnitude for each separate antenna. The same can be done at the receiver side to be more sensitive to a specific direction. MIMO and beamforming technology attracts more and more attention and is already widely used in a significant number of fields like for example Wi-Fi and cellular networks.

Beamforming leads to separate radio frequency (RF) front-ends and analog to digital converters (ADCs) for each antenna, resulting in increased circuit size and higher power consumption. Next to that, effectively performing beamforming proves to be a computationally intensive task, requiring very powerful digital hardware for processing of the signals. To reduce the computational load, a hybrid structure with both an analog and a digital part can be implemented where analog preprocessing networks (APNs) are used to already cancel interference before reaching the digital domain. Employing APNs reduces the required ADC resolution since interference can be pre-canceled and does not have to be digitized, which also translated to lower power consumption [1].

To perform beamforming, optimal weights for the antennas have to be calculated constantly to be able to recover the transmitted data streams in the best way. This thesis work will use an application-specific instruction set processor (ASIP) for that purpose, running the search algorithm as proposed in [1]. An ASIP is a processor

that can be tailored to the specific application that it is designed for, by adding to its instruction set architecture (ISA) or modifying its existing one. This means that for their specific applications, ASIPs can be made relatively fast and energy efficient. This thesis work concerns the ASIP used for calculating the optimal beamforming weights and builds upon [2], a thesis work serving as a foundation for this research. An ISA is already chosen, the basis for the ASIP is constructed and also already a complex instruction is added to the ASIP which can perform square root operations. This work will continue the work that was started by [2], with one of the main challenges being scaling up from a 2-antenna system as used in [2] to a 4-antenna system.

1.1 Research questions

The final goal of the research assignment is to continue the work that was started by [2]. Given the previous research work, the research question for this research thesis can be formulated as follows: *Which changes should be made to the ASIP as proposed in [2] to increase performance and efficiency to find the beamforming coefficients for a 4-antenna system?*

The following sub-questions are formulated:

- What impact does the change of a 2-antenna system as used in [2] to an N-antenna system have on performance and required memory?
- Does analysis of the algorithm without the context of the target architecture yield interesting results? Is there a relation between this analysis and profiling results after compilation for the target architecture? Is this relation as can be expected?
- Which changes can be made to the ISA by arithmetical (e.g. complex-number operations) or control-oriented (e.g. hardware loop controller) instructions, to increase performance and efficiency?

1.2 Report organization

The remainder of this report is organized as follows. In Chapter 2 background about beamforming and ASIPs is discussed, as well as related work on the subject. The beamforming algorithm is elaborated upon in Chapter 3. That chapter will introduce the formal description of the algorithm and perform analysis on it. The memory

requirement and implementation of the algorithm are discussed as well. Chapter 4 will then present results of running the implementation as discussed in Chapter 3 on the ASIP. The software implementation will be optimized, and finally a discussion will be held on what is best to be implemented into the ASIP. Chapter 5 discusses the modification of the ASIP and the new results due to these modifications. Finally, Chapter 6 will draw conclusion based on the results and provide recommendations for future work.

Chapter 2

Background

2.1 Beamforming

Beamforming is a technique used in antenna arrays to electronically steer the transmission or reception of signals into a certain direction. Instead of simply broadcasting the signals equally in all directions, the system tries to determine the direction of interest and transmits or receives a more concentrated beam to or from that direction. The technique of beamforming is widely used in a large number of fields, like for example 5G. The operation of 5G requires very high data rates, where beamforming proves to be a good technique to transmit and receive at these high speeds efficiently [3].

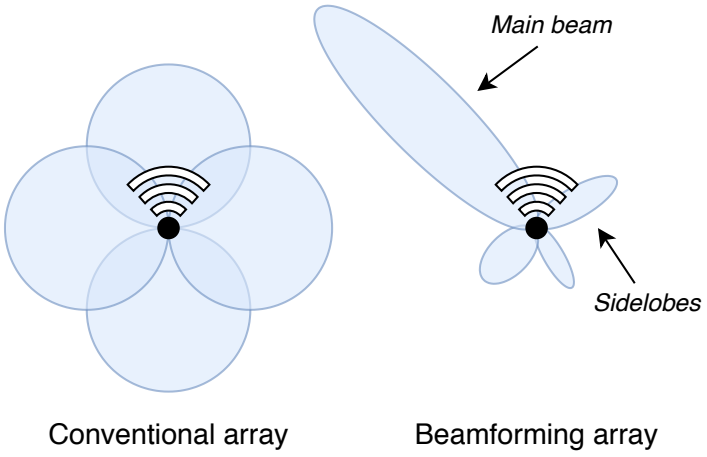


Figure 2.1: Antenna arrays with and without beamforming

Beamforming tries to solve the problem of capacity and throughput limitation that is caused by the omni-directional nature of transmission through antennas. The distribution of energy by a node in directions other than the intended one not only causes unnecessary interference to other nodes, but also reduces the potential range of

the transmission (due to a lower strength of the signal and multipath components). With directional communications, however, spatial reuse and range would be simultaneously enhanced, having a higher signal energy in the desired direction as an effect. [4].

The use of multiple antennas in a beamforming antenna array leads to the use of separate RF front ends and ADCs resulting in increased circuit size and power consumption [1]. In existing multi-antenna receivers, an ADC operation requires the same power as that of hundred thousands of logic gates [5], showing that cutting down on either the number of required ADCs or the ADC resolution would greatly benefit power consumption. To achieve this, hybrid beamforming systems consisting of both APNs and digital hardware can be used, where the APNs reduce the load on the ADCs and digital hardware.

APNs can be used in several ways. A relatively simple way to reduce the number of RF and ADC chains is to use antenna selection, where the antenna with the highest signal energy is selected to be used for the actual signal processing. A big drawback of this is that interference cannot be canceled this way. Ideally, the APNs should be used to linearly combine the received signals using weighing coefficients [1]. In such a setup, signals with higher signal energy receive a higher weight, resulting in these signals having a greater influence on the linearly combined signal than signals with lower signal energy. In an actual system it is very likely for the optimal weighing coefficients to change over time, as the receiver and transmitter can move with respect to each other. To make use of beamforming in its optimal way, an algorithm which tries to find the optimal weighing coefficients should run in real-time, constantly updating the weighing coefficients in the APNs. As stated in Section 1, this research will use an ASIP as the processor in a hybrid beamforming system to calculate the beamforming coefficients that are needed in the APNs.

2.2 Application-specific instruction set processor

Most engineers are familiar with the concept of an application-specific integrated circuit (ASIC). An ASIC usually provides high performance and efficiency on a relatively small area (die) compared to a general-purpose processor, due to it being designed specifically and only for its target application. For this reason, ASICs are widely used. However, the development of ASICs usually takes a long time because of a couple of reasons. Firstly, as ASICs are designed with one specific application in mind, development often starts from scratch. Another disadvantage to the use of ASICs is that once the design is finished and production has started, there is no way to alter its functionality. An ASIC can often be configured to some degree, for example by writing to its internal registers, but the core functionality remains the same.

This makes ASICs inflexible once their design is finished.

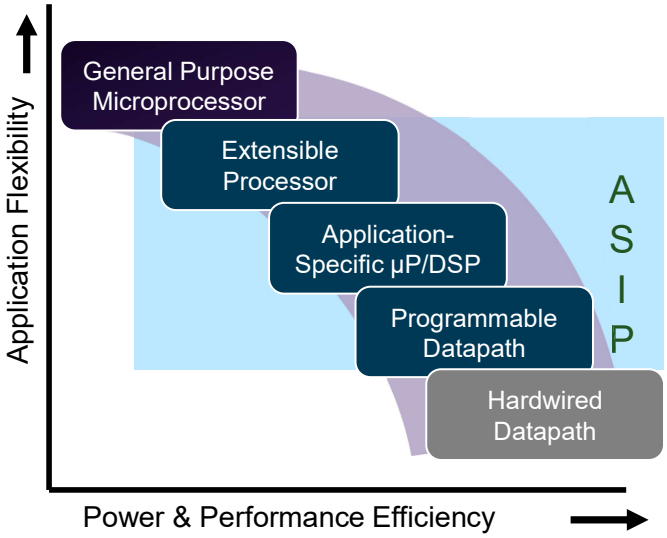


Figure 2.2: Flexibility and performance trade-off for different hardware solutions [6].

On the other end of the extreme would be the general-purpose processor, designed for maximum flexibility and fitting as many different applications as possible. Designing with general-purpose processors mostly consists of writing software to perform the required tasks. Where in general-purpose processors flexibility on the hardware level is still constrained by the architecture of the processor, a great amount of flexibility is achieved by the ability to easily write and change the software running on the processor. Especially higher-level languages like C or C++ enable the designer to implement complex behavior in a relatively short time, while a compiler takes care of the translation to machine code to match the specific architecture. Also, using these higher-level languages in combination with their compilers significantly increases reusability, as software can easily be ported to different architectures. All of this flexibility comes at the cost of higher power consumption and lower speed compared to ASICs. Where both options have clear benefits and drawbacks, an ASIP tries to be the best of both extremes and is the balance between an ASIC and pure software-based solutions [7]. Figure 2.2 draws a picture of how an ASIP fits in between the extremes of the general-purpose processor and the ASIC.

An ASIP is a processor with an instruction set tailored to the exact application that it is designed for. Customization of a processor for a specific application holds the system cost down, which is especially important for embedded consumer products manufactured in high volume [8]. The trade-offs involved in designing ASIPs differ considerably from the design of general-purpose processors, because they are not optimized for a wide variety of applications but target a specific problem class just like ASICs. Regardless of this specific design, flexibility is maintained since in the

end the behavior of the system is still determined by the software running on the ASIP, which can be altered and updated easily. Also, using modern design tools the ISA can be altered and extended relatively easily to suit different applications. For instance, a processor for DSP applications generally shows MAC units, dual memory banks, and address generation hardware, while a network processor provides instructions for fast data packet manipulation and routing table lookup [9].

2.2.1 Design flow

The design of an ASIP is a hardware/software co-design problem that often consists of the same set of steps, which starts with a target application and finally ends with hardware synthesis of the ASIP. Most often, the following 5 steps can be identified in the design process [10]:

1. **Application analysis**
   Input in the ASIP design process is an application along with potential test data and design constraints. Often the application consists of one or multiple algorithms that need to perform complicated mathematical operations, or operate on large chunks of memory, both of which are not always supported by conventional ISAs. The application should be profiled to find the potential ”hotspots” which are interesting to try and optimize using additions to the hardware. The insights gained in this stage are used in the subsequent design steps.

2. **Architectural design space exploration**
   An exploration into different architectures that could fit the need of the application is performed. Most often an already existing architecture is chosen as the base for the ASIP. Some of these architectures can allow for customization on many levels, including the micro-architecture and the pipeline, while others are more constrained but often adhere to certain standards.

3. **Instruction set generation**
   An instruction set is to be generated to suit the need for the applications under design. Most often a base ISA is provided with the architecture that is chosen for the ASIP. The base ISA is often relatively limited, but it is of course extended with custom instructions to suit the applications' needs. The instruction set is used during the code synthesis and hardware synthesis steps.

4. **Code synthesis**
   A compiler is generated for the ISA which can then be used to compile the application on the new hardware. Compiler generation is part of most modern ASIP design tools. After compilation of the application on the new ISA, the

software can be profiled again, specifically for the new ISA. Most often there are multiple design cycles that alternate between this step and the instruction set generation step. Once the designer is satisfied with the performance, the next step can be done.

5. **Hardware synthesis**
   In this step the hardware is synthesized using the ASIP architectural template and ISA. Synthesis can be performed for both FPGA and ASIC implementations, after which also post-synthesis simulations can be performed.

As ASIPs are designed to run software (algorithms) specifically written for the target application, the design of ASIPs always starts with step 1, the algorithm itself. Finding useful alterations and extensions to the ISA requires a deep understanding of the application algorithm which for a large part can be gained by analyzing the corresponding source code. Therefore, source code profiling is the first important step in ASIP design [11]. Traditional source code profiling can be divided into two classes: assembly level profiling and high-level profiling.

The difference between the two originates from the fact that assembly language always corresponds to a specific ISA while a high-level language does not. This means that assembly level profiling is specific to the architecture, and can thus provide architecture-specific information like the number of clock cycles required to execute a certain function. This is not the case with high-level profiling, where profiling is done only at the source code level. This makes high-level profiling a little less useful for ASIP design, as it cannot gather architecture-specific information. However, it can still be useful since this type of profiling can be used to already tackle inefficiencies on the software level. Because of these reasons, it would be interesting to consider both profiling options.

The environment used for ASIP design in this research is called 'ASIP Designer', by Synopsys [12]. It is a retargetable compiler, simulation and hardware generation environment. ASIP Designer is a high-level synthesis (HLS) environment, where the nML and PDG languages are used to describe the processor and its behavior. Figure 2.3 shows the design flow to be used with the environment. The steps shown in the diagram slightly differ from the steps as discussed before. The first step, SDK generation, assumes that an architectural design space exploration was already performed, and that a base architecture was chosen as a result of that. The SDK generation step generates a compiler specifically for the ASIP and also an instruction set simulator (ISS) that can be used to simulate the behavior of the processor when the application code would be running on it. This makes the ISS an ideal tool for performing assembly-level profiling. The ASIP Designer environment does not incorporate any high-level profiling. As a result of that, this step will need

to be performed apart from the ASIP Designer environment. The architectural optimization step indicates the process of modifying the ASIP by adding or modifying instructions. The SDK generation step can then be performed again to generate a new SDK for the modified ASIP. These first two steps can be done multiple times until the designer is satisfied with the simulated performance. The last steps, hardware generation and verification, indicate the hardware synthesis process where synthesizable RTL is generated. The RTL can be used to perform simulations, verify behavior at the hardware level, or implement the ASIP on an FPGA.

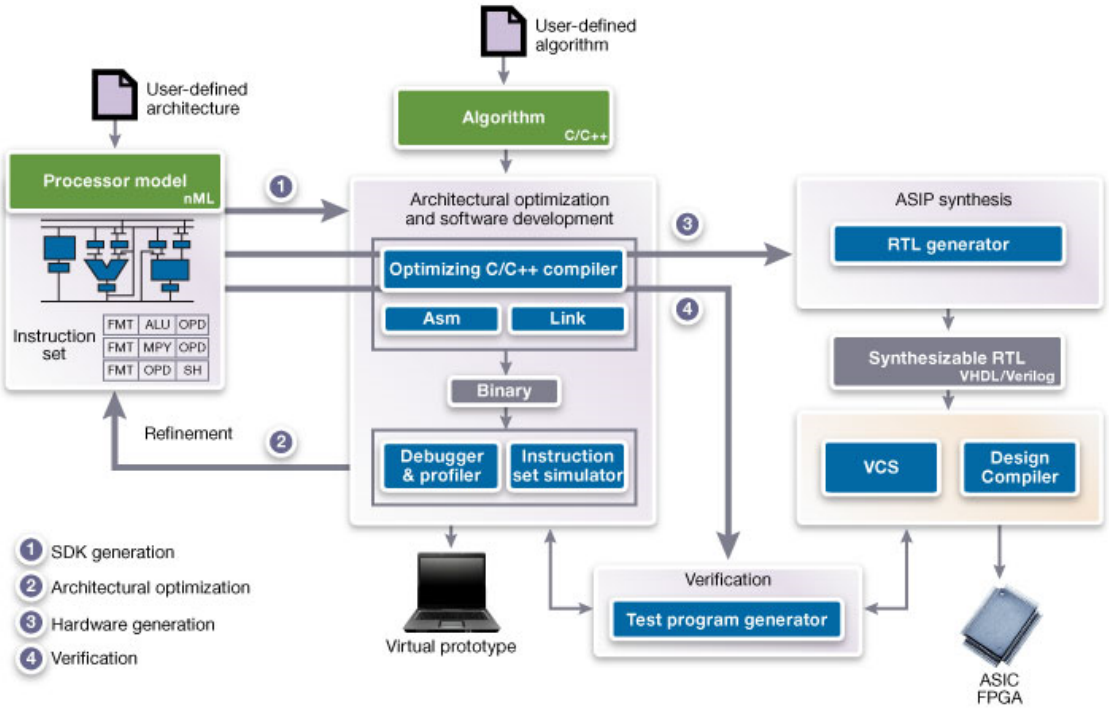


Figure 2.3: ASIP design process in the ASIP Designer tool [12].

2.3 Related work

Section 2.1 and 2.2 introduced beamforming and the ASIP. This section will cover some of the research that has been done, specifically concerning ASIPs and their use in wireless communication related scenarios. No literature was found regarding the exact same topic of this research, i.e., literature about ASIP design on behalf of a beamforming algorithm. Instead, the literature study was focused on closely related topics. For example, ASIP design for other communication related algorithms. The difference in context in which the algorithm operates should not be a problem, since ASIP design specifically targets the operations and actions performed by the algorithm, regardless of the algorithm's application.

[13] shows the design of an ASIP for an MD5 hash algorithm. This is a good example of a straightforward ASIP design problem: an algorithm that performs a lot of operations on a large data set. The ISA was extended with multiple instructions, e.g. complete instructions for specific rounds of the algorithm. These alterations significantly decrease the required number of clock cycles required to run the algorithm.

[14] use an ASIP to run three different beamforming algorithms used in hearing aids. The trade-off between performance of the algorithms and power consumption is analyzed, the latter of which is especially important in this case as hearing aids are battery powered. The proposed application-specific hardware optimizations were implemented in a VLIW-SIMD hearing-aid processor, where the processor's datapath width was modified. A total of 24 different processor configurations were studied. It appeared that the optimizations resulted in either a reduced silicon area requirement of up to 2 times or a decreased power consumption up to 11 times, with only a slight decrease in algorithm performance.

In [15] an ASIP is used for a real-time control module for a retrodirective antenna array. The developed digital control module was capable of updating the current coefficients of an eight-element antenna array in less than 1s with a power consumption of 5.3mW for beamsteering angles from 0 to 60 degrees.

[16] presents an ASIP-based flexible MMSE-IC Linear Equalizer for MIMO turbo-equalization applications. The proposed ASIP has a single instruction multiple data (SIMD) architecture with an ISA extended with specialized instructions and a 7-stage pipeline. Several efforts are made on behalf of efficient computational and storage resource sharing: matrix register banks (MRB) multiplexing, a 16-bit complex arithmetic unit (CAU) made up of 4 combined complex adder/subtractor/multiplier units, 2 real multipliers, 5 complex adders, and 2 complex subtractors, and flexible 32-bit to 16-bit data conversion at multipliers' output. The proposed ASIP achieves a throughput of 273 MSymbol/Sec. The Processor Designer frame-work from CoWare Inc. was used for this research, which allows for automatic generation of ASIP software development tools along with VHDL and Verilog descriptions for hardware synthesis and system integration

An ASIP architecture implementation of channel equalization algorithms for MIMO systems in WCDMA downlink is presented in [17]. The instruction set of transport triggered architecture (TTA) processors is extended with several user-defined operations specific for channel equalization algorithms that significantly optimize the

architecture for the physical layer of the mobile handset. The implemented ASIP solutions meet the real-time requirements for the 3GPP wireless standard with reasonable clock speed and power consumption.

[18] describes the development of an ASIP used as an singular value decomposition (SVD) processor in SVD-MIMO systems for 4 to 16 transmitter and receiver antennas. The proposed design uses floating-point units for computation requiring high precision. Also, an optimization was done to the SVD algorithm, showing the importance of source code profiling. The optimization was done by looking at the minimum number of required QR method iterations according to each size of the MIMO channel matrix. The resulting architecture is able to do real-time processing for 4x4 to 16x16 MIMO-OFDM.

