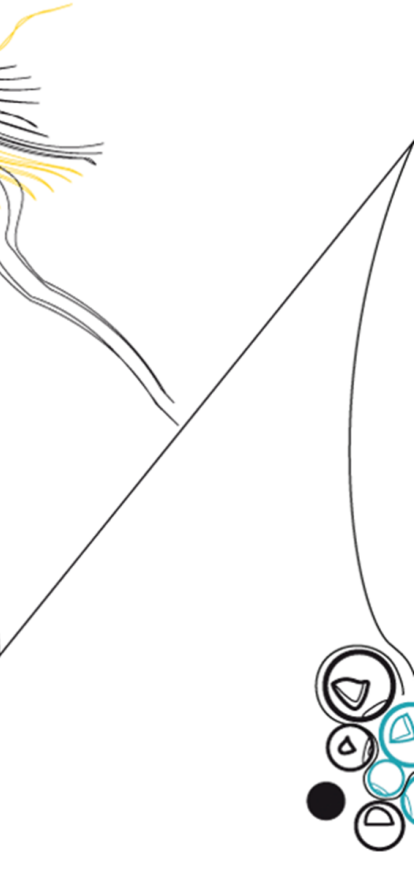
UNIVERSITY OF TWENTE.

**Faculty of Electrical Engineering,
Mathematics & Computer Science**

ASIP Design on behalf of hybrid beamforming in MIMO communication system

Ashwini Pohekar

Thesis Report
October 2019



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

In this thesis, an Application Specific Instruction Set Processor (ASIP) is developed to calculate optimum analog beamforming coefficients for a hybrid beamformer in a Multiple Input Multiple Output (MIMO) communication system. MIMO technology offers promising solutions to meet the increasing data-rate requirements. A lot of research is being carried out to improve the feasibility of these systems. Hybrid beamforming systems aim at reducing the problems faced by MIMO. Hybrid beamforming essentially involves beamforming in the analog as well as the digital domain. The ASIP proposed in this assignment is aimed at calculating optimum coefficient values for the analog beamformer. This thesis presents the different design decisions taken while developing the ASIP, the detailed design flow undertaken in the processor modeling tool and the implementation of the target application on a reference design. Additionally, comparison results against a floating point processor have also discussed to show the performance (and energy) efficiency of the designed ASIP.

Preface

This research is the product of collective efforts put in by many people and I take this opportunity to acknowledge their contributions. First and foremost, I would like to thank my daily supervisor Masoud Abbasi Alaei and my main supervisors dr. ir. Sabih Gerez who have been of immense help to me and without their guidance, this project would not have been possible. I express my gratitude for their interesting solutions for the problems I faced during work and all the encouragement that pushed me forward to deliver my best. I am also highly grateful to them for providing me with all the possible facilities required for the successful completion of the project.

I would also like to thank my committee members dr. ir. A. B. J. Kokkeler and dr. ir. M. S. Oude Alink for their valuable advice. Furthermore, I would like to thank A.C.R. Wijesundara Ranasinghe Appuhamilage for assisting me in the synthesis process and working with UMC 65 nm technology.

I would also like to extend my gratitude to L. J. Helthuis for all the assistance he provided in the tool installation process and while dealing with any technical issue.

At last, I would like to express my hearty gratitude to my parents and my friends for their unwavering faith in me and undying support that kept me strong emotionally through the entire journey of my graduate program.

Contents

List of acronyms

ASIP	Application Specific Instruction Set Processor
MIMO	Multiple Input Multiple Output
LNTA	Low Noise Transconductance Amplifier
ISA	Instruction Set Architecture
CMT	Chip Multi-Threaded
SIMD	Single Instruction Multiple Data
SDK	Software Development Kit
ISS	Instruction Set Simulator
PDG	Primitive Definition and Generation
ADC	Analog to Digital Converter
MMSE	Minimum Mean Squared Error
SISO	Single Input Single Output
TDMA	Time Division Multiple Access
FDMA	Frequency Division Multiple Access
CDMA	Code Division Multiple Access
MISO	Multiple Input Single Output
SIMO	Single Input Multiple Output
SDMA	Space Division Multiple Access
MMSEIC	Minimum Mean Square Error-Interference Canceller
MRB	Matrix Register Banks
CAU	Complex Arithmetic Unit
CU	Control Unit
SFU	Special Functional Unit
TTA	Transport Triggered Architecture

CGRA Coarse Grain Reconfigurable Architecture

MCMC Marko Chain Monte Carlo

WCDMA Wide Code Division Multiple Access

MCFU Multi Cycle Functional Unit

PULP Parallel processing Ultra Low Power platform

AWGN Additive White Gaussian Noise

ASIC Application Specific Integrated Circuit

List of Figures

List of Tables

Chapter 1

Introduction

Over the past several decades, use of Multiple Input Multiple Output (MIMO) technology in communication systems has increased substantially. Wi-Fi networks, cellular 3G / 4G LTE & 5G massive MIMO systems are a few prominent examples where MIMO technology is being used in modern communication infrastructure. MIMO is a promising technology to meet the growing demands of high data rate wireless communication. More recently, MIMO has been finding its way into rapidly growing markets such as professional broadcast video, law enforcement, and government sectors. Although, MIMO technology has already been put to use, a lot of research is being carried out in this field and still many questions are raised over its viability.

The introduction of multiple antennas at the transmitter and receiver increases the overall complexity of the system. This increased complexity is seen in terms of increased circuit size, power consumption and higher computation capacity requirement [1]. A promising solution to these problems lies in the concept of hybrid beamforming in MIMO communication systems. Hybrid beamforming involves the usage of analog beamforming in the RF domain and digital beamforming in the baseband domain. This concept was introduced by one of the authors in [2] and [3], in the mid-2000s. Hybrid beamforming was originally formulated keeping in mind MIMO communication systems which have arbitrary number of antennas but it was later also applied to massive MIMO systems. The interest in hybrid MIMO systems has accelerated over the past three years and various transceiver structures have been proposed in literature.

With this brief glimpse into the history of MIMO communication systems and short introduction to hybrid beamforming, in the next subsection, MIMO communication systems and their operational complexity are summarized.

1.1 MIMO communication systems

MIMO stands for Multiple Input Multiple Output. Figure 1.1 shows a 4x4 MIMO communication system. MIMO can be referred to as the communication channel created with multiple transmitters and receivers to improve performance of a communication system [4]. The data to be transmitted is split into multiple streams at the transmission point and recombined on the receiver side by another MIMO system configured with the same or different number of antennas. The receiver is designed to take into account the slight time difference be-

tween reception of each signal, any additional noise or interference, and even lost signals. MIMO is able to ascertain different paths over the air interface by using multiple antennas at both ends, thus creating sub-channels within one radio channel and increasing the data transmission (or capacity) of a radio link (or channel).

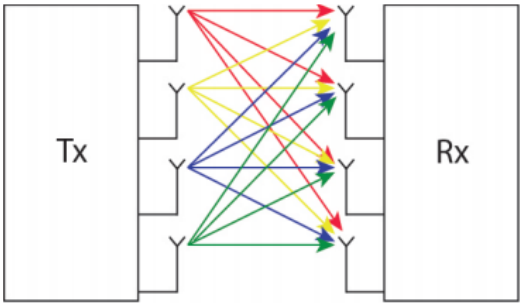


Figure 1.1: A 4x4 MIMO communication system [5]

Although, multiple transmitters and receivers help in overcoming the shortcomings of signal reflection and providing high data-rates, the design of such systems is a demanding task. In order to facilitate the transmission of multiple data streams, signal processing is involved both at the transmitter and the receiver. Precoding (done at the transmitter) and equalization (done at the receiver) are some of the signal processing operations involved in MIMO systems. All these operations are computationally complex and come at a reasonable computational cost (processing power).

In the presence of multiple data streams beamforming needs to be performed for directional transmission or reception of data. Traditionally in MIMO systems, this beamforming is performed in the baseband domain. This beamforming is generally performed by a digital signal processor. When beamforming is performed only in the digital domain, the area of the hardware required is large. The power consumed for beamforming is also quite high in such situations, especially for the Analog to Digital Converter (ADC)s. In case of a hybrid receiver system, the analog as well digital domain contribute towards the beamforming operation. This reduces the number of ADCs required and as result the area and power consumption also reduces.

Beamforming operation requires calculation of the optimal weights which help in recovering the original transmitted data streams. In case of a hybrid receiver, the process of calculation of these optimal weights needs to be performed for both the analog beamformer and digital beamformer. In this research assignment, the Application Specific Instruction Set Processor (ASIP) is used to perform the calculation of optimum weights for the analog beamformer. The next section introduces the concept of ASIPs and also explains the motivation of choosing ASIP design in this assignment.

1.2 ASIP

ASIP stands for Application Specific Instruction-set Processor and it refers to a special class of processors which are designed for an application domain. As a rule of thumb, general-purpose processors are designed keeping in mind that the maximum performance and flex-

ibility is achieved. The instruction set of these processors is such that, it is generic enough to support different types of common applications. Additionally, the compiler is such that it is capable of offering compilation for all programs and adapting to all programmers‘ coding behaviors. However, in case of ASIPs the instruction set is specifically developed such that execution of complex and frequently used functions in a given application is accelerated. So in contrast to general-purpose processors, the flexibility of an ASIP is kept sufficient enough instead of very high, while the performance is kept very high specific to the application.

An ASIP hardware architecture typically will contain a number of suitably designed application specific functional blocks and the necessary interconnects to move around data to/from memory blocks under the control of the top level controller (control circuit) of the processor. Due to their application oriented nature, ASIPs [6] allow alteration of hardware-software boundary to meet the speed and energy constraints of the target application while affording programmability and flexibility in functionality.

Figure 1.2 shows the comparison between flexibility and efficiency for different hardware configurations. On one end there are general-purpose processors which provide very high application flexibility but are relatively low in terms of power and performance efficiency. On the other end, there are hardwired datapaths which in principal offer almost no flexibility but offer very high power and performance efficiency. In between these two extremes are the ASIPs. ASIPs provide with hardware solutions which deploy classic techniques of parallelism and custom datapaths; while maintaining flexibility through software programming. Some examples of ASIPs are application specific DSP processors, accelerators, co-processors etc. Parallel processing Ultra Low Power platform (PULP) [7] and OpenPiton [8] are examples of open source platforms which deploy ASIPs (based on RISC-V, OpenSPARC instruction set resp.) designed for embedded vision, DSP computations, customizable parallel processing, etc.

The optimum weight calculation for analog beamforming to be implemented in the ASIP requires a lot operations based on the complex numbers and matrices. Hence, the design of an ASIP which has custom datapath for handling these complex operations is chosen. While providing the option of a custom datapath, the design of the ASIP will be flexible enough to handle changes in the search algorithm, if any, in the future. The ASIP is a solution which tries to provides the best of both worlds : flexibility and efficiency (power and performance). It is always possible that one or more hardware solutions are better out of the different options presented in the design space shown in Figure 1.2 . Hence, the choice of ASIP design must also be seen as a *design constraint* in this thesis assignment.

1.3 Problem statement

Having briefly discussed MIMO communication systems and ASIPs, the motivation behind this thesis assignment can be discussed. The introduction of multiple antennas at the transmitter and receiver side requires beamforming to be performed to recover the transmitted data streams. The main focus in this thesis assignment is on hybrid beamforming as presented in the following research papers [1], [10] and [11]. The in-depth information on working of MIMO communications systems, its drawbacks and beamforming has been provided in Chapter 2.

The proposed hybrid system in this research assignment is shown in Figure 1.3. In Fig-

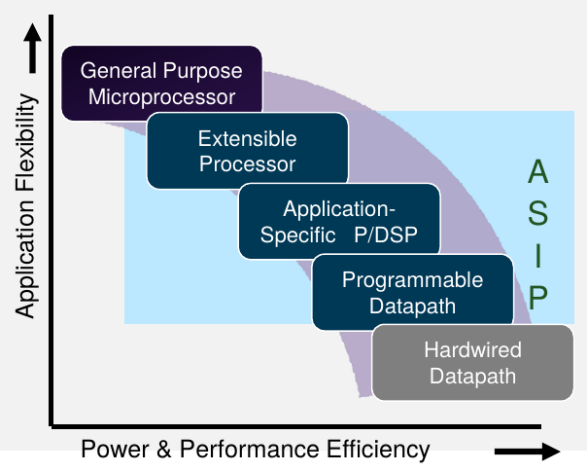


Figure 1.2: Flexibility vs Efficiency for different hardware solutions [9]

ure 1.3 the left most part is used to depict the transmitter along with interference signal. The transmitted signal is denoted by S(t) and the interference signal is denoted by $I_1(t)$. On the right hand side in Figure 1.3, across the multi-path channel H, the hybrid receiver system has been presented. The hybrid receiver considered in this assignment has 2 receiving antennas denoted by RF_1 and RF_2. The output of the receiving antennas is given to an analog beamformer. The analog beamformer consists of Low Noise Transconductance Amplifier (LNTA)s, phase shifters, clock generator and a final amplifier stage. The output of the analog beamformer is connected to the RF chain block. The RF chain consists of down-converter and ADC. The output of the RF chain block is given to the baseband processing block. It can be seen from Figure 1.3 that the baseband processing block consists of 5 components: Dictionary, Estimator, ASIP, Multiplier (Digital Beamformer) and Shift Registers. *The main aim in this assignment is to develop the ASIP in the baseband processing block.*

There is a feedback going from the baseband processing block to the analog beamformer. This feedback is calculated by the ASIP on the basis of a *search algorithm*. The idea is to perform an exhaustive search with the help of an ASIP to find optimum coefficient values for the analog beamformer and adapt the hybrid receiver based on the channel conditions. The output of this ASIP is given to the shift registers where a corresponding bit pattern is obtained. This bit pattern is the final feedback value given to the analog beamformer which determines the coefficient values by turning on/off switches in the analog circuit. The elaboration on the *search algorithm* and role of different blocks involved in baseband processing is provided in Chapter 2.

1.4 Goal(s) of the assignment/Research question(s)

The necessary background for this thesis assignment has been explained and now the research question can be formulated as : ***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***. Further subquestions for this research assignment are:

- Which open source instruction set architecture can be used as a reference upon which

+++

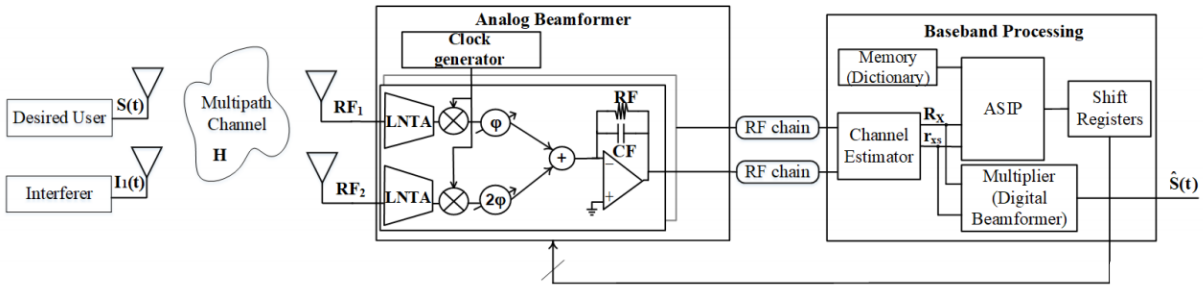


Figure 1.3: Hybrid beamforming structure at the receiver

the ASIP can be developed?

- Given the insights obtained by profiling of the algorithm on the chosen open source architecture, how can its performance be improved by an ASIP architecture optimized for the task?
- What design choices should be made while developing the architecture of the ASIP? For instance, How are complex numbers handled?, What must be the depth of the pipeline?, etc.

1.5 Report organization

The remainder of this report is organized as follows:

Chapter 2 discusses the concepts of MIMO communication, beamforming, formulation of the search algorithm, related work and the role of ASIP and other components in the baseband domain.

In Chapter 3, the choice of a reference open source instruction set architecture is discussed.

In Chapter 4, the work flow of processor modeling tool used in this research assignment is shortly described along with detailed explanation of how each step of processor design works.

Chapter 5 gives the necessary information about the Tzscale processor. The Tzscale processor is the reference open source design (based on RISC-V ISA) upon which the target ASIP is to be developed.

Chapter 6 explains the design methodology followed for the ASIP implementation.

The report then concludes with Chapter 7 on Results and Evaluation, and Chapter 8 on Conclusion and Future Work.

Chapter 2

Hybrid Beamforming in MIMO communication system

This chapter provides a detailed explnation of hybrid beamforming in MIMO communication systems. Initially, Single Input Single Output (SISO) systems along with the concepts of diversity and beamforming are presented. This is followed by a brief explanation about the working of MIMO systems and hybrid beamforming, and the formulation of the *search algorithm*. Subsequently, literature research is presented to show the different ASIPs which are currently being used in MIMO systems. The chapter finally concludes with a summary of the components involved in the baseband processing domain of the hybrid beamformer.

2.1 SISO to MIMO

SISO stands for Single Input Single Output and it is the conventional system technology used in communication. Generally, the signal transmitted from a single antenna is termed as the 'input', whereas signal received on a single antenna is termed as the 'output'. Cellular phones have a single antenna which communicates with a single antenna at the base station. There are multiple users present in a communication system at any given point in time and they require access to the cellular services simultaneously. In order to fulfill the requirements of each user, the signals to the users are separated in time (Time Division Multiple Access (TDMA)), in frequency (Frequency Division Multiple Access (FDMA)), or code (Code Division Multiple Access (CDMA)).

The features of the radio environment influence the quality of the communication link between the transmit and receive antenna. The signal strength will vary as the user moves over both a small and large scale. In some cases this variation can cause the quality of the link to become too low to deliver data successfully. This can cause radio link failure due to unacceptable error rates. This problem can be combated by using a technique called *diversity*. Diversity [12] relies on the use of multiple copies of the same signal, which the receiver can combine or select from. The idea behind it is that, even if one copy of the signal is of poor quality, it is unlikely that all the copies will be so, and therefore this redundancy allows the communication quality to be maintained.

The different types of diversity domains can be distinguished on the basis of how the multiple copies of the transmitted signal are generated. For instance, when multiple copies

of the same signal are transmitting multiple times, it gives rise to time diversity; when multiple copies of the same signal are transmitted at different parts of the spectrum, it gives rise to frequency diversity. Diversity can also be achieved using the space domain. When the same signal is transmitted from several base station antennas and received at a single mobile terminal (large-scale or site diversity), or a receiver has several spatially separated antennas each of which receives a different copy of the signal (small-scale diversity).

Transmit and receiver diversity techniques can be distinguished on the basis of which end of the communication link is under consideration. In transmit diversity techniques multiple copies of the same signal are transmitted from several antennas and their superposition is received at a single antenna. This diversity technique leads to the Multiple Input Single Output (MISO) system. In receive diversity techniques multiple copies of the same signal sent by a single transmit antenna are received at several antennas at the receiver. This diversity technique leads to the SIMO system.

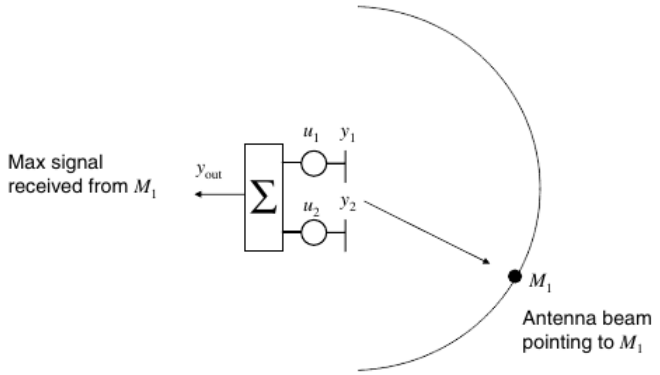


Figure 2.1: An example SIMO system [12]

Another way to classify diversity techniques is according to the *way the multiple copies of the signals are exploited*. In selection diversity the *best* copy of the signal is selected; in equal gain combining the multiple copies of the signal are added; and in maximum ratio combining the multiple copies of the signal are weighted by appropriately selected scaling factors such that a resulting signal of optimum quality is obtained. Figure 2.1 shows a communication system with a transmitting mobile M_1 and a receiving base station with two antennas. The signal transmitted from the mobile station is denoted as x and the signals received at the two base station antennas are indicated as y_1 and y_2 . The relationship between them is given in equation 2.1 [12]. Here, h_1 and h_2 are channel coefficients and, n_1 and n_2 are the noise signals at the two receive stations.

$$\begin{aligned} y_1 &= h_1 x + n_1 \\ y_2 &= h_2 x + n_2 \end{aligned} \tag{2.1}$$

The different diversity techniques for the system in Figure 2.1 can be de defined as:

- Output of a selection diversity receiver would be as shown in equation 2.2 [12]; i is the index of the maximum channel coefficient, h_{ij} is the channel coefficient between the i^{th} transmitted signal (here x_1) and j^{th} receiver antenna.

$$y_{sel} = \max |h_{11}, h_{12}| x_1 + n_i \tag{2.2}$$

- Equal gain combining receiver will align phases of the two signals and add the signals; the output will be as shown in equation 2.3 [12]. Here, u_1 and u_2 are the phase weights.

$$\begin{aligned} y_{equal} &= u_1 y_1 + u_2 y_2 \\ &= (u_1 h_{11} + u_2 h_{12})x + (u_1 n_1 + u_2 n_2) \\ &= (|h_{11}| + |h_{12}|)x + (u_1 n_1 + u_2 n_2) \end{aligned} \tag{2.3}$$

- In maximum ratio combining the phase weights will be adjusted such that the stronger signal is suitably scaled (along with phase alignment). In case of equal average noise power, the phase weights are proportional to channel coefficients $u_1 = h_{1*}$ and $u_2 = h_{2*}$. The output of the system then can then be defined as shown in equation 2.4 [12].

$$\begin{aligned} y_{equal} &= u_1 y_1 + u_2 y_2 \\ &= (u_1 h_{11} + u_2 h_{12})x + (u_1 n_1 + u_2 n_2) \\ &= (|h_{11}|^2 + |h_{12}|^2)x + (h_1^* n_1 + h_2^* n_2) \end{aligned} \tag{2.4}$$

Beamforming is the application of gains (or phase weights) to the signals transmitted or received from multiple antennas to obtain the desired transmitted signal. The phase weights (shown in the previous expressions) determine the formation of a beam. Figure 2.1 is an example of the SIMO system, if the situation is reversed i.e. the two antennas are now transmitters and M_1 is the receiver. This has now become an example of the MISO system. The application of weights at the antennas, for instance, at the transmitter allows it point the energy in specific directions. The appropriate choice of weights can also be used to nullify the energy in undesired directions. This is the basic principle of beamforming. In this way diversity can help in enhancing the system performance. However, there also some disadvantages to applying diversity techniques mainly use of more system resources. For example, when time diversity technique is used more time is used to send copies of the same data whereas this time could have been used to send new data. Use of multiple antennas leads itself to the consideration of space, hardware concerns and increased price. Another disadvantage is that diversity is a process of diminishing returns. This means that the benefit of adding for example an additional third antenna is smaller than the benefit from going from a single antenna to two antennas. Additionally, for diversity techniques to be effective the copies of the signal have to be independent, to minimize the probability that they all face simultaneously bad propagation conditions.

