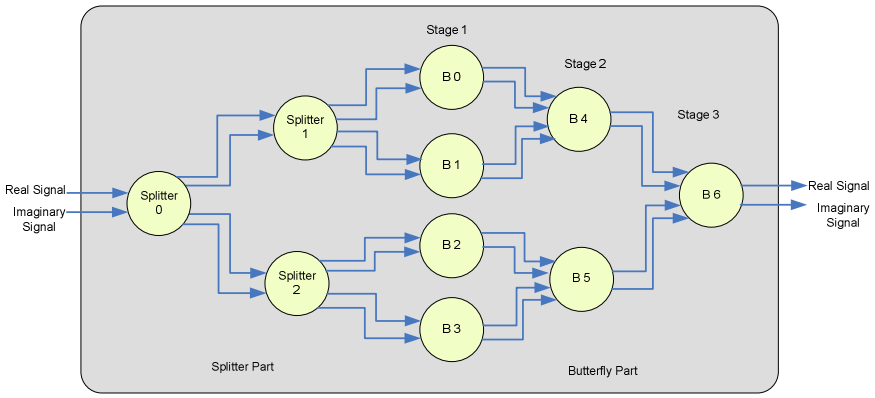
Technical report, IDE0838, June 2008

Signal Processing on Ambric Processor Array: Baseband processing in radio base stations

Master's Thesis in Computer Systems Engineering

Chaudhry Majid Ali, Muhammad Qasim



School of Information Science, Computer and Electrical Engineering
Halmstad University

Signal Processing on Ambric Processor Array: Baseband processing in radio base stations

Master's Thesis in Computer Systems Engineering

School of Information Science, Computer and Electrical Engineering
Halmstad University
Box 832, S-301 18 Halmstad, Sweden

June 2008

Description of cover page picture: A composite object computing 8-Point FFT.

Preface

This Master's thesis is the concluding part of the Master's Program in Computer Systems Engineering Specialization in Embedded Systems at Halmstad University. This project has been carried out as a co-operation between Halmstad University and Ericsson AB. In this project the Ambric processor developed by Ambric Inc. has been used, we would like to thanks Mike Butts for supplying development tools and support. We would also like to thanks our supervisors Professor Bertil Svensson, Zain-ul-Abdin and Per Söderstam from Halmstad University and David Engdal from Ericsson AB for guidance and support throughout this thesis project.

---------- ----------

Chaudhry Majid Ali Muhammad Qasim

Halmstad University, June 2008

Abstract

The advanced signal processing systems of today require extreme data throughput and low power consumption. The only way to accomplish this is to use parallel processor architecture.

The aim of this thesis was to evaluate the use of parallel processor architecture in baseband signal processing. This has been done by implementing three demanding algorithms in LTE on Ambric Am2000 family Massively Parallel Processor Array (MPPA). The Ambric chip is evaluated in terms of computational performance, efficiency of the development tools, algorithm and I/O mapping.

Implementations of Matrix Multiplication, FFT and Block Interleaver were performed. The implementation of algorithms shows that high level of parallelism can be achieved in MPPA especially on complex algorithms like FFT and Matrix multiplication. Different mappings of the algorithms are compared to see which best fit the architecture.

List of Figures

Table of Contents

1 Introduction

Modern communication is growing and system demands also increase. All modern communication systems are using digital signal processing techniques. Digital signal processing is the analysis, interpretation, and manipulation of signals. Signal processing is used in a lot of fields like sound, images, biological signals, radar signals and many others. Processing of such signals includes filtering, storage and reconstruction, separation of information from noise, compression, and feature extraction. Benefits of digital signal processing include increased throughput, reduced bit error rate, and greater stability over temperature and process variation.

Baseband signal specification plays a key role both in selecting the appropriate system architecture and determining the necessary computational speed of all involved algorithms. In the Fourier Domain, a baseband signal is a signal that occupies the frequency range from 0Hz up to a certain cutoff. It is called the baseband because it occupies the lowest range of the spectrum.

Academic world and industry are already looking for means to improve the system performance further. Challenges of baseband signal processing is to provide high bandwidth, spectrum efficiency, higher data rate, reduced device complexity, reduced latency, peak data throughput and flexible channel bandwidth. Some other important issues in the field of baseband signal processing are regularity, power consumption and battery life. Regularity means that the system should provide fast response time because a slower response time can prevent to adopt new technology. To target these challenges new standards are needed to be introduced in the field of communication.

Long Term Evolution (LTE) is one of these most recent standards and it continues to evolve to meet operator requirements to face future challenges. It should address all these targets in a timely and economical manner. LTE is designed to provide peak data rate, high degree of mobility and wide coverage, it must also support variable transmission bandwidths.

The algorithms which are used in signal processing are very complex. They require high computational power, so the need for parallel processing has always existed in signal processing systems. Many Digital Signal Processing (DSP) systems have been developed which incorporated some form of parallelism in order to achieve a required level of performance.

Traditionally embedded computing tasks have been handled by DSPs as well as Field Programmable Gate Arrays (FPGAs). These provide two good alternatives to embedded microprocessors in terms of power, speed and configurability. However, they are still only two regions in a large multi-dimensional space. FPGAs provide low-level fine-grained parallelism with a degree of performance achievable through multiple levels of parallelism. DSPs have higher clock speeds and have lower power requirements, but lack the configurability of an FPGA. Still, there are regions in the design space that can be explored that fit between DSPs and FPGAs. These are Coarse-Grained Reconfigurable Arrays (CGRAs). CGRAs have several advantages over traditional DSP and FPGA approaches [14].

First, most strive for ease of application development, such as a high-level language (HLL) approach for their tools. For example, rather than constructing a design in a hardware description language such as VHDL or Verilog, several of these described platforms use HLLs such as C or C++ [14].

Second, CGRAs are able to achieve higher performance and lower power per operation when compared to FPGAs and DSPs. CGRAs accomplish this through higher levels of raw parallelism as compared to DSPs and more efficient use of silicon per operation as compared to FPGAs. CGRA systems are multi-core systems that fit into two different categories: processor-centric arrays or clusters, and hardware-centric medium-grained processor arrays (MGPA). Processor-centric arrays or clusters are made up of individual processors connected via programmable interconnect such as MIT's RAW, Clearspeed's CSX600, IBM's Cell Broadband Engine and Ambric's Am2000 reconfigurable processing array (RPA) [14].

Over the recent years, several parallel processing architectures have been introduced in order to fulfil high computational application needs, in more and more efficient way. The capabilities of the available technologies such as Ambric are at a level which can support such a parallelism. This thesis attempts to highlight the relevance of parallel processing developments to accomplish signal processing requirements.

This thesis will examine the use of parallel processor architecture in baseband signal processing. This has been done by implementing three demanding algorithms in LTE on Ambric Am2000 family Massively Parallel Processor Array (MPPA). The Ambric chip is evaluated in terms of computational performance, efficiency of the development tools, algorithm and I/O mapping and analyzed that whether energy consumption can be tuned to performance needs.

Implementations of matrix multiplication, FFT and block interleaver were performed. The implementation of algorithms shows that high level of parallelism can be achieved in MPPA especially on complex algorithms like FFT and Matrix multiplication.

The thesis report is organized in several chapters. Ambric Am2000 MPPA is presented in Chapter 2. The LTE standard for mobile communication is presented in Chapter 3. Selected algorithms overview from LTE is presented in Chapter 4. Practical work done within this thesis, including API implementation and detailed description of the algorithm implementation and mapping on the Ambric architecture, is presented in Chapter 5. The results from the implemented algorithms are presented in terms of cycle counts, counts of processor stalls, and speedup for different sizes of input data; this is done in Chapter 6. A summary and interpretation of the results achieved in the simulation, as well as experience gained from working with new processor architecture (Ambric) is presented in Chapter 7, and at the end there are appendices with relevant pieces of code.

2 Ambric MPPA

Ambric first focused on the right programming model and then they invented new hardware architecture and circuit designs to facilitate this programming model, providing massively parallel embedded computing. Ambric chip belongs to the category of MPPA and provides a practical implementation of MIMD programming model.

2.1 Chip Architecture

Ambric chip contains a 5x9 array of brics. Brics within Ambric chip and their interconnection are shown in Figure 1. Ambric chip has 4 GPIO (general purpose input and output) for reading and writing input and output streams, two DDR2 SDRAM (double-data rate 2 synchronous dynamic RAM) interfaces to external memories, a Flash interface for the run-time configuration, a PCIe (Peripheral Component Interconnect express) to host or PC.

An Ambric chip uses two types of processors Streaming RISC with DSP extension (SRD) and Streaming RISC (SR), as shown in Figure 2. A cluster of four processors is a compute unit. The main functionality of SR processor is the administration of channel traffic, producing complex address streams and other service tasks which consistently demonstrate high throughput for SRDs.

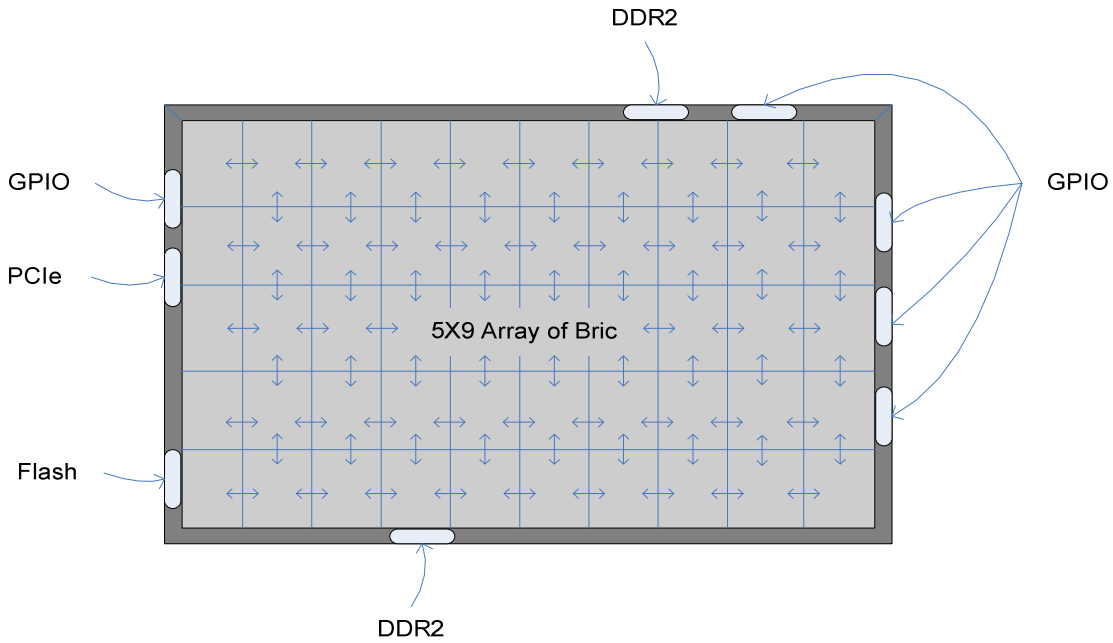


Figure 1: Ambric chip

SR processor is a 32 bit streaming RISC control processing unit. SRD is high performance processor. It is a 32-bit streaming RISC control processing unit with digital signal processing extensions. This processor is used for mathematically demanding processing. It has confined memory for 32 bit instruction and can execute further code from RAM unit directly.