[19] describes the implementation of a multi-mode MIMO detector based on the concept of partially reconfigurable ASIP. The use of reconfigurable hardware allows for the implementation of multiple MIMO detection algorithms each with different antenna and modulation configurations. The rASIP is made up mainly of a coarse-grained reconfigurable architecture (CGRA) coupled with a processor. As matrix operations are required in a lot of MIMO related algorithms the ISA of the processor is extended with support for different matrix operations. Also, special instructions for the implementation of the control path required by the different algorithms are added. Synopsys Processor Designer was used as the tool for the implementation of the ASIP. Feasibility of the proposed approach is shown by implementing three non-iterative MIMO detection algorithms. Post-layout results are generated for 65nm CMOS technology. It appears that the proposed rASIP design is about 1.6–5.4 times more efficient than programmable architectures, and approaches the throughput performance of dedicated ASICs.

A CGRA based approach is also taken in [20]. This paper notes the difficulty to design hardware for MIMO systems with high flexibility and scalability with high hardware efficiency. Three different optimization techniques are applied, targeting flexible and scalable matrix operations for reduced memory access, flexible data access and the support for different bit widths. Fabricated on a 28nm CMOS technology, the chip achieves high flexibility and scalability while supporting various detection algorithms, various MIMO scales, such as 4x4, 32x32, and 128x8 and baseband processing tasks, such as filtering and fast Fourier transformation.

[21] shows a hybrid hardware/software 802.11ac/ax system design with ASIP implementation. In this paper, a design is presented for an 802.11ac/ax system with an

ASIP to implement multiple disjoint system tasks at much faster speed compared to a general purpose processor. Channel SVD, channel compression/decompression and beamforming weight computations were all implemented to run at the same time on the ASIP to support the 802.11ac MU-MIMO beamforming feature. The actual task to be done by the ASIP core at any given time can be programmed at any given time resulting in a very high hardware savings compared to an all hardware implementation.

It becomes clear that ASIP design is a very broad topic. Even when considering it in the context of a lot of the research presented above (wireless communication) it appears that a lot of different options and possibilities can be considered. Looking at all research combined it very well shows that ASIP design is very specific to the design problem and the design always needs to be tailored specifically to that.

Another thing that became clear from the literature research is that ASIP design indeed always starts with a certain algorithm or program and an extensive analysis of this algorithm or program. Clearly understanding the operations of the algorithm and having a good implementation of it are integral to making correct design decisions for the ISA.

Lastly, the thesis carried out prior to this work is addressed. In [2] the main research question was: "*Can a performance and energy efficient ASIP be designed as the baseband processor which performs the search algorithm to find the optimum coefficient value of the analog beamformer in the hybrid MIMO communication system*". This research question was answered successfully, as an ASIP was designed with the RISC-V ISA as a reference design on which the ASIP was developed. The Tzscale processor based on this architecture was taken as the skeleton design which was customized by adding a square root unit in hardware. As a result of this, the efficiency was increased by a decrease in the number of required clock cycles of approximately 50%, while only increasing the required area by approximately 3%. This is a very significant improvement. This work will use the architecture and extensions made to the ISA as proposed by [2], and build upon that to scale the design from 2 antennas to 4 antennas, while at the same time trying to further increase the performance and efficiency.

2.4 Hybrid beamformer top-level view

The structure of the hybrid beamforming system as discussed in [2] can be seen in Figure 2.4, with the analog beamformer and the baseband processing connected to each other via RF chains. There is an RF chain for each of the antennas. The

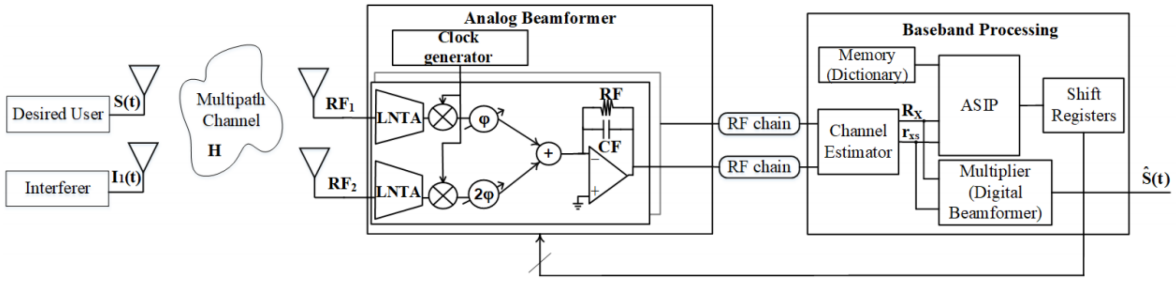


Figure 2.4: Hybrid beamforming structure at the receiver [2]

baseband processing block is the part of interest in this research, focusing on the ASIP. A short explanation for each block is given as follows [2]:

- *Dictionary*. The dictionary contains all possible values of the quantized analog beamforming coefficients. As the number of entries in the dictionary increases exponentially with more antennas, the dictionary will be stored in an external memory unit. The memory unit requires an interface with the ASIP since the ASIP is running the algorithm and therefore needs access to this dictionary.

- *Channel Estimator*. The calculation of the optimum analog beamforming coefficients will be elaborated on in Section 3. The algorithm requires the cross-correlation vector C_{rx} and its corresponding whitened vector value $\underline{C}_{rx}$, which are estimated by the channel estimator block. The block will also calculate the co-variance matrix of the received signal.

- *ASIP*. The ASIP is responsible for running the algorithm and find the optimum analog beamforming coefficients. It will use the values from the channel estimator to find the best set of coefficients in the dictionary, and then pass these to the shift registers.

- *Shift registers*. The shift registers are used as an interface between the ASIP and the analog beamforming circuit. The values produced by the ASIP are shifted to the analog beamformer, which will result in its switches being turned on or off.

- *Multiplier*. The multiplier is expected to perform the digital beamforming on the signals coming from the RF chains. The ASIP is not involved in the digital beamforming part.

Chapter 3

The QMP algorithm

The algorithm to find the best set of coefficients for the APNs considered in this research is the quantized matching pursuit (QMP) algorithm and is described in [1]. The sections in this chapter will discuss a formal representation of the algorithm, perform analysis in terms of complexity, memory requirement for storing the dictionary used by the algorithm and the mathematical operations that require implementation and finally discuss the implementation of the algorithm in the C language.

3.1 Formal description

The APN uses different coefficients for each antenna for receiving the signal that is being transmitted to it. The coefficients therefore have to be chosen in such a way that the signal coming from the APN matches the transmitted signal in the best way. This is can be achieved by transmitting a reference signal which is known to both the receiver and the transmitter. Once the reference signal is received and arrives at the channel estimator, the channel estimator can then pass the required data to the ASIP after which an exhaustive search can be performed where all different sets of coefficients are tried on the received signal. The set of coefficients that yields the best correlation to the reference signal is then chosen as the set of coefficients to be used in the APN.

An algorithm that achieves this is proposed by [1] and is called the quantized matching pursuit (QMP) algorithm. This algorithm performs an exhaustive search for the highest correlation in a dictionary containing coefficient values. (3.1) serves as a mathematical representation of the QMP algorithm as proposed by [1], where $\underline{\mathsf{w}}_{opt}$ refers to the optimal whitened APN coefficients.

$$\underline{\mathsf{w}}_{opt} = \arg\max_{\underline{\mathsf{w}}_i \in \underline{\mathsf{D}}} \frac{|\underline{\mathsf{w}}_i^H \underline{\mathsf{r}}_{\mathsf{xs}}|}{\|\underline{\mathsf{w}}_i\|} \tag{3.1}$$

Here, $\underline{\mathsf{w}}_i$ represents the i^{th} set of whitened coefficients from the dictionary and $\underline{\mathsf{w}}_i^H$

represents the complex conjugate transpose of that vector. $\underline{r}_{xs}$ is the input vector to the algorithm and is obtained by (3.2), where R_x is the covariance matrix of the received signal and r_{xs} is the cross-correlation vector between the received signal and the known reference signal. The size of R_x and r_{xs} depends on the number of antennas in the system. Given a system of N_r antennas, R_x is an $N_r \times N_r$ square matrix and r_{xs} is a vector with N_r elements.

$$\underline{r}_{xs} = R_x^{-\frac{1}{2}} r_{xs} \tag{3.2}$$

Lastly, the whitened dictionary $\underline{D}$ can be found using (3.3), where D is the normal unwhitened dictionary. The dictionary contains all permutations (with repetition) of the set of possible coefficient values, which are represented by column vectors of N_r elements. The set of possible coefficient values depends on the number of bits used for these coefficient values, which is denoted by R_w. Given N_r and R_w the dictionary is a matrix of size $N_r \times 2^{R_w N_r}$.

$$\underline{D} = R_x^{\frac{1}{2}} D \tag{3.3}$$

A Matlab function which implements the behavior of the algorithm can be found in Appendix A. The analysis and implementation of the algorithm in the rest of this chapter are based on this Matlab implementation.

3.2 Analysis

To gain insight into the requirements of the algorithm in terms of computational capability and memory, analysis is performed in three different ways: complexity in big-O notation, a breakdown of the dominant operations that are performed while running the algorithm and the required memory size (mainly for the dictionary that is required in the algorithm).

3.2.1 Complexity

In computer science, the complexity of algorithms is often expressed using the big-O notation. As the size of the inputs to an algorithm increases, its runtime will increase, and the big-O notation mathematically represents the curve of this runtime given the size of the inputs. The representation is useful because it offers quick insight and a means of comparison between the performance of different algorithms, and how the performance of an algorithm scales given the size of its input.

Table 3.1: Complexities of the main mathematical operations in the algorithm loop

Operation	**Complexity**
Matrix vector multiplication	$O({N_\mathrm{r}}^2)$
Vector norm	$O(N_\mathrm{r})$
Complex conjugate of vector	$O(N_\mathrm{r})$
Dot product	$O(N_\mathrm{r})$
Complex number absolute	$O(1)$
Division	$O(1)$

For the QMP algorithm, only the operations of the algorithm inside the main algorithm loop are considered. This is because the order of the loop combined with order of the operations within it will outweigh the order of any of the operations that are outside of the loop. To find the complexity of the QMP algorithm, the Matlab code as per Appendix A will be considered. It is important to mention that some lines of the Matlab code break down into several separate mathematical operations. Table 3.1 shows each separate mathematical operation that happens inside of the algorithm loop, and their complexities in big-O notation. It can be seen that the complexities of all operations inside the algorithm loop are either a constant, or depend only on the number of antennas. Also, it becomes clear that only the complexity of the matrix-vector multiplication has to be considered for the final expression as it is the fastest growing term. To get the final expression, $O({N_\mathrm{r}}^2)$ should be multiplied by the loop variable. As explained in Section 3.1, the size of the loop depends on the size of the dictionary, which is a matrix of size $N_\mathrm{r} \times 2^{\mathrm{R_w} N_\mathrm{r}}$. Since the loop will access this dictionary vector by vector, the final expression of the big-O complexity can be written down as follows:

$$O\left(2^{\mathrm{R_w} N_\mathrm{r}} {N_\mathrm{r}}^2\right) \tag{3.4}$$

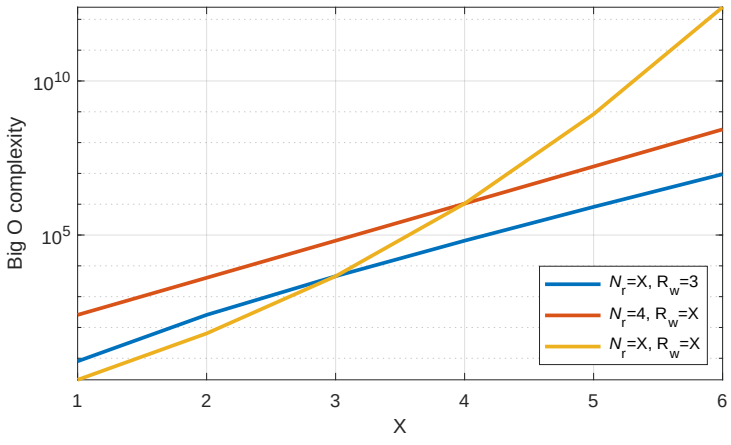


Figure 3.1: Influence of varying N_r and $\mathrm{R_w}$ on execution time

The final expression shows a complexity that depends on two variables and is exponential in nature. Increasing both variables a the same time would imply a drastic increase in execution time. This is best visualized by the graph in Figure 3.1. As the y-axis is displayed logarithmically, every step on the y-axis represents an order of magnitude.

3.2.2 Dominant operations

The big-O notation is a useful and widely used metric for the order of an algorithm. However, the expression does not provide much information about the algorithm except for how it scales given different input sizes. For ASIP design a more in-depth analysis providing insight into the dominant mathematical operations would be helpful. An analysis like this aids in anticipating potential hot-spots of the algorithm running on a processor. Next to that, it can serve as a means of comparison between this analysis and profiling results of the algorithm after it is simulated using an instruction-set simulator. Profiling results that differ too much from this analysis might be an indication that the algorithm was not implemented very efficiently.

To get insight into the most dominant operations that are performed in the algorithm, the mathematical operations shown in Table 3.1 are considered. Each of these mathematical operations is broken down into a number of simpler operations: (complex) addition, (complex) multiplication, division, square root, the complex conjugate and the absolute value of a complex number. Table 3.2 shows this breakdown, where the number of simpler operations that are required for each operation from Table 3.1 is written down as how it depends on the number of antennas.

Table 3.2: Breakdown of main mathematical operations into simpler operations

	add	**Cadd**	**mul**	**Cmul**	**div**	**sqrt**	**Cconj**	**Cabs**
Matrix vector mult.	0	${N_r}^2$	0	${N_r}^2$	0	0	0	0
Vector norm	$2N_r$	0	$2N_r$	0	0	1	0	0
Complex conj. of vector	0	0	0	0	0	0	N_r	0
Dot product	0	N_r	0	N_r	0	0	0	0
Complex number abs.	0	0	0	0	0	0	0	1
Division	0	0	0	0	1	0	0	0
Symbolic subtotal	$2N_r$	$N_r + {N_r}^2$	$2N_r$	$N_r + {N_r}^2$	1	1	N_r	1
Subtotal	8	20	8	20	1	1	4	1
Total	32768	81920	32768	81920	4096	4096	16384	4096

The 'Subtotal' row of the table represents the same as the symbolic subtotal given

that $N_r = 4$ and the 'Total' row represents the same as the subtotal row multiplied by 4096. This number is equal to the number of vectors in the dictionary for $N_r = 4$ and $R_w = 3$. The table clearly shows that complex addition and complex multiplication are the most dominant operations. However, since most processors do not natively support complex-number operations, it would be interesting to also simplify all complex operations into a number of basic (non-complex number) operations. Table 3.3 shows which regular operations are required for each complex-number operation.

Table 3.3: Regular operation for each complex operation

	add	**sub**	**mul**	**sqrt**
Cadd	2	0	0	0
Cmul	1	1	4	0
Cconj	0	0	1	0
Cabs	1	0	2	1

Table 3.4 is the result of combining the results of Table 3.2 and 3.3. It contains only 5 mathematical operations: addition, subtraction, multiplication, division and the square root and it draws a picture of what can be expected of the profiling results.

Table 3.4: Total number of required regular operations

	add	**sub**	**mul**	**div**	**sqrt**
Matrix vector mult.	$3{N_r}^2$	${N_r}^2$	$4{N_r}^2$	0	0
Vector norm	$2N_r$	0	$2N_r$	0	1
Complex conj. of vector	0	0	N_r	0	0
Dot product	$3N_r$	N_r	$4N_r$	0	0
Complex number abs.	1	0	2	0	1
Division	0	0	0	1	0
Symbolic subtotal	$3{N_r}^2 + 5N_r + 1$	${N_r}^2 + N_r$	$4{N_r}^2 + 7N_r + 2$	1	2
Subtotal	69	20	94	1	2
Total	282624	81920	385024	4096	8192

3.2.3 Memory requirement

The QMP algorithm as described in Section 3.1 makes use of a dictionary to store all possible coefficient vectors. Since the length of each of these vectors is equal to N_r, and the number of vectors is equal $2^{R_W N_r}$ the number of bytes required to store the dictionary grows rather rapidly when increasing either N_r or R_W. Considering that each number in every vector is a complex number, and assuming that 4 bytes

are used to represent each number, 8 bytes are required to represent a complex number. The full amount memory required is then shown by (3.5).

$$\text{bytes}_{\text{Dictionary}} = 8 \cdot N_r \cdot 2^{R_W N_r} \tag{3.5}$$

This means that for $N_r = 4$ and $R_W = 3$ a total of 131072 bytes are required to completely store the dictionary. For simple embedded processors this is a relatively large amount of memory, especially since this is the required memory only for the coefficient vectors and it does not include memory required for other variables or the program itself. For $N_r = 4$ and $R_W = 3$ this amount of memory is most likely realisable, however, when one would scale to higher numbers this memory requirement likely becomes a problem. Section 2.4 mentioned that in [2] it was decided to store the dictionary in an external memory unit as a means of tackling this memory requirement problem.

Having an external memory unit to store the dictionary solves the problem of memory requirement. The external memory unit can be chosen exactly such that it fits the dictionary values for a certain N_r and R_W. However, even though it solves the memory requirement problem, it also introduces multiple new problems, most of which are either practical or financial in nature. Now that there is an external memory unit, there needs to be an interface between this external memory and the CPU with its proprietary memories. An interface like this is almost guaranteed to be slower in terms of the required clock cycles to access the memory as compared to the CPU's proprietary memories. Other practical problems are the need for the external memory to be preloaded with its values before it can be used and the fact that if a change was to be made to the software that also requires an update to the coefficients in the dictionary, the external memory needs a separate update. These practical problems all together result in a financial one, since they imply that a slightly longer development time is required compared to an implementation without external memory, not to mention the slight added cost of the external memory unit itself.

Given the problems introduced by having an external memory raises the question if explicitly storing the dictionary is necessary. Section 3.1 mentions that the dictionary itself is the set of all permutations with repetition of the set of possible coefficient values. Permutations with repetition of a set of numbers are just all possible combinations that can be made using this set and a given length. In the case of the QMP algorithm, the set of numbers called W_{angle} is equal to all unique coefficients of which the size is determined by R_W, and the length is simply the number of antennas N_r. Given the set W_{angle}, generating the dictionary is rather simple and could even be achieved in software by using N_r nested loops. Since in the algorithm the dictionary

would be accessed on a vector by vector basis, it is very well possible to generate each vector from W_{angle} as soon as it is needed. This requires the program to store W_{angle} in memory, but it eliminates the need of storing the complete dictionary.

$$\text{bytes}_{W_{angle}} = 8 \cdot 2^{R_W} \tag{3.6}$$

The required number of bytes to store W_{angle} is shown by (3.6), and is significantly lower compared to the dictionary. For $N_r = 4$ and $R_W = 3$ no external memory would be required to store the 64 bytes of W_{angle}. However, generating the vectors of the dictionary as they are needed does require some time, and therefore a test should be done comparing the performance of an implementation where the algorithm makes use of a pre-computed dictionary and an implementation that generates each vector separately as they are needed.

3.3 Implementation

This section will discuss the considerations regarding the implementation of the QMP algorithm. It is important to mention this, since the way in which the algorithm is implemented decides the efficiency of the algorithm to start with. As mentioned in Section 2.2.1, the software side is always the first step of ASIP design and might arguably be the most important one. Since ASIP design mainly relies on software profiling for identifying potential hotspots, poorly designed software could badly influence the design choices made for the ASIP if the profiling results were to indicate a hotspot that would not necessarily need to be there.