The evolution of diversity techniques specifically space diversity has lead to idea of using multiple antennas at the both the transmitter and receiver. This is the principle cornerstone of MIMO communication systems. Along with the added benefits of diversity, an additional benefit of using multiple antennas at the both communication ends is the ability to send several data streams simultaneously. This is termed as *spatial multiplexing*.

Since, MIMO allows multiple data streams to be transmitted simultaneously it allows to increase the data rate as opposed to conventional ways of increasing data rate : increasing transmitted power or increasing the bandwidth. The multiple antennas also allow for the accommodation of multiple users within the limited bandwidth.

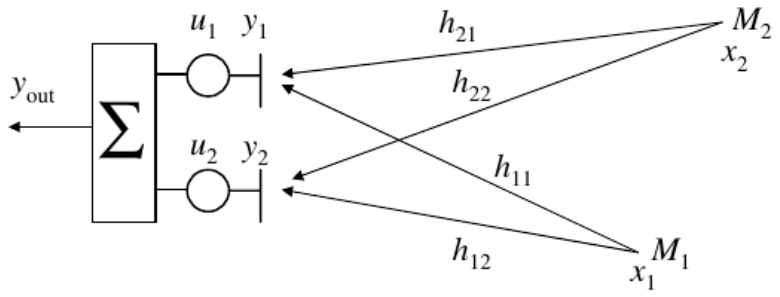


Figure 2.2: Two element array antenna [12]

2.2 How does the MIMO system work?

Consider one of the simplest forms of antenna array at a receiving base station, while there are two mobile devices at different locations, each transmitting a signal at the same frequency to the receiver as shown in Figure 2.2. The mobile users M_1 and M_2 are simultaneously transmitting signals x_1 and x_2 respectively. The superposition of the two signals at each of the two receiving antennas, y_1 and y_2 is shown in equation 2.5 [12] (for simplicity receiver noise has been omitted).

$$y_1 = h_{11}x_1 + h_{21}x_2 y_2 = h_{12}x_1 + h_{22}x_2 \tag{2.5}$$

In the above equation, h_{11} is the complex channel coefficient between mobile M_1 and the receiving antenna 1. Likewise h_{21} is the complex channel coefficient between mobile M_2 and the receiving antenna 1. This works the same way with channel coefficients h_{12} and h_{22}. u_1 and u_2 are the phase weights at the transmitter antennas. The final output at the transmitter can be formulated as shown in 2.6 [12].

$$\begin{aligned} y_{out} &= u_1 y_1 + u_2 y_2 \\ &= (u_1 h_{11} + u_2 h_{12})x_1 + (u_1 h_{21} + u_2 h_{22})x_2 \end{aligned} \tag{2.6}$$

The weights can be set appropriately so that the signal contains only terms with x_1 and not x_2 , which means only the signal from mobile M_1 is received, while the signal from M_2 is suppressed and vice versa. A further step is the application of a second set of weights. By the application of two sets of weights, the receiver has essentially formed two beams, such that y_{out1} only receives from M_1 and y_{out2} only receives from M_2. This technique is referred to as Space Division Multiple Access (SDMA) and an example system is shown in Figure 2.3. Therefore, MIMO can be seen as an evolution of MISO and SIMO that includes the ability to handle multiple users as well as providing a higher data rate communication link.

The selection of suitable weights is crucial to the design of MIMO communication system. Additionally, certain conditions need to be met for the MIMO system to work. Two such conditions are discussed here.MIMO Communication is not possible if both the transmit and receive antennas are close together. No possible values of the weights can be determined in this scenario. Another condition requires the presence of an object called *scatterer* in the communication path. The scatterer will reflect signals leading to different paths. In this situation, the distance of the scatterer from the direct path of communication determines the

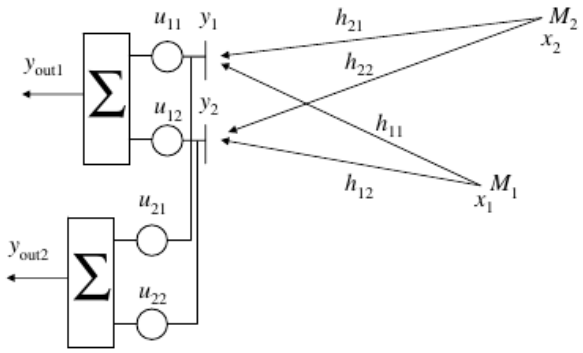


Figure 2.3: Two element array antenna for SDMA [12]

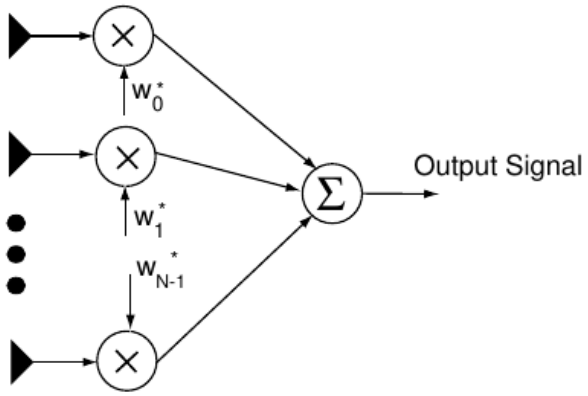


Figure 2.4: Beamforming at the receiver [13]

viability of the system.

2.3 Beamforming

As stated previously, beamforming is the process of transmission or reception of a signal in the desired direction. Figure 2.4 illustrates the receiver beamforming concept [13]. The signal from each element x_n is multiplied with a weight w_n , where the superscript $*$ (in Figure 2.4) represents the complex conjugate. The weighted signals are added together to form the output signal. The output signal r is therefore given by

$$\begin{aligned} r &= \sum_{n=0}^{N-1} w_n^*.x_n \\ &= w^H.x \end{aligned} \tag{2.7}$$

In equation 2.7 [13], w represents the vector of 'N' weights , x represents the vector of 'N' received signals and the superscript H represents the Hermitian of a vector (the conjugate transpose), i.e., $w^H = [w_0, w_1, ...w_{N-1}] = [w^T]^*$.

The array, of N elements, receives message signals from M + 1 users. In addition, the

signal at each element is corrupted by thermal noise, modelled as Additive White Gaussian Noise (AWGN). The received signals are multiplied by the conjugates of the weights. The resultant multiplication terms are added together. The weights shown here are equivalent to the phase weights explained in the previous sections. The value of these weights is adjusted based on the type of combining mechanism chosen at the receiver.

The received signal is shown in equation 2.8 [13]. The goal of beamforming or interference cancellation is to isolate the signal of the desired user, contained in the term α, from the interference and noise. The vectors h_m are the spatial signatures of the m^{th} user.

$$x = \alpha.h_0 + n \tag{2.8}$$

Now that the theory behind beamforming has been understood, here, beamforming operation is observed from the point of view of its implementation in practical scenarios. Beamforming can be performed in the analog as well as the digital domain. In analog domain beamforming [14], the phase weights can be applied either using time delay elements or phase shifters. The available values of weights are limited in the analog domain since all possible values cannot be realized using analog circuits. In digital beamforming [14], the processing for beamforming is done using a digital signal processor which provides greater flexibility with more degrees of freedom to implement efficient beamforming algorithms. The pure digital beamforming method requires a separate RF chain for each antenna element, which results in a complex architecture and high power consumption. A comparison between analog and digital beamforming is presented in Table 2.1 respectively.

Beamforming	Degree of freedom	Complexity	Power consumption	Cost	Inter-user interference
Digital	High	High	High	High	Low
Analog	Low	Low	Low	Low	High

Table 2.1: Analog vs Digital beamforming [14]

2.4 Hybrid Beamforming

The analog and digital beamforming systems by themselves are not sufficient to form an efficient receiver design for MIMO systems. Hence, hybrid beamforming system which deploy both analog and digital beamformers have been proposed as a solution for the design of an efficient beamformer. A brief summary of the different architectures for hybrid receivers proposed in literature has been presented below.

A hybrid architecture reduces the number of paths required for digital baseband processing. In the presence of strong interference, the ADCs spend energy to digitize not only the desired signal but also interference. If interference can be pre-cancelled before the ADCs, energy can be saved. In [1], a combination of the antennas with analog preprocessing has been applied, and a quantized matching pursuit algorithm is proposed to select optimum analog and digital beamforming weights. Analog preprocessing is used to cancel most of the interference in RF which aims to reduce the number of ADCs (which implies less power consumption).

In [10] a hybrid beamforming system is presented with the goal to reduce quantization error in the analog preprocessing network. This quantization error occurs in the analog

phase shifter and amplifier of the analog preprocessing network. The quantized matching pursuit algorithm is used to find the optimum analog and digital beamforming values as presented in [1].

In [11] a design framework for hybrid beamforming for multi-cell multiuser massive MIMO systems over mmWave channels has been presented. This paper presents a new approach for designing analog beamforming using Kronecker decomposition [1]. Kronecker decomposition is aimed at removing the constraints put on analog beamforming due to the use of phase-arrays for obtaining the coefficient values. In addition to these systems, there are many more hybrid system [15], [16] which have been proposed over recent years to make MIMO communication more efficient using hybrid beamforming.

The hybrid beamforming receiver (as shown in Figure 1.3) presented in this research assignment is mainly based on the system in proposed [1]. With this discussion on the different hybrid receiver architectures, in the following sections the concepts which explain the exact mechanism of working of hybrid beamforming system used in this research have been explained.

Minimum Mean Squared Error (MMSE)

Beamforming can be performed under different optimal conditions. In this assignment the focus is on MMSE algorithm for optimal beamforming. The MMSE [13] algorithm minimizes the error with respect to a reference signal d(t). In this model, the desired user is assumed to transmit this reference signal, i.e., $\alpha = \beta d(t)$, where β is the signal amplitude and d(t) is known to the receiving base station. The MMSE tries to find the weights *w* of the beamformer that minimize the average power in the error signal i.e. the difference between the reference signal and the output signal obtained using equation 2.7. The equation which tries to find the optimum value for weights w using MMSE has been shown in equation 2.9 [13].

$$w_{MMSE} = \arg\min_{w} E|e(t)|^2 \tag{2.9}$$

$$\begin{aligned} E|e(t)|^2 &= E[|r(t) - d(t)|^2] \\ &= E[|w^H x(t) - d(t)|^2] \\ &= E[w^H xxHw - w^H xd^* - x^H wd + dd^*] \\ &= w^H Rw - w^H r_{xd} - r_{xd}^H w + dd^* \end{aligned} \tag{2.10}$$

$$r_{xd} = E[x.d^*] \tag{2.11}$$

The calculation of the mean square error value has been shown in equations 2.10 [13] and 2.11 [13]. Finding the minimal value of w as shown in equation 2.9 requires differentiation w.r.t. w^H. This results in the value of w as shown in equation 2.12 [13]. This solution is known as the *Wiener Filter*. Here, w_{MMSE} denotes the optimal beamformer value.

$$w_{MMSE} = R^{-1}.r_{xd} \tag{2.12}$$

$$\text{R is the covariance matrix given by the equation, } R = E[x.x^H]$$

[1] Kronecker decomposition is an operation on two matrices of arbitrary size which results in a block matrix.

The MMSE technique minimizes the error with respect to a reference signal. Therefore, it does not require knowledge of the spatial signature (channel information), but does require knowledge of the transmitted signal. This is an example of a training based scheme: the reference signal acts to train the beamformer weights.

Now, the application of the MMSE algorithm in the hybrid receiver in this research assignment is explained with the help of the hybrid receiver design proposed in this research assignment. Figure 2.5 shows hybrid receiver with 2 receiving antennas. The desired user transmits the signal S(t) and there is also an interference signal $I_1(t)$. The received signals at the two antennas as denoted by x_1 and x_2, together they are denoted by the received signal vector $x = [x_1\ x_2]$.

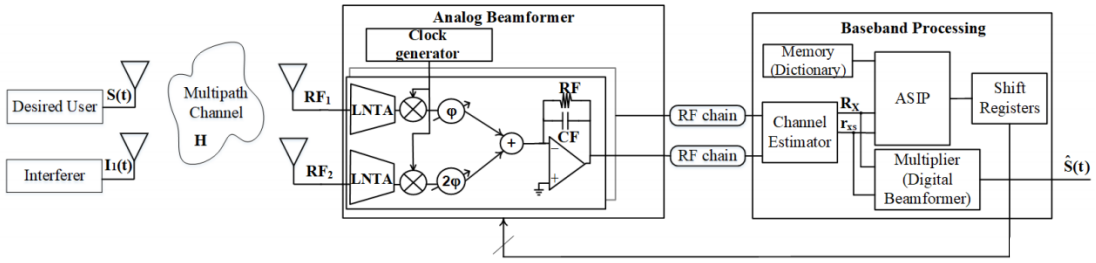


Figure 2.5: Hybrid beamforming structure at the receiver

The Wiener beamformer when applied for the receiver in Figure 2.5, results in the equation 2.13 [1] (as followed from 2.12).

$$\theta_{opt} = R_x^{-1}.r_{xs} \tag{2.13}$$

where $R_x^{-1} = E[xx^H]$ is the co-variance matrix of the received signal x and r_{xs} is the cross-correlation vector between the received signal x and the reference signal s(t). The reference signal s(t) is assumed to be known at the receiver base station (equivalent to the d(t) signal presented previously). Equation 2.13 will serve as reference equation for overall optimal hybrid beamforming.

The optimal hybrid beamformer can be also expressed as shown in equation 2.14 [1]; where W is the analog beamforming vector and λ is the digital beamforming vector. The analog beamforming vector W is used to compose a dictionary matrix D. The size of D is given by N x 2^{N*R_w}; where N is the number of receiver antennas and R_w is the resolution of the analog beamformer (quantization of the phase shifters in the analog beamformer). Each column of the Dictionary matrix represents one possible combination of W for the N receiver antennas. The goal is to find the optimum value of W which reduces the average power in the error signal.

$$\theta_{opt} = W.\lambda \tag{2.14}$$

Using equations 2.13 and 2.14, it can be said that $\theta_{opt} = R_x^{-1}.r_{xs} = w.\lambda$. This result implies that $R_x^{-1}.r_{xs}$ is in the column span of w. This gives the necessary and sufficient condition on W that $\underline{r}_{xs}$ is in the column span of $\underline{W}$ [1].

The mean square error for the receiver system can now be expressed as shown in equa-

tion 2.15 [10], where $s^1[k]$ is the discretized version of the signal transmitted by the desired user (S(t)) and x[k] represents the discretized version of the receiver signal(x).

$$\begin{aligned} MSE &= E[|s^1[k] - \theta^H x[k]|^2] \\ &= E[|s^1[k] - (W\lambda)^H x[k]|^2] \\ &= E[|s^1[k] - \lambda^H W^H x[k]|^2] \end{aligned} \tag{2.15}$$

For any value W and the corresponding optimal $\lambda = (W^H R_x W)^{-1} W^H r_{xs}$, the MSE equation in 2.15 can be re-written as shown in equation 2.16 [1].

$$\begin{aligned} MSE &= 1 - r_{xs}^H W (W^H R_x W)^{-1} W^H r_{xs} \\ &= 1 - \underline{\mathsf{r}}_{xs}^H P_{\underline{\mathsf{W}}} \underline{\mathsf{r}}_{xs} \end{aligned} \tag{2.16}$$

In equation 2.16, $P_{\underline{\mathsf{W}}}$ is the orthogonal projection matrix given by $P_{\underline{\mathsf{W}}} = \underline{\mathsf{W}}(\underline{\mathsf{W}}^H \underline{\mathsf{W}})^{-1} \underline{\mathsf{W}}^H$ and $\underline{\mathsf{W}} = R_x^{\frac{1}{2}} W$ is the whitened analog beamforming vector. The solution $\underline{\mathsf{W}}_0$ which satisfies the MMSE equation is given by equation 2.17 [1].

$$\underline{\mathsf{W}}_0 = \arg\max_{\underline{\mathsf{W}}} \underline{\mathsf{r}}_{xs}^H P_{\underline{\mathsf{W}}} \underline{\mathsf{r}}_{xs} \tag{2.17}$$

The MSE presented in equation 2.16 will have minimum value when the term $\underline{\mathsf{r}}_{xs}^H P_{\underline{\mathsf{W}}} \underline{\mathsf{r}}_{xs}$ has the maximum value for a given value of $\underline{\mathsf{W}}$. This implication has been presented in equation 2.18 [1].

$$\begin{aligned} \underline{\mathsf{W}} &= \arg\max_{\underline{\mathsf{W}} \in \underline{\mathsf{D}}} \underline{\mathsf{r}}_{xs}^H P_{\underline{\mathsf{W}}} \underline{\mathsf{r}}_{xs} \\ &= \arg\max_{\underline{\mathsf{W}} \in \underline{\mathsf{D}}} ||P_{\underline{\mathsf{W}}} \underline{\mathsf{r}}_{xs}||^2 \end{aligned} \tag{2.18}$$

These results are equivalent to equation 2.19 [1].

$$\begin{aligned} \underline{\mathsf{W}} &= \arg\min_{\underline{\mathsf{W}} \in \underline{\mathsf{D}}} ||(I - P_{\underline{\mathsf{W}}}) \underline{\mathsf{r}}_{xs}||^2 \\ &= \arg\min_{\underline{\mathsf{W}} \in \underline{\mathsf{D}}} ||\underline{\mathsf{r}}_{xs} - \underline{\mathsf{W}}(\underline{\mathsf{W}}^H \underline{\mathsf{W}}) \underline{\mathsf{W}}^H||^2 \\ &= ||\underline{\mathsf{r}}_{xs} - \underline{\mathsf{W}}\lambda||^2 \end{aligned} \tag{2.19}$$

To reduce the complexity, the columns of $\underline{\mathsf{W}}$ are selected one-by-one. The quantized matching pursuit algorithm [1] is used tp recursively choose the dictionary elements to obtain the best approximation of the input vector ($\underline{\mathsf{r}}_{xs}$ in this case). Following this algorithm, the problem reduces to finding the solution for the equation 2.20 [1].

$$\underline{\mathsf{w}}_{opt} = \arg\max_{\underline{\mathsf{w}}_i \in \underline{\mathsf{D}}} \frac{|\underline{\mathsf{w}}_i^H \underline{\mathsf{r}}_{xs}|}{||\underline{\mathsf{w}}_i||} \tag{2.20}$$

In equation 2.20, $\underline{\mathsf{w}}_{opt}$ refers to the optimum whitened analog beamformer value, $\underline{\mathsf{w}}_i$ refers to the column i of the whitened dictionary $\underline{D}$ of whitened analog beamformer $\underline{\mathsf{W}}$, $||\underline{\mathsf{w}}_i||$ refers to the norm of the whitened column vector $\underline{\mathsf{w}}_i$. The process of calculating the value of $\underline{w}_{opt}$ which maximises the value of right hand side in equation 2.20 has been termed as *Search Algorithm* in the context of this assignment.

The Search algorithm can be summarized as follows: Given an input covariance matrix C_{rr} , a cross-correlation vector C_{rx_i} , and a dictionary of quantized analog beamforming vectors D.

- Transform the analog beamforming vector W to the whitened matrix $\underline{W}$.
- Compute the value of $\underline{w}_i$ which gives the maximum value of right hand side in equation 2.20.

2.5 Related Work

Chapter 1 provides the information about the problem statement that is tackled in this thesis assignment and the previous sections provides the necessary background information to understand this problem statement. In this section, some of the ASIP implementations in MIMO communication systems are discussed.