2.2 Brics and Interconnects

Brics and Interconnects is another part of the Ambric chip architecture. A bric is the basic building-block that is replicated to make a core. Each bric has two compute units (CU) and two RAM units (RU), totalling 8 CPUs and 13 KB of SRAM. RUs and CUs are pooled on top stage in a physical structure block called a bric. Ambric chip is consisting of array of brics. Computational power can be determined by the number of brics present in system. The array of brics are ad*join*ing control units and ad*join*ing RAM units.

Brics are connected through channels. The physical architecture is highly scalable. These channels are word wide and run at up to 10 Gigabits per second. Figure 2 shows how CUs, RUs, and their interconnects are placed in a single bric.

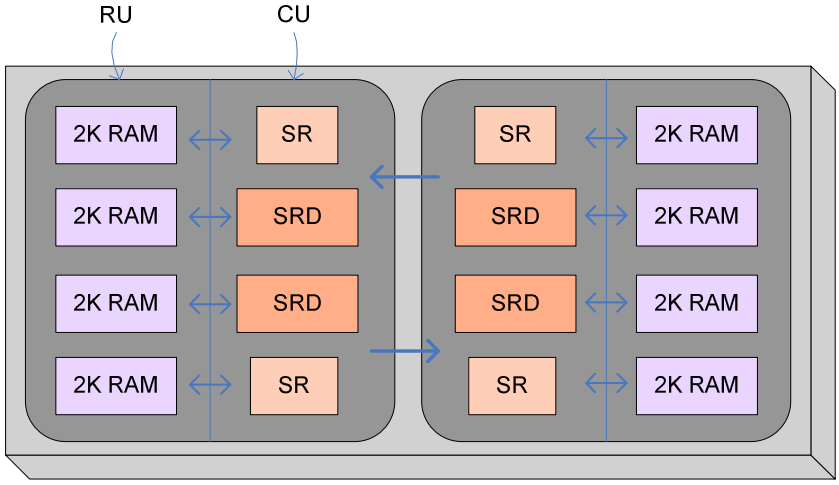


Figure 2: A Brick and Interconnects

Ambric core contains 45 brics arranged in array-like fashion. It has in total 336 SR and SRD processors, 7.1 Mega bits of distributed SRAM and running on 350 MHz. If all processors work collectively then they are able to perform 1.2 trillion operations per second. This performance is supported by the interconnect's 792 Gbps bisection bandwidth, 26 Gbps of off-chip DDR2 memory, PCI Express at an effective 8 Gbps each way, and up to 13 Gbps of parallel general-purpose I/O. [2]

2.3 Processor Architecture

Ambric processor architecture is designed to perform data processing and control through channels. The processor architecture is depicted in Figure 3. The read and write operation in memory is performed through channels, similarly instructions are also passed through channels which make channel communication a prominent feature of Ambric architecture.

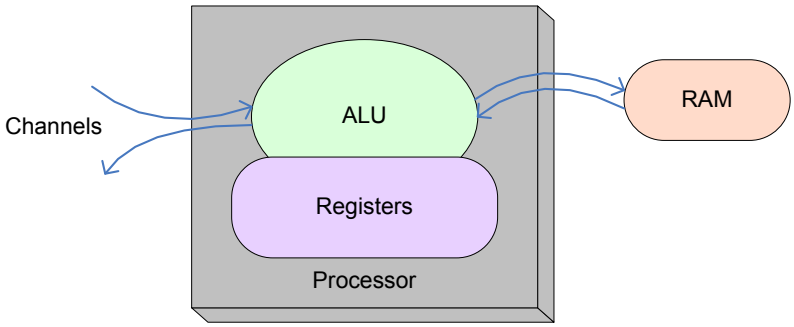


Figure 3: Processor Architecture

Ambric processor is very lightweight 32-bit streaming RISC CPUs. In this architecture RAM is mostly used for buffering rather than a global memory. Since Ambric uses hundreds of processors to perform computing in parallel, it is very important to have simple, efficient and fast implementation of instruction set to take advantage of instruction-level parallelism.

In this architecture, every data path is a self-synchronizing Ambric channel, which makes pipeline control easy. Memory locations are composed of general registers instead of Ambric registers, since they can be read and overwritten at anytime.

2.4 Ambric Registers and Channels

A chain of Ambric registers is called Ambric channel as shown in Figure 4. These channels are fully scalable and encapsulated for passing control and data between objects. In the place of ordinary synchronous registers, Ambric registers are used throughout the chip. The stages of channel are fully local, there is no wire longer than one stage. If we change the length of wire then there is no effect on the functionality but it will affect latency only. If the channel length increases then latency increases and if channel length decreases then latency also decreases.

Ambric register has data in and data out. It also has two control signals known as *valid* and *accept*. These control signals make the Ambric register self-synchronized and asynchronous. When a register can accept input it asserts its *accept signal* upstream and when it has output available it asserts its *valid signal* downstream. When both signals are true then transfer has taken place independently without any negotiation or acknowledgment.

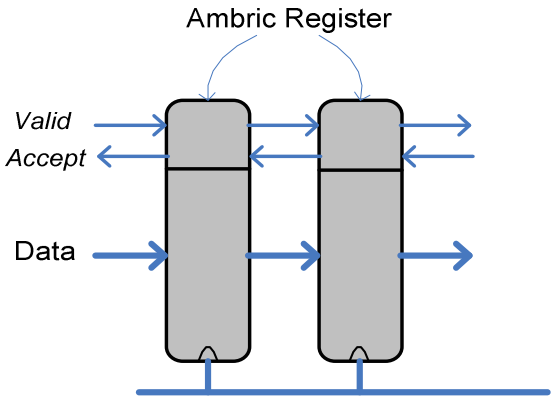


Figure 4: Ambric Channel and Registers

Processors are interconnected in hardware through Ambric channels. Every processor runs independently. It only responds to the own local channels which is why local changes have local effects. We can change clock speed for each processor separately at runtime because it is a globally asynchronous and locally synchronous architecture.

2.5 Structural Object Programming Model

The structural object programming model (SOPM) describes how Ambric chip is programmed. Figure 5 shows the basic elements and their structure in the programming model.

The Ambric chip consists of an array of hundreds of 32-bit RISC processors that can be programmed with ordinary software, and hundreds of distributed memories. These are known as Objects because they follow the strict encapsulation rule. Objects do not depend on each other. They run independently on their own hardware. Structural objects have no implicitly shared memory so there is no side effect to other objects.

The channels act as FIFO buffers. Channels are used for the communication. Objects communicate through a parallel structure of hardware channels. Data and control tokens send through channels. Objects are synchronized through hardware channels at each end, not scheduled at compile time but dynamically used as needed at run time.

Inter-processor communication and synchronization are combined in these channels. Sending a word through a channel is both a communication and synchronization event. Ambric channels synchronize transparently and locally, so that application can achieve high performance without complex global synchronization.

Design re-use is practical and robust in this system. Channels provide a common hardware level interface for all objects. It is easy for the objects to be assembled into higher-level composite objects, the same way as the leaf objects since objects are encapsulated and only interact through channels.

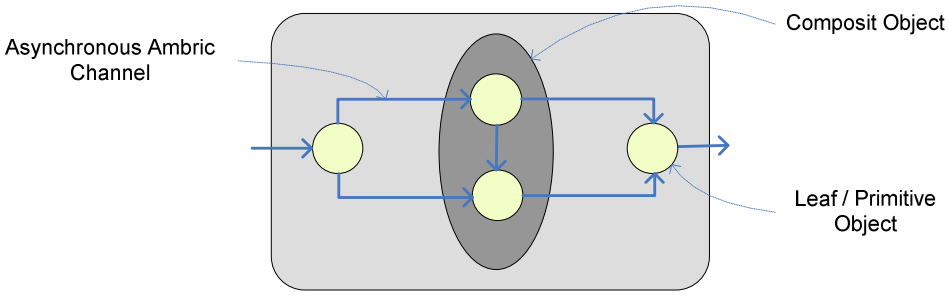


Figure 5: Programming Model

Programming is a combination of familiar techniques. Application developers develop block diagrams to express object-level parallelism. First a hierarchical structure of composite and primitive objects is defined, connected by channels that carry structured data and control messages. Then ordinary sequential software is written to implement the primitive objects [1].

2.6 Application Structure

Here we shall describe briefly about application structure in aDesigner. Ambric developer's environment provides two types of programming languages, one is *astruct* and the other is *ajava*. The *ajava* is a subset of java language and is used to program objects in terms of java classes, or objects can also be written in assembly language. While *astruct* language is used to create the design of application in terms of interfaces and channels, then these interfaces are connected each other with the help of channels. In this way, application design and actual algorithm implementation will remain separate. The *astruct* also provides a programming construct called binding through which java classes are bounded with the interfaces. Each object will run independently on a single processor.

2.7 Development Tools

We have discussed programming model earlier. In this section we will provide an overview of aDesigner, an integrated development environment. aDesigner is based on Eclipse IDE, an open development platform developed by Eclipse foundation. Objects can be written in standard assembly or subset of Java language or loaded from libraries. We can define the structure in text based structural language or graphical block diagrams. Figure 6 shows the graphical block diagram and Figure 7 shows text based structural language views from aDesigner.