Another part to the implementation are the simplifications that were made in the software. As can be seen in Appendix A, two matrices called `Rx_sqrt` and `Rx_sqrt_inv` are passed to the QMP algorithm. However, in the actual algorithm as proposed by [1], only the matrix `Rx` is passed to the algorithm, and the program itself needs to calculate the matrix square root and the inverse of the matrix square root. These two operations are left out, as they are relatively difficult to implement and their impact on performance of the algorithm will be rather small. This is because these operations happen outside of the algorithm loop, meaning that they only need to be performed once for every full run of the algorithm.

Furthermore, it should be mentioned that the implementation of the program will be done in such a way that the program scales with differing values of N_r and R_W. Even though most results of the ASIP design in this research focus on $N_r = 4$ and $R_W = 3$, it is very useful to have a program that scales for different values of these parameters. This way, any future research that considers other parameter values can still easily reuse this implementation by only updating the parameters in the software. Also, in this research it allows the program to easily scale to $N_r = 2$

and $R_W = 3$ so that a comparison can be made between the performance of this implementation and the implementation proposed in [2].

Lastly, after implementation of the algorithm the program will first be tested on a regular PC[1], as it allows for easier debugging and verifying the correctness of the outcome. The functional correctness of the program will be tested by comparing the results of the C program to the results of the Matlab script.

3.3.1 General considerations

The programming language of choice for this project is the C language. The use of either C or C++ is required in the ASIP Designer tool as the tool only generates compilers for these languages. The C language is a relatively low-level language providing a lot of control and most insight into what will happen on the assembly level. This insight and low-level control are important because the purpose of ASIP design is to run software/algorithms in the most efficient way possible (either in terms of power consumption, speed or both), implying the requirement for an efficient programming language.

The way in which software is implemented is an interesting point to consider, both from an efficiency standpoint as well as an ease of development standpoint. Often software is written in such a way that not a single concept or operation is written twice, but is rather implemented through a function (or method). Especially from an ease of development standpoint, this makes a lot of sense. It enables the programmer to write concise code in which the functionality of certain concepts or operations can easily be altered (as only the function that implements them requires change). Also, this provides easy reusability as the programmer can use the functions in other projects as well. However, implementing every single repeated operation through functions does result in overhead in the program, compromising its efficiency. Every time the program calls a function, a context switch is required where all required function arguments and the return address need to be pushed on the call stack, after which the program can jump to the function code and execute it. When the function has finished executing, the context needs to switch back to the point from which the function call was made while cleaning up the call stack accordingly. This indicates that a low-level software implementation sometimes is a trade-off between ease of development and efficiency of the software.

Implementing software as efficiently as possible is critical for ASIP design. This

[1] Compiled using GCC version 7.5.0 and executed on an Ubuntu 18.04.4 LTS distribution running on an Intel Pentium G5400

might make it seem like an obvious choice to not perform small and simple repeated operations through a function, so that a lot of overhead through function calls is prevented. However, in ASIP design it is useful to implement especially these types of operations (relatively simple, often used) through functions because of two reasons. Firstly, the ASIP designer tool performs its profiling mainly on a function level, meaning that most insight can be gained when there are a significant number of functions to profile. Secondly, especially the smaller kinds of operations that are often repeated are the perfect target to explore, since they might have potential to be implemented as an instruction into the instruction-set of the ASIP. Because of this reason, most of the software required for the QMP algorithm is implemented through functions, which are grouped together by category to form small libraries. These libraries are written for a fixed-point number representation (discussed in the next section), for handling complex numbers and for the required matrix and vector operations.

The C language does come with a way of reducing the overhead caused by function calls while maintaining ease of development with a technique called inlining. A compiler hint can be used to tell the compiler to try and inline certain functions at the place from where they would be called. This will be elaborated upon in Chapter 4.

3.3.2 Fixed-point number representation

Most of the time when real or fractional numbers are needed in a program, a floating-point number representation is used. Floating-point numbers are widely used in computing due to their ability to represent fractional numbers and their high dynamic range. However, operations on floating-point numbers are significantly more complex compared to integer arithmetic. Because of this reason, a lot of the more relatively simple processors do not come equipped with dedicated floating-point hardware. The behavior of floating-point numbers can often still be emulated on these kinds of processors, but then operations will take even longer. The Tzscale processor that is used for the ASIP design also does not come equipped with dedicated floating-point hardware. Because of this reason, in [2] it was decided to make use of a fixed-point number representation instead. A fixed-point number representation functions very similarly to an integer number representation, meaning that operations on these numbers are still relatively easy. The effect of a binary point is achieved by scaling the number to be represented in fixed-point format by a constant factor (a power of 2). A simple example of why this works is to imagine a number that could not be represented by a regular integer, like 0.5. If a scaling factor of 2 was applied to this number, the result would be 1, indicating that 0.5 can be represented in a fixed-point number format as long as the number is scaled with at least

a factor of 2. The original number can always be retrieved by dividing the fixed-point number by its scaling factor.

Figure 3.2 shows an example of a 32-bit fixed-point number, where a scaling factor of 2^{26} is used. The scaling factor results in a binary point being placed between bit 25 and 26. The binary point divides the number into an integer and a fractional part, where the fractional part represents the part of the number that is lower than 0. The plain 32-bit binary number of Figure 3.2 is equal to 1630597136 in decimal. But when scaling this value back by the scaling factor of 2^{26}, it is revealed that this fixed-point number actually represents the value 24.29779. A fixed-point number representation does not require its binary point to be placed at a specific fixed location. Arithmetical operations can still be done on fixed-point numbers that have their binary points at different locations. To achieve this, the binary points of the different fixed-point numbers need to be aligned before performing the operation.

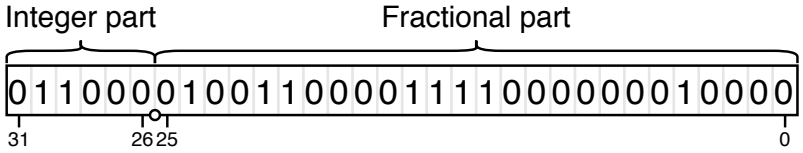


Figure 3.2: A fixed-point number with a scaling factor of 2^{26}, resulting in the binary point being placed between bit 25 and 26

For the QMP algorithm, the fractional part of the fixed-point numbers requires a granularity of 0.0001 [2]. In [2], it was decided to use 28 bits for the fractional part of the number, or in other words, a scaling factor of 2^{28} was used. This leaves 4 bits to store the integer part of the fixed-point number. Given the fact that this research will mainly consider a 4-antenna system instead of a 2-antenna system, it was decided to reduce to scaling factor to 2^{26}, in order to have some more integer bits. The algorithm could require this, since now intermediate results in matrix and vector operations could become a bit larger, since they result from larger matrices and vectors. All numbers will use the same scaling factor. This can be done, as 26 bits for the fractional part of the fixed-point number is sufficiently precise.

Given the fact that all fixed-point numbers in the implementation of the QMP algorithm have the same scaling factor, addition and subtraction works in the same way as when using a regular integer representation, i.e. no alignment of the binary point is required. Multiplication and division are a little bit more complicated. Let's say two numbers A and B are converted to a fixed-point format using scaling factor S and are then multiplied to become the result R. The result would be $AS \cdot BS = RS^2$. This multiplication yields an unwanted answer, as the result is now scaled with a factor

S^2 instead of just S. To get to the wanted answer the result should now be divided once by the scaling factor S. In order to keep full precision, the division by scaling factor S should be done after the multiplication is performed. This implies that ideally the multiplication needs to happen in a 64 bit number representation, as the result of a 32 bit number multiplication could require at most 64 bits to completely store the result. A similar problem appears when dividing two fixed-point numbers. The operation would look as follows: $AS/BS=R$. Now the result lost the scaling factor completely. In order to maintain full precision, number A needs to be scaled by the scaling factor S, resulting in $AS^2/BS=RS$, yielding the required result. To completely fit this operation, it needs to be performed on a type of 32 bits plus at least the number of bits required to represent S. So, in the case of a 32 bit CPU a 64 bit representation is a convenient choice.

The QMP algorithm also requires square root operations. Using the same logic as in the multiplication and division example, the ideal fixed-point square root operation goes as follows: $\sqrt{AS^2}=RS$, thus ideally requiring a 64-bit square root operation. Because a 64-bit square root operation is rather long and complex, in this implementation the square root operation will be performed first, after which the scaling factor is restored by multiplying the result by $\sqrt{S}$, just like it was done in [2]. The fact that the results will be a little less precise is not a problem, as Chapter 4 will explain that the square root operation is not strictly required for the algorithm and can thus be left out.

A summary of all fixed-point operations can be found in Table 3.5. The table clearly shows that the addition and subtraction operations can be performed normally, but that the other operations require extra steps to get to the required answer. The multiplication and divisions by the scaling factor S are handled by bit shifts in the software (therefore requiring the scaling factor to be a power of 2).

Table 3.5: Ideal implementation of fixed-point operations

Operation	**Implementation**
Addition	AS+BS = (A+B)S
Subtraction	AS-BS = (A-B)S
Multiplication	$AS \cdot BS=ABS^2 \rightarrow ABS^2/S=ABS$
Division	$AS \cdot S=AS^2 \rightarrow AS^2/BS=(A/B)S$
Square root	$AS \cdot S=AS^2 \rightarrow \sqrt{AS^2}=\sqrt{A}S$

All functionality required for the fixed-point number representation is bundled in a small library that can be included in the application code.

3.4 Verifying functional correctness

To verify that the C implementation functioned correctly, the results of the C implementation were compared to the results of the Matlab implementation, where the Matlab implementation was assumed to be correct. It was interesting to see that both implementations finally decided on a different 'best' set of coefficients for the same input, making it seem as if the C implementation was not correct. However, after closer inspection it appeared that the C implementation also calculated a very high correlation for Matlab's best result and vice versa. This indicates that the C implementation could still be correct, but calculates slightly different values. To be able to definitively decide on the correctness of the implementation, all correlations calculated by the C implementation were compared to the ones calculated by Matlab, for the same input data and the same order of coefficient vectors. The percentage difference for each of the outputs can be seen in the histogram of Figure 3.3.

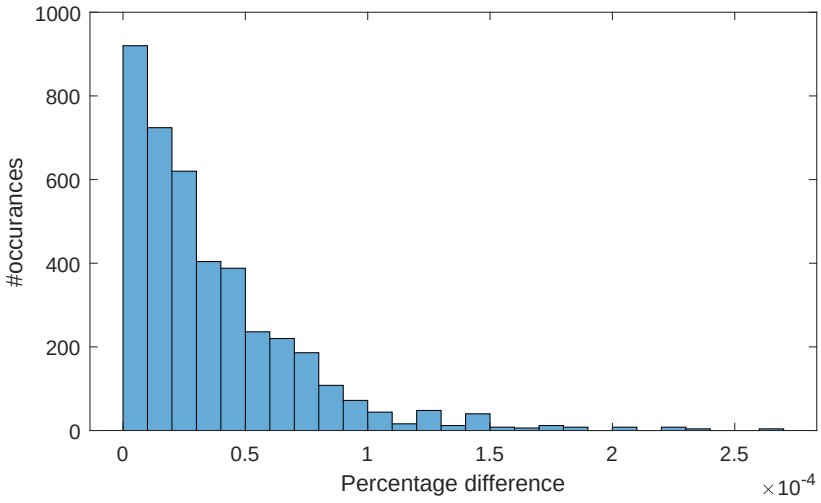


Figure 3.3: Histogram of all relative differences between the results of the C implementation compared to the results of the Matlab implementation using the default Tzscale processor

The histogram includes the percentage difference for áll calculated correlations, indicating that none of the correlations calculated by the C implementation differed significantly from the results of the Matlab implementation. This histogram shows that the C implementation does produce the correct results. The slightly different outcomes of the Matlab and C implementations is most likely due to their different levels of precision. The C implementation is written to make use of a 32 bit fixed-point number representation, where in Matlab its standard 64-bit double type was used.

Chapter 4

Running on Tzscale

The previous chapter discussed the QMP algorithm and its implementation in detail. This chapter will focus on taking that implementation to the Tzscale processor. First, the Tzscale processor (and the extended version as proposed in [2]) and its features are discussed. The algorithm will be executed on the processor using its instruction set simulator, after which the profiling results can be analyzed. The first profiling results then allow for some software optimizations, after which the best course of action for extending the processor is discussed.

4.1 The Tzscale processor

The Tzscale processor is based on the RISC-V ISA and comes modeled as an example project in the ASIP designer tool. The use of this processor for the ASIP is due to this research being a continuation of the research started by [2]. The processor is very well suited for this design problem, as it is a relatively simple processor built upon the open source extendable instruction-set RISC-V [22]. The main features of the Tzscale are:

- 32 bit wide data path, with an ALU and shifter.
- 16 or 32 field (configurable) central register file.
- load/store architecture, which supports 8, 16 and 32 bit memory transfers and an indexed addressing mode.
- 3 stage pipeline.
- single cycle multiplier and multicycle division/remainder unit.

The compiler that the ASIP designer tool generates for the Tzscale processor comes with a C runtime library, emulation of 64-bit integer operations, a floating-point emulation library and a math library. Especially the emulation of 64-bit integer operations

is useful for the fixed-point number representation as discussed in Section 3.3.2.

Instruction-set
The exact instruction-set that the Tzscale processor is built upon is the RV32I integer instruction-set of the RISC-V specification [23]. The implementation of RV32I does deviate slightly from the official specification, as it allows the designer to configure the processor to have either 16 or 32 registers. RV32I requires the design to have 32 registers, where a design with only 16 registers would be closer to the smaller RV32E specification. RV32I is a relatively simple instruction set, supporting most regular load/store, bitwise, control and integer arithmetic instructions. In RISC-V, the base ISA is often extended with the RV32M extension, providing the ISA with multiplication and division instructions.

Multiplication and division
The Tzscale implements its multiplication and division instructions in a way that is close to the standard RV32M extension, using the same base instructions and their encoding. However, the way in which the Tzscale implements the division instructions does not adhere to the RISC-V standard. The RISC-V explicitly specifies the required behaviour for division by 0, or overflow in the case of signed division. The Tzscale does not implement these cases. This means that the multiplication and division instructions of the Tzscale processor should be considered as a custom extension made to the base integer ISA. This is also in line with what is stated in the processor manual, as it does not mention RV32M in any way [23].

Square root
In [2], an extension to the Tzscale processor was made in the form of a 32-bit integer square root instruction. This instruction was added to support the QMP algorithm, as it appeared that the square root instruction required a significant amount of the required runtime. This square root instruction was added as a new separate extension to the ISA of the Tzscale. This implementation will also be considered in this work.

4.2 Generating the dictionary vectors

Section 3.2.3 discussed that the QMP algorithm as proposed by [1] would require a rather large amount of memory to store the complete dictionary of possible vectors, but that these vectors might just as well be generated in software using only the much smaller set of numbers called W_{angle}. To test if generating the dictionary vectors from W_{angle} in real time while running the algorithm does not require too

much time, two software versions were made: a version that uses a completely pre-computed dictionary and a version that generates the dictionary vectors from W_{angle} as they are needed. This generation of the dictionary vectors happens through a function which emulates the dictionary behavior, i.e., it gets passed an index i for which it returns what would be the i^{th} vector of the original pre-computed dictionary. Using the compiler generated by the ASIP designer tool, both implementations were compiled for an unmodified Tzscale processor. The results can be seen in Table 4.1.

Table 4.1: Results for a pre-computed dictionary vs. generation in real time

	Clock cycles		**Program size**		**Data size**	
Version	**#cycles**	**%**	**#bytes**	**%**	**#bytes**	**%**
Pre-computed	23387517	100.0	1826	100.0	164384	100.0
Generated in real time	23465332	100.3	1804	98.8	33376	20.3

The clock cycles column shows the number of clock cycles required to fully run the program. The program size represents the number of bytes required to store the complete program in the program memory, and the data size represents the number of RAM bytes required to store the data used by the algorithm. It should be noted that always at least 32768 bytes are reserved in the data memory. This memory is used for the (call) stack. The table clearly shows that generating each vector when it is needed is not significantly slower than loading the values from a pre-computed dictionary, as only 0.3% additional clock cycles are required. Also the program size remains roughly the same. Then the great advantage of not having to store the complete dictionary becomes very clear from the data size column, since 80% less data memory is required compared to the version where a pre-computed dictionary is used. The fact that generating each vector in real time is not much slower should not come as a big surprise, as generating a dictionary vector from W_{angle} is not a very computationally intensive task. It only requires the original index i passed to the emulation function to be translated to N_r indexes used to select the correct values from W_{angle}. This translation only requires an additional right shift (dividing by a power of 2) and a bitwise-and operation (modulo of a power of 2) for each index.

Since the advantage of the significantly reduced amount of memory outweighs the disadvantage of 0.3% extra clock cycles, the software version which emulates the behavior of the dictionary will be used in the remainder of this research.

4.3 Profiling

Using the compiler generated by the ASIP designer tool, the C implementation of the QMP algorithm as discussed in Section 3.3 that emulates the dictionary behav-

ior is compiled for an unmodified Tzscale processor. This implementation will be referred to as the 'naive' implementation, because all fixed-point number operations are implemented as function calls to the fixed-point number library, even though, for example, fixed-point addition and subtraction can be done in the exact same way as integer addition and subtraction. The compilation results in a program that requires 1804 bytes in the program memory. The ASIP designer tool also generates an ISS that is used to simulate the cycle-accurate behavior of the processor. The ISS produces all kinds of insights into the performance of the program and the processor, as it produces results on, for example, the number of clock cycles used by each function or the number of times that certain functions are called. The naive implementation will yield the most complete profiling results, as all major operations are shown as function calls. Completely running the algorithm takes 23465332 clock cycles. The profiling results can be seen in Table 4.2 below.

Table 4.2: Profiling results for the naive software implementation

Function	**% of total cycles**	**Function calls**	**Cycles per call**
Long division	22.04%	4096	1222
Fixed-point multiplication	21.56%	368704	13
Complex multiplication	17.57%	81936	51
Matrix vector multiplication	12.72%	4097	717
Complex addition	9.08%	81936	26
Integer sqrt	4.62%	8192	132
Fixed-point addition	3.61%	282672	3
Dot product	3.25%	4096	186
Norm of vector	2.04%	4096	117
Complex conjugate of vector	1.19%	4096	68
Dictionary emulation	1.06%	4096	61
Fixed-point subtraction	1.05%	81936	3

The table shows the most dominant functions in terms of relative clock cycle usage after a run of the algorithm. The number of function calls made is an interesting metric for two reasons. Firstly, a function that takes up a significant portion of the clock cycles with only a relatively small number of function calls could be an interesting target to explore for optimization. This is also represented by the cycles per call column. Secondly, the number of function calls can be compared to Table 3.2 and 3.4 to check if the number of function calls compares to the expected number of operations. All fixed-point operations should be compared to Table 3.4, and the complex operations should be compared to Table 3.2. The functions 'Long division' and 'Integer sqrt' in the table should be considered to be the fixed-point operations, since the

implementation of these fixed-point functions mainly relies on those two functions. The number of functions calls from Table 4.2 compares very well to the anticipated number of function calls as shown in Table 3.2 and 3.4, indicating expected behavior.

Another interesting thing that can be noticed in Table 4.2 is the appearance of the long division function. This function comes from the library for emulating 64-bit integer operations that is generated for the Tzscale along with its compiler. As mentioned in Section 3.3.2, a 64-bit number representation is used in the implementation of fixed-point multiplication and division. This results in the call to the long division function. The fact that this 64-bit division is completely emulated in software is the reason that it takes over a 1000 clock cycles to complete. The impact on 64-bit multiplication is less significant, as the compiler is still able to use the multiplication instructions of the processor[1] to achieve 64 bit behavior.