In [17] an ASIP is used for implementing a flexible Minimum Mean Square Error-Interference Canceller (MMSEIC) linear equalizer for MIMO turbo-equalization applications. The proposed 16-bit ASIP has an Single Instruction Multiple Data (SIMD) architecture with a specialized instruction set and 7 stage pipeline. The special instruction set architecture supports complex numbered matrix operations. The ASIP is mainly composed of Matrix Register Banks (MRB), Complex Arithmetic Unit (CAU) and Control Unit (CU) along with a memory interface. The MRBs are used to store complex number in two 16-bit registers. The CAU has the computational resources to perform 4 concurrent complex additions, subtractions, complex conjugation and multiplications. The ASIP is synthesized using 90 nm technology for a frequency 546 MHz.

In [18], 32-bit ASIPs are used for realizing channel equalization algorithm for MIMO system in Wide Code Division Multiple Access (WCDMA) downlink. The ASIPs are designed on the principle of Transport Triggered Architecture (TTA) [2]. Similar to ASIP presented in [17], here also there are Special Functional Unit (SFU)s which deal with the handling of complex number processing. The SFUs are evidenced to provide significant reduction in bus traffic and connection between buses in the proposed ASIPs. Another ASIP implementation is proposed in [19] and [20] where it is used realize a low complexity iterative precoder for multi user MIMO.

[21] presents an ASIP design used for implementing singular value decomposition in MIMO systems. The processor has special instructions for complex value multiplication, vector norm computation and concurrent matrix processing operations. Singular value decomposition is used for beamforming in MIMO system in [21] hence the architectural choices in this paper can serve as a reference for the design methodology as expected in this thesis assignment. However, the instruction encoding is quite wide given that a 102-bit wide instruction bus is used. In addition to complex arithmetic handling as seen in the previous designs, this design also provides special hardware to perform floating point arithmetic.

Reconfigurable ASIPs have also been proposed in MIMO systems as seen in [22]. The

[2] A TTA is a kind of processor design in which programs directly control the internal transport buses of a processor. Computation happens as a side effect of data transports: writing data into a triggering port of a functional unit triggers the functional unit to start a computation

reconfigurable ASIP (termed as rASIP) is composed of a Coarse Grain Reconfigurable Architecture (CGRA) along with a processor. The reconfigurability of the processor is exploited by implementing 4 MIMO detection algorithms based on the requirement of the system. The detection algorithm are : zero forcing, linear MMSE, MMSE and Marko Chain Monte Carlo (MCMC) based detection algorithm. Along the same lines [23] proposes a system where the processor is configured to perform multiple tasks. These tasks are disjoint processes viz. beamforming and channel feedback. ASIP implementation saves resources since it can be used to implement multiple tasks on the same platform as long as these tasks are multiplexed in time. The instruction set is designed such that many other tasks such as encryption-decryption, checksum generation etc. can also be performed without any additional hardware costs. The baseband processor in this thesis assignment can be designed along similar lines i.e. with an instruction set which can support multiple operations which are generally a part of MIMO communication systems.

The systems presented here are by no means exhaustive and many more implementations might be present. The operations implemented in the systems presented earlier are similar to the operations expected to be performed in this assignment. Hence, these designs have been considered. The investigation to determine ASIP implementation in MIMO systems has revealed that an ASIP design for computing optimal coefficient values in a hybrid beamforming system (as shown in Figure 1.3) has not yet been proposed.

2.6 Role of the ASIP Baseband processor

The expected design of the baseband processing block as a part of the hybrid beamforming system has been shown in Figure 2.6. The figure essentially consists of 5 blocks viz. the Dictionary, the Estimator, the ASIP, the Multiplier(Digital Beamformer) and the Shift registers block. The function of each block is explained as follows:

- Dictionary: This block comprises of all possible values of the quantized analog beamforming coefficients. For a system with N antennas and a resolution of R_w (resolution of the phase shifters in the analog beamformer), the dictionary consists of $N * 2^{R_w}$ possibilities. As the number of antennas and their corresponding resolution will increase, the size of the dictionary will increase exponentially. Considering this, at the beginning of this assignment it was decided to store the dictionary in an external memory unit which is interfaced with the ASIP.

- Estimator : The calculation of the optimum analog beamformer requires the calculation of the cross-correlation matrix C_{rx} and the corresponding whitened matrix value $\underline{C}_{rx}$. This operation has been assigned to the estimator block. This block is expected to be an Application Specific Integrated Circuit (ASIC) dedicated for this purpose since calculation of cross-correlation values and whitened matrix values for complex values is a computationally demanding task and it is also required to be fast (in terms of calculation speed). In addition to this, this block will also calculate the co-variance value of the received signal. This block takes input from the RF chain to perform the mentioned operations. It will also be provided with the reference signal value which is assumed to be known at the receiver.

- ASIP: This block is expected to perform the task of determining the optimum analog

beamformer coefficient values following the *search algorithm* [3] explained previously. It will take input values from the Dictionary and Estimator blocks and the output of this block is given back to the analog beamformer.

- Multiplier (Digital Beamformer): This block is expected to perform the digital beamforming on the signals obtained from the RF chain in the baseband domain. The ASIP is not involved in the digital beamforming operation.
- Shift registers: The ASIP will produce vector values at the output. These values need to be converted to bit patterns which will turn on/off the switches in the analog beamformer to achieve different coefficient values. The Shift registers deliver this bit pattern to the analog beamformer. The conversion operation can either be performed in the ASIP or in the shift register block.

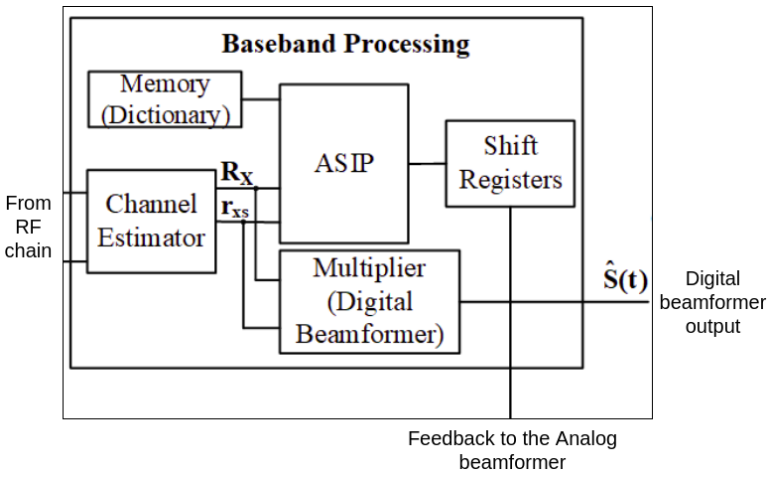


Figure 2.6: Typical expected design of the baseband processing block of hybrid receiver

[3] Matlab Implementation snippet available in Appendix A

Chapter 3

Choice of Instruction Set Architecture

An Instruction Set Architecture (ISA) represents a abstract computer model. Realization of ISA is termed as implementation. Multiple implementations of a computer model are possible based on variation in performance, size and cost etc. The ISA acts as the mediating layer between hardware and software. There are different variants of ISAs : licensed, custom or open source. In this research assignment the focus is on the use of open source instruction set architectures.

The cornerstone of ASIP design is the customization of the instruction set with respect to a given application. In that sense, a completely new instruction set can be developed with the *search algorithm* at its focus. On the other hand, if the foundation of the ASIP is built on existing open source architectures, it ensures certain support on the software end, insights from the community of users and developers, etc. There are several processor (or cores) and system on chip platforms with hardware and software support based on the open source instruction set architectures readily available on the open source platforms. These are a few reasons because of which the ASIP architecture is chosen to be developed on an existing open source instruction set architecture. A few of these open source ISAs are discussed in this chapter.

The choice of the right open source architecture depends on several factors such as the support provided by the developers' and users' community, available software tools for proper experimentation (for example instruction set simulator), scope and ease of instruction set extension. etc. Based on these criteria, selection of the reference open source architecture can be performed. The motivation behind this selection is also discussed in further sections of this chapter.

In the next section, the key features of three ISAs are discussed. They are : OpenRISC, UltraSPARC and RISC-V.

3.1 OpenRISC

The OpenRISC (or OpenRISC 1000) [24] architecture is an open source RISC based architecture. It targets the medium and high performance networking and embedded computer environments. Some of its important features are :

- A linear, 32-bit or 64-bit logical address space with implementation-specific physical address space.
- Simple and uniform-length instruction formats featuring different instruction set extensions:
  - OpenRISC Basic Instruction Set (ORBIS32/64) with 32-bit wide instructions aligned on 32-bit boundaries in memory and operating on 32- and 64-bit data
  - OpenRISC Vector/DSP extension (ORVDX64) with 32-bit wide instructions aligned on 32-bit boundaries in memory and operating on 8-, 16-, 32- and 64- bit data
  - OpenRISC Floating-Point extension (ORFPX32/64) with 32-bit wide instructions aligned on 32-bit boundaries in memory and operating on 32- and 64-bit data
- Optional branch delay slot for keeping the pipeline as full as possible
- A flexible architecture definition that allows certain functions to be performed either in hardware or with the assistance of implementation-specific software.
- Fast context switch support in register set, caches, and memory management units.
- Memory is byte-addressed with half word accesses aligned on 2-byte boundaries, single word accesses aligned on 4-byte boundaries, and double word accesses aligned on 8-byte boundaries.
- The OpenRISC architecture specifies a weakly ordered memory model for uniprocessor and shared memory multiprocessor systems. This model has the advantage of a higher-performance memory system but places the responsibility for strict access ordering on the programmer (through special instructions which specify no reordering).

3.2 UltraSPARC

UltraSPARC architecture [25] is another RISC based open source ISA wherein SPARC stands for Scalable Processor Architecture. Some of the features of SPARC architecture are:

- The SPARC Architecture supports 32-bit and 64-bit integer and 32 bit, 64 bit, and 128 bit floating-point as its principal data types.
- The 32-bit and 64-bit floating-point types conform to IEEE Std 754-1985. The 128 bit floating-point type conforms to IEEE Std 1596.5-1992.
- It supports a linear 64-bit address space with 64-bit addressing. The instructions are 32-bit wide instructions and are aligned on 32-bit boundaries in memory. Only load and store instructions access memory and perform I/O.

- The architecture defines general-purpose integer, floating-point, and special state/status register instructions, all encoded in 32 bit wide instruction formats.
- The load/store instructions address a linear, 264-byte virtual address space.
- The instruction set comes with many extensions, including the Virtual Instruction Set (VIS) for “vector” i.e. SIMD operations.

  An important highlight of this architecture is the support for Chip Multi-Threaded (CMT) technology. CMT is an application of parallel processing. It can be seen as being similar to software multi-threading where multiple processor activities can be done in a single process. The only difference is that CMT is hardware-based so that the processor handles the different threads instead of the software. The key advantage of this compared to older processor technologies is improved throughput. The SPARC architecture supports CMT design by providing a control architecture.

3.3 RISC-V

The name RISC-V was chosen to represent the fifth major RISC ISA design from UC Berkeley (RISC-I , RISC-II, SOAR, and SPUR were the first four). RISC-V ISA [26] allows efficient implementation of different particular microarchitecture styles (e.g., microcoded, in-order, decoupled, out-of-order) and different implementation technologies (e.g., full-custom, ASIC, FPGA) combinations.

- The ISA is separated into a small base integer ISA, usable by itself as a base for customized accelerators or for educational purposes, and optional standard extensions, to support general-purpose software development.
  - Each base integer instruction set is characterized by the width of the integer registers and the corresponding size of the user address space. There are 4 base instruction set variants: RV32I, RV32E, RV64I and RV128I.
  - The RV32E is a reduced version of the RV32I ISA especially aimed at embedded system applications. There are a lot more standard extensions for e.g. extension which supports compressed instruction format RV32C. The naming convention for the base instruction set, standard and custom extensions can be found in [26].
- Support for the revised 2008 IEEE-754 floating-point standard.
- RISC-V supports extensive user-level ISA extensions and specialized variants.
- Both 32-bit and 64-bit address space variants for applications, operating system kernels, and hardware implementations.
- It supports highly-parallel multi core implementations, including heterogeneous multiprocessors.
- Optional variable-length instructions to both expand available instruction encoding space and to support an optional dense instruction encoding for improved performance, static code size, and energy efficiency.

3.4 Comparison between the open source ISAs

The general criteria for choosing the right instruction architecture was briefly described at the beginning of this chapter. Here, the three architectures are compared on the basis of the following factors : Design flexibility, hardware and software development, standard extension availability, instruction encoding possibility and currently available hardware designs based on these ISAs.

Table 3.1 and 3.2 summarize the comparison between the three instruction sets.

ISA	Design Flexibility	Hardware development	Software development
RISC-V	High	Low	Medium
OpenRISC	Medium	High	High
UltraSPARC	Medium	Low	Low

Table 3.1: Comparison between open source instruction set architectures (part 1)

ISA	Standard Extensions availability	Instruction Encoding	Available processor designs
RISC-V	High	Variable (16-bit multiples)	PULP, Boom, Rocket
OpenRISC	Medium	32/64-bit only	minSoC, OpTiMSoC, MiSoC
UltraSPARC	Medium	32/64/128-bit only	OpenSPARC T1 and OpenSPARC T2

Table 3.2: Comparison between open source instruction set architectures (part 2)

The elaborate comparison between the three ISAs is presented here.

1. Design flexibility: This factor refers to the different implementations styles that are supported by the instruction set. The RISC-V architecture supports different types of architecture styles as mentioned previously. The OpenRISC architecture supports specifically the weakly ordered memory model. The UltraSPARC architecture supports three types of memory models and the architecture is well designed to support branch prediction and elimination functions natively.

2. Hardware development: OpenRISC is a mature open source instruction architecture, hence it has good hardware development support available provided by the OpenRISC community. RISC-V lacks a bit in this regard because of its relatively new establishment. Although, there are certain companies for example SiFive or Green Wave Technologies which are bringing RISC-V hardware to the market. The UltraSPARC architecture finds its application only in the development of the two open source processor designs : OpenSPARC T1 and OpenSPARC T2.

3. Software Development: The community for OpenRISC and RISC-V for software support in terms of linux compiler support, instruction set simulators, linkers, etc. is quite high. In that sense, the support for UltraSPARC can be considered to be a bit limited since a lot of alternatives for the same software tool are not available.

4. Number of Standard Extensions available: All these instruction set architectures have different types of standard extensions already available. RISC-V is the instruction set architecture with a lot of extensions already standardized, for example, an atomic extension. Such type of standardization for different extensions directly implies availability of software support. At the same time, the usage/requirement of a particular extension is also an application dependent factor.

5. Instruction Encoding: The RISC-V ISA provides the most flexible instruction encoding option by supporting any instruction encoding format which is multiple of 16. This means based on the design requirements it can support a varying range of instruction encoding formats from compressed instruction formats to VLIW formats.

6. Available processor designs: This last factor lists the available implementations of the three instruction sets.

From this discussion it can be gathered that RISC-V provides high design flexibility along with good software development support. This is due to its modular design, which features a common base of roughly 40 integer instructions (I) that all cores must implement, with ample opcode space left over to support optional extensions, of which the most canonical have already been standardized.

The RISC-V instruction set architecture allows for easy integration of user defined custom instructions in the existing base or standard extensions sets. This allows to develop design of the core such that specific speed or energy efficiency can be achieved in certain operations. Along with support for user level ISA extension, a key feature of this instruction set is the support for compressed instructions. The existing standard custom instruction set supports the compressed format for some basic instructions like add, sub etc. The advantage of using such compressed instructions is that it reduces the length of the program code; saving memory and power at the same time. This feature is not found in OpenRISC and UltraSPARC architectures. However, whether a compressed instruction format is needed was kept an open question.

Additionally, the user base for RISC-V is increasing exponentially. There is ample support provided by the members of the RISC-V foundation and the community of programmers and developers. Tech giants such as Qualcomm, Western digital and many others are investing in RISC-V as an open source alternative to licensed ISAs such as ARM.

Hence, RISC-V has been chosen as the reference open source architecture upon which the instruction set of the ASIP will be developed.

Chapter 4

Processor Modeling tool and flow of design

In this Chapter, the tool used to design the ASIP is discussed along with the design flow followed for processor modeling in this tool.

4.1 Processor Modeling tool

The Synopsys ASIP designer has been chosen to design the ASIP for the hybrid receiver. The overall tool flow has been shown in Figure 4.1.

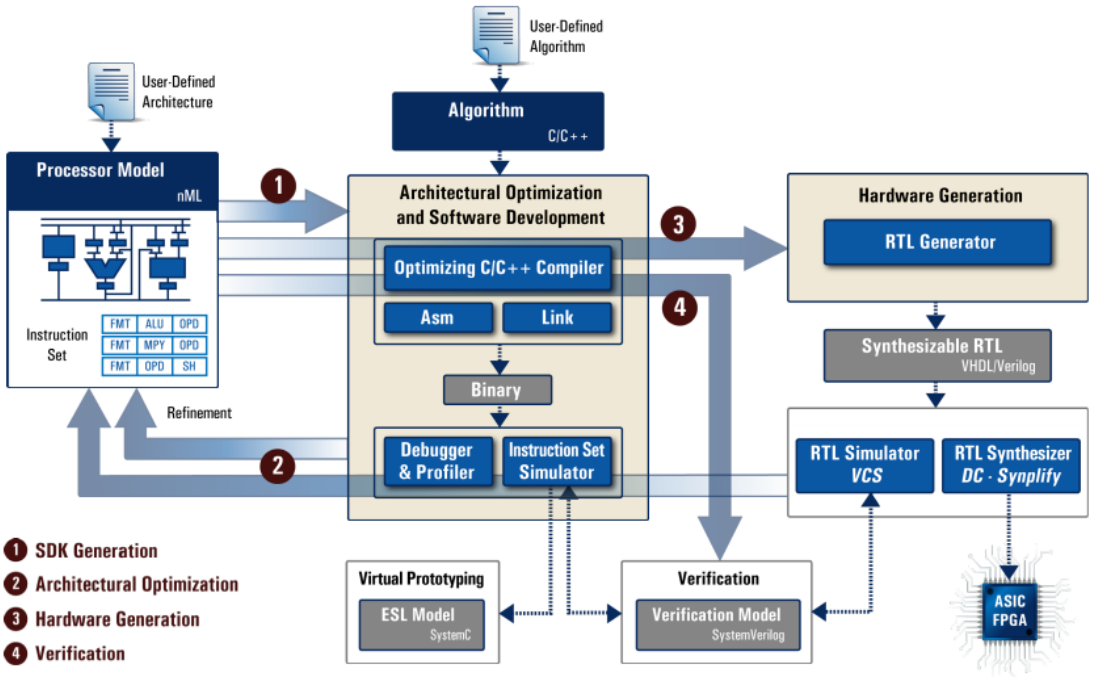


Figure 4.1: Synopsys ASIP designer tool flow [9]

ASIP Designer supports the following features:

- Modeling of ASIP instruction set architectures in the nML processor description language. nML is a high-level definition language for describing a processor architecture and instruction set.
- Once, the processor architecture has been defined using nML, the tool generates the entire Software Development Kit (SDK). The SDK comprises of the Compiler, assembler, linker, Instruction Set Simulator (ISS), etc. Using the ISS the simulation for a desired algorithm (in C/C++) can be performed. Based on the results obtained refinement can be performed in the processor model. This process is known as the *compiler-in-the-loop* architectural exploration.
- The SDK comprises of the following elements:
  - An optimizing compiler which provides efficient code generation and, quick and automatic retargetability to new ASIP architectures. The compiler supports C (optionally extended with user-defined data types and operators using C++ classes and function overloading), C++, and OpenCL C (OpenCL kernel language). The compiler can cope well with architectural peculiarities of DSP cores. It supports instruction-level and data-level parallelism, deeply pipelined instructions, specialized arithmetic functions, custom data-types, specialized address generation units, heterogeneous register structures, and various degrees of instruction encoding (ranging from VLIW to highly encoded instruction sets).
  - A linker that builds an executable file from separately compiled Elf/Dwarf object files for different C functions.
  - An assembler and disassembler that translates machine code from assembly into binary format and back.
  - A fast ISS, offering both cycle-accurate and instruction-accurate abstraction levels generated from the same nML model, and easy integration into cycle-accurate and transaction-level virtual prototypes.
  - A flexible (multicore) debugger, which can be used in connection to both ISSs and on-chip debug hardware (via JTAG).
  - Multi-faceted profiling capabilities to analyze the instruction-set architecture for hot-spots and to drive the architectural optimization process.
- The tool also automatically generates a power and area efficient hardware implementation of each ASIP, in *synthesizable Verilog or VHDL*. The RTL design can be verified via simulation and again refinement can be performed in the processor model. This process is known as the *synthesis-in-the-loop* architectural exploration

  A JTAG interface and a debug controller can optionally be generated, to support on-chip debugging.
- The automatic generation of ASIP-specific test programs in C and assembly code allows extensive verification of the ASIP.
- Another important feature of the ASIP Designer tools is that it comes with a wide range of example ASIP designs, with highly differentiating architectures provided in nML source code.

The compiler-in-the-loop and synthesis-in-the-loop architectural exploration provided by the ASIP designer make it the ideal choice for designing the target ASIP in this assignment. Along with these architectural exploration processes the example processors that are provided are also an added advantage. These processor models can be used as references while making different design decisions during the ASIP development. Hence, the Synopsys ASIP designer is chosen as the tool suite for the development of the ASIP in this research assignment. The next section explains different steps involved in the design of the processor model.

4.2 Processor model design flow

The flow chart shown in Figure 4.2 shows the different design steps that must be taken by the user to model the processor in Synopsys ASIP design tool.