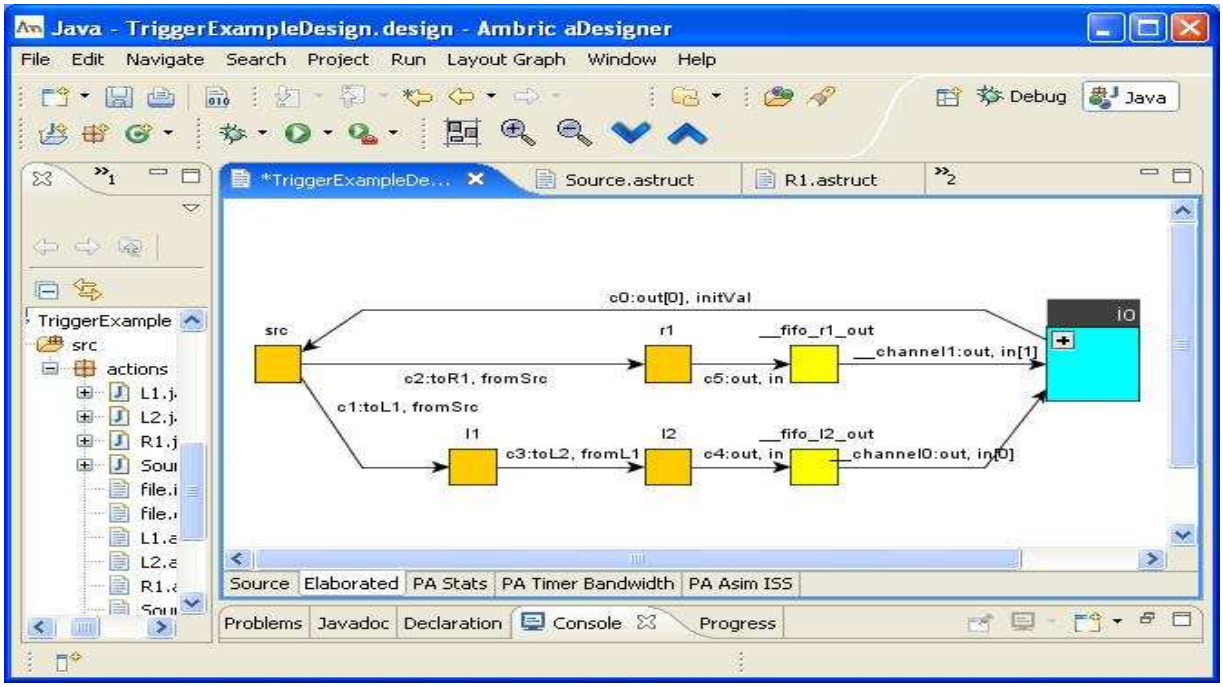


Figure 6: Graphical Diagram View

Real-time hardware debugging uses the vacant processors, memories and channels. There is also available a dedicated debug network. Programmers can use standard debug features like halt, step in, restart a processor and can view channel values and events.

The aDesigner IDE also provides behavioural simulator (aSim) and functional simulator. The behavioural simulator, aSim, is used to simulate the execution and debugging applications on a workstation without having a chip [1].

Functional simulator is used to run and perform initial software debugging. Ambric developer board creates a configuration file for the application deployment. A host can configure the chip on runtime or it may be configured by itself from flash just like FPGA configuration.

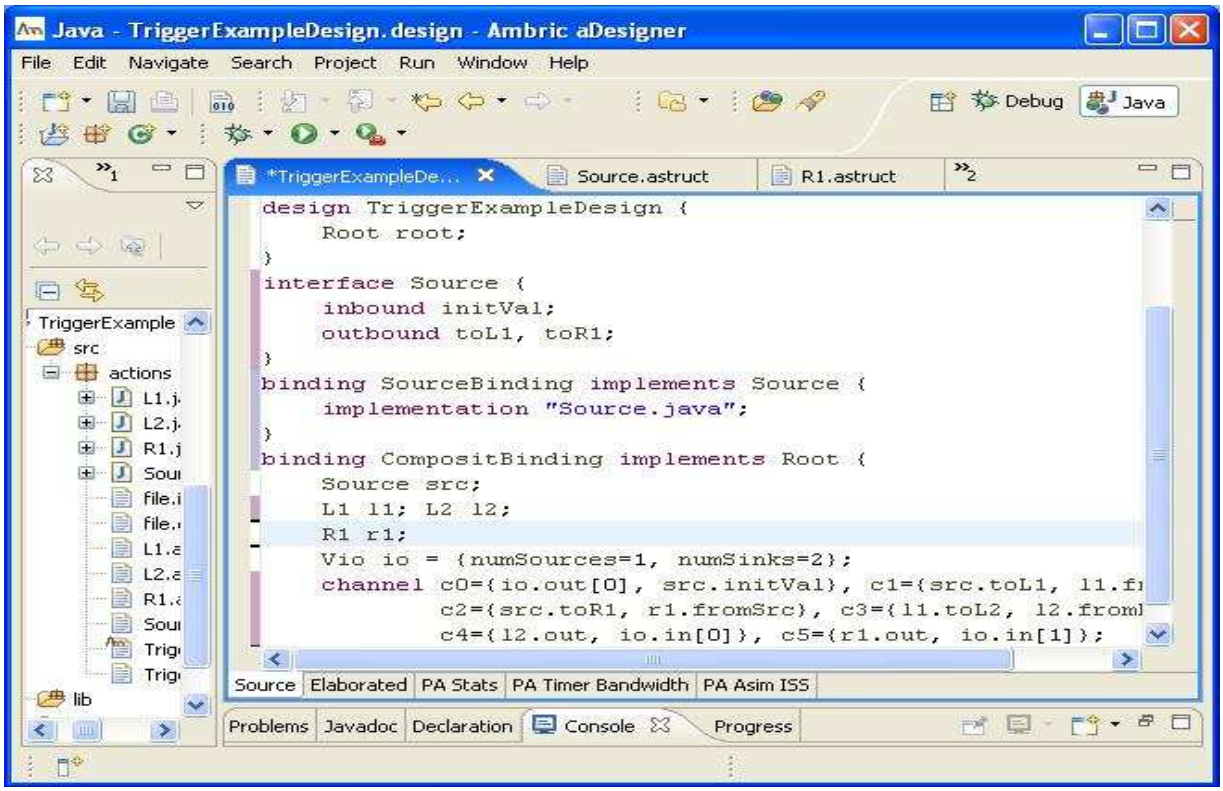


Figure 7: Text Base View

3 Long Term Evolution

Long Term Evolution (LTE) is emerging technology, which would be commercially available in 2010 [6]. Much more could be written on this technology, but topics which are directly related to this thesis are discussed here. This chapter will provide an overview of LTE by summarizing the requirements, objectives and system architecture.

3.1 Technology Overview

With advancement in mobile technology, broadband is becoming a reality. It is estimated that by 2012, around 1.8 billion people will have broadband and two-thirds of them will be mobile broadband consumers. To meet these requirements we need to have a technology which can provide high bandwidth with low cost and reduced device complexity. To achieve this goal, scientist and researchers are working on a technology named LTE. It is a project of 3GPP and its current organizational partners are ARIB, CCSA, ETSI, ATIS, TTA, and TTC. [4]

3.1.1 Targets and Objectives

Long Term Evolution is designed to provide high bandwidth, high spectrum efficiency, higher data rate, reduced device complexity, reduced latency, peak data throughput and flexible channel bandwidth.

3.1.2 LTE Requirements

LTE is designed to provide peak data rate, high degree of mobility and wide coverage. It should support up to 200 active users in a cell with less than 5 ms user-plane latency. System should be optimized for 0 to 15 km/h but it should support 15 to 120 km/h with high performance.

Another requirement of LTE is to provide uplink peak rate of 50 Mbps and downlink peak rate of at least 100Mbit/s. Furthermore, Radio Access Network (RAN) round-trip times (RTT) should be less than 10ms.

LTE must also support the variable transmission bandwidths, including 1.25 MHz, 2.5 MHz, 5 MHz, 10 MHz, 15 MHz, and 20 MHz. Each transmission bandwidth corresponds to a fast Fourier transform (FFT) size of 128, 256, 512, 1024, 1536, and 2048 points, respectively [5].

3.2 System Architecture

LTE has two main components that are uplink and downlink. Together with advanced antenna technologies, it uses Orthogonal Frequency Division Multiplexing (OFDM) and Single Carrier

Frequency Division Multiple Access (SC-FDMA) as its radio access technology for downlink and uplink respectively. A brief overview of these two multiplexing technologies is provided in the next section.

3.2.1 Single-carrier FDMA (SC-FDMA)

SCFDMA is a modified form of OFDMA. Using this technique high data rate uplink communication could be achieved in future cellular systems. This technique provides similar throughput performance and essentially the same overall complexity as OFDMA. It provides peak-to-lower average power ratio (PAPR) as compare to OFDMA. Like any other communication systems, there are complex tradeoffs between design parameters and performance in an SC-FDMA system. Figure 8 depicts block diagram of SC-FDMA transmitter and receiver structures.

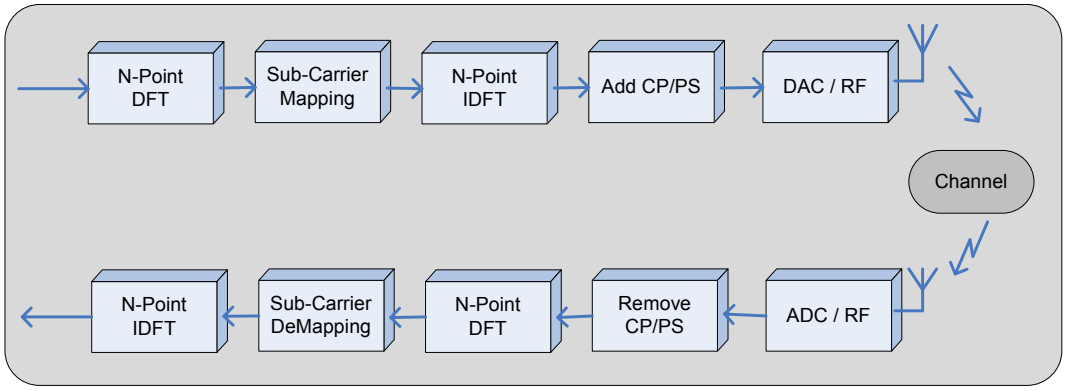


Figure 8: Transmitter and Receiver structure of SC-FDMA

3.2.2 Orthogonal frequency-division multiplexing (OFDM)

This multiplexing technique is used as digital multi-carrier modulation method. To carry data, it uses a large number of closely-spaced orthogonal sub-carriers. Several parallel data streams or channels are created from that data. This sub-carrier are then modulated at a low symbol rate using conventional modulation schemes such as Quadrature Amplitude Modulation (QAM) or Phase Shift Keying (PSK), maintaining total data rates similar to the conventional single-carrier modulation schemes in the same bandwidth. Figure 9 depicts OFDMA transmitter and receiver structures.