4.4 Software optimizations

Table 4.2 shows the most complete profiling results for the algorithm, where every single operation is called as a function. Now that this information is known, some software optimizations could be done which potentially remove functions and operations from the list. As discussed in Section 3.3.1, having function calls to a library with only relatively little and simple implementation provides a relatively large overhead. A prime example of this in the case of this implementation is the fixed-point number library, that sits at the bottom of the software and is used in every other user implemented library as well (complex-number and matrix operations). One way to remove overhead is to use the C keyword `inline`. The inline keywords hints to the C compiler to not actually call the function, but rather inline its implementation at the place from which the function is called. Inlining every single function in a software implementation is not a good idea, as it becomes increasingly difficult for the compiler to do so efficiently. Also, in such a case, a new kind of overhead appears in the form of more required program memory. Especially when larger function are inlined, a copy of that function is to be placed at every position in the software where it is called. For this reason, inserting the `inline` keyword is only done for the fixed-point library in this software implementation.

Another optimization can be done in the complex-number library. In the naive approach all arguments of the complex-number functions are passed by value. Since

[1] Especially the 'multiply high' or 'mulh' instructions are useful as they return the upper 32 bits of the 64 bit result

each complex number consists of a real and an imaginary part, every complex-number argument requires 2 values to be copied to the call stack. To try and reduce the required store word operations to store all these values on the call stack, most complex-number operations were updated to have their arguments passed by reference instead.

Lastly, a small change can be made to the algorithm: removing a mathematical operation while the end result stays the same. In the implementation of the algorithm shown in Appendix A, it can be seen that the decision of which coefficient vector is considered best is based on the if statement within the main algorithm loop. This if statement compares each end result of the mathematical operations to the previous result, where each time the biggest value is considered best. The latter is important: the mathematical correctness of the result is irrelevant, as long as the biggest value is chosen. The values that are compared in the if statement ultimately result from an operation like $\frac{\sqrt{A}}{\sqrt{B}}$, where A and B represent other mathematical operations. If the only thing of interest of all $\frac{\sqrt{A}}{\sqrt{B}}$ operations is to find the highest value, then considering the highest value of all $\frac{A}{B}$ operations would yield the same result. So, as a final optimization to the software the square root operations are not required and are therefore left out. The new profiling results for the optimized software can be found in Table 4.3 below.

Table 4.3: Profiling results for the optimized software implementation

Function	**% of total cycles**	**Function calls**	**Cycles per call**
Long division	37.08%	4096	1217
Complex multiplication	33.06%	81936	50
Matrix vector multiplication	10.93%	4097	372
Complex addition	6.07%	81936	10
Vector norm	3.47%	4096	107
Dot product	2.93%	4096	99

As expected, no fixed-point operations are shown anymore since the compiler was able to inline all fixed-point functions. The result of passing arguments by reference can be very clearly recognized in the matrix-vector multiplication function. This function calls a lot of complex-number functions in order to perform its calculation. Now that these function arguments are passed by reference, it saves on a lot of load and store word operations. A similar effect can be seen in the complex addition function. In the naive approach, this function required 26 clock cycles to complete, whereas now it only requires 10 clock cycles. In the case of the complex addition function,

this is mainly due to the fact that now the fixed-point addition function is inlined, requiring a lot less load and store word operations moving arguments and return addresses around on the call stack. The effect becomes very clear when looking at the disassembly of the complex addition function for both software versions.

```
PC    Instruction   Assembly
---   -----------   --------------
172   10 11         addi x2, 16
174   fe c1 2e 23   sw x12,-4(x2)
178   fe b1 2c 23   sw x11,-8(x2)
182   46 0c         lw x11, 0(x11)
184   48 10         lw x12, 0(x12)
186   fe a1 2a 23   sw x10,-12(x2)
190   fe 11 28 23   sw x1,-16(x2)
194   f4 ff f0 ef   jal x1, -178
198   ff 81 20 83   lw x1,-8(x2)
202   ff c1 21 83   lw x3,-4(x2)
206   00 a0 a0 23   sw x10,0(x1)
210   00 41 a6 03   lw x12,4(x3)
214   00 40 a5 83   lw x11,4(x1)
218   f3 7f f0 ef   jal x1, -202
222   ff 81 21 83   lw x3,-8(x2)
226   ff 41 20 83   lw x1,-12(x2)
230   00 a1 a2 23   sw x10,4(x3)
234   00 01 a2 03   lw x4,0(x3)
238   ff 01 21 83   lw x3,-16(x2)
242   00 40 a0 23   sw x4,0(x1)
246   00 a0 a2 23   sw x10,4(x1)
250   10 15         addi x2, -16
252   00 01 80 67   jalr x0, x3, 0
```

Listing 4.1: Cadd() calling xadd()

```
PC    Instruction   Assembly
---   -----------   --------------
20    00 46 21 83   lw x3,4(x12)
24    00 05 a2 03   lw x4,0(x11)
28    00 06 26 03   lw x12,0(x12)
32    00 45 a5 83   lw x11,4(x11)
36    96 12         add x12, x4
38    95 8e         add x11, x3
40    00 c5 20 23   sw x12,0(x10)
44    00 b5 22 23   sw x11,4(x10)
48    00 00 80 67   jalr x0, x1, 0
```

Listing 4.2: Cadd() after optimization

It is clear that the software optimizations have already drastically improved the performance of the implementation. The new results in terms of the required number of clock cycles and the program size for each extra step of software optimization can be seen in Table 4.4 below.

Listing 4.1 shows the disassembly of the complex addition function for the naive approach. It is very clear that load and store word instructions are by far the most executed instructions in this function, while its only purpose is to add together two numbers. The call to the fixed-point addition function (`xadd()`) can be recognized from the two `jal` instructions. Listing 4.2 shows the disassembly of the complex addition function for the optimized software version. It is clear that inlining the fixed-point addition function has a significant effect, both on the size of the complex ad-

dition function as well as its speed. Also, the behavior of the function can now be recognized more easily. The function starts by loading the 4 values that it requires, then simply adds them together, and stores only the two values of the result back on the stack.

Table 4.4: Results of software optimization for $N_r = 4$

	Clock cycles		Program size	
Software version	**#cycles**	**%**	**#bytes**	**%**
Naive	23465332	100.0	1804	100.0
Inlining	16275588	69.4	1772	98.2
Inlining, pass by ref.	14313224	61.0	1544	85.6
Inlining, pass by ref., without sqrt	13228786	56.4	1428	79.2

After all optimizations, the algorithm runs almost twice as fast and requires 20% less space in program memory. In addition, the significant effect on speed of inlining the fixed-point library becomes clear, as inlining alone cuts off over 30% of the clock cycles.

At this point, it could be interesting the compare the optimized software to the software implementation used in [2]. Table 4.5 shows the results while running the algorithm for $N_r = 2$, and also displays the results acquired in [2].

Table 4.5: Comparison to the results acquired by [2] for $N_r = 2$

	Clock cycles		Program size	
Software version	**#cycles**	**%**	**#bytes**	**%**
[2]	387642	100.0	18366	100.0
Naive	188349	48.6	1810	9.9
Inlining	150721	38.9	1778	9.7
Inlining, pass by ref.	140191	36.2	1542	8.4
Inlining, pass by ref., without sqrt	124153	32.0	1426	7.8

4.5 Discussion on profiling results

The results of the different optimized software versions show that moving around data inside the processor is a task that often takes up a lot of time. Optimizing the software using the `inline` keyword and passing values by reference instead of by value can drastically reduce the amount of data that needs to be moved around, therefore improving the speed of the algorithm. It might very well be possible that more of these kinds of optimizations are possible to further increase the performance of the algorithm by altering the software alone. However, for this research

the software optimizations were left at this point so that a sufficient amount of time can also be spent on modifying the processor.

Another interesting thing that appears from the profiling results is that the fixed-point number representation takes a heavy toll on the speed of the algorithm. This can mainly be attributed to the long division that is required for fixed-point division, but Table 4.2 shows that also fixed-point multiplication is responsible for a lot of the clock cycles. This makes it an interesting idea to maybe implement native support for fixed-point multiplication and division. Something like this should be possible, as the multiplications and division operations on fixed-point numbers are very similar to integer operations, except that they ideally require a larger type (i.e. a larger register in hardware) in order to keep full precision, and they require an additional bit-shift which by itself is not a difficult operation.

One could argue that adding SIMD instructions to assist the complex-number and matrix operations would also make sense. It could be very beneficial to have a small additional ALU such that some simple arithmetic operations could happen in parallel. This happens quite often, given that most complex-number operations require calculation for both the real and the imaginary parts and that matrix operations are just the same repeated operations on different data. However, adding this kind of support becomes quite difficult when the underlying data type is not yet natively supported by the processor.

These reasons make it the logical step to try and add native fixed-point support for the mathematical operations that are happening on fixed-point numbers in the QMP algorithm. This implementation will be discussed in Chapter 5.

Chapter 5

Modifying the Tzscale

This chapter will discuss the modifications that are made to the Tzscale processor. Chapter 4 and Section 4.5 showed that it would be very beneficial for the Tzscale processor and the QMP algorithm to have native support for fixed-point operations. The implementation of these new instructions is elaborated upon, as well as their verification and new results of the processor using the new instructions.

5.1 Native fixed-point support

In Section 3.2.2 it is shown that the mathematical operations that are required to run the QMP algorithm are: addition (add), subtraction (sub), multiplication (mul) and division (div). In Section 4.4 it was shown that the square root operation is not required for the algorithm, meaning that this operation does not require native fixed-point support in the processor. Fixed-point addition and subtraction can be done using the regular integer adder/subtractor, as long as the binary point in the two fixed-point numbers are aligned with each other. Given the fixed-point implementation as discussed in Section 3.3.2, this will always be the case. This means that an unmodified Tzscale processor already has native support for fixed-point addition and subtraction.

The Tzscale only requires new instructions for multiplication and division to natively support the required fixed-point operations. The multiplication and division will need to happen in the way that is shown in Table 3.5. Given that the scaling factor is a power of 2, i.e., the binary point is placed between two bits in the 32 bit representation, the problem of the scaling factor can be accounted for with additional bit shifts. Say the number of bits after the binary point is called N_f, then after a multiplication of two fixed-point numbers the result should be shifted N_f bits to the right to get the final answer. For a division instruction that divides A by B, A should first be shifted

N_f to the left before being divided by B. In the fixed-point library that was written for this algorithm, N_f is set by a macro definition, meaning that N_f cannot change during runtime (as it is not required to). It would be very convenient to still have control over N_f in software, as for different values of N_r and W_{angle} the requirements of the fixed-point representation might change. Given this, and the fact that N_f is known at compile time, the number of bits that should be shifted in a fixed-point multiplier and divider could be encoded into the instruction by an immediate value. A 5-bit immediate value would support the full range of bits in the 32-bit number representation, allowing the user to place the binary point at any place.

5.1.1 New instruction encoding

Now that the new required multiplication and division instructions are known, their encoding can be determined. Section 4.1 explained that the Tzscale processor adhered to the encoding and instruction format of the standard RV32M extension, but that it does not fully implement it and should thus be considered as a custom extension. RISC-V has reserved space in its opcode map specifically for custom extensions to the ISA. The base opcode map used for RISC-V can be seen in Table 5.1.

Table 5.1: RISC-V base opcode map, inst[1:0]=11 [22]

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

It is interesting to note that the non-standard multiplication and division extension of the Tzscale is still placed at the spot of the standard RV32M extension in the base opcode map (the standard place for RV32M is at OP, having the full opcode of 0110011). Since the default Tzscale implementation is already non-standard and will be differentiated from even further with the implementation of the fixed-point functionality, it would be a good idea to change the opcode from OP to one of the places reserved for custom instructions. In [2], the extension was implemented in the $custom$-0 space, but given the fact that this extension implementing the square root instruction is no longer required, the multiplication and division extension will be placed there. This means that every instruction encoding for a multiplication or division instruction starts with the opcode of 0001011 (on the LSB side).

When a custom extension is added to the RISC-V architecture, it does not necessarily have to adhere to the encoding formats as already specified by the RISC-V specification. Still, it would be wise to stay as close as possible to the already existing encoding formats, as a large deviation most likely requires a larger and more complex decoder. The original encoding format used for the multiplication and division instructions for the Tzscale can be seen in Figure 5.1 below.

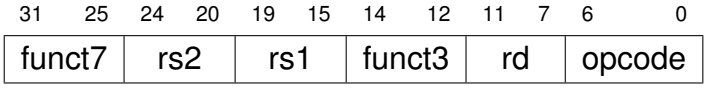


Figure 5.1: Standard R-type encoding as per the RISC-V specification

The previously discussed opcode can be seen at the LSB side. All registers are specified through the 5-bit fields rs2, rs1 and rd, being the two source registers and the destination register, respectively. The 3-bit and 7-bit funct parts can be used to select the exact instruction further, since an opcode is often used for multiple instructions of the same category. Sadly, the standard R-type encoding used in the default Tzscale cannot accommodate the 5-bit immediate value required for the new fixed-point instructions, so a change will have to be made.

A couple of things are important to keep in mind while designing the new encoding. Firstly, the location of the registers should not be changed in the encoding, as the decoder can then be constructed in such a way to always take the register addresses from the same location. Secondly, the opcode should also stay in its original position, as this is the first thing to be looked at by the decoder. And lastly, the MSB of any immediate value should ideally be placed at bit position 31, so in the case of a signed immediate value the decoder can always use that bit to check signedness [22]. Even though the 5-bit immediate value is unsigned the MSB can still be placed there. The new proposed instruction encoding including the 5-bit immediate value can be seen in Figure 5.2.

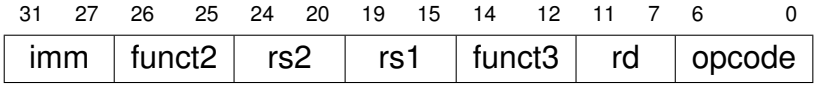


Figure 5.2: Custom instruction encoding including a 5-bit immediate

In the original encoding, funct7 was completely redundant in the case of a multiplication or division instruction, since it was set to 0x01 regardless of the chosen type of instruction. The choice between each division and multiplication instruction was made through funct3 only. In the new encoding, funct2 is used to distinguish multiplication from division. Only one bit is explicitly required for this, but since the immediate value never requires more than 5 bits, these 2 bits remained from the

original encoding that used funct7. Funct3 is used to specify the specific type of multiplication/division. Now that multiplication and division are distinguishable from the funct2 part, the funct3 part has new free space that could be used for new multiplication/division instructions in the future. The 5-bit immediate value is present in this new encoding as 5 bits on the MSB side. This immediate value can now be used to make a fixed-point implementation for any of the existing multiplication and division instructions. The nML file responsible for the definitions of the instruction encoding (”opcode.n”) was updated to support the new format.

5.1.2 Multiplication

The default Tzscale comes with 4 different multiplication instructions: `mul`, `mulh`, `mulhsu` and `mulhu`. The `mul` instruction performs a regular integer multiplication instruction yielding the lower 32 bits as the result, while the several different `mulh` instructions yield the upper 32 bits as the result. This means that the multiplier inside the Tzscale processor already works on a 64-bit register. This enables the processor to implement something like 64-bit integer multiplication fairly easily. The nML description for the multiplication instruction can be seen in the listing below.

```
1  opn alu_rrr_mul(op: funct3_mul_div, rd: mR1, rs1: mR1, rs2: mR2)
2  {
3      action {
4      stage DE:
5          switch (op) {
6          case mul    : mpyC  = mul    (mpyA=rs1,mpyB=rs2) @mpy;
7          case mulh   : mpyC  = mulh   (mpyA=rs1,mpyB=rs2) @mpy;
8          case mulhsu : mpyC  = mulhsu (mpyA=rs1,mpyB=rs2) @mpy;
9          case mulhu  : mpyC  = mulhu  (mpyA=rs1,mpyB=rs2) @mpy;
10         }
11         PD = mpyC;
12      stage WB:
13         rd = PD;
14     }
15     syntax : op " " rd "," rs1 "," rs2;
16     image  : funct7_muldiv.mul_div::rs2::rs1::op::rd::opc.mul_div_e;
17 }
```

Listing 5.1: nML description for the multiplication instructions of the default Tzscale

The nML description consists of a declaration, an action block, a syntax definition and an image definition. The declaration part has to contain all parameters that are

```
1 w32 mul(w32 a, w32 b)
2 {
3     return a * b;
4 }
```

Listing 5.2: PDG description for the `mul` instruction of the default Tzscale

required for the instruction, and names each of them. In the case of the mul instruction, this is the `funct3` part of the instruction encoding (as that is used to decode which multiply operation to execute), and all the registers on which the instruction operates. The action block describes what should happen for each pipeline stage. In the case of a mul instruction, a call is made to the corresponding PDG function in the decode execute (DE) stage, and the result is then put in its register in the write-back (WB) stage. The syntax definition represents the disassembly corresponding to the instruction and the image definition contains the exact instruction encoding. All parameters in the image definition are defined in a separate nML file called ”opcode.n”.

The `mul` instruction is the instruction of interest for adding native fixed-point support. When implementing a fixed-point mul instruction with an immediate 5-bit value that indicates the number of required bit shifts, the regular `mul` instruction can be executed when choosing that immediate value equal to 0. This means that the original `mul` instruction can be completely replaced by the new fixed-point one. The PDG description for the regular multiplication instruction is fairly simple, and can be seen in Listing 5.2.

New multiplication instruction

To implement the new `mulfxdpt` instruction, both the nML and the PDG descriptions should be altered. The nML description for the new fixed-point instruction can be seen in Listing 5.3. An extra parameter can be seen in the declaration, namely the immediate value as a `c5u` type. This type specifically represents an unsigned 5-bit constant, where the constant type indicates that it functions as an immediate. This immediate is passed to the new fixed-point multiplication PDG function through a new 5 bit transitory called `mpyI`. The other multiplication instructions do not require the immediate value, and thus remain unchanged. In the new nML description, the image definition now describes the new instruction encoding as discussed in Section 5.1.1. The new parameters `funct2_mul_div` (in the image definition) and `funct3_mul` (in the declaration) are added to the opcode file. The 5-bit immediate value can be seen at the start of the image definition.

```
opn alu_rrr_mul(op: funct3_mul, rd: mR1, rs1: mR1, rs2: mR2, imm: c5u)
{
    action {
    stage DE:
        switch (op) {
        case mulh     : mpyC  = mulh    (mpyA=rs1,mpyB=rs2) @mpy;
        case mulhsu   : mpyC  = mulhsu  (mpyA=rs1,mpyB=rs2) @mpy;
        case mulhu    : mpyC  = mulhu   (mpyA=rs1,mpyB=rs2) @mpy;
        case mulfxdpt : mpyC  = mulfxdpt(mpyA=rs1,mpyB=rs2,mpyI=imm) @mpy;
        }
        PD = mpyC;
     stage WB:
        rd = PD;
    }
    syntax : op " " rd "," rs1 "," rs2 ",imm:" imm;
    image  : imm::funct2_mul_div.mul_type::rs2::rs1::op::rd::opc.mul_div_e;
}
```

Listing 5.3: nML description for the multiplication instructions of the modified Tzscale

The PDG description should be altered such that it receives an extra argument representing the immediate value and that a shift is performed on the multiplication result. It is implemented in such a way that the complete instruction can still execute in one cycle. This should be possible, as shifting in hardware is not very complex. The new PDG description for the `mulfxdpt` instruction can be seen in Listing 5.4.