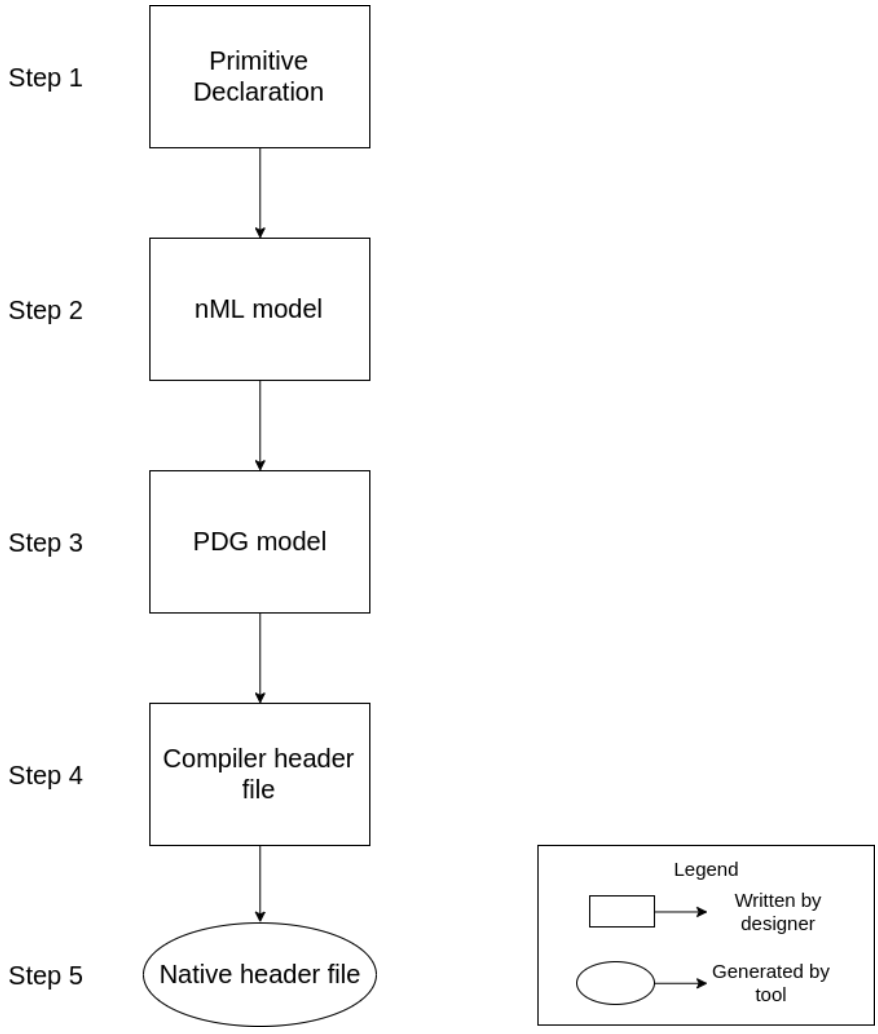


Figure 4.2: Design Steps for processor modelling in Synopsys ASIP designer

The steps are explained as follows:

- Primitive Declaration: The first step is the declaration of user-defined processor specific data types and functions. These data types are defined as C++ classes and functions respectively. They are defined in the <processor>.h file in the primitive namespace.

- nML model: The data path of the processor model is defined in the nML model. This model can be broken down into 2 components: structural skeleton and instruction set description. The structural skeleton comprises of storage elements, functional units, registers, wires, etc. The instruction set description consists of grammar rules which define the behavior of the data path. The nML model uses the primitive types and functions for defining the data path.

- PDG model: The PDG model is written in Primitive Definition and Generation (PDG) language which is based on "C". It has operators from "C" and some from "Verilog". All the primitive functions declared in the <processor>.h file must be defined here. The controller of the processor model is also defined in the PDG model. The I/O interfacing is also done using PDG.

- Compiler Header File: The final step in processor design is writing the compiler header file. The mapping of the C built-in types and operator to primitive processor types and functions is performed in this file. Along with these, additional processor optimization directives and specification of the subroutine call convention is also done in this step.

- Native Header File : This file is automatically generated by the ASIP designer tool from the compiler header file. The tool maps custom data-types and functions from the application code to data-types and functions that the host machine can understand. This file plays a quite an important role in the verification on the application level. The C code is compiled for the host and target machines and then verification can be performed by comparison.

Once all the design steps have been performed by the user the entire processor model can be said to be *available*. This resultant model represents the processor model shown on the left side in Figure 4.1. The processor model is used by the Synopsys ASIP designer to generate the SDK. Once the SDK is available, verification and different tests can be run on the processor. Different C/C++ applications can be run on the processor to perform required evaluation of the processor performance, for instance, running the Coremark Benchmark [27]. Using the compiler-in-the-loop technology, refinement based on verification and performance evaluation can be implemented and results seen immediately via the simulator. Once the user is satisfied with the performance characteristics of the processor, the synthesis of the processor can be performed using the Synopsys ASIP Designer. The synthesis-in-the-loop technology can be used to perform finer refinement in the processor model. In this way, the processor modeling can be performed using Synopsys ASIP designer.

4.2.1 In-Depth insight into each step of processor model design flow

To understand the processor model design flow steps more clearly and gain better perspective about the complexity involved in processor design in the Synopsys ASIP designer, here in this subsection, each step from Figure 4.2 is explained with examples.

Step 1: Primitive Declaration

As stated previously, all the primitives are declared in the <processor>.h file called the primitive processor header file (where *processor* refers to actual given name of the ASIP). Inside the primitive processor header file, all the primitive data types and functions are declared. Primitive declaration is performed because the compiler maps C types and operators onto primitive types and functions.
A sample namespace for *tinycore2*[1] processor is shown in Figure 4.3.

```
namespace tinycore2_primitive {

// declaration of primitive types

   class word property(16 bit signed);

// declaration of primitive functions

   word add(word,word);

};
```

Figure 4.3: Primitive namespace for tinycore2 processor [9]

Primitive data-types are specified using a C++ class declaration inside the primitive namespace. The sample primitive types declared for processor *tinycore2* are shown in Figure 4.4. The primitive data-type *word* can then be used to define registers or inputs to functional units of 16-bit signed type. The primitive data-type *pmtype* will be used to define the type of program memory data; *sbyte* will be used to define signed 8-bit values.

```
namespace tinycore2_primitive {
    class word
          property(16 bit signed);
    class pmtype
          property(13 bit unsigned);
    class sbyte
          property( 8 bit signed);
};
```

Figure 4.4: Primitive data type declaration for tinycore2 processor [9]

Similarly, primitive functions are also declared in the primitive namespace. An instruction is mapped to a primitive function (either directly or indirectly). Following this, the structure of the primitive function will be such that: the operands in the instruction being added become the input arguments for the function, and the output of the function will be the resultant operand of the instruction. Additionally, the primitive function might have additional input or output status signals based on the type of instruction that will be mapped to it. For example, an extra input argument will be required when dealing with control signals which might not be directly visible in the instruction being added.In Figure 4.5, for the primitive function *word sub(word,word,stat&)* stat is an additional signal which will be used to indicate the status of the subtraction operation. From this figure, it can also be seen that function overloading

[1]The tinycore2 is one of the example processor models provided alongside the Synopsys ASIP designer tool suite

is allowed and data-type conversions such as *word(sbyte)* are also done in the primitive namespace.

```
namespace mycore_primitive {
  class stat property(3 bit unsigned);

  word sub(word,word);
  word sub(word,word,stat&);

  class word {
    word(sbyte);
  };
};
```

Figure 4.5: Primitive function declaration for tinycore2 processor [9]

Step 2: nML model design

Once, the primitive types and functions have been defined they are used in the nML model of the instruction. The abstraction level of nML is such that it corresponds to that of a typical processor manual. The data-path and instruction set behaviour are defined using nML.

The instruction behavior is captured using grammar rules and the structural components such as registers, memories, connecting units (wires) are also declared here. There are two types of grammar rules : OR and AND. A grammar rule has a syntax similar to that of a normal C function. It has a name along with input parameters and return value. In addition to this a grammar rule in nML also has attributes viz. action, syntax and image. The AND rule has all three attributes, whereas the OR rule only has the image attribute.

An example of OR rule for the *tinycore2* processor is shown in Figure 4.6. In Figure 4.6, *opn* is a keyword to describe a grammar rule; *image* is the keyword of the image attribute of the grammar rule. It captures the binary encoding of the instructions for which the OR rule has been written

```
opn tinycore2 (alu_opn | compare_opn | ... | generate_byte)
{
  image : "000000"::alu_opn
        | "000001"::compare_opn
        ...
        | "100"::generate_byte;
}
opn control_opn (cjump | ujump | bsr | rts | nop);
```

Figure 4.6: Illustration of OR rule for tinycore2 processor [9]

The OR rules are used to list out alternative instructions or alternative part of instructions. Once all the instructions for the processor have been defined, they can be grouped in a large OR group. This will be a superset of all the OR rules and will define the entire instruction set of the processor. For example, as can be seen in Figure 4.6, there is an OR rule *control_opn* which is listing the alternative control instructions. The OR rule *tinycore2* is listing the different alternatives like *alu_opn* and *compare_opn* which are themselves an OR rule combination of different instructions like the *control_opn* OR rule.

The AND rules dictate the composition of the independently controlled alternative (parts) of the instruction. Simply put, AND rules provide the grammar to define the behavior of a

single instruction. This is done by defining 3 attributes of the AND rule: action, syntax and image. The register transfer behavior of the instruction will be written in the action attribute. The primitive functions and data types defined in the primitive namespace are used in the action attribute. The syntax attribute is used to define assembly view of the instruction. The image attribute is used to define the binary encoding, just like in the case of OR rule. Figure 4.7 shows an example of the AND rule for the *tinycore2* processor.

```
enum alu_op {ADD " + ", SUB " - ", AND " & ", OR " | " };
enum eR { r0, r1, r2, r3, r4, r5, lnk, sp };

opn alu_opn (op : alu_op, a : eR, b : eR)
{
  action {
    stage E1:
    aluA = R[a];
    aluB = R[b];
    switch (op) {
      case ADD : aluC = add (aluA, aluB);
      case SUB : aluC = sub (aluA, aluB);
      case AND : aluC = band(aluA, aluB);
      case OR  : aluC = bor (aluA, aluB);
    }
    R[a] = aluC; // read-modify-write
  }
  syntax : a " = " a op b;   // for example:  r1 = r1 - sp
  image : op::a::b;
}
```

Figure 4.7: Illustration of AND rule for tinycore2 processor [9]

Figure 4.7 shows the AND rule *alu_opn* with 3 parameters (*op, a, b*) as input arguments which are used in the action attribute. In the action attribute the order of the statements is not important just like while using a hardware description language. The keyword *stage E1* refers to first execution stage in the pipeline stages of the *tinycore2* processor. In this way, the stage by stage execution of the instruction in the pipeline of the processor can also be defined. *aluA = R[a]* refers to the reading of the register file at address *a*. Additionally, *add, sub, band, bor* are the primitive functions defined for processor *tinycore2*.

In the presence of a pipelined processor hazards are introduced. These hazards can be managed using nML image attribute. Keyword *cycles* in the image attribute is used to indicate the number of clock cycles for which the pipelined must be stalled. An example of such hazard management is shown in Figure 4.8. In this case, when the condition jump instruction is encountered during program execution, the newly fetched instruction will not be executed for 3 cycles.

```
enum stage_names { IF,    // instruction fetch stage
                   ID,    // instruction decode stage
                   E1 };  // first execute stage
opn cjump (offs : c_8s)
{
  action : stage E1 : jump(tc=SREG,pc_off = offs);
  syntax : "id cc goto " offs ;
  image  : "000011"::offs, cycles(3);
}
```

Figure 4.8: Image attribute changes for hazard management for tinycore2 processor [9]

In addition to this, hazard rules are also written separately as a part of the nML model to

ensure proper instruction execution. These hazard rules can either use software stalls,hardware stalls or bypass forwarding to achieve the necessary stalling.

Step 3: PDG model design

Till Step 2, primitives have been declared and they have been used in the nML model. In Step 3, the bit-true behavior of the primitive functions is defined using the PDG language. As stated previously, the PDG language is a combination of *C* and *Verilog*. Figure 4.9 shows the example where the definition of three primitive functions viz. *add, sub, mul* is described using PDG.

```
w32 add(w32 a, w32 b) { return a + b; }

w32 sub(w32 a, w32 b)
{
    uint33_t aa = (uint32_t)a;
    uint33_t bb = (uint32_t)b;
    uint33_t rr = aa - bb;
    return rr[31:0];
}

w32 mul(w32 a, w32 b)   { return a * b; }
```

Figure 4.9: Definition of primitive functions using PDG

In addition to this, the PDG language is also used to define the controller of the processor. Figure 4.10 shows the skeleton structure of the processor controller unit as written in PDG. This unit is defined in the <processor>_pcu.p file. Keyword *pcu_storages* in Figure 4.10 is used to declare local storages which will be used by the controller unit. These storage units are in addition to the units the defined in the nML model. The *user_issue* function is used to move the instruction execution from the fetch to the decode state. The *user_next_pc* function is used to prepare the next address in the program counter (the preparation can be either a simple increment or new address in case of control instruction execution).

```
// PCU specific storages

pcu_storages {
    reg reg_booting<bool>;
}

// Auxiliary functions

...

// PCU functions

void tinycore2::user_issue()
{
}

void tinycore2::user_next_pc()
{
}
```

Figure 4.10: Skeleton structure of the processor controller unit

The nML model can take care of defining instructions which require a single instruction cycle for execution. However, to define the behavior of multi-cycle instructions a PDG

model is defined. In such case, the nML action will only be used to map the primitive on a multi-cycle functional unit. More on multi-cycle functional units and their implementation is explained in Chapter 7.

Step 4: Writing the compiler header file

The definitions for mapping the C built-in types and operators to the processor primitive data types and functions is done in the compiler header file. The <processor>_chess.h is the main compiler header file and it includes a collection of different header files based on the C data types viz. <processor>_int.h, <processor>_float.h, <processor>_double.h etc. For each C data type, the operators and data types are mapped to primitive types and functions.

```
promotion int operator+(int,int) = word add(word,word);
```

Figure 4.11: Mapping of C operator onto primitive function

Figure 4.11 shows an example of mapping the '+' operator onto the *add* primitive function (here *promotion* is a keyword). The overall mapping of different steps from the application code to the nML model are illustrated in the Figure 4.12.

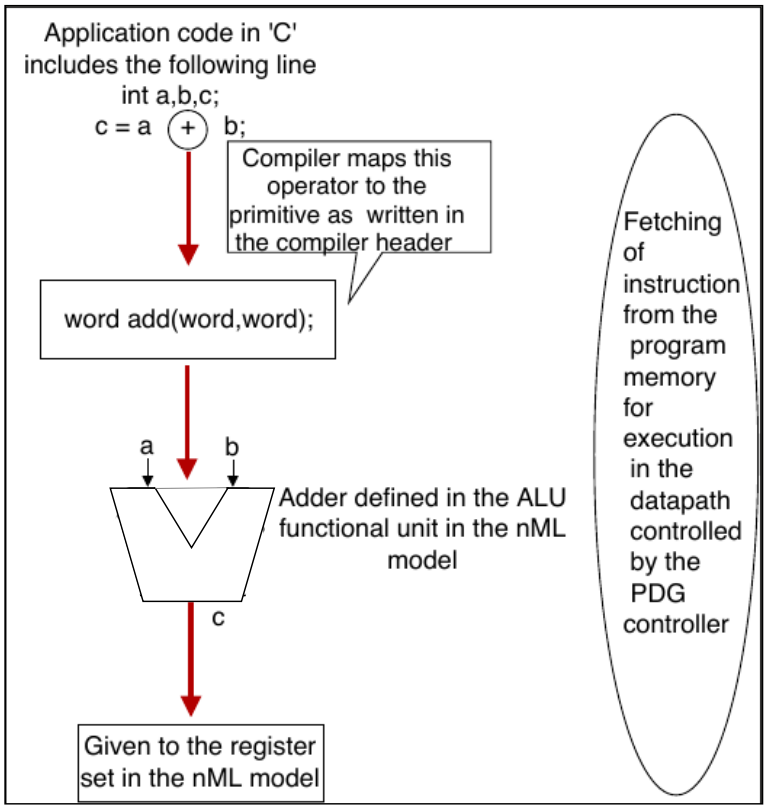


Figure 4.12: Processor modeling in Synopsys ASIP designer

Step 5: Native header file generation

The native header is automatically generated by the Synopsys ASIP design tool once the compiler header file for the processor has been completely defined. Figure 4.13 shows the an example of the conversion of the primitive data-type *w08* to a definition which can be understood by the host machine (for an example processor model).

```
//  Primitive types (definitions)

class w08
{
public:
    typedef VBit<8, true> BitType;
    static const int bits = BitType::bits;
    static const bool isSigned = BitType::isSigned;
    BitType val;
public:
    w08() = default;
    template <int B, bool S> w08(const VBit<B, S>& a) : val(a) {}
    w08(const VBitWrapper& a) : val(a) {}
    template <typename DataType, int E> w08(const VBitVector<DataType, E>& a) : val(a) {}
public:
    // backwards compatibility api
    w08(BitType::ValueType a) : val(a) {}
    BitType::ValueType& value() { return val.value(); }
    const BitType::ValueType& value() const { return val.value(); }
    void value(const BitType::ValueType& v) { val.value(v); }
    BitType::UBaseType to_unsigned() const { return val.to_unsigned(); }
    BitType::SBaseType to_signed() const { return val.to_signed(); }
    friend std::ostream& operator<<(std::ostream& os, const w08& x) { os << x.val; return os; }
    friend std::istream& operator>>(std::istream& is, w08& x) { is >> x.val; return is; }
public:
};
inline const w08::BitType& toBitType(const w08& v) { return v.val; }
```

Figure 4.13: Primitive definition in the native header file

More details on the specific syntax and semantics followed in each design step of processor modeling can be found in the documentation provided alongside the Synopsys ASIP design tool. This chapter is to give the reader an idea about the design steps involved in processor modeling in Synopsys ASIP designer. The elaborations presented here also act as the foundation on the basis of which the Chapter 6 on Design Methodology can be understood better.

Chapter 5

Tzscale RISC-V processor

5.1 Introduction

In the previous two chapters the choice of the reference ISA and design tool suite has been explained. The RISC-V ISA and Synopsys ASIP designer tool suite have been chosen. As stated previously, the ASIP designer provides with a set of example processor designs. Among these examples one example design is the Tzscale processor. The Tzscale processor has been chosen as the main reference design for development of the ASIP. The main reasons for choosing this design as a reference are:

- The processor design is built on the open source RISC-V architecture
- The processor implementation is quite simple and minimalisitic. This provides with a good skeleton design upon which the desired custom instructions can be built.

The Tzscale processor design is similar to the Z-scale [28] processor design proposed by the Berkeley research group in the California. The Berkeley research group has also developed the RISC-V open source instruction set architecture. The Z-scale processor uses the RV32E RISC-V base instruction set. The original Z-scale processor provided by the Berkeley research group has been written in Scala and is no longer supported. In further sections a little background about the Z-scale processor is provided, followed by a summary of the two base instruction sets of RISC-V (RV32I and RV32E), and it concludes with the explanation about the architecture of the Tzscale processor.

5.2 RV32I Base Integer Instruction set

RV32I has been designed to be sufficient to form a compiler target and to support modern operating system environments. The ISA has also been designed to reduce the hardware required in a minimal implementation. RV32I contains 47 unique instructions [1]. RV32I can emulate almost any other ISA extension (except the A extension, which requires additional hardware support for atomicity). Existing standardized extensions include multiply and divide (M), atomics (A), single-precision (F) and double-precision (D) floating point. These

[1] The detailed instruction set description has been presented in Appendix C

common extensions (RV32/64IMAFD) are collected into the (G) extension that provides a general-purpose, scalar instruction set. A compressed (C) extension provides 16-bit instruction formats to reduce static code size. Opcode space is also reserved for non-standard extensions, so designers can easily add new features to their processors that will not conflict with existing software compiled to the standard.

There are 31 general-purpose registers x1-x31, which hold integer values. Register x0 is hardwired to the constant 0. There is no hardwired subroutine return address link register, but the standard software calling convention uses register x1 to hold the return address on a call. For RV32, the x registers are 32 bits wide, and for RV64, they are 64 bits wide. There is an additional user-visible register: the program counter pc. It holds the address of the current instruction.

5.3 RV32E Instruction Set Architecture

In this section, the instruction set of the Tzscale (also Z-scale) processor is discussed. The RV32E is one out of the four base instruction sets for RISC-V viz. RV32I, RV64I, RV128I and RV32E. The RV32E [26] is a reduced version of the RV32I and is designed specially for embedded systems. The main changes are: reduction in the number of integer registers from 32 to 16 and removal of counters that are mandatory to RV32I. The E variant has been developed only for the 32-bit address space width. The main motivation behind development of this ISA has been a general observation discussed by the authors of the RISC-V specification. The observation is as follows: in the small RV32I core designs the upper 16 registers consume around one quarter of the total area of the core excluding memories, thus their removal saves around 25% core area with a corresponding core power reduction. The choice of RV32E offers the ability to provide an area and energy efficient design.

RV32E uses the same instruction set encoding as the RV32I except that use of register specifiers for the higher 16 registers will result in an illegal instruction exception being raised. Another point to note is that, the RV32E is only used with a soft-float calling convention. Systems with hard-floating point must use I-base.