This is very much similar to SC-FDMA. The difference is the presence of Discrete Fourier Transform (DFT) in the SC-FDMA transmitter and the Inverse Discrete Fourier Transform (IDFT) in the SC-FDMA receiver. That is why SC-FDMA is sometimes referred to as DFT spread OFDMA.

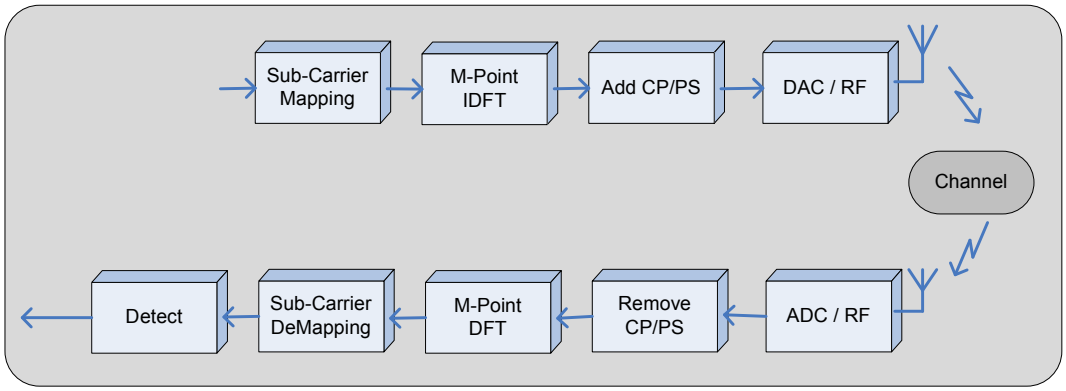


Figure 9: Transmitter and Receiver structure of OFDMA

3.2.3 Block Interleaving

Interleaving is used to make a burst error into random errors. These errors can be corrected by error correcting codes. In mobile communication channel suffers from noise and a fading due to multi path propagation. Burst errors occur in transmitted data due to fading. Block interleaving is used in OFDMA [7].

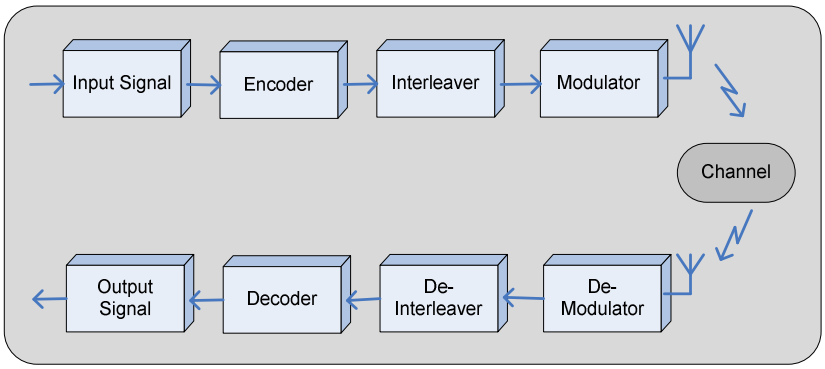


Figure 10: System block diagram for mobile data communication

Interleaver is used at the transmit end of channel shown in Figure 10, the inverse of the interleaver must be used at the receiver end to recover the original signal. The inverse interleaver is referred to de-interleave. Block interleaving and DFT is further discussed in Chapter 4.

In this chapter, Long Term Evolution technology is discussed. System architecture is examined and brief overview of multiplexing techniques i.e. SC-FDMA and OFDMA are provided. An overview of block interleaving is also provided. On analyzing LTE, it is observed that DFT is the most important component of the system which is common in both transmitter as well as receiver structures.

4 Algorithm Overview

This chapter will describe an overview of the selected algorithms which we are going to implement on the Ambric architecture in the next chapter.

4.1 Matrix Multiplication

Matrix multiplication is one of the most fundamental operations in numerical linear algebra, a bit more challenging algorithm. Its importance is magnified by the number of other problem (e.g. computing determinants, solving system of equation, matrix inversion, QR decomposition, etc).

By the definition of matrix multiplication, the number of columns in matrix A is equals the number of rows in B, this range of numbers is called as inner dimension. Finally, the number of rows in A is equal to the number of rows in the resultant matrix C and the number of columns in B is equal to the number of columns in C, this range of numbers is called as outer dimension [8]. Therefore the inner dimension of the matrices must be satisfied.

$$C_{n\times p} = A_{n\times m}, B_{m\times p} \qquad (4.1)$$

The product C of two matrices A and B is defined as;

$$C_{i,j} = \sum A_{i,k} B_{k,j} \qquad (4.2)$$

4.2 Fast Fourier Transforms (FFT)

FFT is an optimized and fastest way to calculate the Discrete Fourier Transform (DFT). It is used to convert the samples in time domain signal to frequency domain signal. FFT is optimized to remove the extra calculation in DFT. Number of samples to be transformed should be an exact power of two. Mathematically, the Fourier transform can be performed without the demand of the number of samples, but the speeding up of the algorithm to an FFT adds this demand [12].

There are two approaches for the calculation of FFT. One is the decimation-in-frequency and the other is decimation-in-time. Both approaches require the same number of complex multiplications and additions. The key difference between the two is that decimation-in-time takes bit-reversed input and generates normal-order output, whereas decimation-in-frequency takes normal-order input and generates bit-reversed output [9]. The manipulation of inputs and

outputs is carried out by so-called butterfly stages. The use of each butterfly stage involves multiplying an input by a complex twiddle factor shown in Figure 11.

4.2.1 Radix-2 FFT

The most common method for computing the FFT is the radix-n algorithms. The radix-2 algorithm is illustrated in Figure 12. It is applicable only to sequences of length N=nm, where m is a positive integer. The most important advantage of this method is a regular and efficient computational scheme.

The basic computation in the radix-2 decimation-in-time algorithm is the butterfly 1, graphically described in Figure 11. It is applied in log2 (N) consecutive steps, N/2 times in each step giving the algorithm a complexity of O (N log2 N).

Figure 12 shows the 8-point FFT computation sequence x(n) using the radix-2 decimation-in-time algorithm. Observe the shuffled order of the input samples, the order is found by reversing the binary representation of a normally ordered sequence. Equivalently, the order can be reached by the log2 (N), 1shuffels of the original linear sequence. One may also rearrange the structure of the calculation depicted in Figure 12, so the input sequence is in order producing a shuffled output [13].

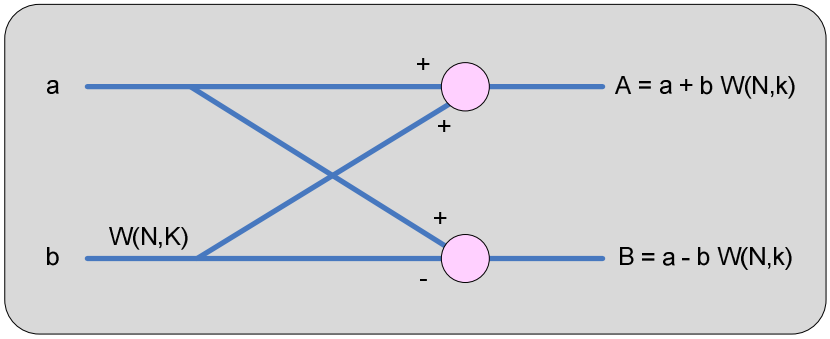


Figure 11: Butterfly Computation of radix-2 with decimation-in-time

4.2.2 Complexity analysis of radix-2 FFT

Obtaining the butterfly as the basic unit of computation then there is one complex multiplication and two additions involved in each butterfly. With N/2 butterflies in each of log2 N steps for a N-point FFT then will be a total of 2N log2 N real multiplications and 3N log2 N real additions, e.g. 5N log2 N floating point operations. In all is the computation in O(N log2N) [13].

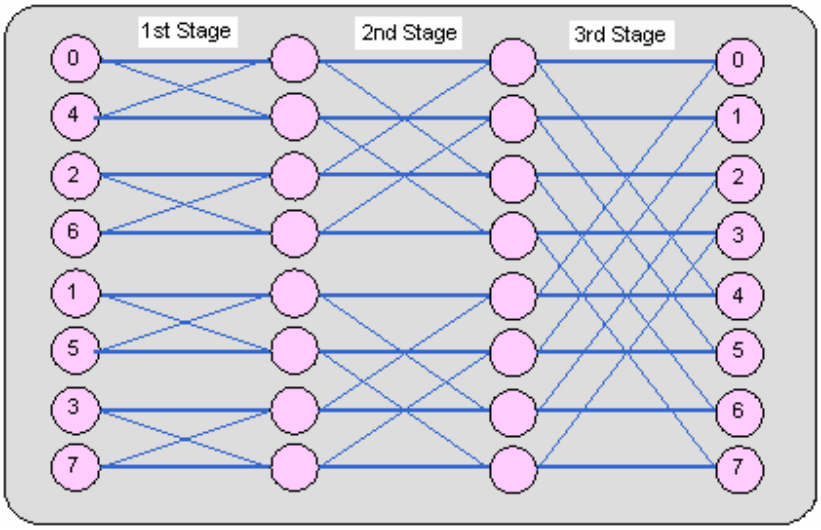


Figure 12: 8-point FFT butterfly using the radix-2 decimation-in-time

4.3 Block Interleaver

There are various techniques used for the block interleaving and we have selected two of them, one is matrix interleaver and the other is helical scan interleaver. Now we will describe these techniques one by one.

4.3.1 Matrix Interleaver

In matrix interleaver, the information stream is written row by row in a matrix of “n” rows and “m” columns and read out column by column. The column size n is called the depth and the row size m is the span. Such an interleaver is completely defined by n and m and is thus refereed to as M(n,m) matrix interleaver. At the De-Interleaver, information is written column-wise and read out row-wise. The capability of burst error scattering for the matrix interleaver depends on the values of n and m [11].

4.3.2 Matrix Helical Scan Interleaver

The Matrix Helical Scan Interleaver performs block interleaving by filling a matrix with the input data row by row and then sending the matrix data in a helical fashion. The number of row and number of column parameters are the dimensions of the matrix.

Helical fashion means that the block selects output data by selecting elements along diagonals of the matrix shown in Figure 13. The block traverses diagonals so that the row index and column

index both increases. Each diagonal after the first one begins one row below the first element of the previous diagonal.