Just like the PDG descriptions for the different `mulh` instructions, the new PDG description makes an explicit 64-bit multiplication by using an `int64_t` type. The immediate value is received through the standard `w32` datatype, that is used in most Tzscale PDG descriptions. This is a requirement, since the `c5u` type cannot be

```
w32 mulfxdpt(w32 a, w32 b, w32 s)
{
    int64_t p = a * b;
    p >>= s[4:0];
    return p[31:0];
}
```

Listing 5.4: PDG description for the `mulfxdpt` instruction of the modified Tzscale

used in the PDG description. Defining the multiplication like this is sufficient, as it is possible for the multiplication operator to be synthesized.

5.1.3 Division

The default Tzscale comes with 4 different division related instructions: `div`, `rem`, `divu` and `remu`. The `div` and `divu` instructions perform signed and unsigned division, whereas the `rem` and `remu` perform the signed and unsigned modulo operation, respectively. All of these instructions make use of the same multi-cycle functional unit that is called div. This multi-cycle functional unit is required as a division operator often cannot be synthesized, meaning that its behavior has to be explicitly defined. This is mainly due to the fact that division in hardware is a lot more complex compared to multiplication. This also explains the need for a multi-cycle functional unit: the operation is best performed in multiple clock cycles. A division or remainder operation in the Tzscale can take up to 34 clock cycles to complete: 1 initialization cycle, up to 32 division iterations and 1 write back cycle. The regular `div` instruction is shown in the nML description in Listing 5.5.

```
1 opn div_rr(rs1: mR1, rs2: mR2, rd:c5unz)
2 {
3     action {
4     stage DE:
5          R[div_wad=rd] = wd = divC = divs(divA=rs1, divB=rs2) @div;
6     }
7     syntax : "div " "x"rd ", " rs1 "," rs2;
8     image  : funct7_muldiv.mul_div::rs2::rs1::funct3_mul_div.div::rd::opc.mul_div_e,
      ↪  class(div);
9 }
```

Listing 5.5: nML description for the `div` instruction of the default Tzscale

Much of the same nML can be recognized from the nML used for the multiplication instruction. The most striking difference is the lack of the write back stage in the action block. This is due to the division being done in a multi-cycle functional unit, where the write back is handled by the multi-cycle functional unit itself. The other instructions are all handled by their own separate nML descriptions. They can be found in Appendix C.

The behavior of the division and remainder instructions are again defined in the PDG language. Two different implementations are used: a behavioral description, to

be used by an instruction-accurate ISS, and an implementation model that is used for a cycle-accurate ISS and the RTL generation and synthesis. The behavioral description is more straightforward, and can use the division and modulo operators. This is because the ISS itself is compiled into a program to simulate the behavior of the hardware, where no RTL generation or synthesis is involved. The behavioral PDG descriptions can be seen in Listing 5.6. The description is easy to read, and is exactly equal to what would be a C code equivalent.

The behavioral description is used specifically for an instruction-accurate ISS. An instruction-accurate ISS makes an abstraction of the pipeline stages, and simulates a single instruction at a time. This is a simpler (and therefore faster) method of simulating the processor behavior compared to a cycle-accurate ISS. A cycle-accurate ISS does not require a behavioral description, as it uses the implementation model. The cycle-accurate ISS requires all information provided by the implementation model to provide cycle-accurate results. This is mainly due to the fact that not all instructions always take the same number of clock cycles to complete. This is also the case for the division instruction, as will be discussed later. The cycle-accurate ISS is the one that is used in this research. For completeness, a behavioral model is still included.

```
1 w32 divs(w32 a, w32 b) { return a / b; }
2 w32 rems(w32 a, w32 b) { return a % b; }
3
4 w32 divu(w32 a, w32 b) { return (uint32_t)a / (uint32_t)b; }
5 w32 remu(w32 a, w32 b) { return (uint32_t)a % (uint32_t)b; }
```

Listing 5.6: Behavioral PDG description for the division and remainder instructions of the default Tzscale

The PDG description for the implementation model is significantly longer, and a lot more complex to understand. It requires knowledge and understanding of how division is performed in hardware. According to the Tzscale processor manual, the algorithm that is used for division is called the ”iterative division algorithm” [23]. What is most likely meant by this statement, is that a division result is achieved through an algorithm that requires multiple iterations, as is often the case. More specifics on the implementation are not given in the manual.

A relatively simple method for doing division in hardware that is very common is a shift/subtract division algorithm [24]. After close inspection, it appears that the Tzscale division implementation is also an iterative shift/subtract division algorithm. Lets say that in a division of $\frac{A}{B}$ A is called the dividend and B is called the divider. The

result of this division is called the quotient, and the value that remains is called the remainder. In a shift/subtract division algorithm the dividend is placed in a double register, where it is loaded on the rightmost side of the register. The divisor is stored in a separate register. Lets consider a 4-bit example as shown in Figure 5.3. A 4-bit example can fully explain the inner workings of the algorithm, as it functions exactly the same for a higher number of bits, except that it will take more iterations before the final answer is calculated.

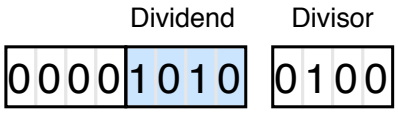


Figure 5.3: 4-bit example where the dividend is placed in a double register

The algorithm works by shifting the dividend one bit to the left, thus shifting the MSB of the right register into the LSB position of the left register. The left register is then added to the 2's complement of the divisor to calculate what is called a trial difference. The result of the trial difference is placed in a 4+1 bit register, where the extra bit is used to store the correct signedness of the answer. Until this point it was unknown what bit was to be placed at the LSB side of the right register after its shift operation, but this is where the trial difference comes in. The trial difference essentially indicates whether or not the divisor could be subtracted from the left register. If this was the case (sign bit equals 0), a 1 should be inserted at the LSB side of the left register, otherwise (sign bit equals 1, indicating a result that is still negative) a 0 should be inserted, indicating that the inverse sign bit is inserted. At the next shifting iteration these bits shift to the left along with all other bits. When the sign bit of the trial difference is set to 0, the contents of the left register will be replaced by the 4 bit trial difference result, since it was possible to subtract the divisor without resulting in a negative answer. The operation is completed once all bits of the dividend have been shifted into the left register. At that point the left register contains the remainder result and the right register contains the quotient. This behavior describes the exact way in which the Tzscale implements the algorithm. A visual representation of the algorithm performing the full operation for 4-bit numbers can be seen in Figure 5.4.

It should be noted that this implementation does not yet account for signed division, while signed division is possible given the instructions of the Tzscale. The Tzscale implements this as follows. When a signed division or signed remainder instruction is used, the division unit first checks the signedness of both operands. In the case of signed operands, the operands are converted to unsigned using 2's complement. The division unit then keeps track of two status bits to indicate what should be the

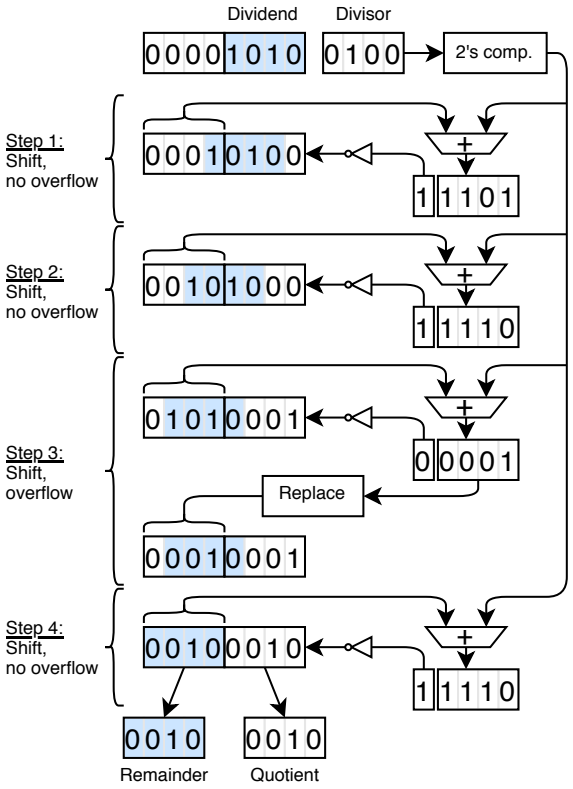


Figure 5.4: 4-bit example of the division algorithm in the Tzscale

signedness of the end result. The PDG description used for loading the operands into the division unit based on the signedness of the instruction can be seen in Listing 5.7. `divA` and `divB` are the transitories used to load the operands into the division unit. In the case of a division instruction, the end result is negative if the dividend was negative, or if the divisor was negative, but not both. In the case of a remainder instruction, the end result is negative if the dividend was negative. The operations determining the value of these status bits can be seen on line `10` and `11` in Listing 5.7. After the division unit has completed all its iterations, the correct signedness of the final answer is restored using these status bits, indicating whether or not to again take the 2's complement of the final result.

The Tzscale processor also implements initialization functionality which speeds up the division result. The default Tzscale processor uses 32-bit operands for the division, in 32 bit registers, meaning that it would take 32 clock cycles to fully run the shift/subtract algorithm. However, all consecutive 0 bits on the LSB side of the dividend will never result in a sign bit of 0 on the trial difference once they are shifted into the left register. Only when the first 1 is shifted into the left register there is a possibility for the sign bit to become 0. To speed up the division, the Tzscale counts this number of X leading 0 bits in the dividend on initialization of the division unit. The counter keeping track of the number of iterations is then initialized to $33 - X$,

```
if( divC_divs_divA_divB_div_DE_sig ||
    divC_rems_divA_divB_div_DE_sig ) {

    if(divA < 0) { divA_loc = -divA; }
    else         { divA_loc =  divA; }

    if(divB < 0) { divB_loc = -divB; }
    else         { divB_loc =  divB; }

    is_neg_div = (divA<0 ^ divB<0) ? 1:0;
    is_neg_rem = (divA<0) ? 1:0;
}
else {
    divA_loc =  divA;
    divB_loc =  divB;
    is_neg_div  = 0;
    is_neg_rem  = 0;
}
```

Listing 5.7: PDG description used to load the operands for the signed and unsigned case

and the dividend is shifted X bits to the left into the left register, inserting only 0's at the LSB side. The initialization of the counter is explained by the way in which the counter is used. Previously, it was mentioned that the write back stage was not part of the nML description, as this should now be handled by the multi-cycle functional unit. The write back of the division unit is done in its last clock cycle, when the counter has a value of 1. When the division unit is idle, the counter should have a value of 0. So, in order to achieve the correct number of cycles for the division operation, the counter receives an offset of 1, resulting in the expression $33-X$.

New Division instruction

Now that the behavior of the multi-cycle functional unit implementing division is known, its behavior can be altered to support the fixed-point variant. The nML description for the new division instruction can be seen in Listing 5.8. Just like the nML for the new multiplication instructions, the `c5u` type is used to indicate the immediate value. The image definition is also updated to represent the new instruction encoding.

As mentioned previously, the algorithm for division used by the Tzscale can be scaled up to a higher number of bits. To implement fixed-point division in the Tzscale, the register in which the dividend is placed will be double the size, so that

```
opn div_rr(rs1: mR1, rs2: mR2, rd:c5unz, imm:c5u)
{
    action {
    stage DE:
         R[div_wad=rd] = wd = divC = divs(divA=rs1, divB=rs2, divI=imm) @div;
    }
    syntax : "div " "x"rd ", " rs1 "," rs2",imm:"imm;
    image  :
    ↪  imm::funct2_mul_div.div_type::rs2::rs1::funct3_div.div::rd::opc.mul_div_e,
    ↪  class(div);
}
```

Listing 5.8: nML description for the modified `div` instruction

the full range of possible scaling factors (through the 5-bit immediate) can be used. During the first cycle of the division where the dividend is placed in its register, it should be shifted the number of bits specified by the immediate value to the left, implementing the required behavior described in Section 3.3.2. Then, the division can be performed through all shift/subtract iterations. A division can now take at most 64 clock cycles, because the size of the dividend register has doubled. When the division unit is finished with all its iterations, the division result can be found in the lower 32 bits of the right register, and the remainder result can be found in the lower 32 bits of the left register. Also, all internal counters and the PDG functions supporting the multi-cycle functional unit should be updated to support the larger data type and larger number of cycles.

Lastly, division by 0 should be discussed. Section 4.1 mentioned that the Tzscale does not support division by 0. The reason for this lack of support is not due to the algorithm that is used for division. To the contrary, if the algorithm is executed exactly as shown in Figure 5.4 division by 0 would yield the result as specified in the RISC-V manual, namely that all quotient bits are set to 1 and the remainder is set to the value of the dividend). The RISC-V specification dictates specifically that no exception is raised in the case of division by zero [22]. Since the Tzscale counts the number of leading zero bits and skips all of these iterations, zeros are inserted into the quotient resulting in the wrong answer. The modified implementation of the Tzscale correctly accounts for division by 0, by considering it as a special case in the initialization cycle. In this special division by 0 case, all bits in the quotient register are set to 1, and the remainder register is set to the value of the dividend, therefore adhering to the RISC-V specification. The complete modified PDG implementation for the Tzscale can be found in Appendix C.

5.2 Implementation

The previous section explained the modifications to the Tzscale in detail, where both fixed-point multiplication and division are added through new instructions and alterations to the underlying hardware. Two more steps are required to be able to implement the new instructions into the already existing software: the use of the new instructions should be detailed in the compiler header files of the Tzscale processor after which the software should be updated to make use of the new instructions.

The compiler header files are a set of C++ header files forming the link between the PDG/nML implementation and the software side. The CHESS part of the ASIP designer tool can use the header files to compile the application code for the ASIP. An example from the processor header file regarding the int datatype, describing regular multiplication, can be seen in Listing 5.9.

```
1 promotion int operator*(int,int)   = w32 mul(w32,w32);
2 promotion unsigned operator*(unsigned,unsigned)   = w32 mul(w32,w32);
```

Listing 5.9: Operator promotion for multiplication on the default Tzscale, linking the multiplication operator to the PDG function

```
1 promotion int MUL_FXDPT(int,int,unsigned) = w32 mulfxdpt(w32,w32,w32);
2
3 inline int operator*(int a, int b) {return MUL_FXDPT(a,b,0);}
4 inline unsigned operator*(unsigned a, unsigned b) {return MUL_FXDPT(a,b,0);}
```

Listing 5.10: Making the new multiplication functionality accessible through an intrinsic function

Defining the behavior for certain operators can be done in multiple ways. In Listing 5.9 the promotion keyword is used, which is the most common way. Now, the new PDG function that implements the fixed-point behavior should be accessible in the application code. To achieve this, an intrinsic function can be used. Given the fact that the fixed-point multiplication instruction replaces the original one, regular multiplication should then execute the instruction through a call to the intrinsic function. This can be done using the inline keyword for the multiplication operator. The updated compiler header code that implements this behavior can be seen in Listing 5.10.

The division and modulo operators can be updated in the exact same way: making intrinsic functions for the different instructions, and linking them to their operators by inlining the intrinsic function. This implementation can be seen in Listing 5.11.

```
promotion int DIV_FXDPT(int,int,unsigned) = w32 divs(w32,w32,w32);
promotion int REM_FXDPT(int,int,unsigned) = w32 rems(w32,w32,w32);

promotion unsigned DIVU_FXDPT(unsigned,unsigned,unsigned) = w32 divu(w32,w32,w32);
promotion unsigned REMU_FXDPT(unsigned,unsigned,unsigned) = w32 remu(w32,w32,w32);

inline int operator/(int a, int b) {return DIV_FXDPT(a,b,0);}
inline int operator%(int a, int b) {return REM_FXDPT(a,b,0);}

inline unsigned operator/(unsigned a, unsigned b) {return DIVU_FXDPT(a,b,0);}
inline unsigned operator%(unsigned a, unsigned b) {return REMU_FXDPT(a,b,0);}
```

Listing 5.11: Making the new division functionality accessible through intrinsic functions

Given the implementation of the new PDG functions through the compiler header files, the behavior of regular integer arithmetic stays exactly the same. The fixed-point behavior can now be accessed by calling the intrinsic functions, and passing the number of bits after the binary point as a third argument. The fixed-point library was updated to make use of the new functionality, which can be seen in Listing 5.12.

```
//fixed-point multiplication
inline fixedpt xmul(fixedpt A, fixedpt B){
    return MUL_FXDPT(A, B, FIXEDPT_FBITS);
}

//fixed-point division, dividing A by B
inline fixedpt xdiv(fixedpt A, fixedpt B){
    return DIV_FXDPT(A, B, FIXEDPT_FBITS);
}
```

Listing 5.12: Implementation of the new intrinsic functions in the application code

5.2.1 Verifying functional correctness

Now that the ASIP is modified with new fixed-point functionality, the functional correctness of the result of the application code should be re-verified to see whether the design behaves as expected. The verification will be done in exactly the same way as in Section 3.4: by a histogram showing all percentage differences between all outcomes of the application code running in the ISS and all outcomes of the Matlab implementation. Section 4.4 discussed that the square root operation was removed from the application code, as removing it did not influence the end result. However, since the Matlab implementation does make use of the square root operation, it was temporarily re-enabled in the C application code so their results could be compared. The result can be seen in Figure 5.5.

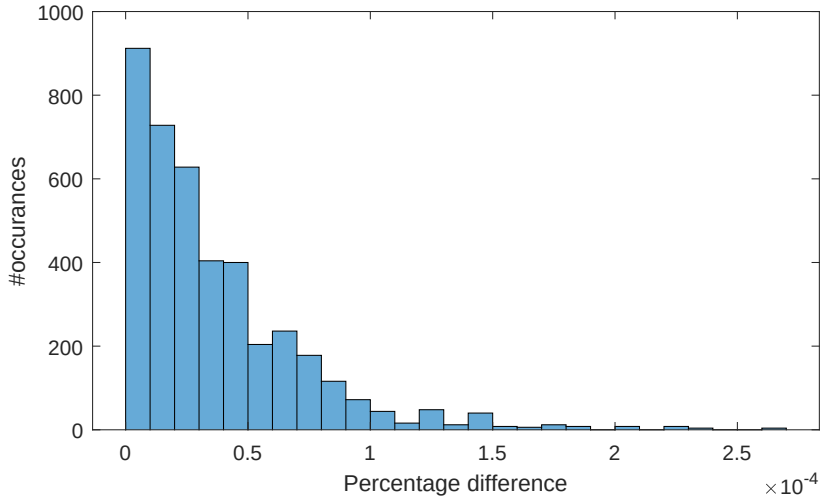


Figure 5.5: Histogram of all relative differences between the results of the C implementation compared to the results of the Matlab implementation using the modified Tzscale processor

The histogram shows an almost exactly equal percentage difference distribution as can ben seen in Figure 3.3. This is the expected result in the case that the new instructions function correctly.

5.2.2 VHDL simulation

Complete verification of the new instructions also includes a VHDL simulation. A simulation like this requires a complete hardware description of the processor. The ASIP designer tool provides GO for this purpose: it can translate nML and PDG to synthesizable RTL. A GO configuration file is used to instruct the tool of the required settings for the translation. In this case, it was chosen to generate VHDL along with

```
PC      Instruction     Assembly
____    ___________     ________________
20      00 40 21 83     lw x3,4(x0)
24      00 80 22 03     lw x4,8(x0)
28      01 80 65 13     ori x10,x0,24
32      d2 32 45 8b     mulfxdpt x11,x4,x3,imm:26
36      d4 41 c2 0b     div x4, x3,x4,imm:26
```

Listing 5.13: Top of the disassembly of the test program

a testbench specifically constructed to be used in ModelSim. The testbench can be used to start the simulation, where the testbench then loads a memory configuration file that should be generated from compiled application code. To test the new instructions, a simple test program was written to use the newly implemented instructions through the fixed-point library. The test program has two randomly chosen fixed-point numbers stored in its data memory: 0.6 and 0.21. Given the scaling factor of 2^{26}, these two numbers are stored in memory as the integers 40265318 and 14092861. The disassembly at the beginning of the program can be seen in Listing 5.13. The top shows two `lw` (load word) instructions, where both parameters are loaded from the data memory into registers `x3` and `x4`. The `ori` (or immediate) instruction is placed there by compiler optimization as this instruction is not required for the multiplication and division operations. The result of this instruction is later used at the end of the program. Then, both new instructions can be recognized, where the multiplication is executed before the division. It is interesting to note that the compiler placed the two source registers `x3` and `x4` in a different order for both instructions. The compiler can choose to swap the source registers for the multiplication instruction, since it is given the commutative property. This property tells CHESS that swapping the operands will still result in the same outcome.