Like any other base instruction set the RV32E instruction set can also be extended. M and C user level standard extensions are possible for RV32E.

5.4 Architecture of the Tzscale Processor

The architectural features of the Tzscale processor are:

- 32 bit wide data path with 3 stage pipeline
- 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.
- ALU, shifter, single cycle multiplier and multi-cycle division/remainder unit

5.4.1 Register Structure

The register file has 2 read ports and one write port. Any register x1-x15 (or x31) can be used to read and write an operand value. Register x0 is set to the value 0. register x1 is used to save the return address of a subroutine (has the alias LR (Link Register)). Register x14 is reserved for the stack pointer and has the alias SP. Alongside the register file, there is also a program counter.

5.4.2 Pipeline

The Tzscale has 3 pipeline stages namely Instruction Fetch (IF)- Decode and Execute (DE) - Write Back (WB). Unlike the 5 stage RISC architecture, in Tzscale the DE and EX stages are combined in a single stage, an Memory Access (MA) stage is missing. The memory load results are available on the bus in the WB stage. The operations performed in each stages are listed as follows:

- IF : A new instruction is fetched from program memory and is issued.
- DE :
  - The instruction is decoded and the operands are read from the register file. The target address is sent to program memory.
  - This is the stage in which the ALU and shifter units execute their operation.
  - The multiply unit executes in this stage.
  - The multi-cycle iterative division is started in this stage and can take variable number of cycles to finish.
  - For memory load operations, the effective address is computed and is sent to the memory. For store operations both address and data are sent to the memory. The load or store operation is started.
  - The unconditional jump instruction executes in this stage.
  - The conditional branch instructions execute in this stage.
- WB :
  - The result of memory load operations is available on the data bus.
  - The result of ALU, shift, multiply, control and load operations is written to the destination field on the register file.

5.4.3 Data path

The data path of the Tzscale processor is shown in Figure 5.1. The DE and WB stages are marked on the left side of Figure 5.1. In the DE stage the operands are read from the register file (R as shown in Figure 5.1) using read ports r1 and r2. These read ports are connected to the inputs of the functional units ALU (via aluA and aluB), shifter (via shA and

shB), multiplier (mpA and mpB), store (SX) unit (dm_addr) and iterative divider (not shown in Figure 5.1). The ALU, shifter and multiplier produce results at the end of DE. Their results are then stored in the register PD before being written to the register file in the WB stage. The multiplier has a single cycle latency and hence it behaves in the same way as the ALU and the SH units. The result of a load is also available in WB. It goes through a sign/zero extension unit LX, and is written to the register file. In case of a store operation the relevant part of the data is extracted in the SX unit. For both load and store operations, the effective address is computed on the ALU. The output aluC is copied to the address bus dm_addr.

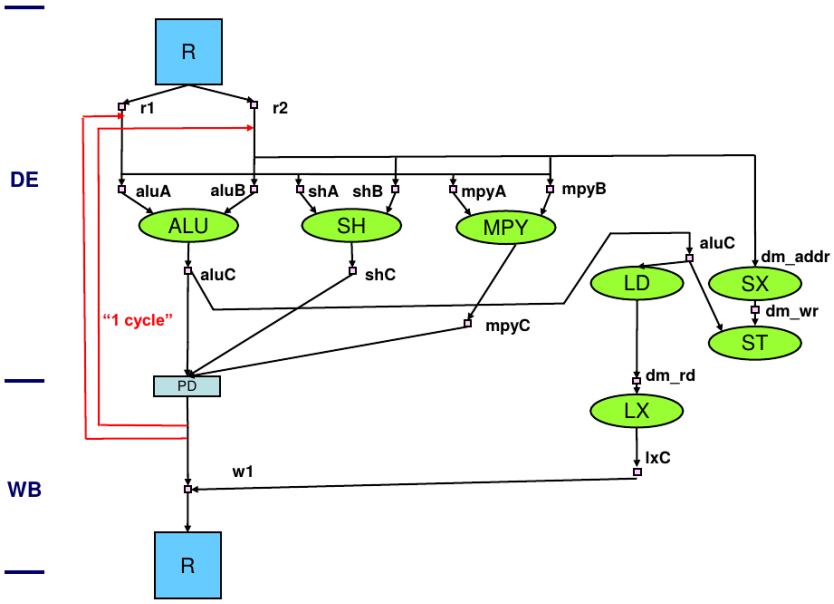


Figure 5.1: Data path of the Tzscale processor

5.4.4 Instructions

The Tzscale processor implements the following instructions of the base RV32 ISA : LUI, JAL, JALR, BEQ, BNE, BLT, BGE, BLTU, BGEU, LB, LH, LW, LBU, LHU, SB, SH, SW, ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, ADD, SUB, SLL, SLT, SLTU, XOR, SRL, AND, SRA and OR. The AUPIC instruction is not modeled. In addition to these instructions, the multiplication instruction is also implemented as en extension to the RV32E ISA. The different multiply instructions that are supported are : MUL, MULH, MULHSU, MULHU.

Another point to note is that C floating types float and double are supported by Tzscale. However, floating point operations are emulated in software.

Chapter 6

Design Methodology

In this chapter, the top-down design methodology followed is explained step-by-step. Figure 6.1 shows the overall design approach undertaken in this research assignment. It follows the generic approach [29] taken while design of an ASIP for a target application. Initially, the target application code is implemented on a reference design. This step is then followed by the profiling the application to identify performance bottlenecks. The reference processor design is then customized through the addition of application specific instructions. Finally, the necessary changes for updating the software toolchain (compiler, linker, etc.) are performed. In this thesis, some extra steps are performed in addition to the steps explained in the generic approach. These steps are: search algorithm, simulation and verification, and synthesis for a particular technology. All the steps shown in Figure 6.1 are addressed in further sections.

6.1 Target Application Code Implementation

As a first step, the search algorithm (as described in Chapter 2) is implemented on the Tzscale processor. The search algorithm is implemented in C on the Tzscale processor.

A MATLAB implementation for the given algorithm is readily available. This MATLAB realization is used as a reference against which the results of the C algorithm are validated. The C realization of the search algorithm is simplified keeping in mind the complexity of the overall ASIP development process. The MATLAB version of the search algorithm involves calculation of the coefficient values using high complexity functions such as the user developed function as ”permn” [1](refer Appendix A) etc. In the MATLAB version of the search algorithm, the coefficient values for a single antenna and the subsequent dictionary are generated each time the search operation needs to be performed. This generation of the coefficient values and dictionary is not performed in the same way inside the processor. As discussed in Chapter 2, the Dictionary is stored in an external memory unit. By that logic the coefficient values will be fetched from the memory unit every time optimum coefficient search needs to be performed. In the current implementation, the coefficient values are made available directly inside the processor.

[1]The permn function takes a the coefficient values and then calculates all possible permutations of the given set of values for a defined number of antennas.

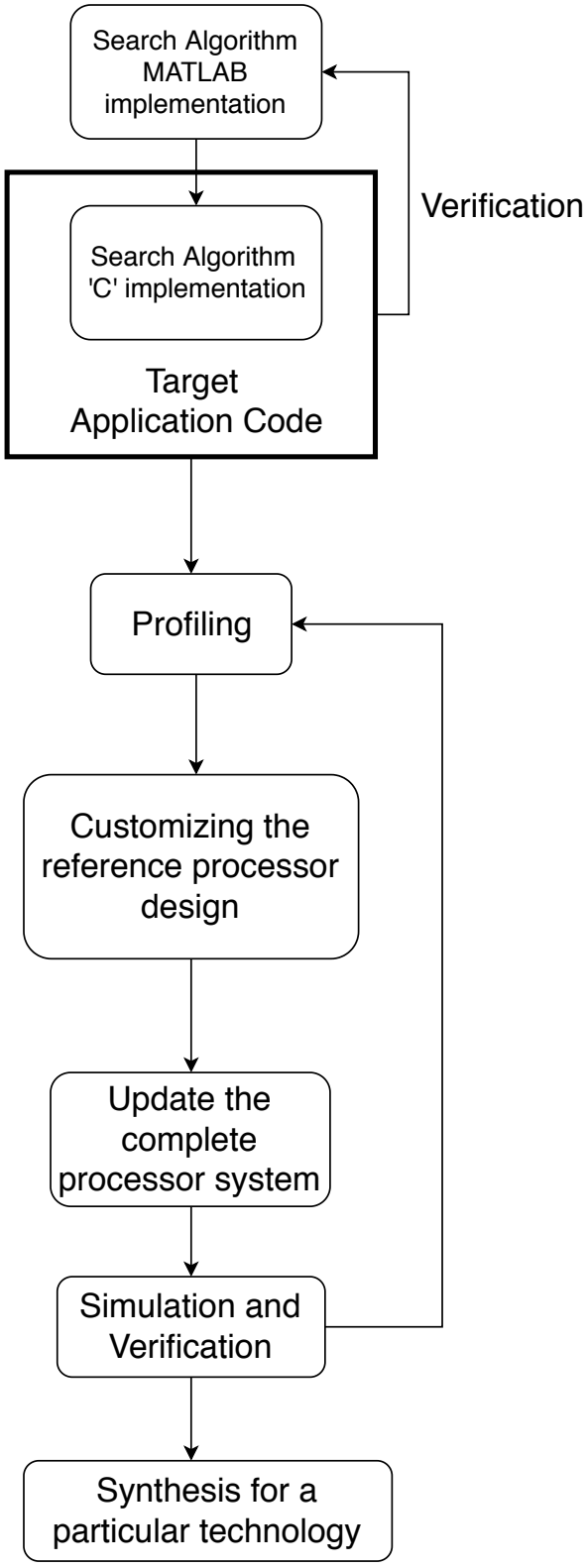


Figure 6.1: Top-down design approach

The search algorithm also requires the calculation of the whitened matrix of the analog beamforming coefficient values. The whitened matrix calculation involves calculating square root of the covariance matrix. The matrix square root calculation is a complicated operation.

Hence, for now these values are directly imported in the processor (these values are hard-coded). These sum up the simplifications that have been made to the search algorithm from the MATLAB to C.

All the coefficient values are complex in nature. The implementation of the C code must be such that the complex numbers and their operations are dealt with properly. In order to do so, a *struct* was made to handle the real and imaginary part of the complex numbers. All the basic operations to deal with complex number arithmetic (such as addition, multiplication etc.), handling of vector and matrix operations were made into a library. Once, the basic operations were implemented and their proper functioning verified, the core search part was implemented (refer Appendix B). Initially, the code was tested on the native machine to verify the correctness of the program. After validating the results of the C implementation with the MATLAB results, the code was ported to the Tzscale processor.

Another aspect of the search algorithm that needs to be considered is that, apart from being complex in nature the analog beamforming coefficients also have floating-point values. The Tzscale does not have hardware which supports floating-point operation. Floating-point arithmetic is emulated using SoftFloat library. SoftFloat is a software implementation of floating-point that conforms to the IEC/IEEE Standard for Binary Floating-Point Arithmetic.

6.1.1 Fixed-point implementation of the search algorithm

The Synopsys ASIP tool designer provides with an example processor model viz. FLX [2] processor which has hardware floating-point arithmetic unit. The search algorithm is also implemented on this processor to get an idea of the impact of emulating floating-point arithmetic (in Tzscale) in reference to usage of hardware floating-point (in FLX). The comparison of instruction count and cycle count for the search algorithm for the FLX and Tzscale processor is shown in Table 6.1.

Processor	Tzscale floating-point implementation	FLX
Cycle count	536880	225667
Instruction count	466765	222196

Table 6.1: Search algorithm instruction and cycle count comparison between Tzscale and FLX

From Table 6.1 it can be seen that in the presence of floating-point hardware $\approx$50% less cycles as well as instructions are required for execution. The range of values that are currently being used in the search algorithm are such that the entire range of floating-point numbers is not being used. This logically leads to the implication that maybe a fixed-point implementation of the same algorithm on the Tzscale processor might give better results in terms of cycle and instruction count.

To define a fixed-point type we need two parameters: width of the number representation and the position of the binary point within the number. Keeping this in mind, the range of values that are currently used for the search algorithm were analyzed and the following requirements were obtained:

[2] Information about the FLX processor is available in Appendix D

Processor	Tzscale Softfloat implementation	Tzscale Fixed-point implementation	FLX
Cycle count	536880	387642	225667
Instruction count	466765	341251	222196

Table 6.2: Search algorithm instruction and cycle count for FLX and 2 different implementations on Tzscale

- 4 bits are required to represent the integer part of the fixed-point representation. On the basis of the current range of values being used 3 bits are also sufficient, but, an extra bit is considered to account for high interference in the channel[3].
- The the fractional part requires a granularity of 0.0001. In order to achieve this granularity, 14 bits are enough ($2^{-14} = 0.00006$).
- Based on the above two requirements, a total of 18 bits are sufficient to implement the search algorithm using Fixed-point notation. The Tzscale processor already provides a 32-bit datapath. Hence, for the fixed-point notation: 4 bits are chosen to represent the integer part and 28 bits are used to represent the fractional part. The 28 bits provide a granularity of $2^{-28} = 3.7252903 \times 10^{-9}$.

A library for fixed-point arithmetic is implemented. Keeping the fixed-point arithmetic software based provides with the advantage that the range of values and the granularity is dynamic and can be changed at any point in time based on the channel conditions and requirements of the MIMO communication system.

6.2 Profiling

The verification of the implementation of the fixed-point form of the search algorithm was performed successfully. The new cycle and instruction count obtained for this implementation is presented in Table 6.2 along with the results from Table 6.1.

This step was then following by the function-level profiling of the algorithm as executed on the Tzscale processor as well as the FLX processor. The profiling operation is performed using the profiler provided by the Synopsys ASIP designer tool. The profiling results for 3 different implementation scenarios are presented in Table 6.3. These three different scenarios are: search algorithm implementation on Tzscale processor using Softfloat library and fixed-point library, and the search algorithm realization on the FLX processor. From the profiling results presented in Table 6.3 it can be seen that for the Tzscale Softfloat implementation, 32-bit floating-point multiplication and floating-point square root are the dominant functions. When the search algorithm is implemented on the same platform using fixed-point library then floating-point square root and integer division become the most dominant functions. On the FLX processor the most dominant functions are the floating-point square root operation followed by the dot production calculation function (user defined function).

Before drawing conclusions from these profiling results, the following points must be noted:

[3] An approximate range for high interference value has been considered

Platform	Tzscale floating-point implementation		Tzscale fixed-point implementation		FLX	
Dominant Functions	Function	% of total cycles	Function	% of total cycles	Function	% of total cycles
	Floating point multiplication	22.51	Floating point square root	22.89	Floating point square root	67.75
	Floating point square root	21.19	Long integer division	19.92	Dot product calculation	9.05
	Floating point round and pack function	14.88	Floating point multiplication	9.22	Complex number multiplication	3.40
	Floating point add and subtract function	14.82	Dot product calculation	7	Floating point less-than function	3.21

Table 6.3: Profiling results for different implementations on different platforms

- For the search algorithm, the square root and division operations are performed repeatedly in a loop.
- Dot product calculation also forms a part of the loop operation.
- The FLX processor takes 29 clock cycles for floating-point division and 26 clock cycles for floating-point square root calculation.
- Certain operations for the Tzscale fixed-point implementation and the FLX implementation still make use of the softfloat functions for e.g. the floating-point “less than” operation which can be seen in Figure 6.3.

Based on the profiling results obtained for the fixed-point implementation, implementing the square root operation in hardware on the Tzscale processor was chosen as the next design decision.

6.3 Square root implementation

The FLX processor implements the square root algorithm using the restoring shift/subtract algorithm. The calculation is performed in 26 clock cycles. This is considered as the reference in terms of clock cycles required for performing square root operation. The square root implementation executed on the Tzscale should be less than or atleast at par with the FLX processor in this regard to justify its addition to the existing architecture.

A comprehensive list of methods to calculate the square root of a given number is presented in the book *Computer Arithmetic* by Behrooz Parhami [30]. This list includes the

following methods of calculating the square root of a given number: restoring (shift/subtract) algorithm, non-restoring algorithm, high-radix (digit recurrence) square rooting and square rooting by convergence (the Newton-Raphson method). Along with the study of the methods that were presented in this book, in order to implement an efficient square root algorithm which can compete with the square root implementation in FLX, a short research was carried out. The results of these research and the final method chosen for implementation is presented further.

[31] proposes a digit recurrence square root design (high-radix square rooting) and [32] proposes a combined square root, multiply and divide unit based on similar lines of digit recurrence. These implementations are quite specific to the problem they want to tackle. At the same time, changing the digit representation system appears to a bit complex and unnecessary operation when it comes to the problem at hand.

[33] presents an FPGA implementation of a 32-bit fixed-point square root based on the non-restoring algorithm. The operation latency achieved is 25 clock cycles for a precision of 8-bits after the decimal point in the fixed-point representation. Although, the latency provided is comparable to that provided by FLX, it is dependent on the number of bits after the decimal point. In this case, it would be more desirable to have a square root implementation which is independent of the placement of the binary point in the fixed-point number.

[34] implements the square root using the non-restoring algorithm with a reduced circuit area without loss in precision. The implementation targets circuit area optimization when compared to a more classical approach to implement the same algorithm. [35] presents an algorithm which improves upon the Newton-Raphson method by providing a novel method to get a better initial guess before starting the convergence iteration. Hence, a faster convergence time is expected using this method.

[36] proposes a modified non-restoring algorithm to calculate the integer square root of a 32-bit number in 17 clock cycles. More elaboration on this method is presented further.

Keeping in the mind the complexity of the various algorithms briefly explained above and the requirements of the system, [36] was shortlisted for implementation. This algorithm focuses on the speed of operation instead of the area reduction. The algorithm presents a fixed iteration value (or definite convergence time) for calculation and is also simple to implement compared to the other approaches. Additionally, selecting algorithms which improve upon the area requirements of existing algorithms will make much more sense after an initial customization based on a simple algorithm has already been made as a starting point.

6.3.1 Modified non-restoring Square root

Here, the modified non-restoring square root algorithm is briefly discussed. The restoring, non-restoring and modified non-restoring square root calculation algorithms calculate the square root and remainder value by an iterative process. The equation for the square root calculation can be written as:

$$D = Q^2 + R \tag{6.1}$$

$$\text{where D is the input data (or radicand), Q is the quotient and R is the remainder}$$

The quotient Q given in equation 6.1 is the square root of the radicand D. A small example is presented below to understand the modified non-restoring algorithm.

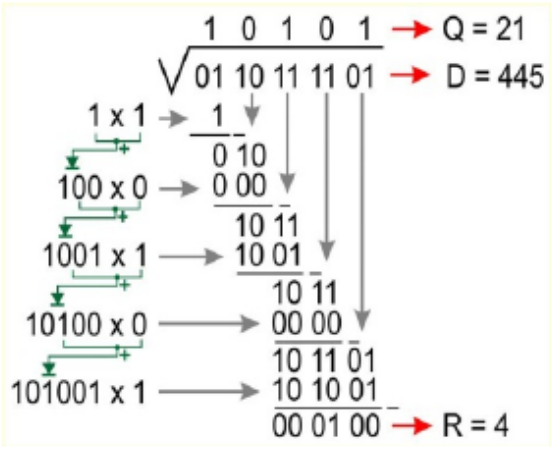


Figure 6.2: Illustration of modified non restoring algorithm [36]

Figure 6.2 shows the example of calculation process step by step. The input D has the value 445 in the decimal representation. Binary representation of D is $(110111101)_2$. The radicand value is divided into groups of 2-bits. If there are odd number of bits, then an extra '0' bit is added at the beginning. Calculation is started with first MSB group. Here, "01" is the first MSB group. The quotient bits are estimated and finalized one bit at a time. An initial guess of '1' is made for the MSB bit of the quotient. The square of the MSB bit of the quotient will be '1'. This value is equal to the current MSB group value, hence a subtraction will be performed and the estimated MSB quotient value will be finalized to '1'. The next iteration is performed by fetching the new group from the radicand value. Here, "10" is the group after the MSB group. If there is any remainder from the previous subtraction then it is also considered along with the new group of bits. To estimate the second MSB bit of the quotient, the current quotient value is shifted left by one bit and also '1' is appended to this shifted value. This value is then compared to the new group of radicand which has been fetched (along with any residual bits from previous subtraction result). In this case, the value "101" will be compared to "010". Since, the estimated value is greater than the fetched value, subtraction is not performed and the second MSB bit of the quotient is set to '0'. The new group of radicand value fetched at each iteration is called the partial remainder, the quotient value shifted by left and appended with '1' at each iteration is called the partial factor. To summarize the working of this algorithm, it can be said that the partial factor value is compared to the partial remainder value at every iteration. If the partial remainder is greater than or equal to the partial factor a subtraction is performed and a new quotient bit for that iteration is set to '1' (otherwise it is set to '0'). This sequence of operations is carried out for all the groups and the entire quotient value is determined. For the example shown in Figure 6.2, the quotient value is 21_{10} and the remainder value is 4_{10}.

For a 32-bit number, 16 groups will be formed. Hence, 16 iterations need to be performed to calculate the 16 quotient bits. This equates to the usage of 16 clock cycles to determine the quotient value. An additional clock cycle is required to perform the necessary initialization of the radicand, partial remainder and partial factor values at the start of the square root operation. Therefore, this method requires 17 clock cycles to calculate the square root of a 32-bit number.