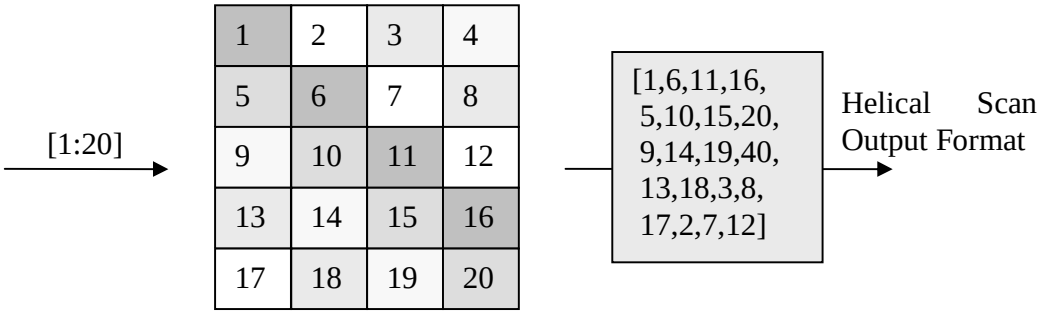


Figure 13: Helical Scan Block Interleaver

5 Implementation

We have chosen three algorithms for the implementation and evaluation of Ambric architecture. We already have presented an overview of these algorithms in Chapter 4. This chapter will describe the practical work done within the thesis, including API implementation and detailed description of the algorithm implementation and mapping on the Ambric architecture. In later discussion whenever we will refer to a processor or object, it means an object running on a single processor so we will use these terms alternatively.

5.1 Application Programming Interface (API)

In implementation we need some functionality like fixed point, binary log and binary power, so first we will discuss about the actual implementation of these functions. We have discussed about fixed point in Chapter 3, now we will describe more about its implementation.

5.1.1 Fixed point

The arithmetic operations like addition and subtraction of fixed point numbers can be performed with normal integer addition and subtraction operators. But care must be taken in the case of multiplication because when we multiply two fixed point numbers of length WL then the result would be 2WL. In Ambirc architecture the length of a word is 32-bit. So we can decide any fix-point format within 32-bit of word length. For instance, we have Q8.24 fix-point format where QI is 8-bits including one sign bit and QF is 24-bits. Ambric architecture supports the multiplication of two 32-bit numbers and store the 64-bit result in the accumulator register from where we can read lo 32-bit and hi 32-bit values of the result separately.

After multiplication our result will be store in the accumulator but we can not read and store 64-bit number in *ajava*. So hi part of the accumulator contains the result of multiplication in Q16.16 format. In order to convert this number back to Q8.24 format we have to shift this number 8-bits left so lower 8 bits of the answer would always be empty. To get more precise answer we can read lo part of the accumulator and move its most significant 8-bits to the left of our answer.

5.1.2 Binary log

Binary logarithm is often used in mathematical calculations so we also have implemented it on Ambric. It is used in butterfly calculation of FFT. In 32-bit integer range, the integer binary log can be computed by rounding down or rounding up. We have implemented this function by assuming the floor value. The two techniques are shown in the following equation;

$$\text{Floor}(\text{Log2}(N)) = \text{Ceiling}(\text{Log2}(N+1))-1 \quad (5.1)$$
$$\text{Where } N >= 1$$

We have implemented the algorithm by using only arithmetic right and arithmetic left shift operators as we do not have support for floating point operation.

5.1.3 Binary power

Taking binary power is very simple, each bit in a word length represents 2N. For instance, in 8 bit word the first bit represent 20=1, second bit represents 21=2, third bit represents 22=4 and so on. We can take power of two by shifting 1 to left N times.

$$\text{Power of } 2 = 1 << N \quad \text{For } N >= 0 \qquad (5.2)$$

5.2 Algorithm Implementation

For each algorithm, we have made two different designs so that we can evaluate the Ambric chip and development tools better. This will make us able to think how can we design and map application efficiently onto the Ambric architecture.

For optimal utilization of parallel processing capabilities, data stream pass to it should also be parallel in its structure. Ambric chip provides only four input and output ports. So we can not read more than four input streams from the chip. To run an algorithm on more than four processors we have to distribute a single stream to parallel processing elements. Also an object can have maximum five input ports and six output ports according to the SRD's instruction set.

Mostly SR processors are used for streaming and SRD processors are used for math intensive operations. So we can distribute our input and output streams through SR processors with the help of *splitter* objects. These objects take single input stream and duplicate it, and send two or more streams on output, although this procedure will take some time to reach input to the objects actually performing some useful work. But this time can be included in setup time of the application. Finally, we can *join* the output stream in the similar way.

5.3 Matrix Multiplication

We have done matrix multiplication in two different ways. The designs differ mainly by the flow of data stream. This will also tell us about how we can provide parallel data streams to different objects efficiently.

5.3.1 Design approach 1

This design reads matrices in two parallel streams and passes the data to other objects in serial order. Simultaneously this data is provided to the objects performing multiplication and finally the result is joined to output stream. Figure 14 show four objects connected in a ladder performing multiplication along with data flow. Each object in this design is a composite object which consists of two *splitters* "A" and "B", one *multiplier* "M" and one *join* "J" object illustrated in Figure 15.

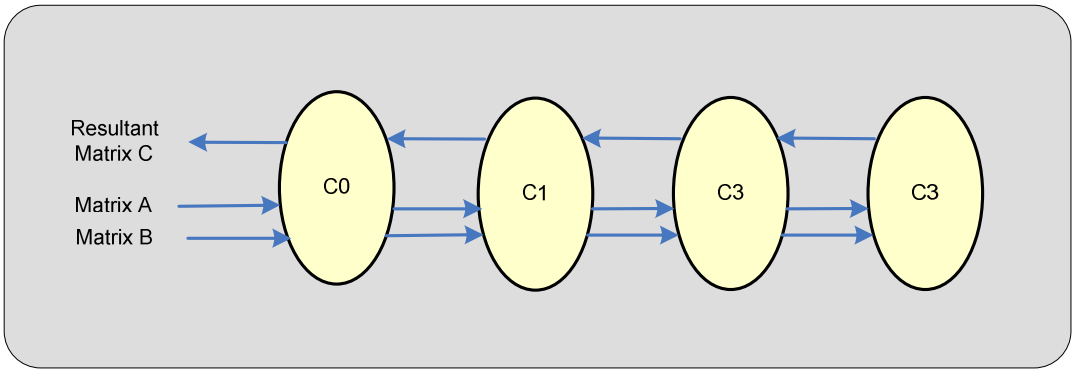


Figure 14: 4 composite objects in ladder

Splitter A reads matrix A and send its first row to its *multiplier* M and then passes the remaining matrix to the next composite object. So each *multiplier* object gets one row of matrix A and stores it for further use. For instance, *multiplier* M of composite object C0 has first row, C1 has second row and so on.

Splitter B reads columns of matrix B one by one and passes each column to its *multiplier* M moreover it sends to the next composite object. Then it will read second column and apply the same phenomenon on it. So in this way, each *multiplier* M object will get all columns one by one.

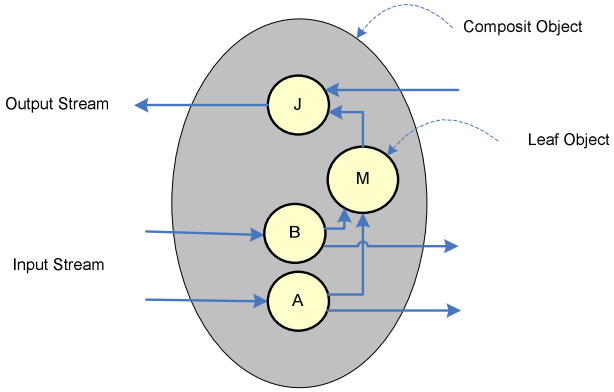


Figure 15: A composite object

In the first iteration of the algorithm, each *multiplier* M will multiply its corresponding row of matrix A with column of matrix B and sends out the result to their respective *join* object. The *join* objects will collect all elements and send out column one of resultant matrix C. In next iteration, each *multiplier* will multiply their respective rows with second column and send out the second column of resultant matrix C and so on. Each composite object in this design will calculate one row of the resultant matrix.

```
int dim = 8; // dimension of matrix
CompositObj co[dim];

for(i=0; i<dim; i++){
    co[i].name = "Element"+i; // this name will be displayed on the elaborated
    co[i].dim = dim;
    co[i].rows = dim - i;
    co[i].id = i+1;

    if((i+1) == dim) {
        co[i].last = true;
    } else {
        co[i].last = false;
    }
}
for(i=0; i<dim; i++){
    if(i == 0){
        channel c1 = {io.out[0], co[i].splitInA};
        channel c2 = {io.out[1], co[i].splitInB};
        channel c3 = {co[i].joinOut, io.in[0]};
    } else {
        channel c4 = {co[i-1].splitOutA, co[i].splitInA};
        channel c5 = {co[i-1].splitOutB, co[i].splitInB};
        channel c6 = {co[i].joinOut, co[i-1].joinIn};
    }
    if( (i+1) == dim ){
        /* for the last Splitter and Join objects we don't need to create chann
         * so mark these input ports as un-used, now we can't read/write on the
         */
        attribute Unused() on co[i].splitOutA;
        attribute Unused() on co[i].splitOutB;
        attribute Unused() on co[i].joinIn;
```

Figure 16: Design file for the multiplication

For the multiplication of 4x4 matrices we need 4 SRD processors and 12 SR processors. So for NxN matrix multiplication we need N SRDs and 3N SRs. In general the formula becomes;

$$\text{Total no of processors} = 4N \qquad (5.3)$$

This design is scalable; it means we can use this design for NxN multiplication within the range of total number of processors illustrated in Figure 16. By changing only one compile time parameter we can scale this design for NxN multiplication.

5.3.2 Design approach 2

This design is similar to the approach 1; the major difference is in the *splitter* part. This design is more efficient and requires less number of objects illustrated in Figure 17.

Input streams provided to *multiplier* objects are more parallelized in this design than the previous design approach. The *splitter* B will read one row and distribute elements of row to *multiplier* objects one by one. If there are four elements in a row then *splitter* B will send first element to *multiplier* M0 and second to M1 and so on. Each *multiplier* object will get one column of matrix B. In this way, we do not need to take transpose of the resultant matrix; it is demonstrated in Figure 18. In the meantime *splitter* A reads one row of matrix A and provides this row to all *multiplier* objects at the same time, then second row to all *multiplier* objects and so on.