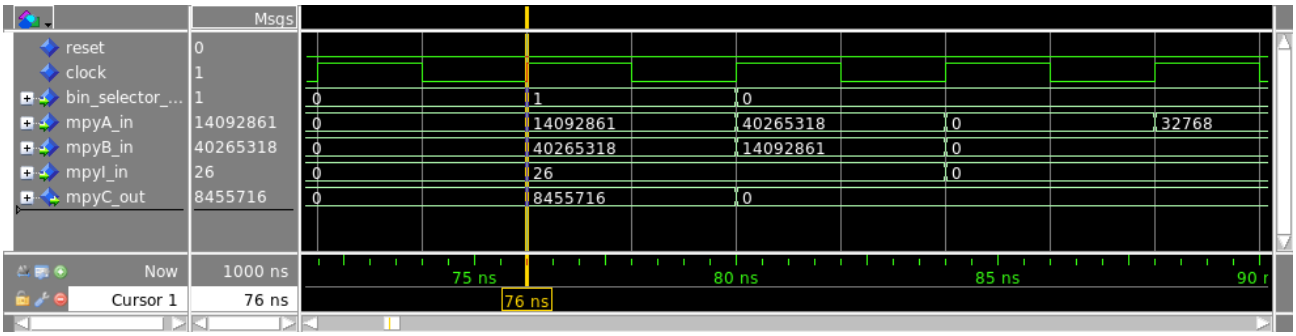


Figure 5.6: Simulation result of the new mul instruction

The VHDL simulation where the new fixed-point multiplication instruction is executed can be seen in Figure 5.6. The execution of the instruction starts at the spot where the yellow cursor is placed. The instruction executes in one clock cycle, thus providing its result on the output transitory `mpyC_out` on the same clock cycle. The input parameters can be seen in `mpyA_in`, `mpyB_in` and `mpyI_in`, where the latter is used for the immediate value. The expected result is the two input transitories multiplied by each other, and then scaled back by a factor of 2^{26}, yielding 8455716. This number can indeed be recognized as the result in `mpyC_out`, indicating correct behavior.

The beginning of the execution of the new division instruction can be seen in Figure 5.7, where the instruction starts executing at the yellow cursor. The input parameters can be seen in `divA_in`, `divB_in` and `divI_in`, just like the multiplication instruction. It appears that the transitories of the multiplication and the division unit share the same input, as they always show the same data. The first clock cycle is the initialization cycle, where the correct values are loaded into the local registers B and PA, with the divisor being stored in B and the (shifted) dividend being stored in PA. The shifted dividend has 12 leading zero bits, so the counter (called `cnt` in the figure) is initialized with $65 - 12 = 53$, meaning that there will be 52 division iterations and 1 final write-back cycle. Also, it can be recognized that a signed division instruction is being executed as the corresponding control signal is set to 1 (recognizable in the figure by `en_divC_divs_...`).

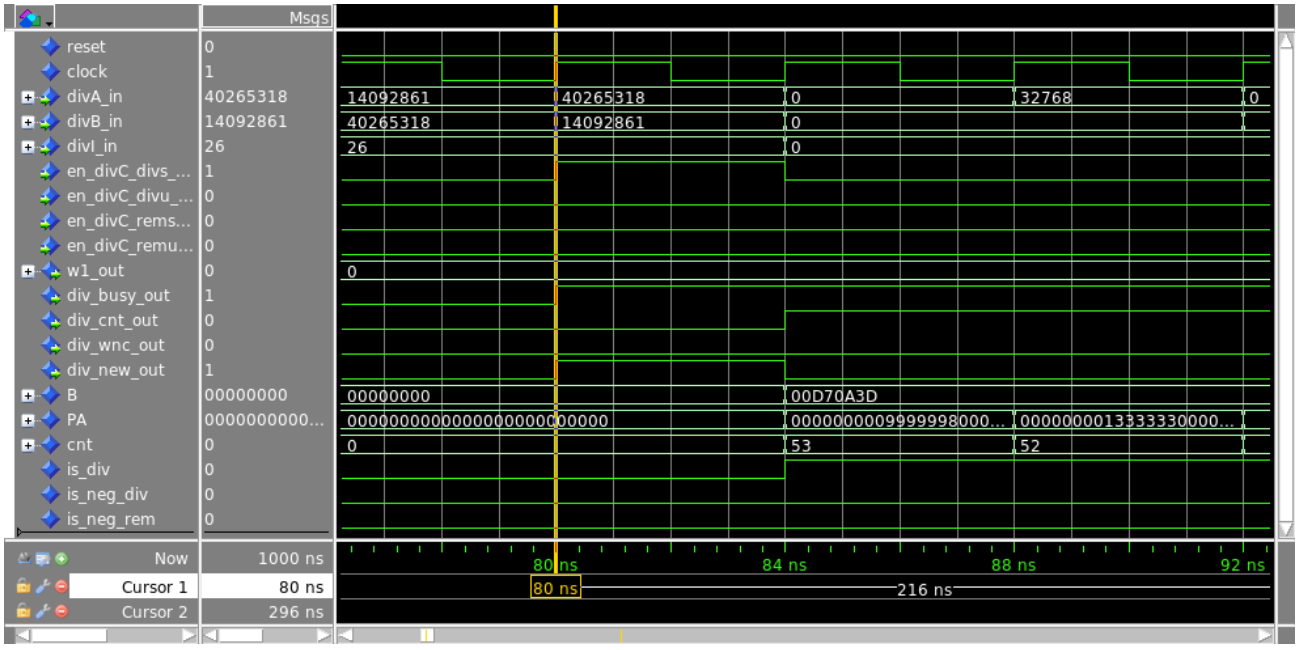


Figure 5.7: Simulation result of the new div instruction (at the start)

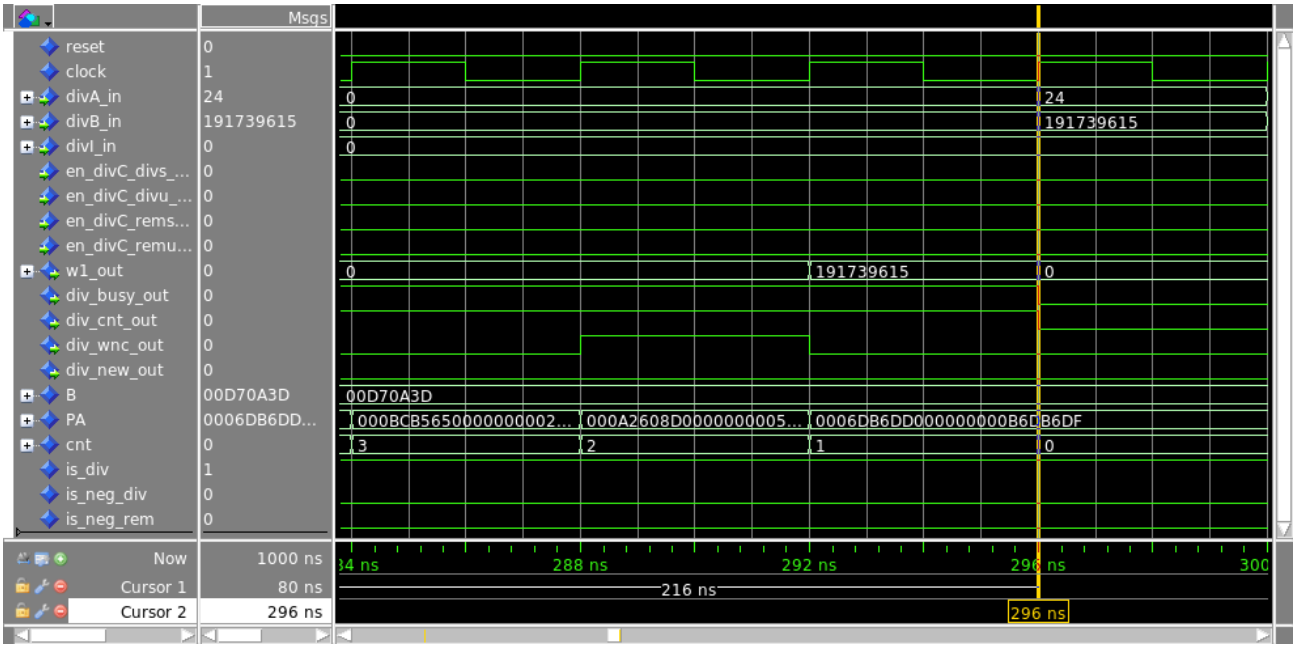


Figure 5.8: Simulation result of the new div instruction (at the end)

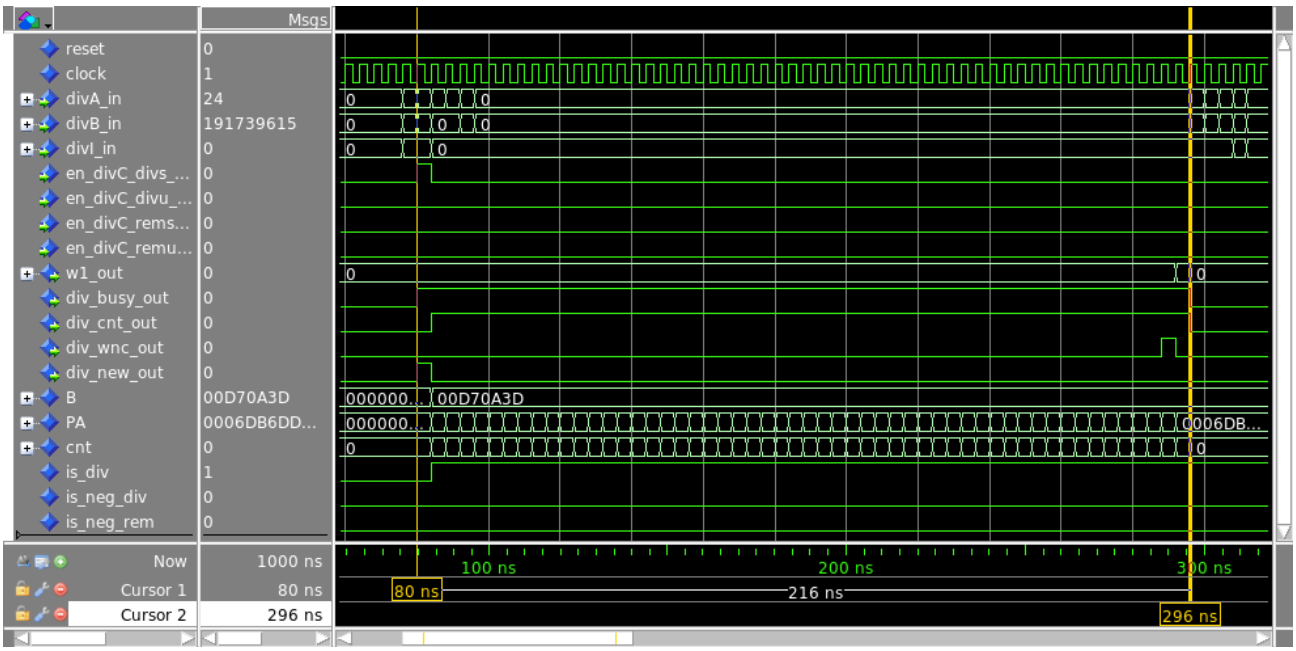


Figure 5.9: Simulation result of the new div instruction (fully shown)

The end of the execution can be seen in Figure 5.8. The cursor is now placed at the very end of the execution, after execution of the write back cycle. The last cycle of the division iterations can be seen in the clock cycle where the counter has a value of 2. After that cycle (counter has a value of 1), the division result is ready and can be seen on the signal `w1_out`. The expected division result is the value of the dividend scaled up by 2^{26} and then divided by the value of the divisor, yielding

191739615. This is indeed the value that can be recognized on `w1_out`, indicating correct behavior.

The complete execution of the division instruction can be seen in Figure 5.9. The time difference between both cursors is 216ns. A clock frequency of 250MHz was used for the simulation, meaning that the division instruction took 54 clock cycles from start to write back. This is to be expected: 1 initialization cycle, 52 division iterations and 1 write back cycle.

5.2.3 Comparison between VHDL and the ISS

The VHDL simulation discussed in the previous section showed correct behavior for the test program. While this shows correct behavior for the parameters used in the test program, the ideal test case would be a full comparison between the ISS and the VHDL simulation both running the application code for the QMP algorithm. This can be done by comparing so-called register log files that both the ISS and the testbench for the VHDL simulation can generate. These log files contain every write action to data memories and the processor's registers, for every clock cycle of the program. Every clock cycle in the file starts with the number of the clock cycle followed with the value of the program counter between parenthesis. If any writing was done in a clock cycle, this information is printed after it on the next lines.

The ISS was configured to output such a log file. After a full simulation, the log file of the ISS was compared to the log file of a VHDL simulation running the application code of the QMP algorithm. At first sight, the log files seemed similar, however, the order of writes to memory and register locations were ordered differently. An example of this, for the 19th clock cycle, can be seen in Listings 5.14 and 5.15. The actions within the 19th clock cycle are the same, however, both simulations added them to the file in a different order.

```
19 (164)
R[3] = 32812
DM1_3[4121] = 0
DM1_2[4121] = 0
DM1_1[4121] = 128
DM1_0[4121] = 44
```

Listing 5.14: Cycle 19 for VHDL

```
19 (164)
DM1_0[4121] = 44
DM1_1[4121] = 128
DM1_2[4121] = 0
DM1_3[4121] = 0
R[3] = 32812
```

Listing 5.15: Cycle 19 for the ISS

The different order of the information in the log files makes it difficult to compare the results by hand. Instead, a Python script was written to compare both log files. The

script gathers all write actions for a clock cycle both for the ISS log file and the VHDL log file. For both log files, these write actions are then put in a list. Both lists are then ordered, so they can be compared element by element. This operation is then done for every clock cycle. After running this python script on the lists is appeared that the contents of all lists were the same. This means that both the ISS and the VHDL implementation performed the same actions in every clock cycle, indicating equal behavior.

5.3 Results and evaluation

New results are produced for the modified Tzscale processor with native fixed-point support. Separate results are produced for a Tzscale with only a new multiplication instruction and a Tzscale with both a new multiplication and a new division instruction. This way the impact of both changes can be reviewed separately. The software version that was used to produce the result was the most optimized software version as discussed in Section 4.4. The new results in terms of the required number of clock cycles and program size are acquired through the newly generated ISSs and can be seen in Table 5.2 below.

Table 5.2: Results of the modified Tzscale for $N_\mathrm{r} = 4$

	Clock cycles		**Prog. size**	
Processor version	**#cycles**	**%**	**#bytes**	**%**
Unmodified	13228786	100.0	1428	100.0
Native fixed-point mul	9951410	75.2	1186	83.1
Native fixed-point mul and div	5151930	38.9	882	61.8

The support for native fixed-point multiplication results both in a faster algorithm as well as a smaller one. Now, only 1 clock cycle is required in order to achieve fixed-point multiplication, where in the first profiling results it was shown that the unoptimized software version required 12 clock cycles to do the same. Furthermore, the addition of native fixed-point division is even more impressive. This was to be expected, as the emulated long division took over a 1000 clock cycles to complete, whereas the new division instruction completes in 66 clock cycles or less.

For reference, the results for $N_\mathrm{r} = 2$ are shown in Table 5.3, where significant improvements are visible as well. It is interesting to note that the impact of native fixed-point multiplication support is relatively smaller compared to $N_\mathrm{r} = 4$, while the impact of fixed-point division support is relatively larger. This can be explained by looking at Table 3.4. The table shows that the number of divisions in one cycle of

the algorithm loop is constant, whereas the number of multiplications increases with higher values of N_r.

Table 5.3: Results of the modified Tzscale for $N_\mathrm{r} = 2$

	Clock cycles		Prog. size	
Processor version	**#cycles**	**%**	**#bytes**	**%**
Unmodified	124153	100.0	1426	100.0
Native fixed-point mul	107113	86.3	1184	83.0
Native fixed-point mul and div	32149	25.9	878	61.6

Now that the new and improved results in terms of the required number of clock cycles and program size are known, it is important to know the cost of the improvements in terms of both the required amount of hardware and the power consumption. To acquire these results, all different designs were synthesized for the UMC 65 nm technology[1], which is the exact same technology used in [2]. The synthesis is done for the same clock frequency as well, namely 250 MHz. This makes it possible to compare the synthesis results of the newly modified Tzscale processor to the synthesis results acquired in [2]. The synthesis results can be seen in Table 5.4.

Table 5.4: Synthesis results for UMC 65nm technology

	Area		Power		Energy	
Processor version	**µm²**	**%**	**mW**	**%**	**nWh**	**%**
Unmodified	45609	100.0	0.5721	100.0	8.41	100.0
Native fixed-point mul	46501	102.0	0.5783	101.1	6.39	76.0
Native fixed-point mul and div	49206	107.9	0.5905	103.2	3.38	40.2
Native integer sqrt [2]	46763	102.5	0.6029	105.4	NA	NA

The power numbers presented in Table 5.4 are for 10% toggle rate switching activity as calculated by the Synopsys Design Compiler. This is purely a metric to provide a means of comparing the power efficiency between different designs. The number does also allow for an approximation of the amount of energy required to run the algorithm. The energy column in Table 5.4 is calculated by multiplying the time required to run the algorithm by the corresponding power number. The results show that the algorithm runs significantly more efficient in terms of the required energy on the modified processors, but that this is mainly due to the increase in speed.

Lastly, the table shows that fixed-point multiplication support requires an additional 2% of area, where fixed-point division support then requires an additional

[1] The UMC 65 nm technology is available to the university via Europractice

5.9%. In total just under 8% extra area is required. The extra hardware that is required can be considered significantly more than the hardware that was required for the integer square root unit as implemented in [2], however, the performance gains are also a lot more significant (especially for $N_\mathrm{r} = 2$).

The increase in required hardware is especially striking for native fixed-point multiplication. A 2% increase might not seem like a lot, but when comparing it to the 2.5% increase in hardware for a completely new dedicated square root unit, it raises the question what all the extra hardware is required for. One way in which it might be explained is the fact that the multiplier already was one of the largest units in the Tzscale processor, only preceded by the register file [23]. The division unit is also large, but this is mainly due to the many internal registers and control logic required to run the algorithm. The multiplication operation performs in one clock cycle, meaning that all electrical signals have to propagate through the multiplier completely before the end of the clock cycle. With the addition of the variable shifter at the end of the multiplier, this path has become even longer. This could be an explanation for the relatively large increase, as all that extra hardware might be required to accommodate this longer path while still being able to reach a speed of 250MHz.

The final profiling results after running the most optimized software version on the Tzscale with full native fixed-point support can be seen in Table 5.5. Long division is no longer emulated in software and is therefore no longer present in the table. The division operation now happens through the inlined fixed-point division function which calls the new intrinsic for fixed-point division. Given the fact that the operation is inlined at the place where it is used, the clock cycles required to perform the division are added to the clock cycles for `main()`. The table clearly shows that the influence of the main function on the total number of clock cycles is not very significant, indicating that the division indeed happens a lot faster.