The restoring algorithm will not perform a comparison with the current bit factor and new remainder value. Instead a subtraction is always performed. Based on the sign of the result

of subtraction, it will be decided whether the bit factor value should be '1' (for a positive subtraction result) or '0' (for a negative subtraction result). If the result of subtraction is negative, then the bit factor value must be '0' and the obtained negative subtraction result will be restored to previous positive value. Hence, it is termed as the restoring algorithm. For the non-restoring algorithm, as the name suggests no restoring operation is performed. Since no restoration is performed, the remainder and quotient value obtained at the end of the iteration cycle need correction. In the modified non-restoring, since an extra comparison is performed before subtraction, neither restoration or correction at the end is required.

The claims of [36] were verified by performing a VHDL simulation. Figure 6.3 shows the simulation results for the case when the input/radicand value is "1000000". The output value/quotient is correctly calculated as "1000" in 17 clock cycles.

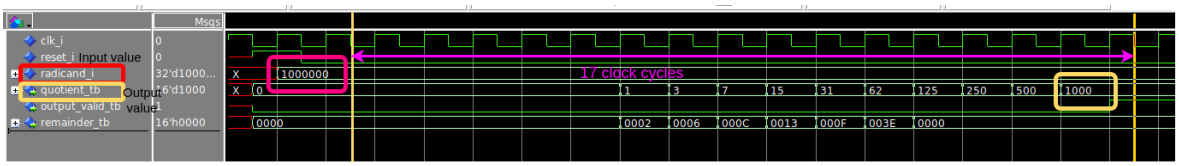


Figure 6.3: Modified non restoring algorithm simulation results as proposed in [36]

Thus, it is verified that the modified non-restoring algorithm calculates the square root of a 32 bit number in 17 clock cycles.

6.4 Customization of the reference design

The necessary information about which customization has be performed on the basis of profiling and a study of the methods which can lead to an efficient implementation has been presented in previous sections. In this section, the addition of the square root unit to the Tzscale processor in Synopsys ASIP designer has been explained.

The modified non-restoring algorithm has been chosen to perform the square root calculation which takes 17 clock cycles to produce the desired value. Now, to implement a hardware module which takes multiple cycles for execution, a special Multi Cycle Functional Unit (MCFU) has to be designed in Synopsys ASIP designer.

6.4.1 MCFU design in Synopsys ASIP designer

A MCFU is a separate module in the design with dedicated hardware resources and local registers. The MCFUs follow a separate thread of execution in parallel to the main thread. This helps in keeping the processor pipeline shallow. The MCFU is modeled as follows:

- The primitive function which starts the multi-cycle operation is defined with "multi-cycle property".
- The instruction which starts the MCFU is modeled as an nML action.
- The MCFU is modeled as a PDG module like the controller of the processor.

```
////////////////////////////////////////////////////////////////////////
//iterative square root primitive
    w32 mnr_sqroot (w32 rad) property(multicycle_17);
////////////////////////////////////////////////////////////////////////
// control primitives
```

Figure 6.4: Primitive function for square root unit

- Hardware stall rules and hazard rules are added to ensure the proper functioning of the multi-cycle unit with the rest of the instructions.

In further sections the different modifications made in the existing Tzscale processor for adding the square root module are discussed in depth.

6.4.2 Definition of the primitive function

The primitive function is defined in the primitives header file ”tzscale.h”. The primitive function is defined with the property *multicycle_17* and is shown in Figure 6.4. This property hints to the compiler about the latency of the MCFU on which the primitive function is mapped. The controller stalls the operation of the pipeline for the whole duration of primitive function operation. A control signal is also generated which remains high for 17 clock cycles. This control signal is used while defining the behavior of the MCFU PDG module.

6.4.3 Definition of the nML action

A screenshot of the nML model for the square root module which uses the primitive function defined earlier is presented in Figure 6.5. The action specifies interface (operands and result registers) for compiler. The output *quotient* is not written as a result of nML action (as in the case of other instructions) but will be written in PDG code of the MCFU. The keyword *cycles(1)* indicate that a new instruction can be started in the next cycle.

6.4.4 Design of the MCFU as PDG module

The iterative square root algorithm can be summarized as:

1. Preparation [4] and initialization of the input data D(radicand), remainder R, square root Q (quotient), partial factor F, and bit index i.
2. Grouping of the radicand into sub-groups of 2 bits each which are fetched at every clock cycle.
3. The current sub-group of the radicand is treated as the current partial remainder.

$$R_t = D_t[i : i-1]; \mathrm{t\ is\ the\ time\ index\ indicator} \tag{6.2}$$

4. Comparison between the current remainder and partial factor value $((F_t << 1)|1)$ is performed. If the current remainder value is greater than or equal to the partial factor

[4] Preparation here refers to the loading of the input values into local registers of the MCFU

```
fu sqroot;
trn radicand<w32>; hw_init radicand = 0;
trn quotient<w32>; hw_init quotient = 0;

//address transitoryto seek out the quotient register in the register set
trn sqr_wad<uint5>;

// Hazard transitories
trn sqr_busy<uint1>;  hw_init sqr_busy = 0;  //to avoid RAW and WAW hazards
trn sqr_cnt<uint1>;   hw_init sqr_cnt = 0;  //to avoid using the square root unit while its still busy
trn sqr_wnc<uint1>;   hw_init sqr_wnc = 0; // writes next cycle, blocks the quotient register so that no one else can wri
trn sqr_addr<uint5>;  hw_init sqr_addr = 0;

opn sqroot_instr (rs1: mR1, rd:c5unz)
{
        action {
        stage DE:
                  //quotient = mnr_sqroot(radicand = rs1) @sqroot;
                  R[sqr_wad = rd] = wd = quotient = mnr_sqroot(radicand = rs1) @sqroot;
                  // mnr_sqroot(radicand = rs1, quotient =rd, remainder=rd) @sqroot; --this would have been possible only
                  //Currently the register set has only one write port and two read ports.

        //stage WB:
        //      R[sqr_wad = rd] = w1 = quotient;
        }
        syntax : "sqroot" "x"rd ", "rs1;
        image  : sqr_imm.imm_sqr::rs1::funct3_mul_div.sqr::rd::opc.sqr_e, class(sqroot), cycles(1);

}
```

Figure 6.5: nML model for square root module

value then :

$$Q_{t+1} = (Q_t << 1)|1; F_{t+1} = ((F_t + F_t[0]) << 1)|1); \tag{6.3}$$

else

$$Q_{t+1} = (Q_t << 1)|0; F_{t+1} = ((F_t + F_t[0]) << 1)|0); \tag{6.4}$$

5. Based on the results of the comparison, also perform subtraction. The result of the subtraction is appended to the next sub-group of radicand that is fetched every iteration.
6. Update the iteration indexes i and t.
7. Continue the process from step 4, till all the sub-groups of the radicand have been fetched.

The implementation of this iterative algorithm in the MCFU PDG module can be seen partly in the Figure 6.6. Since, the entire implementation is quite long, approximately 40% of the implementation has been presented. The control signal generated as a result of the multi-cycle property is supposed to remain high only for one clock cycle. However, during implementation it was found that this signal ("sqroot_start variable") remains high for the entire duration of 17 clock cycles. The PDG module was then adapted accordingly.

6.4.5 Hazard management for the MCFU

One of the implications of the multi-cycle nature of the square root operation is the introduction of the different hazards in the processor. In order to avoid these hazards during program execution, certain additions need to made to the *hazards.n* file of the Tzscale processor. The modifications made have been presented in Figure 6.7. The hazards which are avoided are: read-after-write and write-after-write for the destination register of the square

```
//Transtories and static storages that are declared in nML can be accesssd from inside the process function
multicycle_fu sqroot
{
    //Declaration of local storages
    reg curr_partial_factor<uint16_t>;
    reg res_quotient<uint16_t>;
    reg partial_remainder<uint16_t>;
    reg quotient_reg_addr<uint5_t>;
    reg i<uint5_t>;
    //reg enable<uint1_t>;
    //reg dg_sqr_busy<uint1_t>;
    //reg dg_sqr_cnt<uint1_t>;
    //reg dg_wnc<uint1_t>;

    //initialization of local storage
    hw_init curr_partial_factor = 0;
    hw_init res_quotient = 0;
    hw_init partial_remainder = 0;
    hw_init i = 17;


    //the process function is called every cycle

    void process(){
        // Here, the behaviour of the multicycle unit is defined
        //check for the enable signal written in the tzscale_ctr_enab_iss.txt
        //result of comparison is uint1_t type
        uint16_t copy_partial_rem;
        uint16_t copy_curr_part_fact;
        uint16_t copy_res_quotient;
        uint1_t sqroot_start = quotient_mnr_sqroot_radicand_sqroot_DE_sig;
        //enable = quotient_mnr_sqroot_radicand_sqroot_DE_sig;

        //sqroot_start signal remains high for 17 clock cycles
        if (sqroot_start){
         if (i==17) {
            curr_partial_factor = 0;
            res_quotient = 0;
            partial_remainder = 0;
            quotient_reg_addr = sqr_wad;
            i = 16;
          }
         else if (i < 17 && i>=0){

             //updating the temporary variable values
             copy_res_quotient = res_quotient;
             copy_partial_rem = partial_remainder;
             copy_partial_rem = copy_partial_rem << 2;
```

Figure 6.6: A part of the square root MCFU PDG module

```
// Hardware stalls for square root
//Stall when an instruction reads the result while the unit is still busy (RAW)
hw_stall trn(){
    trn(sqr_busy); address_trn(sqr_addr);
} -> {
    stage DE: ... = R$[#];
}
// Do not write to register before result is written (WAW)
hw_stall trn () {
    trn(sqr_busy); address_trn(sqr_addr);
} -> {
    stage WB: R$[#] = ...;
}
// No new division may be started while the previous is still busy
hw_stall trn () {
    trn(sqr_cnt);
} -> {
    class(sqroot);  // trn(div_new);
}
// No instruction may use register write port when result is written
hw_stall trn () {
    trn(sqr_wnc);
} -> {
    stage WB: R$[] = w1;
}
```

Figure 6.7: Managing hazards in the Tzscale processor

root unit, multiple issuing of square root unit while it is still busy and possibility of no other functional unit writing to the destination register while the square root unit is writing back to the destination register.

6.5 Updating the complete processor system

When the entire process mentioned above has been compiled successfully, the next step is updating the processor system. This process involves updating compiler directives, addition of the custom square root instruction at the application level. This addition is performed based on the C data types. The custom square module implemented here takes as input 32-bit integer values and also produces a 32-bit integer value at the output. Hence, the custom square function is defined only for the int data type.

```
promotion int mysqrt(int) = w32 mnr_sqroot(w32);
```

Figure 6.8: Custom square root instruction to be used at the user level

Figure 6.8 shows the custom square root instruction which will be used at the application level instead of the *sqrt* function from the *math.h* standard library. The name of this custom instruction has been given as *mysqrt* and its usage in C code has been shown in Figure 6.9. The assembly view of this instruction is shown in Figure 6.10.

```
1  #include <stdio.h>
2
3
4  int main()
5  {
6  int a = 10;
7  int b;
8  b = mysqrt(81);
9  b = a+b;
10 printf("First sqaure root= %d \n",b);
11 a = mysqrt(b);
12 printf("Second sqaure root = %d \n",a);
13 b = mysqrt(4765);
14 printf("Third square root = %d \n",b);
15 }
```

Figure 6.9: Usage of “mysqrt” function

```
0  05 10 05 93   addi x11,x0,81
4  80 05 f5 0b   sqroot x10, x11
8  0c 11         addi x2, 12
```

Figure 6.10: Assembly view of the new square root instruction

6.5.1 Opcode addition to the RISC-V instruction set

The RISC-V base opcode map is shown in Figure 6.11. Major opcodes with 3 or more lower bits set have been reserved for instructions with length greater than 32 bits. Opcodes marked as reserved are to be avoided for custom instruction set extensions as they might be used by future standard extensions. Major opcodes which are marked as custom-0 and custom-1 will be avoided by future standard extensions and are recommended for use by custom instruction-set extensions within the base 32-bit instruction format. The opcodes marked custom-2/rv128 and custom-3/rv128 are reserved for future use by RV128, but will otherwise be avoided for standard extensions and so can also be used for custom instruction-set extensions in RV32 and RV64.

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

Figure 6.11: RISC-V base opcode map inst[1:0]= 11 [26]

Based on the above information and Figure 6.11 the opcode for the square root instruction is implemented in the custom-0 space which means instruction bits 2-4 will be “010” (since Tzscale has a 32-bit wide data path). The instruction bits 5-6 will then “00”. The instructions bits 0-1 have already been fixed to “11”. Apart from this, an instruction type also needs to be decided. As discussed previously, Tzscale processor has 4 main instructions types: R-type, I-type, S-type and U-type (refer Appendix C). The square root operation will only have one input operand and one output value. Hence, it makes more sense to use the I-type instruction format which has a single source register and destination register in its encoding. The immediate operand in the I-type instruction format is filled with a random

value of 11-bits to make a uniform 32 bit encoding for the instruction.

The syntax and image defined in the nML model as shown in Figure 6.5 along with modifications to the *opcode.n* file in the Tzscale processor model, together help in defining the opcode and the view of the instruction in assembly. This sums up the opcode addition process. Additionally, the original floating-point square root extension of the RISC-V ISA also encodes the floating-point square root function along similar lines.

6.6 Simulation and Verification

Synopsys ASIP designer provides with the option to perform simulation with the help of an instruction set simulator. The designer can choose to develop two types of instruction set simulators: cycle true or instruction true. Here, the decision was made to go ahead with the cycle true type of instruction set simulator. With the help of a cycle true instruction set simulator, the line-by-line execution of the search algorithm execution can be tracked in the instruction set simulator which shows the view the assembly code. The results obtained using the instruction set simulator have been presented in Chapter 7.

At the same time, using the “GO” tool provided as a part of the Synopsys ASIP designer synthesizable RTL of the complete processor model can be generated. This model can then be used for performing simulation and verification of the entire search algorithm execution at the RTL level. The process of generation of the RTL is managed with the help of a GO configuration file. More details on how to deal with the GO configuration file have been given in Appendix D.

6.7 Synthesis

Once the processor model working has been verified at the instruction-set level and the RTL level, the last step is the synthesis of the processor model for a particular technology. The expectation is to synthesize the processor model for the UMC 65 nm technology. Subsequently power analysis for the search algorithm execution is also a part of the top-down design approach taken in this chapter. The synthesis results have been presented in Chapter 7.

Chapter 7

Results and Evaluation

In this chapter, the results obtained after performing the desired customization are discussed and evaluated. In the first section, the profiling results after modification of the Tzscale processor are discussed. This is then followed by the result verification that is performed using the instruction set simulator and RTL level simulations. The chapter concludes with the presentation of the synthesis results for the modified ASIP design, Tzscale and FLX processor.

7.1 Profiling results after addition of square root module

Here, the different implementations that were created in this research assignment are summarized:

1. The search algorithm (obtained after converting the MATLAB code to 'C' code) is implemented on the host machine. Verification of the solution of this 'C' algorithm is performed against the result provided by the MATLAB code.
2. This search algorithm in 'C' is then implemented on the FLX processor (which has floating point arithmetic hardware support) and Tzscale processor (which emulates floating point arithmetic using Softfloat library). The results from these two platforms are successfully cross-verified against MATLAB implementation. Profiling is performed on both the platforms.
3. Another version of the search algorithm uses user-written fixed-point library. This fixed-point version is implemented only on the Tzscale platform and profiling is performed again.
4. Based on the results of the second profiling of the fixed-point version of the search algorithm on the Tzscale platform, a decision to add the square root unit as a customization to the Tzscale processor is made. The final implementation of the search algorithm is the version which is based on the use of fixed-point library and runs on the modified Tzscale processor (with the square root customization).

In Chapter 6, the profiling results for implementations mentioned in Step 2 and 3 are presented and discussed. Here, the profiling results for the final fixed point version of the search algorithm as implemented on the modified Tzscale processor are presented.

Table 7.1 shows the comparison between 4 different implementations of the search algorithm as summarized previously. Column 1 shows the cycle count and instruction count for the modified Tzscale processor with the added square root unit when the algorithm is implemented.

Processor	Modified Tzscale processor with square root unit	Tzscale floating-point implementation	Tzscale fixed-point implementation	FLX
Cycle count	255552	536880	387642	225667
Instruction count	227044	466765	341251	222196
Program memory usage (bytes)	12710	16072	18366	11788

Table 7.1: Search algorithm instruction and cycle count for the different search algorithm implementation on different platforms

From Table 7.1, the following observations can be made:

1. As compared to the floating-point search algorithm implementation on the Tzscale processor, the same algorithm on the FLX processor takes 57.97% less clock cycles and 52.40% fewer instructions.
2. Comparing the fixed-point implementation and floating-point implementations of the search algorithm on the Tzscale processor, the fixed-point implementation takes 27.8% fewer clock cycles and requires 26.89% less number of instructions.
3. When the fixed-point implementation on the Tzscale is compared with the floating point implementation on the FLX processor, the FLX takes 41.7% less number of clock cycles and 34.88% less number of instructions are required.
4. Comparing the fixed point implementation on the Tzscale and the the modified Tzscale, the modified Tzscale processor requires 34% fewer clock cycles and saves 33.46% of the number of instructions required.
5. As compared to our reference design of floating point hardware i.e. the FLX processor, the modified Tzscale processor requires 13.24% more clock cycles and 2.1% more instructions.
6. The modified Tzscale processor saves 52.4% clock cycles and 51.35% instructions when compared to the unmodified reference Tzscale design which executes the floating point version of the search algorithm.
7. The memory usage for the Tzscale floating-point implementation is 26.6% more than the FLX processor. The fixed-point implementation uses 12.49% more memory compared to the floating-point implementation on the Tzscale processor. The fixed point implementation on the modified Tzscale processor requires 44% less memory compared to the Tzscale fixed-point implementation and 31.41% less memory compared to the Tzscale-floating point implementation. Lastly, the fixed point implementation on the modified Tzscale processor uses 7% more memory compared to the floating point implementation on the FLX processor.
8. For all the three parameters shown above, the modified Tzscale processor is comparable in performance to the FLX processor.

From these results, it can be seen that when compared to the Tzscale design which executes the floating-point version of the search algorithm, it is a combination of fixed-point search algorithm implementation and customized Tzscale processor which yields the best performance.

Additionally, in Chapter 6 it was mentioned that the FLX processor completes a square root operation in 26 clock cycles. The modified Tzscale processor executes the square root operation in 17 clock cycles. The final cycle count of the Tzscale processor is comparable to the FLX processor as shown in Table 7.1. Here, it must be noted that the FLX performs floating point square root whereas the modified Tzscale processor performs integer square root (in addition to fixed point square root arithmetic manipulations). At the same time, there are still certain operation in the fixed point version of the algorithm which make use of the Softfloat library functions, for example, the 32-bit floating point multiplication. This justifies why the results obtained even after performing customization are only comparable between the modified Tzscale and FLX processor.

The final fixed-point version of the search algorithm as executed on the Tzscale processor is again profiled and the new profiling results which show the new dominant functions are presented in Table 7.2.

Function	Percentage of total cycle count
Long integer division operation	30.40
32-bit floating point multiplication	13.99
Dot product calculation	13.10
Fixed point multiplication	9.77
Complex number multiplication	7.66

Table 7.2: Modified Tzscale profiling results

The profiling results before modifying the Tzscale processor which also executes the fixed point version of the search algorithm have been presented again in Table 7.3 (these results were previously presented in Chapter 6).

Function	Percentage of total cycle count
Floating point square root	29.14
Long integer division	20.39
32-bit floating point multiplication	9.73
Dot product calculation	7

Table 7.3: Profiling results for fixed point search algorithm implementation on Tzscale processor

On comparing Table 7.2 with Table 7.3, it can be seen that the floating-point square root function which is the most dominant function in the fixed-point implementation is no longer dominant on the modified Tzscale processor. This is because, the square root operation is not emulated in software on the modified Tzscale processor and does not consume a large percentage of the total clock cycles required. This result proves that the expectation from the implemented customization is met.

Another point to be noted is that, although the performance expectation from the modified Tzscale processor is met, the final result obtained slightly deviates from the floating-point im-

plementation. This is caused due to the fixed-point usage and implementation nature of the square root unit. However, the deviation obtained is under limits and has been considered as acceptable specific to the application and the range of values specified.

Based on the new-profiling results for the modified Tzscale processor it was also investigated if adding a customization w.r.t to the long integer division makes sense. The Tzscale processor has a floating-point hardware divider. This divider is equipped to perform 32-bit floating-point division. The long integer division is not mapped to this divider unit since it involves with dealing with operands which are 64-bit long (hence the name long integer division). Keeping this mind, it would a better idea to focus on either removal of the long integer operation such that the hardware divider unit is used or adding micro-routines to implement a faster long integer division. Adding a separate divider unit which deals with 64-bit division would lead to increased hardware overhead as the available functional unit is not being utilized. It would be more beneficial to focus on other functions which are dominant apart from the long integer division operation.

7.2 Instruction Set Simulator Results and Verification

In this research assignment, the hybrid receiver system is considered to have 2 (N = 2) receiving antennas. Each antenna has a resolution value of 3 (R_w = 3) i.e. 8 possible beamforming coefficient values. This implies the dictionary has a set of 2^{N*R_w} i.e. 2^6 i.e. 64 values. The dictionary can be considered to be a 2x64 matrix, where each column represents a possible combination of coefficients for the 2 antennas (hence 2 rows). Since 64 such combinations are possible, there are 64 columns in the dictionary matrix. The objective of the exhaustive search carried out by the search algorithm in the ASIP is to find the best suited combination which improves the output of the analog beamformer. This is the reference scenario for which the result verification of the search algorithm has been performed in this research assignment.