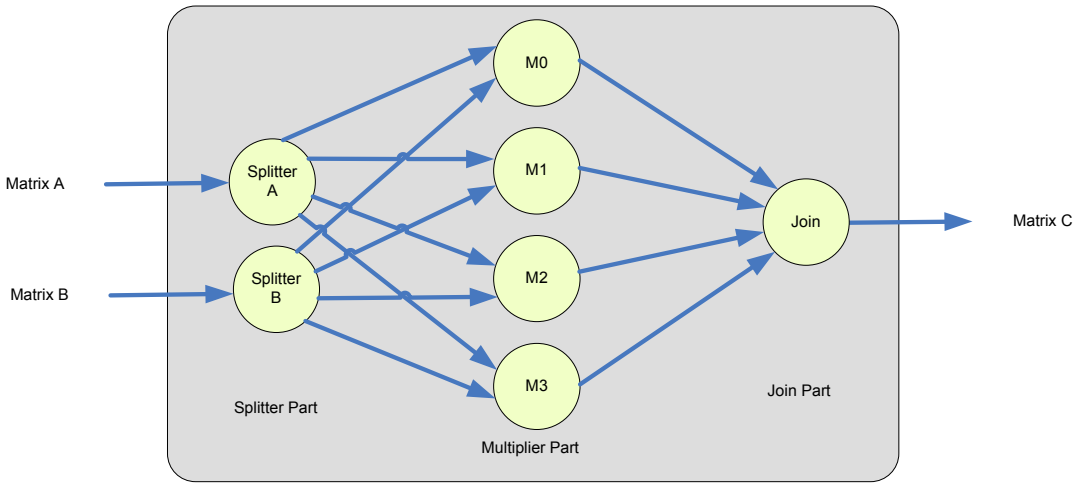


Figure 17: Objects communication of design approach 2

Every *multiplier* object stores the column of matrix B for further use and gets rows of matrix A one by one, and multiplies them in parallel. Finally, *join* objects get rows of resultant matrix C and sends them to output stream. This design requires total 7 processors which include only 3 SR and 4 SRD processors for the 4x4 matrix multiplication.

```
public void run(InputStream<Integer> inB,
        OutputStream<Integer> outB1,
        OutputStream<Integer> outB2,
        OutputStream<Integer> outB3,
        OutputStream<Integer> outB4) {

    /*
     * read matrix B row-wise but distribute it coloumn-wise
     *  so that each Mulitplier object will have corresponding ct
     *  e.g; Mulitiplier 1 will have coloumn#1 and Mulitiplier 2
     */
    for(int i=0; i<dim; i++) {
        outB1.writeInt(inB.readInt());
        outB2.writeInt(inB.readInt());
        outB3.writeInt(inB.readInt());
        outB4.writeInt(inB.readInt());
    }
}
```

Figure 18: Java code for *splitter* B

5.4 Fast Fourier Transform

FFT is implemented with radix-2 described in Chapter 4, here we will discuss about its design and mapping on Ambric architecture. We have also implemented two designs for FFT which we will describe in next section. Here we will discuss about some commonalities used in both designs.

Each design approach has two different versions, one is 8-point and the other is 16-point FFT. We will discuss only 8-point FFT because both versions have similar design principal. It is a better approach to provide pre-calculated twiddle factors to the application rather than computing them on run-time, in this way we can get some speedup. We can store twiddle factors in several ways, in lookup table or in external memories. It can also be provided to processors on run-time through input streams but it will consume more resources. We can also pass twiddle factors to the objects at compile time through *astruct* language and this approach is better.

In design approach 1, each object stores the twiddle factors in lookup tables. We can not pass them through *astruct* language at compile time because we can not pass array of properties to java objects or assembly code. But in design approach 2 each object will need only one value of twiddle factor. So twiddle factors are provided to objects once through static *astruct* design and then stored them in memory permanently, it is shown in Figure 19.

```
// values for stage 4 twiddle factors
int s4Cos[] = {16777216, 15500126, 11863683, 6420363, 0, -6420363, -11863683
int s4Sin[] = {0, -6420363, -11863683, -15500126, -16777216, -15500126, -118

for(i=0; i<nFft_onsides; i++) {
    // objects for stage 1
    fft24[i].name = "FFT_2in4out"+i;
    fft24[i].propPoints=nPoints;
    fft24[i].propCosVal = s1Cos;
    fft24[i].propSinVal = s1Sin;
    // objects for stage 4
    fft42[i].name = "FFT_4in2out"+i;
    fft42[i].propPoints=nPoints;
    fft42[i].propCosVal = s4Cos[i];
    fft42[i].propSinVal = s4Sin[i];
}
```

Figure 19: Twiddle factors assigning at compile time

```
public Splitter(int propN){
    this.N = propN;
}

public void run(InputStream<Integer> inReal,
        InputStream<Integer> inImg,
        OutputStream<Integer> outReal1,
        OutputStream<Integer> outImg1,
        OutputStream<Integer> outReal2,
        OutputStream<Integer> outImg2) {

    // distribute points in even and odd order for bit-reversal
    for(int i=0; i<N; i+=2) {
        // send even point to left output stream
        outReal1.writeInt( inReal.readInt() );
        outImg1.writeInt( inImg.readInt() );
        // send odd point to left output stream
        outReal2.writeInt( inReal.readInt() );
        outImg2.writeInt( inImg.readInt() );
    }
}
```

Figure 20: Java code for *splitter* object

We have used decimation-in-time method for the implementation of radix-2. We have used this technique because it uses bit-reversal mechanism first and then it performs the butterfly computations. Why we have used this mechanism? When input stream is distributed, at that time we can take bit-reversal sorting on the input points.

We do not need to reserve one or more processors for the bit-reversal sorting, so in this way we have saved some execution time. Figure 20 elaborates this mechanism. *Splitter* objects sort out the even and odd elements of the input stream. During distribution of input stream each *splitter* object will send even points to its left object and odd points to its right object. The final stage will have completely separate even and odd points. It is depicted in Figure 21.

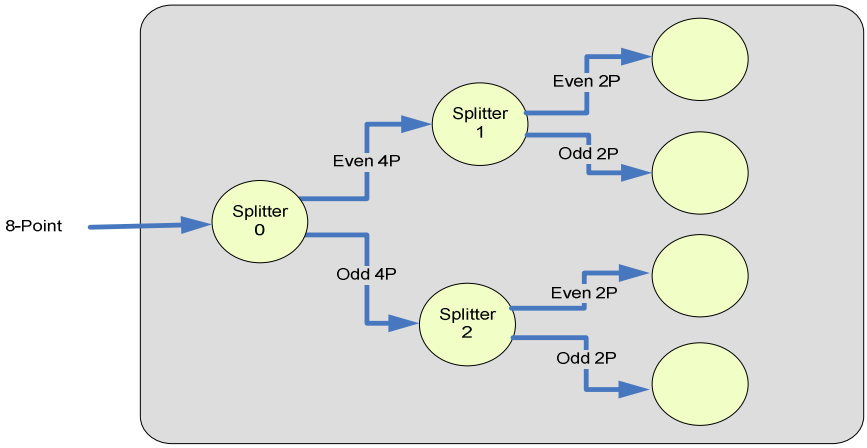


Figure 21: 8-point FFT bit-reversal sorting

This *splitter* stream technique is also common in our both design implementations. Now we will discuss more about the two design approaches.

5.4.1 Design approach 1

Figure 22 depict object mapping of the design approach and their communication through channels for 8 point FFT. Each circle represents the object running on Ambric processor and arrow represents the flow of algorithm. Input stream consists of both real and imaginary part of the time domain signal. It means we are reading real and imaginary part of the signal in two different parallel streams. It is illustrated in Figure 23.

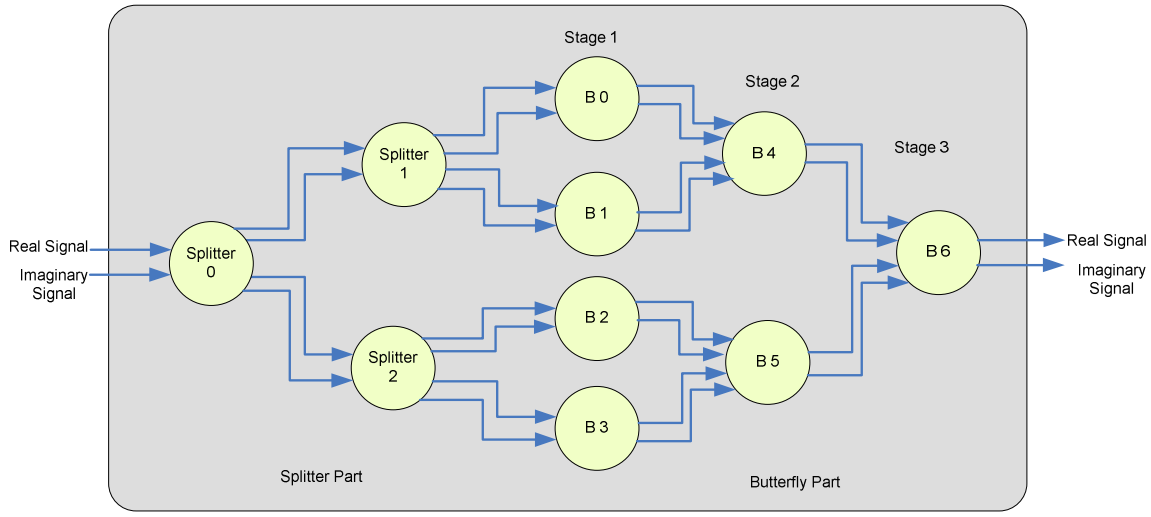


Figure 22: 8-Point FFT Design Approach 1

This algorithm can be divided into two parts; first part is the *splitter* where algorithm splits out the input stream along with performing the bit-reversal sorting and the second part is performing the butterfly calculations along with *join*ing the 8-points in all three stages and sends them to the output stream. The *splitter* part of the algorithm read chunks of 8-points from input stream and sends to the butterfly calculation part of algorithm where each object in stage 1 gets 2-points and perform butterfly calculation on them, in stage 2 each object perform butterfly calculation on 4-points and similarly stage 3 compute 8-point butterfly on a single object.