Table 5.5: Final profiling results

Function	**Percentage of total clock cycles**
Matrix vector multiplication	29.56%
Complex multiplication	22.25%
Complex addition	15.89%
Dot product	7.86%
Main	7.04%
Vector complex conjugate	5.40%

Chapter 6

Conclusion and future work

This chapter draws a conclusion on the work done in this thesis and will then discuss future work. The conclusion section will separately deal with each sub-question from Chapter 1 after which the main research question will be answered.

6.1 Conclusion

The goal of this thesis was to investigate the algorithm for finding the beamforming coefficients, both on the software and the hardware level, to increase speed, reduce the number of bytes required for storing the program and the required data, and decrease power consumption. This work was a continuation of the work started in [2], with the goal of further investigating the topic and improve upon the results. Now, each sub-question that was presented in Section 1.1 will be treated after which the main question can be answered.

What impact does the change of a 2-antenna system as used in [2] to an N-antenna system have on performance and required memory?

Chapter 3 thoroughly discussed how the algorithm scales for different numbers of antennas and the number of bits in the coefficient alphabet. It appeared that the complexity of the algorithm increases drastically for higher numbers of antennas, showing exponential behavior. Figure 3.1 inherently visualizes this, by varying the different parameters and showing the results on a logarithmic scale. The required memory size for the main parameters used in this research was still manageable at 131072 bytes. However, given that the required memory size also exponentially increases for a higher number of antennas, it was decided to not explicitly store the complete dictionary, but rather generate it as needed while executing the algorithm. It appeared that this drastically reduces the required memory size, while the computational overhead for on-the-fly generation is very small. This is best shown by

Table 4.2, where it can be seen that only 1.06% of the clock cycles is spent on generating vectors from the dictionary.

Does analysis of the algorithm without the context of the target architecture yield interesting results? Is there a relation between this analysis and profiling results after compilation for the target architecture? Is this relation as can be expected?

Analysis of the algorithm without the context of the target architecture in this research was mainly done through an extensive complexity analysis. For every part of the algorithm, the number of required operations was determined. This was done to see both the absolute number of operations required for a given number of antennas and how the number of operations scales for different parameters. Table 3.2 and 3.4 show these results. This method mainly provided insight into what kind of profiling results to expect, enabling the check whether the final software implementation performs as expected. It appeared that the actual software implementation running on the Tzscale indeed showed operations of which the number of times they were executed was very similar to the results from Table 3.2 and 3.4.

It must be noted, however, that most insights were gained using the profiling results presented in Chapter 4. These profiling results, along with a look at the disassembly that was generated for the compiled application code, resulted in the software optimizations that are discussed in Section 4.4. The results of the software optimizations might be some of the most interesting ones in this research. Due to software optimizations only, the required number of clock cycles was almost cut in half for $N_\mathrm{r} = 4$. For $N_\mathrm{r} = 2$, it was shown that the most optimized software version only required 32% of the clock cycles compared to the application code used in [2], truly showing the significance of research on the software level.

Which changes can be made to the ISA by arithmetical (e.g. complex-number operations) or control-oriented (e.g. hardware loop controller) instructions, to increase performance and efficiency?

The final profiling results in Chapter 4 and the interpretation in Section 4.5 showed that it would be very beneficial for the Tzscale to have native fixed-point support. Other, more involved arithmetical instructions would not yet make sense when the processor would not have native support for the main used datatype. Native fixed-point support was implemented into the Tzscale by modifying the existing multiplication unit and division unit, with an immediate value encoding into the instruction to indicate the scaling factor to be used for the fixed-point number. After the addition of support for native fixed-point multiplication and division, only 38.9% of the clock cycles were required compared to an unmodified Tzscale. This increase in speed came at a cost of 8% extra required chip area and an increased average

power draw of 3.2%. This is reasonable, however, as the increase in speed results in only 40.2% of the energy being required for the algorithm on the modified Tzscale.

Which changes should be made to the ASIP as proposed in [2] to increase performance and efficiency to find the beamforming coefficients for a 4-antenna system?

The three sub-questions have shown that performance and efficiency were successfully increased with the changes made to both the software and the Tzscale processor. The Tzscale was extended with native support for the required fixed-point operations, being able to find the beamforming coefficients in 5151930 clock cycles for $N_\mathrm{r} = 4$ using the new hardware. Also the effect of software optimizations is clearly shown, with the first implementation requiring 23465332 while the fully optimized software version reduced that to 13228786. This software optimization step was an important one, as it was also shown that the square root operation that was implemented into the Tzscale in [2] was unnecessary for the algorithm, as it yields the same results without it.

6.2 Experience with ASIP Designer

The experience of working with the ASIP Designer environment was one with mixed feelings. A large part of the negative side is due to ASIP Designer being quite overwhelming when starting off. ASIP Designer requires multiple languages to describe the processor and compiler behavior, most of which were unknown to me when starting off. Next to that, the environment comes with a lot of internal tools, most of which have to be used for a full design cycle. For example: CHESS is the retargetable C compiler, CHECKERS is the ISS generator, GO is the synthesizable HDL generator, DARTS is the assembler and disassembler and BRIDGE is the linker. It takes some time to get into it, and to realize which tools require specific interaction and which do not. Luckily, after some experience these things become less of a problem. There is one more thing to ASIP Designer that tends to get a little tedious: the number of different projects that are required for a full design cycle. First, there is the processor project, with its different files containing the full processor description. After this project is compiled, at least 4 more projects are required for the full design cycle: a separate project for each ISS, a project for generating the C runtime libraries, a project for HDL generation and an application project that uses the ISS and the processor model to simulate C code. Given the fact that most of these things belong together, it feels like more of the different projects could have been put together into one larger project.

ASIP Designer comes with a large set of manuals, providing at least one manual for each of the tools mentioned above. The manuals are quite detailed, and

all required information could be found in them. However, some inconsistencies were found in the documentation for the Tzscale. For example, the Tzscale processor manual stated that RV32I was implemented as the base instruction-set, while the read-me provided with the processor files stated that RV32E was implemented. Also, the processor manual could have been a little more extensive. For example, it would have been nice to see a more extensive description on the division unit. In this work, the division unit had to be mainly reverse engineered to find out how it worked.

The user interface, called ChessDE, looks and feels a little old from a visual perspective. Functionally, it works perfectly fine. After some time of working with, it is quite fast and switching between the different projects goes rather smoothly. The environment proved to be a powerful one, where once familiar with the different tools, a designer can extend and test a processor in a very short amount of time. One of the most important reasons for this is the retargetable compiler that comes with the environment, allowing C code to be compiled onto the custom processors. Development would have been significantly more difficult without the presence of this retargetable compiler.

6.3 Recommendations for future work

The work done in this research already was a continuation of the project reported in [2]. After this research, there is still a lot more that can be done. The following points are listed for potential future work:

1. The fixed-point support that was set up in this research can be further investigated.

   (a) A complete redesign of the division unit should yield better results. The existing division unit in the Tzscale can divide an N-bit number by an N-bit number in N bit shifts, while it should be possible to divide a 2N-bit number by an N-bit number in N-steps [24]. This is quite interesting for fixed-point support, as only the dividend requires a larger datatype, while the divisor remains the same size. Redesigning or modifying the division unit to resemble a restoring hardware divider [24] should yield better results.

   (b) It would be interesting to see the effect of hard-coding the number of bits after the binary point into the processor. This research focused on flexibility, allowing the programmer to choose the number of bits after the binary point by encoding it into the instruction through an immediate value. However, this flexibility did require the construction of a new type of instruction,

complicating the decoder of the processor. Also, variable bit shifters are required to implement the behavior for both the multiplier and the division unit. When the number of antennas and bits in the alphabet remain fixed, hardcoding the number of bits after the binary point is possible and should yield a processor requiring less area.

2. Different architectural changes could be explored.

   (a) Given the nature of the algorithm, it could be very interesting to add SIMD instructions. Both complex-number and matrix and vector operations could benefit greatly from this, as simple arithmetical operations could happen in parallel.

   (b) Smaller, less obvious changes to the ISA could be explored. For example, a branch immediate instruction could be added. Currently, the Tzscale only supports register-register branch operations. However, most of the branches in the disassembly are based of a constant number in the application code, often representing the number of antennas or the number of bits in the alphabet. Every time such a branch instruction is to be evaluated, the constant value first needs to be loaded from memory into a register. If the Tzscale would have register-immediate branch instructions, it would not require this constant value to be loaded from memory, as it could be encoded into the instruction (given that the constants are known at compile time).

3. The change denoted by point 2b is cannot be extracted from the analysis and profiling results in this research. This indicates that there might be other changes to the ISA that would be very beneficial but have not yet been found. It might be interesting to try and profile/analyze the behavior and the requirements of the code running on the Tzscale in more different ways than was done in this research.

4. The top-level design shown in Figure 2.4 can be considered. The interfaces between the different blocks connecting to the ASIP should be worked out and implemented, after which the ASIP could be implemented onto an actual FPGA to conduct real-world measurements and testing.

Bibliography

[1] V. Venkateswaran and A. van der Veen, "Analog beamforming in MIMO communications with phase shift networks and online channel estimation," *IEEE Transactions on Signal Processing*, vol. 58, no. 8, pp. 4131–4143, Aug 2010.

[2] A. Pohekar, "ASIP design on behalf of hybrid beamforming in MIMO communication system," Master's thesis, University of Twente, October 2019.

[3] R. Chataut and R. Akl, "Massive MIMO systems for 5G and beyond networks—overview, recent trends, challenges, and future research direction," *Sensors*, vol. 20, no. 10, p. 2753, 2020.

[4] R. Ramanathan, "On the performance of ad hoc networks with beamforming antennas," in *Proceedings of the 2nd ACM international symposium on Mobile ad hoc networking & computing*, 2001, pp. 95–105.

[5] B. Murmann, "Digitally assisted analog circuits," in *2005 IEEE Hot Chips XVII Symposium (HCS)*, Aug 2005, pp. 1–29.

[6] M. Willems, "Multicore design using application-specific instruction-set processors (ASIPs)," accessed: 12 March 2020. [Online]. Available: https://www.synopsys.com/designware-ip/technical-bulletin/multicore-design.html

[7] C. Liem, T. May, and P. Paulin, "Instruction-set matching and selection for DSP and ASIP code generation," in *Proceedings of European Design and Test Conference EDAC-ETC-EUROASIC*, Feb 1994, pp. 31–37.

[8] M. Gschwind, "Instruction set selection for ASIP design," in *Proceedings of the Seventh International Workshop on Hardware/Software Codesign (CODES'99) (IEEE Cat. No.99TH8450)*, March 1999, pp. 7–11.

[9] K. Karuri, M. A. Al Faruque, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr, "Fine-grained application source code profiling for ASIP design," in *Proceedings. 42nd Design Automation Conference, 2005.*, June 2005, pp. 329–334.

[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*, Jan 2001, pp. 76–81.

[11] Q. Chen, Q. Jinguo, Y. Zhang, and J. Ju, “Source code profiling for ASIP design: Strategy and implementation,” in *2011 International Conference on Electronics, Communications and Control (ICECC)*, Sep. 2011, pp. 1032–1035.

[12] Synopsys, “ASIP designer,” accessed: 16 March 2020. [Online]. Available: https://www.synopsys.com/dw/ipdir.php?ds=asip-designer

[13] Y. S. Mehrabani, M. Eshghi, and Y. S. Mehrabani, “Design of an ASIP processor for MD5 hash algorithm,” in *2012 20th Telecommunications Forum (TELFOR)*, Nov 2012, pp. 548–541.

[14] L. Gerlach, G. Payá-Vayá, S. Liu, M. Weißbrich, H. Blume, D. Marquardt, and S. Doclo, “Analyzing the trade-off between power consumption and beamforming algorithm performance using a hearing aid ASIP,” in *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, July 2017, pp. 88–96.

[15] A. M. Engroff, A. G. Girardi, M. V. Heckler, A. Winterstein, and L. A. Greda, “ASIP development of a real-time control module for a retrodirective antenna array,” *AEU - International Journal of Electronics and Communications*, vol. 109, pp. 31 – 42, 2019.

[16] A. R. Jafri, D. Karakolah, A. Baghdadi, and M. Jezequel, “ASIP-based flexible MMSE-IC linear equalizer for MIMO turbo-equalization applications,” in *2009 Design, Automation Test in Europe Conference Exhibition*, April 2009, pp. 1620–1625.

[17] P. Radosavljevic, J. R. Cavallaro, and A. de Baynast, “ASIP architecture implementation of channel equalization algorithms for MIMO systems in WCDMA downlink,” in *IEEE 60th Vehicular Technology Conference, 2004. VTC2004-Fall. 2004*, vol. 3, Sep. 2004, pp. 1735–1739 Vol. 3.

[18] T. Kaji, S. Yoshizawa, and Y. Miyanaga, “Development of an ASIP-based singular value decomposition processor in SVD-MIMO systems,” in *2011 International Symposium on Intelligent Signal Processing and Communications Systems (ISPACS)*, Dec 2011, pp. 1–5.

[19] X. Chen, A. Minwegen, Y. Hassan, D. Kammler, S. Li, T. Kempf, A. Chattopadhyay, and G. Ascheid, “FLEXDET: Flexible, efficient multi-mode MIMO detec-

tion using reconfigurable ASIP," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, April 2012, pp. 69–76.

[20] G. Peng, L. Liu, S. Zhou, S. Yin, and S. Wei, "A 2.92-Gb/s/W and 0.43-Gb/s/MG flexible and scalable CGRA-based baseband processor for massive MIMO detection," *IEEE Journal of Solid-State Circuits*, vol. 55, no. 2, pp. 505–519, Feb 2020.

[21] N. Yoshida, L. Lanante, Y. Nagao, M. Kurosaki, and H. Ochi, "A hybrid HW/SW 802.11ac/ax system design platform with ASIP implementation," in *2017 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS)*, Nov 2017, pp. 827–831.

[22] *The RISC-V Instruction Set Manual*, CS Division, EECS Department, University of California, Berkeley, May 2017.

[23] Synopsys, *ASIP Designer - Tzscale - Processor Manual*, O-2018.09 ed.

[24] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*. USA: Oxford University Press, Inc., 2000.

Appendix A

Matlab implementation

```
%Quantized Matching Pursuit, this runs on the asip
function W = QMP(Rx_sqrt, Rx_sqrt_inv, R_xs, w)
    % Pre-calculation
    rxs  = Rx_sqrt_inv * R_xs;
    w_wh = Rx_sqrt * w;

    % Find best match
    result = -inf;
    for i = 1 : size(w,2)
        w1 = abs(w_wh(:,i)'*rxs)/norm(w_wh(:,i));
        if w1 > result
            w_final = w_wh;
            result = w1;
        end
    end

    % Finalize
    W = Rx_sqrt_inv * w_final;
end
```

Appendix B

C implementation

Main file

```
1  //user file includes
2  #include "settings.h" //contains all setting macros
3  #include "matlab_data_gen.h"
4  #include "matrix.h"
5
6  //system file includes
7  #if TRACK_TIME_ON || SHOW_FINAL_RESULT_ON
8      #include <stdio.h>
9  #endif
10 #if TRACK_TIME_ON
11     #include <time.h>
12 #endif
13
14 //this function can be found at the end of this file
15 void dicionary_emu(const uint32_t idx, complex_n result_vec[NR_A]);
16
17 int main() {
18     complex_n w_vector[NR_A];       //to store a vector of the dictionary
19     complex_n w_vector_final[NR_A]; //the store the resulting vector of the
       ↪  dictionary
20     complex_n w_wh_vector[NR_A];    //to store the whitened version of a vector of
       ↪  the dicionary
21     complex_n rxs[NR_A];            //to store rxs (as a result of multiplying
       ↪  R_x_sqrt_inv with R_xs)
22     fixedpt weight = -2147483648;   //smallest possible value
23
24     #if TRACK_TIME_ON
25         clock_t start_time = clock();
26     #endif
27
```

```
28      ////// QMP - Quantized Matching Pursuit algorithm //////
29      multiply_matrix_by_vector(R_x_sqrt_inv, R_xs, rxs);
30
31      //algorithm loop, performing exhaustive search
32      for(uint32_t idx1 = 0; idx1 < W_SIZE; idx1++){
33          dicionary_emu(idx1, w_vector);
34
35          multiply_matrix_by_vector(R_x_sqrt, w_vector, w_wh_vector);
36
37          fixedpt norm = norm_of_vector(w_wh_vector);
38
39          cconj_of_vector(w_wh_vector); //take the elementwise complex onjugate of the
            ↪ vector
40          fixedpt abs_value = cabsolute(multiply_vector(w_wh_vector, rxs));
41          fixedpt new_weight = xdiv(abs_value, norm);
42
43          if(new_weight > weight){
44              weight = new_weight;                //update the weight to the new best
                ↪ value
45              for(uint32_t i = 0; i < NR_A; i++)  //update the weights vector to the
                ↪ newly found one
46                  w_vector_final[i] = w_vector[i];
47          }
48      }
49
50      #if TRACK_TIME_ON
51          clock_t end_time = clock();
52          double time_spent = (double)(end_time - start_time) / CLOCKS_PER_SEC;
53          printf("Time spent: %f seconds\n\n", time_spent);
54      #endif
55
56      #if SHOW_FINAL_RESULT_ON
57          printf("Final weight:\n%f\n\n", fixedpt_to_flt(weight));
58          printf("Final coefficients whitened:\n");
59          for(uint32_t i = 0; i < NR_A; i++)
60              printf("[%d] .r = %f, .i = %f\n", i,
                ↪ fixedpt_to_flt(w_vector_final[i].r),
                ↪ fixedpt_to_flt(w_vector_final[i].i));
61      #endif
62
63      return 0;
64  }
65
66
67  //fucntion that emulates the dictionary behavior. W_angle contains all unique
    ↪ numbers that reside in memory
```

```
68 //returns the vector (through w_vec[NR_A]) that would be at the idx'st place in
   ↪ memory
69 void dicionary_emu(const uint32_t idx, complex_n w_vec[NR_A]){
70     uint32_t W_angle_idx;
71 
72     //based on idx (ranging from 0 to W_SIZE-1) the required indexes for W_angle are
       ↪ calculated (ranging from 0 to W_ANGLE_SIZE-1)
73     for(uint32_t i = 0; i < NR_A; i++){
74         W_angle_idx = (idx >> ((NR_A - 1) * (NR_A - 1 - i))) & (W_ANGLE_SIZE - 1);
75         w_vec[i] = W_angle[W_angle_idx];
76     }
77 }
```

Complex number type and functions

```
1  #ifndef COMPLEX_N_H
2  #define COMPLEX_N_H
3  
4  #include "fixedptc.h"
5  
6  // Complex number struct, containing a fixedpt real and imaginary part.
7  typedef struct complex_n {
8      fixedpt r, i;
9  } complex_n;
10 
11 // Adds complex number a to b and returns the result.
12 complex_n cadd (complex_n a, complex_n b){
13     a.r = xadd(a.r,b.r);
14     a.i = xadd(a.i,b.i);
15     return a;
16 }
17 
18 // Subtracts complex number b from a and returns the result.
19 complex_n csub (complex_n a, complex_n b){
20     a.r = xsub(a.r,b.r);
21     a.i = xsub(a.i,b.i);
22     return a;
23 }
24 
25 // Multiplies complex number a with b and returns the result.
26 complex_n cmul (complex_n a, complex_n b){
27     complex_n res;
28     res.r = xsub(xmul(a.r,b.r), xmul(a.i,b.i));
29     res.i = xadd(xmul(a.r,b.i), xmul(a.i,b.r));
30     return res;
```

```
}

// Divides complex number a by b and returns the result.
// It is calculated by expanding and then simplifying
// the following: (a / b) * (cconj(b) / cconj(b))
complex_n cdiv (complex_n a, complex_n b){
    complex_n res;
    fixedpt div = xadd(xmul(b.r, b.r), xmul(b.i, b.i));
    res.r = xdiv(xadd(xmul(a.r, b.r), xmul(a.i, b.i)), div);
    res.i = xdiv(xadd(xmul(a.i, b.r), xmul(a.r, b.i)), div);
    return res;
}

// Returns the complex conjugate of the complex number.
complex_n cconj (complex_n a){
    a.i = -(a.i); //complex conjugate
    return a;
}

// Returns the aboslute value of the complex number.
fixedpt cabsolute (complex_n a){
    fixedpt abs_value;
    abs_value = xsqr(xadd(xmul(a.r,a.r), xmul(a.i,a.i)));
    return abs_value;
}

#endif //COMPLEX_N_H
```