The result produced by the implementation of the search algorithm on the host machine is verified against the result produced by the MATLAB implementation. Figure 7.1 shows the MATLAB result compared against the search algorithm implementation as executed on the host linux machine. Both these results show the value of whitened optimum coefficients for a given pair of co-variance and cross-correlation values. Only 4 digits after the decimal point are taken into consideration and hence, it can be said that the result produced by the host machine matches the MATLAB results.

From here on, the results produced by the host machine are considered the standard values against which the results of the instruction set simulator are verified. The results produced by the modified Tzscale processor which executes the fixed point version of the search algorithm are compared against the results produced by the host machine. The host machine executes the floating point implementation of the algorithm, the final results are converted to fixed point representation. This result comparison is presented in Figure 7.2.

The user defined fixed point library (used on the Tzscale) is used to convert floating point values to fixed point values when the search algorithm is executed on the host machine. The slight deviation in results is due to the approximation of values when moving from the floating point system to the fixed point system.

The correctness of the results obtained from the modified Tzscale processor has now

max_correlation.m | testing_max_correlation.m

```
1   function W = max_correlation( Rx,rxs,R_w,Nr )
2
3 -     theta = (2*pi)/(2^R_w);
4 -     c = 0 : (2^R_w-1);
5 -     W_angle = exp(1i*c*theta);
6
7       %% creating all possible values in dictionary %%
8 -     w = conj(permn(W_angle,Nr)');
9 -     w_wh = (Rx^0.5)*w;
10  %    disp(w_wh)
11      %%
12 -     result = -inf;
13 -     for i = 1 : size(w,2)
14 -        w1 = abs(w_wh(:,i)'*rxs)/norm(w_wh(:,i));
15 -        if w1 > result
16 -            w_final = w_wh(:,i);
17 -            result = w1;
18
19 -        end
20 -     end
21 -    W = w_final;
22
23 - end
```

Command Window

New to MATLAB? See resources for Getting Started.

```
>> W

W =

  1.7796 + 0.0096191i
  0.0096191 + 1.7312i
```

(a) MATLAB results

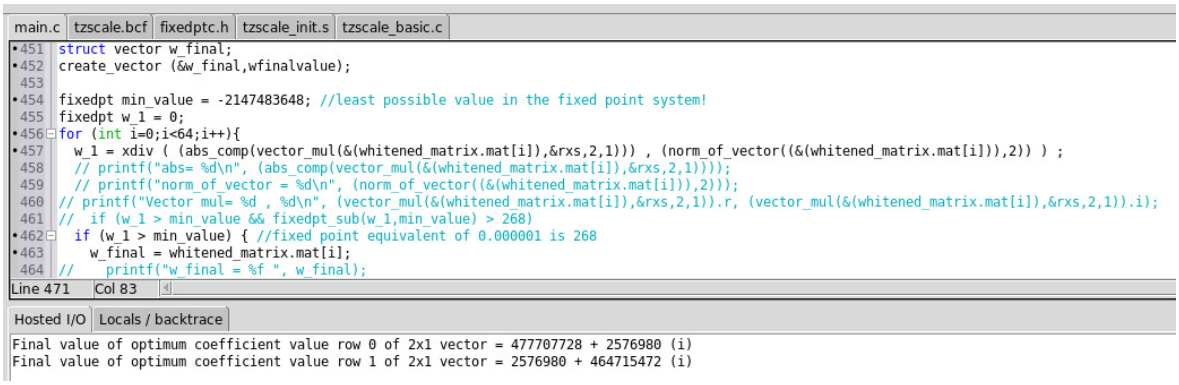
```
ashwini@probook:~/Documents/thesis/matlab_code/tzscale_experiment$ gcc main.c -lm
ashwini@probook:~/Documents/thesis/matlab_code/tzscale_experiment$ ./a.out
Final value of whitened optimum coefficient, Row 0 of 2x1 vector = 1.779600 + 0.009600 (i)
Final value of whitened optimum coefficient, Row 1 of 2x1 vector = 0.009600 + 1.731200 (i)
```

(b) Host machine results

Figure 7.1: Whitened optimum coefficient value result verification

```
ashwini@probook:~/Documents/thesis/matlab_code/tzscale_experiment$ gcc main.c -lm
ashwini@probook:~/Documents/thesis/matlab_code/tzscale_experiment$ ./a.out
Final value of whitened optimum coefficient, Row 0 of 2x1 vector = 477707712 + 2576980 (i)
Final value of whitened optimum coefficient, Row 1 of 2x1 vector = 2576980 + 464715456 (i)
```

(a) Host machine results



(b) Modified Tzscale results

Figure 7.2: Whitened optimum coefficient value result verification in fixed point representation

been established at the instruction set level. It was also verified in the instruction set simulator that the square root operation is in fact mapped to the hardware square root unit. This verification is presented in Figure 7.3. This figure also gives an illustration of the working of compiler-in-the-loop architectural exploration process as presented in Chapter 4. Figure 7.3 shows the assembly code for the custom square root function, the corresponding 'C' code (a function in the user defined fixed point library) and the hardware registers for the custom square root unit. The instruction set simulator offers in this way the ability to change to processor model according to the application requirements.

The cycle count and instruction count results as shown in Chapter 6 and the previous section are results obtained from the instruction set simulator. This sums up the discussion on the verification process performed using the instruction set simulator in Synopsys ASIP designer.

7.3 RTL level Simulation and Verification

The Synopsys ASIP designer is also used to generate the RTL model of the modified Tzscale processor. Pre-synthesis verification of the search algorithm execution on the modified Tzscale processor is performed at the RTL level in Modelsim. Separate program memory and data memory files are generated using the instruction set simulator which are used in the pre-synthesis simulation.

To illustrate the proper generation of the square root unit which is generated from the PDG module a simple example is considered. A 'C' program which calculates the square root of 81 is written and its implementation verified on the RTL level. Figure 7.4 shows the simulation results obtained for the 'C' code. In Figure 7.4, the signals: *pm_rd, pm_addr and*

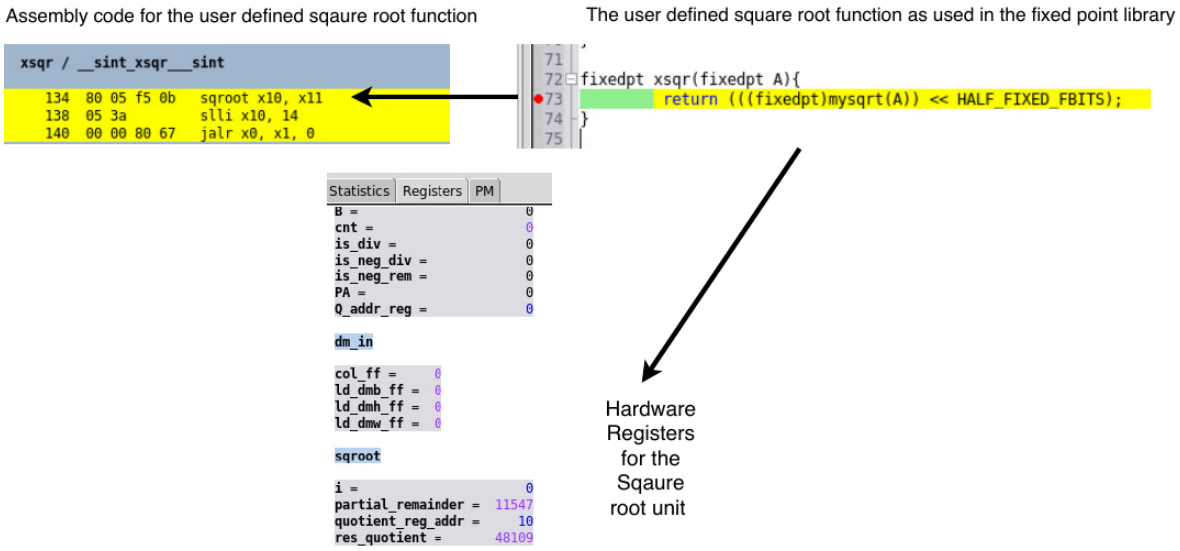


Figure 7.3: Square root unit usage shown with the help of instruction set simulator in Synopsys ASIP designer

Processor	Observed Execution Time(ns)
FLX (floating-point implementation)	902668
Tzscale (floating-point implementation)	2147543
Modified Tzscale (fixed-point implementation)	1022152

Table 7.4: Simulation time for target application execution on FLX, Tzscale and modified Tzscale processor

pm_wr are program memory read bus, address bus and write bus respectively. The signal *radicand_in* is the input register which stores the operand value for one execution period of the square root unit. The output of the square root unit is first written to the local register *res_quotient* which is then copied to the register set of the processor. Additional control signals such as : *sqr_busy* (which shows that the square root unit is busy) have also been shown in this simulation result. The square root unit produces result in exact 17 clock cycles which is the expected execution time as seen from Chapter 6.

In a similar way, the simulations results are verified for the complete execution of the search algorithm on the modified Tzscale processor.

The simulation time for execution of the search algorithm (the respective variant) on the FLX processor, Tzscale processor and modified Tzscale processor for a clock frequency of 250 MHz is presented in Table 7.4.

7.4 Synthesis results

The performance of the modified Tzscale processor has been compared to the original Tzscale implementation and FLX processor in terms of clock cycles required and the instruction count. The three processors are also compared on the basis of the results obtained from

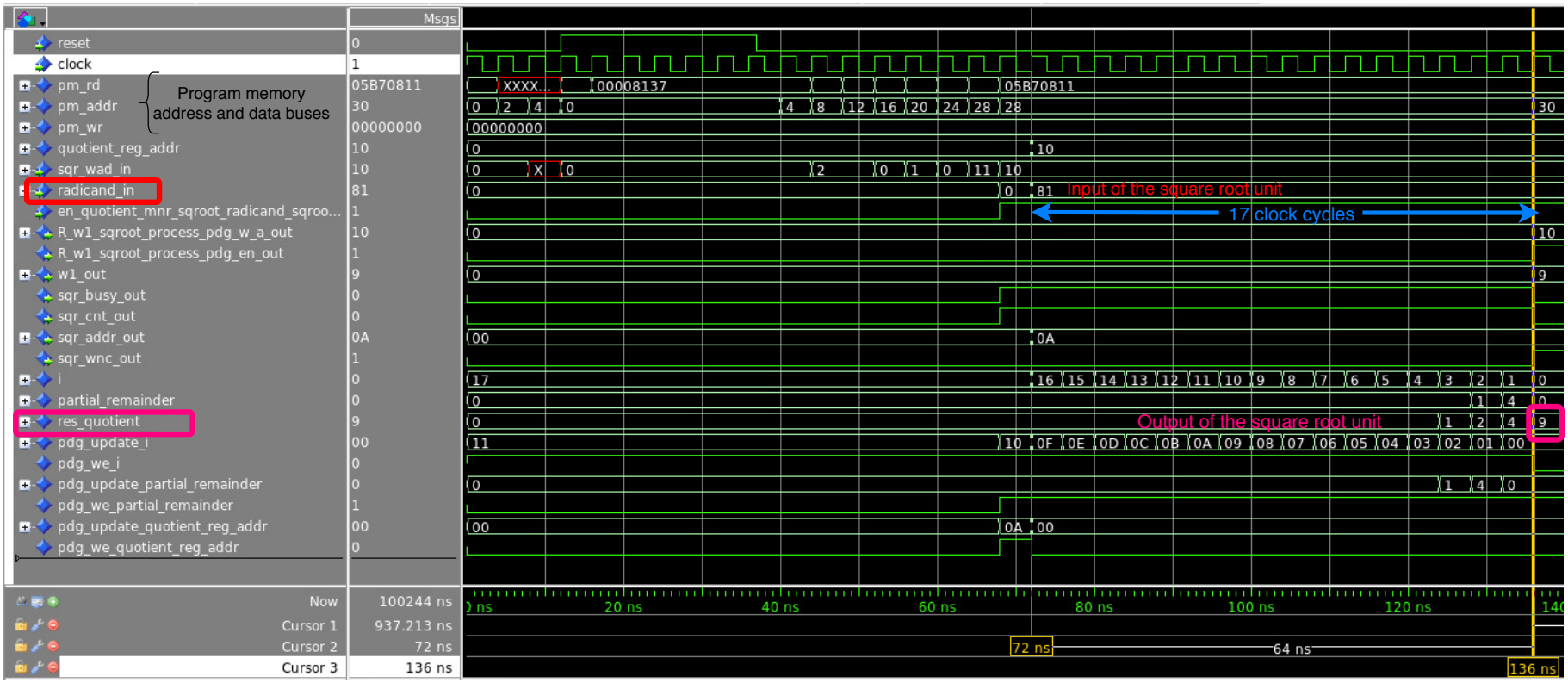


Figure 7.4: Square root unit usage shown with the help of VHDL simulation

Processor	Area (um^2)
FLX	67896.72
Tzscale	45609.12
Modified Tzscale	46763.28

Table 7.5: Area comparison for FLX, Tzcsale and Modified Tzscale processor for UMC 65 nm technology

Processor	Power (mW)
FLX	1.4440
Tzscale	0.5721
Modified Tzscale	0.6029

Table 7.6: 10% toggle rate switching activity power

synthesizing them for the UMC 65 nm technology [1]. All three processors are synthesized for a clock cycle period of 4 ns i.e. a clock frequency of 250 MHz.

The FLX processor is expected to be larger in size compared to the Tzscale processor and its modified version. This is because the FLX processor implements hardware to perform floating point arithmetic. The modified Tzscale processor is 2,53% bigger in area as compared to the Tzscale processor. This increase can be easily attributed to the added square root unit and additional control circuitry. More importantly, FLX processor is 31.13 % larger in size compared to the modified Tzscale processor. Thus, the modified Tzscale processor offers performance which is comparable to the FLX processor while occupying much less area.

The power numbers presented in Table 7.6 are for the 10% toggle rate switching activity performed by the Synopsys Design Compiler. These power numbers for the FLX are higher because of greater area (more hardware involved). The modified Tzscale processor is larger in area compared to the Tzscale processor hence, greater 10% toggle rate switching activity power is obtained. To determine the energy efficiency, the idea was to determine the total amount of energy consumed for the switching activity when the search algorithm runs on each platform. This energy would be the product of the total simulation time for the search algorithm execution multiplied by the power value as produced by Synopsys DC. However, there were issues with the SDF file usage with the synthesised netlist during post-synthesis verification. Hence, an estimate regarding the possible energy value is made.

The simulation time execution of the search algorithm(floating-point) on the Tzscale processor is 52.4% more than the execution time of the search algorithm on the modified Tzscale processor. Based on the 10% switching activity, it can be estimated that calculated power for the modified Tzscale will be more than the calculated power for the Tzscale processor. This is because additional hardware present in the modified Tzscale processor viz. the square root unit and the added control circuitry. The energy consumption which will be a product of the power and time is expected to turn out less for the modified Tzscale design compared to the Tzscale design. This is mainly attributed to the 52.4% reduction in the execution time. This is the best estimate that can be made on energy consumption based on the available parameters.

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

Chapter 8

Conclusion and Future Work

In this chapter, the conclusion and future work is discussed.

8.1 Conclusion

The aim of the ASIP is to conduct an exhaustive search in a given coefficient dictionary to determine optimum coefficient values for the analog beamformer in the hybrid beamforming system.

With a good understanding of the problem statement, having discussed the necessary background, the design methodology and the results it can be concluded *that an ASIP was successfully designed as a part of the baseband processing domain of the hybrid receiver system*.

Here, the research sub-questions framed in Chapter 1 are answered which finally lead to answering of the main research question.

Which open source instruction set architecture can be used as a reference upon which the ASIP can be developed?

In Chapter 3, a thorough analysis of which open source instruction set architecture is best suited as the reference architecture was discussed. Three open source ISAs viz. OpenRISC, UltraSPARC and RISC-V were taken into consideration. Based on parameters such as design flexibility, hardware and software development support, it was established that RISC-V is the best suited choice. The ease of extension of RISC-V and increasing community support were also the contributing factors leading towards this decision. Hence, it was chosen as the reference open source instruction set architecture upon which the ASIP in this research assignment is developed.

Given the insights obtained by profiling of the algorithm on the chosen open source architecture, how can its performance be improved by an ASIP architecture optimized for the task?

Chapter 4 discusses the Tzscale processor which is based on embedded extension of the RISC-V ISA. This processor provides with a good starting point based on which the profiling of the 'C' implementation of the search algorithm is performed. The main research

objective deals with the concept of a performance efficient processor design. The performance efficiency here is analysed on the basis of number of clock cycles required and the total instruction count. This performance efficiency is established in comparison with another processor design viz. FLX processor.

The addition of a custom instruction and the related hardware support (in the form of a square root unit) reduces the instruction count and cycle count required for execution of the search algorithm on the Tzscale processor. This also makes the performance of the modified Tzscale processor comparable to the the chosen comparison design i.e. the FLX processor. Thus, it is through the addition of a new custom square root instruction and the associated hardware square root unit that the performance of the reference architecture i.e. the Tzscale processor is improved. However, this increase improvement in performance comes at the cost of increase in area when comparing the modified Tzscale processor area to the original Tzscale processor (based on the UMC 65nm technology synthesis results).

Shifting the search algorithm implementation from floating-point to fixed-point representation also improves the performance of the processor. Thus, it can be concluded that before adding customizations on the instruction set level, the optimizations should also be checked at the application level through profiling. Once application level implementation has been established as a good starting point, customizations based on the profiling results of this starting point will prove more beneficial.

What design choices should be made while developing the architecture of the ASIP? For instance, How are complex numbers handled?, What must be the depth of the pipeline?, etc.

An ASIP offers the ability to define new data-types based on the application requirements. The requirement in this research assignment was the handling of complex data-types. These data-types were implemented as part of the application code. The profiling results for the modified Tzscale processor indicate that operations such as dot product calculation, complex number multiplication are the new dominant instructions after square root and division. These results indicate that micro-routines which deal with fast implementation of these functions or accelerating these operations through dedicated hardware could be more beneficial than implementing a new-data type. Introducing a new data-type will also bring in other complexities within the system, such as implementation of basic memory elements which can handle such data-types. Hence, it can be concluded that implementing new-data types as a part of the application code is a more sound decision.

From the new profiling results (as presented in Chapter 6) performed on the search algorithm implementation on the modified Tzscale processor it can be seen that operations such as division, dot product calculation, 32-bit floating-point multiplication, etc. form the new set of dominant operations. Most of these operations have an iterative nature which implies an execution period of more than one-clock cycle. In such a scenario, increasing the depth of the pipeline of the processor might not turn out as advantageous as expected. The modified Tzscale processor currently has a shallow pipeline of depth 3. Offloading iterative operations to special hardware units will turn out to be more advantageous compared to experimentation with the depth of the pipeline of the processor. Another important consideration is the complexity of the whole process of pipeline depth increment. Each instruction of the RISC-V embedded instruction set must be modified in the nML model of the processor to accommodate the new pipeline. It can be concluded that a design decision which changes the pipeline depth might not be very useful based on the type of customization requirement and implementation complexity involved.

The power numbers for 10% switching activity, the synthesised area value for the UMC 65nm technology and the target application execution time for the Tzscale and modified Tzscale processor suggests that: the expected total energy consumption for modified Tzscale processor will be less than the Tzscale processor. Although this could not be verified due to issues mentioned in Chapter 7.

The research sub-questions have been answered and this section is concluded with the answer to the final research question.

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

The evidence presented in Chapter 7 and the answers to the previous research sub-questions strongly support that an ASIP with high performance and energy efficiency was designed successfully as the baseband processor of the hybrid receiver system to calculate optimum analog beamformer coefficient values. The RISC-V instruction set architecture is chosen as the reference design upon which the ASIP is developed. The Tzscale processor based on the RISC-V architecture is taken as the skeleton design which is subsequently customized based on the needs of the target application code. The hardware square root unit is added as the customization to the Tzscale processor which improves the efficiency in terms of clock cycles by approximately 50% while the increase in the area usage is approximately 3%. Design decisions were taken at the target application level as well as at the processor model level to ensure high performance and energy efficiency.

8.2 Future work

The processor modeling perform in this research assignment is the first step and there are numerous possibilities for further work which can be built on this starting point. The future work possibilities have been listed as follows:

1. The search algorithm currently calculates the whitened optimum coefficient values for the analog beamformer value. To obtain the exact optimum coefficient values, a negative square root operation of the cross-correlation matrix Rxs needs to be performed. The new design can include the provision to implement the required operation to recover the actual value of the analog beamforming coefficients.

2. The profiling results are based on the function profiling of the search algorithm. It can be investigated whether instruction level profiling is also a possibility within the Synopsys ASIP designer. Instruction level profiling can provide details about which instructions rather than functions are most frequently used. Subsequently, the implementation of these more frequently-used functions could be improved. For example, consider the case where the conditional jump instructions are the most frequent instructions. In such a case, implementing a hardware loop controller could vastly improve the performance of the processor at very little hardware cost.

3. Micro-routines or hardware customizations which target the new set of dominant instructions on the modified Tzscale processor can be implemented.

4. An FPGA prototype can be developed for the modified Tzscale processor which can

then be interfaced with the analog front end and the feasibility of the processor design can be tested in a more practical scenario.

5. A system level model of the hybrid receiver system can be developed. This system level model will consist of the analog beamformer as well as the ASIP. This model can be used to verify that the coefficient values calculated by the ASIP are indeed the correct values based on the beamformer output.