For this algorithm we need 3 SR and 7 SRD processors totalling 10 processors while for 16-point FFT we need 7 SR and 15 SRD processors totalling 22 processors. In general this design will need (N/2 -1) SR processors and (N-1) SRD processors. So for N-point FFT the formula becomes;

$$\text{Total Processors} = \left(\frac{N}{2} - 1\right) + (N - 1) \qquad (5.4)$$

Each stage in the butterfly calculation part of algorithm contains half number of objects than the previous one. For this reason objects in each butterfly stage have more work load than the previous one. For instance, in stage 1 four processors perform the butterfly computation on only two points while in stage 2 two objects perform the butterfly computation on four points so that they need more execution time to complete the calculation. Although we can utilize channel buffers but they have very limited memory so when continuous streaming is passed through the design then processors suffers from stall conditions after certain interval of time.

```
interface Splitter {

    inbound inReal; // input port for real signal
    inbound inImg;  // input port for imaginary signal

    outbound outReal1, outImg1;
    outbound outReal2, outImg2;

    property int propPoints; // total no of point in FFT
    property int propN;      // no of points to read from previous splitter
}

binding SplitterBinding implements Splitter {
    implementation "Splitter.java";
    attribute CompilerOptions(targetSR = true) on SplitterBinding;
}
```

Figure 23: *Astruct* code of *splitter* object and binding with java

This is one of the main reasons behind creating this design that we can evaluate the most interesting and powerful feature of the Ambric architecture that each object can run independently on its own speed or different processors can run on different frequencies. Figure 22 specifies that processors in each stage runs on different frequencies according to the application requirements. Processor in stage 3 run on its full frequency "1f" and processors in stage 2 run on half of the frequency "½f" and processors in stage 1 run on the quarter frequency "¼f". The speed of the object is directly proportional to the clock frequency of the processor. So we can utilize this feature in the design by scaling the clock frequency of processors on each stage.

5.4.2 Design approach 2

This design is enhanced and more efficient than the design approach one. It is also more parallelized version and somewhat more flexible than the other one. The *splitter* part of this design is the same as the design approach 1. There is an additional part of the algorithm which will *join* the output stream at the end of butterfly calculations.

In this design each object only computes 2-point butterfly calculation so all objects will have the same work load. As all objects have same work load so processor stalls are reduced in this design. This design does not require any change in the clock frequency to scale the execution speed of objects. Figure 24 demonstrate 3 stages of 8-point FFT where each object gets 2 point from input stream and multiplies them.

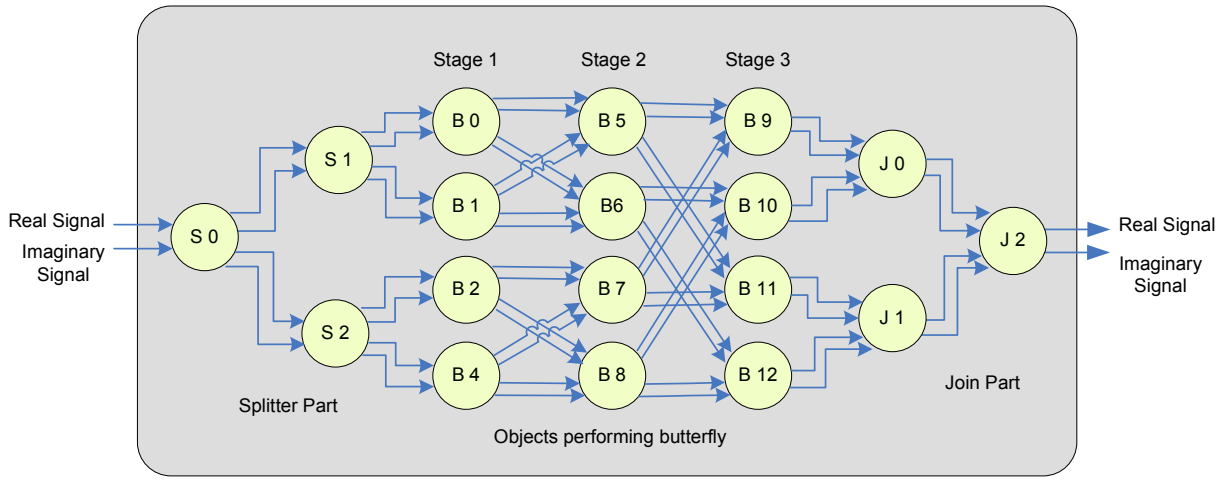


Figure 24: 8-Point FFT Design Approach 2

This design is efficient from the design approach 1 because each object only computes one butterfly. In this design we do not have loop overheads because each object only multiplies two points. The java code is shown in Figure 25.

```
for(i=0; i<N; i+=2){ // read real and imaginary signal
    x[i].real = inReal1.readInt();
    x[i].img = inImg1.readInt();
    x[i+1].real = inReal2.readInt();
    x[i+1].img = inImg2.readInt();
}

// Butterfly calculation using fixed point numerics
int TR = fp.subtract(fp.multiply_32(x[1].real, cos), fp.multiply_32(x[1].img, sin));
int TI = fp.add(fp.multiply_32(x[1].real, sin), fp.multiply_32(x[1].img, cos));
x[1].real = fp.subtract(x[0].real, TR);
x[1].img = fp.subtract(x[0].img, TI);
x[0].real = fp.add(x[0].real, TR);
x[0].img = fp.add(x[0].img, TI);

for(i=0; i<N; i+=2) {
    outReal1.writeInt(x[i].real);
    outImg1.writeInt(x[i].img);
    outReal2.writeInt(x[i+1].real);
    outImg2.writeInt(x[i+1].img);
}
```

Figure 25: Java code computing one butterfly

This design requires more number of processors. For 8-point FFT we need 18 objects (3 *splitter*, 3 *join* and 12 objects for butterfly computation) and for 16-point FFT we need 46 total objects. So for N-point FFT 2(*N*/*m* -1) number of SR processors and (*N*/*m* log2N) SRD processors and the formula become;

$$\text{Total Processors} = 2\left(\frac{N}{m}-1\right)+\left(\frac{N}{m}\times\log_2(N)\right) \qquad (5.5)$$

Where *m* is the number of points calculating on each object and should be in the power of 2. *N* is the total number of points in the FFT and $\text{Log}_2(N)$ represents number of stages in the FFT.

When we increase number of points in FFT algorithm then number of processors also increases gradually. But we have limited number of processors on the Ambric chip that is 336; half of them are SR. Note that we can not use SR processors for the multiplication. So we can only implement this design for maximum of 32-point FFT. But if we want to use it for calculating larger point FFT then we can increase value of *m*. So in this way, we can use this design to calculate any point FFT more efficiently.

5.5 Block Interleaver

We have implemented two types of Block Interleaver, Matrix Interleaver and Helical Scan Interleaver. Structure of the design, illustrated in Figure 26, is same for both techniques and is much similar to matrix multiplication design. Only technique used on the multiple processors is different and single data stream is used for input and output.

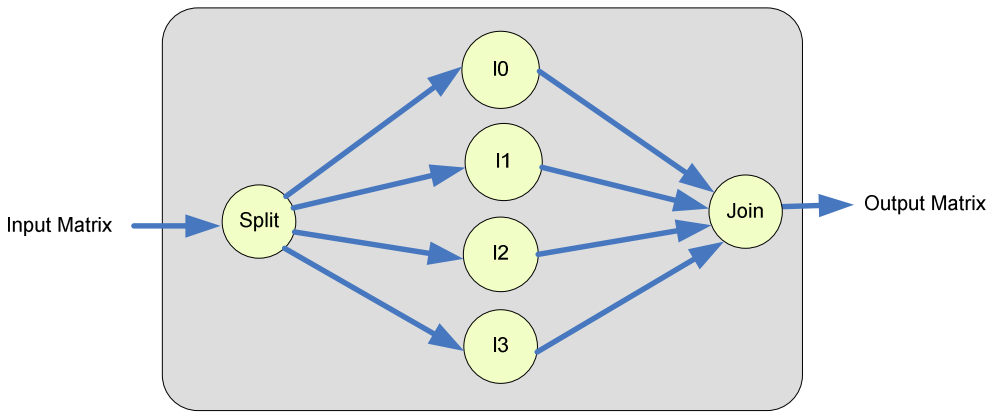


Figure 26: Design approach for Block Interleaver

5.5.1 Matrix Interleaver

This design is much similar to the matrix multiplication approach 2; the major difference is in the *splitter* part shown in Figure 26.

In this design only one *splitter* is required. *Splitter* will read one row and send elements of row to their respective *Interleaver* objects. If there are four elements in a row then *splitter* will send first element to *interleaver* 0 and second to *interleaver* 1 and so on. In this way each *interleaver* object will get one column of input matrix. First *interleaver* 0 starts sending first column to output stream immediately while others have to wait until the antecedent *interleaver* has sent its data to output. So other *interleaver* objects have to store inputs except first one. *Join* object gets first column from *interleaver* 0 and after sending it to output stream it will get second column from *interleaver* 1. Afterwards this method will be applied to all other columns.

5.5.2 Helical Scan

Helical scan interleaver is again similar to the Matrix Interleaver depicted in Figure 26. The only difference is in the sorting of the data. Helical scan interleaver reads matrix data diagonally; a detailed discussion on the algorithm is given in Chapter 4. Here we will discuss its implementation technique.

All *interleaver* objects will get their corresponding row number, store and sort them diagonally.
For instance, *interleaver* 0 will get first row, *interleaver* 1 will get second row and so on. After sorting, each *interleaver* object will send data towards *join* objects. *Join* object will get first element from all *interleaver* objects and make first row of output matrix, then it gets second element of each *interleaver* and sends it to output stream and so on. To get first row of output matrix *join* object has to wait until whole input matrix loaded on the *interleaver* objects. The reason is that the last element of first row of output matrix is the last element of input matrix.

6 Evaluation

The main objective of this thesis is to evaluate whether parallel (Ambric) processor architecture is useful in the baseband signal processing. In this chapter we will investigate the efficiency of the development tools, algorithm and I/O mapping.

The results from the implemented algorithms are presented in terms of cycle counts and processor stalls for different sizes of input data. Whenever a processor is waiting for an input or output (waiting for other processor to get input from it) is known as stall. The total number of cycle counts is equal to the number of instructions executed plus the number of processor stalls.

$$\text{Total Cycle Counts} = \text{Instructions} + \text{Stalls} \qquad (6.1)$$

This information is extracted from the aSim simulator. In the simulator we can make both interval and window measurements. Intervals have a period, measured in processor clock cycles. Windows have a start address and an optional stop address. The simulator returns cycle counts of processor execution and stalls for each interval or window. On processors, intervals and windows measure the number of instructions executed, cycles taken, and stalls caused by instruction execution or memory accesses. Since the intervals and windows on a single processor use the same time unit, the processor cycles, they can be correlated.