Matrix operations

```
#ifndef MATRIX_H
#define MATRIX_H

// Calculates the product of a NR_A by NR_A matrix and a 1 by NR_A vector.
// Returns the result as a 1 by NR_A vector stored in res.
void multiply_matrix_by_vector(complex_n mat[NR_A][NR_A], complex_n vec[NR_A],
↪   complex_n res[NR_A]){
    for(uint32_t i = 0; i < NR_A; i++){
        res[i].r = 0;
        res[i].i = 0;
        for(uint32_t j = 0; j < NR_A; j++){
            res[i] = cadd(res[i], cmul(mat[i][j], vec[j]));
        }
    }
}
```

```

// Multiplies two vectors of 1 by NR_A and returns the scalar result.
complex_n multiply_vector(complex_n vec1[NR_A], complex_n vec2[NR_A]){
    complex_n sum = {.r = 0, .i = 0};

    for(uint32_t i = 0; i < NR_A; i++)
        sum = cadd(sum, cmul(vec1[i], vec2[i]));

    return sum;
}

// Returns the norm of a 1 by NR_A vector.
fixedpt norm_of_vector(complex_n vec[NR_A]){
    fixedpt sum = 0;

    for(uint32_t i = 0; i < NR_A; i++)
        sum = xadd(sum, xadd(xmul(vec[i].r, vec[i].r), xmul(vec[i].i, vec[i].i)));

    return xsqr(sum);
}

// Returns the complex conjugate of each complex element in the vector.
void cconj_of_vector(complex_n vec[NR_A]){
    for(uint32_t i = 0; i < NR_A; i++)
        vec[i] = cconj(vec[i]);
}

#endif //MATRIX_H
```

Fixed point implementation

```
#ifndef FIXEDPTC_H_
#define FIXEDPTC_H_

//system file includes
#include <stdint.h>

#ifndef FIXEDPT_BITS
        #define FIXEDPT_BITS 32
#endif

#ifndef FIXEDPT_WBITS
        #define FIXEDPT_WBITS 6
#endif

```

```
15 #if FIXEDPT_BITS == 32
16         typedef int32_t fixedpt;
17         typedef         int64_t         fixedptd;
18         typedef         uint32_t fixedptu;
19         typedef         uint64_t fixedptud;
20 #else
21         #error "FIXEDPT_BITS must be equal to 32"
22 #endif
23
24 #if (FIXEDPT_WBITS % 2) != 0
25         #error "FIXEDPT_WBITS must be an even number"
26 #endif
27
28 #if FIXEDPT_WBITS >= FIXEDPT_BITS
29         #error "FIXEDPT_WBITS must be less than or equal to FIXEDPT_BITS"
30 #endif
31
32 #define flt_to_fixedpt(ARG) ((fixedpt) (((float)ARG) * (1 << (FIXEDPT_BITS -
   ↪  FIXEDPT_WBITS))))
33 #define fixedpt_rconst(ARG) flt_to_fixedpt(ARG)        //legacy support, this macro
   ↪  was first called fixedpt_rconst
34 #define fixedpt_to_flt(ARG) ((float) (((float)ARG) / (1 << (FIXEDPT_BITS -
   ↪  FIXEDPT_WBITS))))
35
36 #define FIXEDPT_FBITS         (FIXEDPT_BITS - FIXEDPT_WBITS)
37 #define FIXEDPT_FMASK         (((fixedpt)1 << FIXEDPT_FBITS) - 1)
38 #define HALF_FIXED_FBITS (FIXEDPT_FBITS / 2)
39
40 #define FIXEDPT_ONE         ((fixedpt)((fixedpt)1 << FIXEDPT_FBITS))
41 #define FIXEDPT_ONE_HALF (FIXEDPT_ONE >> 1)
42 #define FIXEDPT_TWO         (FIXEDPT_ONE + FIXEDPT_ONE)
43 #define FIXEDPT_PI        fixedpt_rconst(3.14159265358979323846)
44 #define FIXEDPT_TWO_PI         fixedpt_rconst(2 * 3.14159265358979323846)
45 #define FIXEDPT_HALF_PI         fixedpt_rconst(3.14159265358979323846 / 2)
46 #define FIXEDPT_E        fixedpt_rconst(2.7182818284590452354)
47
48 fixedpt xadd(fixedpt A, fixedpt B);
49 fixedpt xsub(fixedpt A, fixedpt B);
50 fixedpt xmul(fixedpt A, fixedpt B);
51 fixedpt xdiv(fixedpt A, fixedpt B);
52 fixedpt xsqr(fixedpt A);
53
54 fixedpt xadd(fixedpt A, fixedpt B){
55         return A + B;
56 }
57
58 fixedpt xsub(fixedpt A, fixedpt B){
```

```
59          return A - B;
60  }
61
62  fixedpt xmul(fixedpt A, fixedpt B){
63      return ((fixedpt)(((fixedptd)(A) * (fixedptd)(B)) >> FIXEDPT_FBITS));
64  }
65
66  fixedpt xdiv(fixedpt A, fixedpt B){
67          return ((fixedpt)(((fixedptd)(A) << FIXEDPT_FBITS) / (fixedptd)(B)));
68  }
69
70  // calculates the integer square root of the fixedpt input number
71  // rounds the answer to the nearest integer
72  // this is based on the code found here (last accessed: 18-05-2020):
73  //
    ↪  https://stackoverflow.com/questions/1100090/looking-for-an-efficient-integer-square-root-alg
    ↪
74  fixedpt xsqr(fixedpt A){
75    uint32_t op  = (uint32_t) A;
76    uint32_t res = 0;
77    uint32_t one = 1uL << 30; // The second-to-top bit is set
78
79    // "one" starts at the highest power of four <= than the argument.
80    while(one > op)
81      one >>= 2;
82
83    while(one != 0){
84      if(op >= res + one){
85        op = op - (res + one);
86        res = res +  2 * one;
87      }
88      res >>= 1;
89      one >>= 2;
90    }
91
92    // Do arithmetic rounding to nearest integer
93    if(op > res)
94      res++;
95
96    return ((fixedpt) res) << HALF_FIXED_FBITS;
97  }
98
99  #endif //FIXEDPTC_H_
```

Setting macros

```
#ifndef SETTIGS_H
#define SETTIGS_H

/**
 * if set to 1, <time.h> will be included to keep track
 * of the time required to run an algorithm cycle
 */
#define TRACK_TIME_ON 1

/**
 * if set to 1, the final result (weight and coeffcients)
 * will be displayed
 */
#define SHOW_FINAL_RESULT_ON 0

#endif //SETTIGS_H
```

Appendix C

Tzscale Division

Default nML

```
opn div_rru(rs1: mR1, rs2: mR2, rd:c5unz){
    action {
    stage DE:
         R[div_wad=rd] = wd = divC = divu(divA=rs1, divB=rs2) @div;
    }
    syntax : "divu " "x"rd ", " rs1 "," rs2;
    image  :
    ↪  funct7_muldiv.mul_div::rs2::rs1::funct3_mul_div.divu::rd::opc.mul_div_e,
    ↪  class(div);
}

opn rem_rru(rs1: mR1, rs2: mR2, rd:c5unz){
    action {
    stage DE:
         R[div_wad=rd] = wd = divC = remu(divA=rs1, divB=rs2) @div;
    }
    syntax : "remu " "x"rd ", " rs1 "," rs2;
    image  :
    ↪  funct7_muldiv.mul_div::rs2::rs1::funct3_mul_div.remu::rd::opc.mul_div_e,
    ↪  class(div);

}

opn div_rr(rs1: mR1, rs2: mR2, rd:c5unz){
    action {
    stage DE:
         R[div_wad=rd] = wd = divC = divs(divA=rs1, divB=rs2) @div;
    }
    syntax : "div " "x"rd ", " rs1 "," rs2;
```

```
26      image   : funct7_muldiv.mul_div::rs2::rs1::funct3_mul_div.div::rd::opc.mul_div_e,
        ↪  class(div);
27  }
28
29  opn rem_rr(rs1: mR1, rs2: mR2, rd:c5unz){
30      action {
31      stage DE:
32           R[div_wad=rd] = wd = divC = rems(divA=rs1, divB=rs2) @div;
33      }
34      syntax : "rem " "x"rd ", " rs1 "," rs2;
35      image   : funct7_muldiv.mul_div::rs2::rs1::funct3_mul_div.rem::rd::opc.mul_div_e,
        ↪  class(div);
36  }
```

Modifed nML

```
1   opn div_rru(rs1: mR1, rs2: mR2, rd:c5unz, imm:c5u){
2       action {
3       stage DE:
4            R[div_wad=rd] = wd = divC = divu(divA=rs1, divB=rs2, divI=imm) @div;
5       }
6       syntax : "divu " "x"rd ", " rs1 "," rs2",imm:"imm;
7       image   : imm::funct2_mul_div.div_type::rs2::rs1::
        ↪  funct3_div.divu::rd::opc.mul_div_e, class(div);
8   }
9
10  opn rem_rru(rs1: mR1, rs2: mR2, rd:c5unz, imm:c5u){
11      action {
12      stage DE:
13           R[div_wad=rd] = wd = divC = remu(divA=rs1, divB=rs2, divI=imm) @div;
14      }
15      syntax : "remu " "x"rd ", " rs1 "," rs2",imm:"imm;
16      image   : imm::funct2_mul_div.div_type::rs2::rs1::
        ↪  funct3_div.remu::rd::opc.mul_div_e, class(div);
17
18  }
19
20  opn div_rr(rs1: mR1, rs2: mR2, rd:c5unz, imm:c5u){
21      action {
22      stage DE:
23           R[div_wad=rd] = wd = divC = divs(divA=rs1, divB=rs2, divI=imm) @div;
24      }
25      syntax : "div " "x"rd ", " rs1 "," rs2",imm:"imm;
26      image   : imm::funct2_mul_div.div_type::rs2::rs1::
        ↪  funct3_div.div::rd::opc.mul_div_e, class(div);
```

```
}

opn rem_rr(rs1: mR1, rs2: mR2, rd:c5unz, imm:c5u){
    action {
    stage DE:
         R[div_wad=rd] = wd = divC = rems(divA=rs1, divB=rs2, divI=imm) @div;
    }
    syntax : "rem " "x"rd ", " rs1 "," rs2",imm:"imm;
    image  : imm::funct2_mul_div.div_type::rs2::rs1::
    ↪  funct3_div.rem::rd::opc.mul_div_e, class(div);
}
```

Modified PDG

```
// Behavioural models for IS ISS

w32 divs(w32 a, w32 b, w32 s) { return (int32_t) ((int64_t) ((((int64_t) a) << s) /
↪  (int64_t) b)); }
w32 rems(w32 a, w32 b, w32 s) { return (int32_t) ((int64_t) ((((int64_t) a) << s) %
↪  (int64_t) b)); }

w32 divu(w32 a, w32 b, w32 s) { return (uint32_t) ((uint64_t) ((((uint64_t) a) << s)
↪  / (uint64_t) b)); }
w32 remu(w32 a, w32 b, w32 s) { return (uint32_t) ((uint64_t) ((((uint64_t) a) << s)
↪  % (uint64_t) b)); }

// Implementation model

uint97_t div_step(uint97_t pa, uint32_t b)
{
    uint97_t new_pa = pa << 1;
    uint34_t diff = (uint34_t)(new_pa[96:64]) - b;
    if (diff[33] == 0) {
        new_pa[96:64] = diff;
        new_pa[0] = 1;
    }
    return new_pa;
}

uint6_t div_clb(uint64_t a) // count number redundant leading bits
{
    uint6_t r;
    uint64_t tmp = a;
    if      (tmp[63: 0] == 0) r = 63;
    else if (tmp[63: 1] == 0) r = 63;
```

```
28      else if (tmp[63: 2] == 0) r = 62;
29      else if (tmp[63: 3] == 0) r = 61;
30      else if (tmp[63: 4] == 0) r = 60;
31      else if (tmp[63: 5] == 0) r = 59;
32      else if (tmp[63: 6] == 0) r = 58;
33      else if (tmp[63: 7] == 0) r = 57;
34      else if (tmp[63: 8] == 0) r = 56;
35      else if (tmp[63: 9] == 0) r = 55;
36
37      else if (tmp[63:10] == 0) r = 54;
38      else if (tmp[63:11] == 0) r = 53;
39      else if (tmp[63:12] == 0) r = 52;
40      else if (tmp[63:13] == 0) r = 51;
41      else if (tmp[63:14] == 0) r = 50;
42      else if (tmp[63:15] == 0) r = 49;
43      else if (tmp[63:16] == 0) r = 48;
44      else if (tmp[63:17] == 0) r = 47;
45      else if (tmp[63:18] == 0) r = 46;
46      else if (tmp[63:19] == 0) r = 45;
47
48      else if (tmp[63:20] == 0) r = 44;
49      else if (tmp[63:21] == 0) r = 43;
50      else if (tmp[63:22] == 0) r = 42;
51      else if (tmp[63:23] == 0) r = 41;
52      else if (tmp[63:24] == 0) r = 40;
53      else if (tmp[63:25] == 0) r = 39;
54      else if (tmp[63:26] == 0) r = 38;
55      else if (tmp[63:27] == 0) r = 37;
56      else if (tmp[63:28] == 0) r = 36;
57      else if (tmp[63:29] == 0) r = 35;
58
59      else if (tmp[63:30] == 0) r = 34;
60      else if (tmp[63:31] == 0) r = 33;
61      else if (tmp[63:32] == 0) r = 32;
62      else if (tmp[63:33] == 0) r = 31;
63      else if (tmp[63:34] == 0) r = 30;
64      else if (tmp[63:35] == 0) r = 29;
65      else if (tmp[63:36] == 0) r = 28;
66      else if (tmp[63:37] == 0) r = 27;
67      else if (tmp[63:38] == 0) r = 26;
68      else if (tmp[63:39] == 0) r = 25;
69
70      else if (tmp[63:40] == 0) r = 24;
71      else if (tmp[63:41] == 0) r = 23;
72      else if (tmp[63:42] == 0) r = 22;
73      else if (tmp[63:43] == 0) r = 21;
74      else if (tmp[63:44] == 0) r = 20;
```

```
75      else if (tmp[63:45] == 0) r = 19;
76      else if (tmp[63:46] == 0) r = 18;
77      else if (tmp[63:47] == 0) r = 17;
78      else if (tmp[63:48] == 0) r = 16;
79      else if (tmp[63:49] == 0) r = 15;
80
81      else if (tmp[63:50] == 0) r = 14;
82      else if (tmp[63:51] == 0) r = 13;
83      else if (tmp[63:52] == 0) r = 12;
84      else if (tmp[63:53] == 0) r = 11;
85      else if (tmp[63:54] == 0) r = 10;
86      else if (tmp[63:55] == 0) r =  9;
87      else if (tmp[63:56] == 0) r =  8;
88      else if (tmp[63:57] == 0) r =  7;
89      else if (tmp[63:58] == 0) r =  6;
90      else if (tmp[63:59] == 0) r =  5;
91
92      else if (tmp[63:60] == 0) r =  4;
93      else if (tmp[63:61] == 0) r =  3;
94      else if (tmp[63:62] == 0) r =  2;
95      else if (tmp[63]    == 0) r =  1;
96      else                      r =  0;
97
98      return r;
99  }
100
101
102 multicycle_fu div
103 {
104     reg cnt<uint7_t>;
105     reg PA<uint97_t>;
106     reg B<uint32_t>;
107     reg Q_addr_reg<uint5_t>;
108     reg is_div<uint1_t>;
109     reg is_neg_div<uint1_t>;
110     reg is_neg_rem<uint1_t>;
111
112
113     void process () {
114         uint7_t new_cnt = 0;
115         int64_t divA_loc;
116         int32_t divB_loc;
117         int32_t div_res;
118         int32_t rem_res;
119
120         uint1_t div_start = (cnt == 0) &&
121                             (divC_divu_divA_divB_divI_div_DE_sig ||
```

```
122                         divC_divs_divA_divB_divI_div_DE_sig ||
123                         divC_remu_divA_divB_divI_div_DE_sig ||
124                         divC_rems_divA_divB_divI_div_DE_sig );
125         if (div_start && (cnt == 0)) {
126
127             divA_loc = divA;
128
129             if( divC_divs_divA_divB_divI_div_DE_sig ||
130                 divC_rems_divA_divB_divI_div_DE_sig ) {
131
132                 if(divA < 0) { divA_loc = -divA; }
133                 else         { divA_loc =  divA; }
134
135                 if(divB < 0) { divB_loc = -divB; }
136                 else         { divB_loc =  divB; }
137
138                 is_neg_div = (divA<0 ^ divB<0) ? 1:0;
139                 is_neg_rem = (divA<0)? 1:0;
140             }
141             else {
142                 divA_loc =  divA;
143                 divB_loc =  divB;
144                 is_neg_div  = 0;
145                 is_neg_rem  = 0;
146             }
147
148             if(divB_loc == 0){ //to tackle division by 0
149                 uint97_t PA_tmp;
150                 PA_tmp[63:0] = 0xffffffffffffffff;
151                 PA_tmp[95:64] = divA_loc;
152                 PA = PA_tmp;
153                 new_cnt = 1; //skip directly to write stage
154             }
155             else{
156                 divA_loc <<= divI; //shift the bits as described in the immediate
157                 uint6_t n = div_clb((uint64_t)divA_loc);
158                 PA = (uint64_t)divA_loc << n;
159                 B = divB_loc;
160                 new_cnt = 65 - n;
161             }
162
163
164             Q_addr_reg = div_wad;
165             is_div = divC_divs_divA_divB_divI_div_DE_sig ||
166                      divC_divu_divA_divB_divI_div_DE_sig;
167
168         } else if (cnt > 1) {
```

```
169             PA = div_step(PA, B);
170             new_cnt = cnt - 1;
171         } else if (cnt == 1) {
172 
173             uint97_t pa = PA;
174 
175             if(is_div) {
176                 if( is_neg_div ) { div_res = -pa[31:0]; is_neg_div = 0; }
177                 else             { div_res =  pa[31:0];                 }
178                 R[Q_addr_reg] = w1 = div_res;
179             }
180             else {
181                 if( is_neg_rem ) { rem_res = -pa[95:64]; is_neg_rem = 0; }
182                 else             { rem_res =  pa[95:64];                 }
183                 R[Q_addr_reg] = w1 = rem_res;
184             }
185             new_cnt = cnt - 1;
186         }
187 
188         cnt = new_cnt;
189         div_busy = div_start || cnt > 0;
190         div_cnt = cnt > 0;
191         div_addr = div_start ? 0 : Q_addr_reg;
192         div_wnc = (cnt == 2);
193         div_new = (divC_divu_divA_divB_divI_div_DE_sig ||
194                    divC_divs_divA_divB_divI_div_DE_sig ||
195                    divC_remu_divA_divB_divI_div_DE_sig ||
196                    divC_rems_divA_divB_divI_div_DE_sig );
197     }
198 }
```