Appendix A

This is an older version of the MATLAB code and a newer version of the report will include the latest MATLAB code Matlab realization of the search algorithm in the form of a MATLAB function is shown below:

```
function W = max_correlation( Rx,rxs,Rw,Nr )

   theta = (2*pi)/(2^Rw);

   c = 0   (2^(Rw-1));

   W_angle = exp(1i * c * theta);

%% creating all possible values in dictionary %%

   w = conj(permn(W_angle,Nr)');

   w_wh = (Rx^0.5) * w;

   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(:,i);
         result = w1;
      end

   end

   W = w_final;

end
```

Appendix B

The C implementation of the search algorithm

```
% fixed point library square root function fixedpt xsqr(fixedpt A){
   return (((fixedpt)mysqrt(A)) << HALF_FIXED_FBITS);
}

% Function which calculates absolute value of a complex number fixedpt abs_comp (struct
complex_no a){
   fixedpt abs_value;
   abs_value = xsqr(complement_mul(a));
   return abs_value; }

% Function which calculates norm of a vector fixedpt norm_of_vector(struct vector *vector,
int size){
   int i;
   fixedpt sum = fixedpt_rconst(0);
   for (i = 0; i ¡ size; i++){
      sum = xadd(sum,complement_mul(vector->ve[i]));
}
   sum = xsqr(sum);
   return sum; }

float min_value = -17179869184; //least possible float value possibly !  float w_1 = 0; %
Exhaustive search for (int i = 0; i < 64; i++ ) {

                w_1 =((abs_comp(vector_mul(&(whitened_matrix.mat[i]), &rxs, 2, 1)))/
                      (norm_of_vector((&(whitened_matrix.mat[i])), 2)));

   if (w_1 > min_value && (w_1 − min_value > 0.000001)) {
      w_final = whitened_matrix.mat[i];
      min_value = w_1;

   }
}
```

Appendix C

Here, the instruction set format for the RISC-V integer base instruction set has been explained in detail.

Base Instruction Formats

In the base ISA, there are four core instruction formats (R/I/S/U), as shown in Figure 8.1 . All instructions are 32 bits in length and must be aligned on a four-byte boundary in memory. An instruction address misaligned exception is generated on a taken branch or unconditional jump if the target address is not four-byte aligned. No instruction fetch misaligned exception is generated for a conditional branch that is not taken. In the Figure 8.1, funct3 and funct7 are used to select a type of operation within an instruction format. For example in R-type operation, funct7="00000" and funct3= "ADD" specific addition operation.

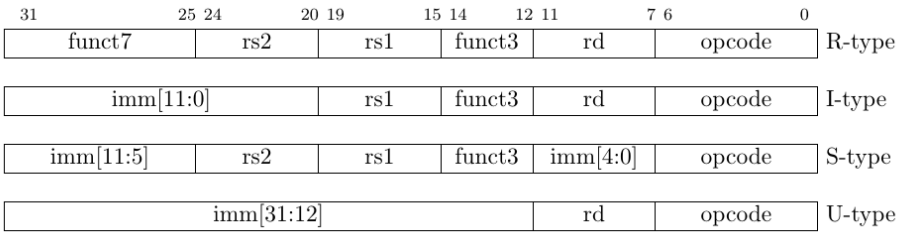


Figure 8.1: RV32I base instruction format [26]

The RISC-V ISA keeps the source (rs1 and rs2) and destination (rd) registers at the same position in all formats to simplify decoding. Except for the 5-bit immediates used in CSR instructions (explained later), immediates are always sign-extended, and are generally packed towards the leftmost available bits in the instruction and have been allocated to reduce hardware complexity. In particular, the sign bit for all immediates is always in bit 31 of the instruction to speed sign-extension circuitry. There are two further variants of the instruction formats (SB/UJ) based on the handling of immmediates. These two variants are shown in Figure 8.2. In Figure 8.2, each immediate subfield is labeled with the bit position (imm[x]) is the immediate value being produced, rather than the bit position in the instruction's immediate field. The only difference between the S and B (or SB) formats is that the 12-bit immediate field is used to encode branch offsets in multiples of 2 in the B format. Instead of shifting all bits in the instruction-encoded immediate left by one in hardware as is conventionally done, the middle bits (imm[10:1]) and sign bit stay in fixed positions, while

the lowest bit in S format (inst[7]) encodes a high-order bit in B format. Similarly, the only difference between the U and J (or UJ) formats is that the 20-bit immediate is shifted left by 12 bits to form U immediates and by 1 bit to form J immediates. The location of instruction bits in the U and J format immediates is chosen to maximize overlap with the other formats and with each other.

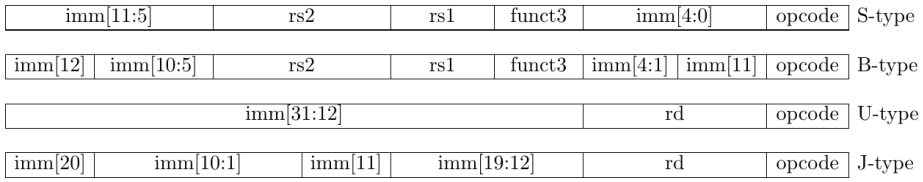


Figure 8.2: RV32I base instruction format showing the immediate variants [26]

Integer Computational Instructions

Integer computational instructions are either encoded as register-immediate operations using the I-type format or as register-register operations using the R-type format. No integer computational instructions cause arithmetic exceptions. The integer register-immediate instructions in RV32I are : ADDI, SLTI, SLTIU, ANDI, ORI, XORI, SLLI, SRLI, SRAI, LUI, AUIPC. Apart from LUI and AUPIC all the integer immediate instructions are encoded in I-type format. The LUI and AUPIC instructions are encoded in the U-type format. The integer register-register instructions are : ADD, SUB, SLT, SLTU, ADN, OR, XOR, SLL, SRL, SRA. The NOP instruction along with the integer register-register instructions is encoded as a R-type instruction. The NOP instruction does not change any user visible state except for advancing the pc.

Control Transfer Instructions

RV32I provides two types of control transfer instructions: unconditional jumps and conditional jumps. *Control Instructions in RV32I do not have architecturally visible delay slots*. The unconditional jump instructions are JAL and JALR. The JAL instruction uses the UJ type encoding. These instructions can generate a misaligned instruction fetch exception if the target address is not aligned. The conditional branch instructions are BEQ, BNE, BLT, BLTE, BGE, BGEU. All conditional branch instructions are encoded using the SB-type format.

The conditional branches have been designed to include arithmetic comparison operations between two registers (as also done in PA-RISC [37] and Xtensa ISA [38]), rather than use condition codes (x86 [39], ARM [40], SPARC [25], PowerPC [41]), or to only compare one register against zero (Alpha [42], MIPS [43]), or two registers only for equality (MIPS). The authors of RISC-V [26] have motivated this design decision by stating that: a combined compare-and-branch instruction fits into a regular pipeline, avoids additional condition code state or use of a temporary register, and reduces static code size and dynamic instruction fetch traffic. Also, comparisons against zero require non-trivial circuit delay (especially after

the move to static logic in advanced processes) and so are almost as expensive as arithmetic magnitude compares. Another advantage of a fused compare-and-branch instruction is that branches are observed earlier in the front-end instruction stream, and so can be predicted earlier (if branch prediction mechanisms are supported). The advantage to a design with condition codes is when multiple branches can be taken based on the same condition codes, but the authors consider this case fairly rare and hence fused comparison-arithmetic instructions have been implemented as a part of the ISA.

Load and Store Instructions

RV32I is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers. RV32I provides a 32-bit user address space that is byte-addressed and little-endian. The execution environment will define what portions of the address space are legal to access. The load instructions are encoded using the I-type format; they are as follows: LW, LH, LHU, LB, LBU. The store instructions are encoded using the S-type format; they are as follows: SW, SH, SB.
Note: For best performance, the effective address for all loads and stores should be naturally aligned for each data type (i.e., on a four-byte boundary for 32-bit accesses, and a two-byte boundary for 16-bit accesses). The base ISA supports misaligned accesses, but these might run extremely slowly depending on the implementation. Furthermore, naturally aligned loads and stores are guaranteed to execute atomically, whereas misaligned loads and stores might not, and hence require additional synchronization to ensure atomicity.

The base RISC-V ISA supports multiple concurrent threads of execution within a single user address space. Each RISC-V thread has its own user register state and program counters, and executes an independent sequential instruction stream. In the base RISC-V ISA, each RISC-V thread observes its own memory operations as if they are executed sequentially in program order. RISC-V has a relaxed memory model between threads, requiring an explicit FENCE instruction to guarantee any specific ordering between memory operations from different RISC-V threads. The FENCE instruction is used to order device I/O and memory accesses as viewed by other RISC-V threads, external devices or co-processors. The FENCE.I instruction is used to synchronize the instruction and data streams.

Control and Status Register Instructions

System instructions are used to access system functionality that might require privileged access and are encoded using the I-type instruction format. These can be divided into two main classes: those that atomically read-modify-write Control and Status Registers (CSRs), and all other potentially privileged [1] instructions. The full set of CSR instructions are : CSRRW, CSRRS, CSRRC, CSRRWI, CSRRI and CSRRCI. In the standard user-level base ISA, only a handful of read only counter CSRs are accessible.

[1] Privileged instruction is an instruction that can be executed only by an operating system in a specific mode (generally the Kernel mode)

RV32I provides a number of 64-bit read-only user-level counters, which are mapped into 12-bit CSR address space and accessed in 32-bit pieces using CSRRS instructions. They are as follows: RDCYCLE, RDTIME, RDINSTRET, RDCYCLEH. The RDCYCLEH is a RV32I only instruction that reads bits 63-32 of the same cycle counter. Similarly, RDTIMEH is an RV32I only instruction that reads 63-32 of a real time counter. Along with this RDINSTRETH is also an RV32I only instruction that reads 63-32 of the an instruction counter.

These basic counters are essential for basic performance analysis, adaptive and dynamic optimization, and to allow an application to work with real-time streams. These counters are kept 64 bit even for 32-bit address width implementations.

Environment Call and Breakpoints

There are two more instructions viz. ECALL and EBREAK instructions which complete the RV32I instruction set. These instructions are used to interact with the supporting execution or debugging environment. The instruction encoding is done using I-type format; the immediate field is replaced by funct12 which is to distinguish between the two operations.

Appendix D

The FLX processor is a RISC based processor architecture with the following features:

- 32-bit wide datapath, with an ALU, shifter and multiplier unit.
- 32-bit wide instruction word with orthogonal instruction encoding
- load/store architecture, which supports 8,16 and 32 bit memory transfers and an indexed addressing mode.
- Follows the conventional RISC 5 stage pipeline (IF-ID-EX-MA-WB).
- Supports a multi-cycle iterative division unit.
- Most importantly supports IEEE single precision floating point unit
- The central register file consists of 32 registers.

The FLX processor is chosen as a design against which the performance of the Tzscale processor can be compared due to the presence of the floating point unit. At the same time, both processor have a 32-bit datapath which ensures fair comparison. Additionally, this design is available as an example processor model along with the Synopsys ASIP designer. This allows the comparison to performed at the instruction set as well as the synthesis level (RTL generation and subsequent synthesis is possible for desired technology).

Appendix E

The synthesis-in-loop architectural exploration techniques explained in Chapter 4 shows that the Synopsys ASIP designer can be used to generate the RTL model of the processor. This RTL generation is achieved with the help of a GO tool which is provided as a part of the Synopsys ASIP designer. To generate the RTL from the nML model, certain configuration parameters need to be passed to the GO tool. In the Synopsys ASIP designer, this is done with the help of a GO configuration file. An example of the GO configuration file is shown in Figure 8.3.

go_options.cfg

```
#include "tzscale_define.h"

//-------------------------------------------------------
//By default VHDL description is generated.
// 3.1 General options

//includes the lib directory in the path to search for the processor descriptions.
include_directory: "../lib";
//this option generates the HDL testbench
generate_testbench;
//makes sure that primitive operations which write to the same transitories are mapped to the same functional unit, usage of for rg @alu becomes unnecessary.
assign_primitive_operations;

//-------------------------------------------------------
// 3.2 Merging of files and units

//identical entities are searched and merged into the same entity.
merge_identical_entities;
//Each package declaration is put in the same file as the corresponding package body.
merge_package_body_files;
//Each entity and architecture is put in the same file.
merge_entity_architecture_files;

//-------------------------------------------------------
// 3.4 HDL configuration

//The generated HDL is annotated with comments. The higher the number for the annotation level the more are the comments added.
annotation_level: 100;
//only for verilog, sets the timescale for verilog simulation
timescale: "1ns/1ps";
//models memory as synchronous memory. Next value of program counter is already latched at teh address bus of program memory.
synchronous_program_memory;
//The address must be specifically written in the controller
program_memory_address_from_controller;
//Multiplexers with only single input are expanded as a concurrent assignment in the top level net list of processor entity
expand_trivial_multiplexers;
//literal contstants are generated locally--> avoids input ports otherwise needed to feed those constants
local_literal_constants;
//Go ignores the hw_init value of a transitory--> is it even needed
local_hw_init;
//pdg generates the vhdl implementation of primitives
pdg;
//constant definitions are written in the file itself instead of a separate package
selector_constants: 2;
//naming of output ports is done from transitories and not their copies
original_port_names;
//generates extra hdl code to log register writes during simulation. This extra code has no influences on the synthesis
log_register_writes;
//same as logging of register writes.
log_memory_writes;
//no configuration files are generated. Generic values are given a default value.
configuration_files: 0;
```

Figure 8.3: Sample Go configuration file

The configuration parameter such as “generate_testbench” is used to direct the GO tool to generate the testbench for the processor model. This testbench can then be used to perform simulations in a simulation tool like Modelsim. There are numerous HDL configuration parameters such as “annotation_level”, synchronous_program_memory, etc. which the

designer is free to decide based on the requirements from the RTL model. The entire list of parameters can be found in “go-manual.pdf” in the documentation of the Synopsys ASIP designer.

Once, all the parameters have been decided and the configuration file has been written the GO tool can be used to generate the RTL model of the processor.

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] Xinying Zhang, A. F. Molisch, and Sun-Yuan Kung, "Variable-phase-shift-based rf-baseband codesign for mimo antenna selection," *IEEE Transactions on Signal Processing*, vol. 53, no. 11, pp. 4091–4103, Nov 2005.

[3] P. Sudarshan, N. B. Mehta, A. F. Molisch, and J. Zhang, "Channel statistics-based rf pre-processing with antenna selection," *IEEE Transactions on Wireless Communications*, vol. 5, no. 12, pp. 3501–3511, December 2006.

[4] "MIMO communication systems," https://www.edgefx.in/multiple-input-and-multiple-output-mimo-wireless-communications/, Accessed: 2019-05-15.

[5] A. Krumbein, "MIMO basics," https://www.rfmw.com/data/SWA-MIMO-Basics.pdf, Accessed: 2019-05-15.

[6] C. Shekhar, Raj Singh, A. S. Mandal, S. C. Bose, R. Saini, and P. Tanwar, "Application specific instruction set processors: redefining hardware-software boundary," in *17th International Conference on VLSI Design. Proceedings.*, Jan 2004, pp. 915–918.

[7] F. Conti, D. Rossi, A. Pullini, I. Loi, and L. Benini, "Pulp: A ultra-low power parallel accelerator for energy-efficient and flexible embedded vision," *Journal of Signal Processing Systems*, vol. 84, no. 3, pp. 339–354, Sep 2016. [Online]. Available: https://doi.org/10.1007/s11265-015-1070-9

[8] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang, M. Matl, and D. Wentzlaff, "Openpiton: An open source manycore research framework," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 217–232. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872414

[9] "Synopsys ASIP Designer," https://www.synopsys.com/dw/ipdir.php?ds=asip-designer, Accessed: 2019-05-03.

[10] Taedong Shin, Gibum Kim, Hyuncheol Park, and H. M. Kwon, "Quantization error reduction scheme for hybrid beamforming," in *2012 18th Asia-Pacific Conference on Communications (APCC)*, Oct 2012, pp. 243–247.

[11] G. Zhu, K. Huang, V. K. N. Lau, B. Xia, X. Li, and S. Zhang, "Hybrid beamforming via the kronecker decomposition for the millimeter-wave massive mimo systems," *IEEE Journal on Selected Areas in Communications*, vol. 35, no. 9, pp. 2097–2114, Sep. 2017.

[12] T. Brown, E. D. Carvalho, and P. Kyritsi, *Practical Guide to MIMO Radio Channel with MATLAB ® Examples*. Wiley, 2012.

[13] "Optimal beamforming," https://www.comm.utoronto.ca/~rsadve/Notes/BeamForming.pdf, Accessed: 2019-05-06.

[14] I. Ahmed, H. Khammari, A. Shahid, A. Musa, K. S. Kim, E. De Poorter, and I. Moerman, "A survey on hybrid beamforming techniques in 5g: Architecture and system model perspectives," *IEEE Communications Surveys Tutorials*, vol. 20, no. 4, pp. 3060–3097, Fourthquarter 2018.

[15] X. Huang, Y. J. Guo, and J. D. Bunton, "A hybrid adaptive antenna array," *IEEE Transactions on Wireless Communications*, vol. 9, no. 5, pp. 1770–1779, May 2010.

[16] J. Nsenga, A. Bourdoux, and F. Horlin, "Mixed analog/digital beamforming for 60 ghz mimo frequency selective channels," in *2010 IEEE International Conference on Communications*, May 2010, pp. 1–6.

[17] 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.

[18] 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.

[19] Y. Yokota, S. Yoshizawa, and H. Ochi, "Asip implementation of a low complexity iterative bd precoder for mu-mimo system," in *2015 15th International Symposium on Communications and Information Technologies (ISCIT)*, Oct 2015, pp. 277–280.

[20] S. Shahabuddin, O. Silvn, and M. Juntti, "Asip design for multiuser mimo broadcast precoding," in *2017 European Conference on Networks and Communications (EuCNC)*, June 2017, pp. 1–4.

[21] 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.

[22] X. Chen, A. Minwegen, S. B. Hussain, A. Chattopadhyay, G. Ascheid, and R. Leupers, "Flexible, efficient multimode mimo detection by using reconfigurable asip," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 10, pp. 2173–2186, Oct 2015.

[23] 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.

[24] "OpenRISC 1000," https://raw.githubusercontent.com/openrisc/doc/master/openrisc-arch-1.2-rev0.pdf, Accessed: 2019-04-25.

[25] "OpenSPARC Specification," https://www.oracle.com/technetwork/systems/opensparc/opensparc-internals-book-1500271.pdf, Accessed: 2019-04-25.

[26] "Risc V Specification," https://riscv.org/specifications/, Accessed: 2019-04-25.

[27] "Coremark Benchmark," https://www.eembc.org/coremark/, Accessed: 2019-06-03.

[28] Y. Lee, "Z scale 32 bit risc v microcontroller," https://riscv.org/wp-content/uploads/2015/06/riscv-zscale-workshop-june2015.pdf, Accessed: 2019-06-15.

[29] K. Küçükçakar, "An asip design methodology for embedded systems," in *Proceedings of the Seventh International Workshop on Hardware/Software Codesign*, ser. CODES '99. New York, NY, USA: ACM, 1999, pp. 17–21. [Online]. Available: http://doi.acm.org/10.1145/301177.301190

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

[31] A. Hosseiny and G. Jaberipur, "Decimal square root: Algorithm and hardware implementation," *Circuits Syst. Signal Process.*, vol. 35, no. 12, pp. 4195–4219, Dec. 2016. [Online]. Available: http://dx.doi.org/10.1007/s00034-015-0215-1

[32] J. H. P. Zurawski and J. B. Gosling, "Design of a hih-speed square root multiply and divide unit," *IEEE Transactions on Computers*, vol. C-36, no. 1, pp. 13–23, Jan 1987.

[33] K. Piromsopa, C. Aporntewan, and P. Chogsatitvataa, "An fpga implementation of a fixed-point square root operation," 09 2002.

[34] S. Samavi, A. Sadrabadi, and A. Fanian, "Modular array structure for non-restoring square root circuit," *J. Syst. Archit.*, vol. 54, no. 10, pp. 957–966, Oct. 2008. [Online]. Available: http://dx.doi.org/10.1016/j.sysarc.2008.04.004

[35] R. Hashemian, "Square rooting algorithms for integer and floating-point numbers," *IEEE Transactions on Computers*, vol. 39, no. 8, pp. 1025–1029, Aug 1990.

[36] R. V. W. Putra, "A novel fixed-point square root algorithm and its digital hardware design," in *International Conference on ICT for Smart Society*, June 2013, pp. 1–4.

[37] "Pa risc," https://en.wikipedia.org/wiki/PA-RISC, Accessed: 2019-04-20.

[38] "Xtensa ISA," https://0x04.net/~mwk/doc/xtensa.pdf, Accessed: 2019-04-22.

[39] "x86 isa," https://en.wikipedia.org/wiki/X86, Accessed: 2019-04-20.

[40] "Arm isa," https://en.wikipedia.org/wiki/ARM_architecture, Accessed: 2019-04-22.

[41] "PowerPC isa," http://math-atlas.sourceforge.net/devel/assembly/ppc_isa.pdf, Accessed: 2019-04-22.

[42] "Alpha isa," https://en.wikipedia.org/wiki/X86, Accessed: 2019-04-20.

[43] "Mips isa," https://en.wikipedia.org/wiki/X86, Accessed: 2019-04-20.