In each algorithm, there are some processors running in sequence while some processors are running in parallel. For the processors which are running in parallel we have picked the one which has the maximum number of cycle counts, and then we add up cycle counts of processors which are running in sequence. For instance, in matrix multiplication design approach 2, *splitter* A and B is running in parallel so we picked the *splitter* which has maximum cycle count value. All *multiplier* objects are running in parallel so we have again picked up one processor from them and there is only one *join* processor. Finally, we have added cycle counts of these three processors and find out the total number of cycle counts of the whole algorithm.

We will also evaluate the implemented algorithms in terms of speedup values. We can not analyze the performance of algorithms in terms of time because the aSim simulator only returns cycle counts, processor stalls and number of instructions executed. We will calculate the speed up by total number of cycles taken by each algorithm and these cycle counts also include processor stalls, while processor stalls can be increased or decreased depending on the implementation of the algorithm. So we can not evaluate the speed up of algorithms exactly, but a rough estimation could be made. Because of the processor stalls a surprising difference in speed up values will be noticed. The original formula for the calculation of speedup is given below;

$$\text{Speed up} = (\text{Time taken by 1 processor} / \text{Time taken by N processors}) \qquad (6.2)$$

In this formula we have used the total cycle counts instead of time.

6.1 Results from Matrix Multiplication

In Chapter 5 we have discussed two types of algorithm mappings for the matrix multiplication. This section compares the two techniques.

The results from three different approaches for the multiplication of matrices are given in Table 1. After calculating the cycle counts on the single Ambric processor and on multiple processors we observe that design approach 1 results in a cycle count that is approximately half of the single processor implementation while design approach 2 results in three times fewer cycles than the design approach 1.

In design approach 2 the stalls on the processors are reduced. The difference between these two results is because of the input and output streams provided. The I/O provided to the design approach 2 is mapped more efficiently. Table 1 shows that design approach 2 is much faster than the approach 1 and it takes less number of processors, so obviously it will also consume less power.

In design approach 1 the *splitter* objects are connected in a sequence and reading the data in serial order, because of it the application gets more processor stalls. The I/O stream provided to the approach 2 is more parallelized; a detailed discussion is available in section 5.3. Both designs use same number of SRD processors for the computation e.g. 4 processors for the matrix size of 4x4 and 8 processors for 8x8 matrix size while both designs use different number of SR processors for splitting and joining of the data stream.

Matrix Multiplication Matrix Size:	Total Cycle count		Total Stalls	
	4x4	8x8	4x4	8x8
Sequential implementation	4208	27051	2298	15820
Design Approach 1	2783	9640	612	2303
Design Approach 2	832	3398	324	1509

Table 1: Results for Matrix Multiplication

Matrix Multiplication	No of processors		Speed up	
	4x4	8x8	4x4	8x8
Sequential implementation	1	1	1	1
Design Approach 1	16	32	1.512	2.806
Design Approach 2	7	17	5.058	7.960

Table 2: Speed up table for Matrix Multiplication

Table 2 shows the speedup along with number of processors used from the matrix multiplication design approaches. The design approach 2 is approximately 3.3 times faster than the approach 1 for the 4x4 matrix size and for 8x8 matrix size the approach 2 is approximately 3 times faster, in spite of the fact that it uses only about half as many processors.

6.2 Results from FFT

In this section we will compare the two algorithm mappings of FFT, called design approach 1 and 2. The design approach 1 is one of the parallelized versions of FFT. Table 3 shows that design approach 1 takes more cycle counts than the single processor implementation. If we compare total stall counts of design approach 1 with the single processor implementation, then we came to know that its total cycle counts are increased due to the number of stall counts. The reason is already described in the Section 5.4.1. If we scale the frequency of processors in each stage then processors in each stage can be compatible with each other in terms of input and output points, this will reduce the number of stalls on each processor and hence total cycle counts will also be reduced but the time will not decrease.

FFT	Total Cycle count		Total Stalls	
	8-Point	16-Point	8-Point	16-Point
Sequential implementation	5066	11472	1581	3578
Design Approach 1	5195	15368	2985	7995
Design Approach 2	1085	1828	394	594

Table 3: Results for the 8-Point and 16-Point FFT

Table 4 illustrates the speedup along with number of processors from the FFT design approaches. The speedup value of design approach 1 is less than the sequential implementation for 8-point FFT and it slows down for the 16-point FFT. The design approach 2 is approximately 5 times faster than the design approach 1 for 8-point FFT and approach 2 is approximately 8 times faster than the approach 1 for 16-point FFT. The Table 4 shows that the design approach 1 is getting slower for larger point FFTs while the design approach 2 is getting speedup over larger point FFTs.

The design approach 2 takes fewer cycle counts and stalls than the approach 1. Although it uses more processors for butterfly computation, but it gives speedup with respect to the design approach 1. For instance, the design approach 2 use double processors than the approach 1 for 8-point FFT but it gives 4 times speedup, so it requires less energy consumption as compared to the approach 1.

FFT	No of processors		Speed up	
	8-Point	16-Point	8-Point	16-Point
Sequential implementation	1	1	1	1
Design Approach 1	10	22	0.975	0.748
Design Approach 2	18	46	4.669	6.275

Table 4: Speed up table for FFT

6.3 Results from Block Interleaver

This section compares the two techniques of block interleaver with each other and with the sequential implementation.

The performance analysis of sequential and parallel implementation of matrix interleaver is given in Table 5. The parallel version of matrix interleaver for 4x4 matrix takes almost one third cycle counts and for 8x8 matrix it is almost half of the sequential implementation, same is the case with helical scan interleaver. As the order of matrices increases the total number of cycle counts and number of stalls also increases. The results from Table 5 and 6 shows that the matrix interleaver technique is performing better than the helical scan interleaver.

Matrix Interleaver	Total cycle count		Total stalls	
	4x4	8x8	4x4	8x8
Sequential implementation	893	3157	454	1622
Parallel implementation	382	1784	51	208

Table 5: Results for Matrix Interleaver

Helical Scan Interleaver	Total cycle count		Total stalls	
	4x4	8x8	4x4	8x8
Sequential implementation	1215	3738	518	1864
Parallel implementation	427	1922	24	351

Table 6: Results for Helical Scan Interleaver

Speedup and number of processors used are given in Table 7. Matrix interleaver gets 2.3 times speedup for the 4x4 matrix over the sequential implementation then it slows down to 1.7 times for 8x8 matrix. Helical scan interleaver gets 2.8 times speedup for 4x4 matrix and then it also slows down to 1.9 times for 8x8 matrix. Helical scan get speedup of 0.5 over the matrix interleaver for 4x4 matrixes and then it decelerate to 0.2 times for the 8x8 matrix.

Block Interleaver	No of processors		Speed up	
	4x4	8x8	4x4	8x8
Sequential implementation	1	1	1	1
Matrix Interleaver	6	14	2.337	1.769
Helical Scan Interleaver	6	14	2.845	1.945

Table 7: Speed up table of Block Interleaver

7 Discussion

A summary and interpretation of the results achieved in the simulation, as well as experience gained from working with new processor architecture (Ambric) is presented in this section.

Tailored algorithms for the parallel architectures are needed to achieve maximum performance. As a first step in introducing new processor architecture on the market a high level development environment and a library of optimized algorithms have to be supported. It would be helpful to reduce the learning period and the development time.

7.1 FFT

The comparison of FFT implementation made in this thesis on the Ambric provides a deeper understanding of the Ambric architecture. In FFT we have focused on the algorithm mapping, processor computation and communication between objects. After evaluation we have observed that approach 2 is more efficient than approach 1 based on the fact that each object only computes one butterfly. In this design we did not have loop overheads because each object only multiplies two points. So the design approach 2 takes fewer cycle counts and stalls than the other one. It takes more processors but a significant speedup is achieved.

FFT design approach 1 is not suitable for the computation of larger FFTs. When the number of butterfly stages is increased in the design then scaling the frequency of processors could not be practical, as processors in each stage have half work load than the next stage and we have to run processors on the half frequency than the next stage.

The maximum frequency at which the Ambric processors can operate is MaxClock and the value of MaxClock is 350 MHz. The frequency is controlled by writing the clock generator configuration register. The clock generator register can be written at any time. Ambric architecture provides eight different options for scaling the clock frequency of Ambric processors, illustrated in Table 8. This information is extracted from the Am2045DataBook manual provided by the Ambric Inc.

Register Value	Frequency
0	MaxClock
1	MaxClock/2
2	MaxClock/4
3	MaxClock/8
4	MaxClock/16
5	MaxClock/32
6	MaxClock/64
7	MaxClock/128

Table 8: Clock generator register fields

According to Table 8, we can not use design approach 1 for FFT which has more than eight stages because we do not have the option to select the frequency for ninth stage. Within eight option we can use this design for maximum 256-point FFT. It is not useful to utilize this design approach for larger FFTs. We can use design approach 1 for smaller FFTs and where we require less power consumption.

Although FFT design approach 2 uses more processors but we can reduce number of processors by increasing number of points multiplying on each processors. The FFT design approach 2 is computing 8-point FFT but we can use the same design for 16-Point FFT. So by using the limited number of processors we can use this technique for larger FFTs.

We can also get code reusability in *ajava* or assembly language code. As in design approach 2 we have created only three java classes a Splitter, a Join and an FFT. Each *splitter* processor runs the same code for the distribution of input points, each processor uses the same java code for the butterfly computation and processors which are joining the output stream also run the same java code. If some processor requires different information that can be provide to processor through java properties at the creation time of objects.

As *ajava* is a subset of java language, we can not use standard features of java language. There is no standard API support available in current version of development tools (version 1.0) for programming *ajava* language. As the Ambric architecture does not support floating point and development tools does not provide any functionality for the fixed point calculation, we have to implement fixed point API by ourselves. Because of this our implementation takes more time. In order to reduce the development time some built-in functionality should be provided along with development tools such as complex numbers, fixed point, log, power etc.

Ambric architecture provides the interesting feature that different processors can run on different frequencies, but the current version of development tools (version 1.0) did not support this feature. Hopefully this feature will be provided in next versions of aDesigner. So we can not implement this feature. If we could reduce processor stalls by implementing this feature, then design approach 1 could also become efficient than the serial implementation. The design approach 1 also uses less number of processors than the approach 2 so it will consume less power than the approach 2.

From these two implementations we have learnt that the engineers should have to concentrate seriously on reducing processor stalls when designing the algorithms for the Ambric architecture in order to decrease the computation time and energy consumption.

7.2 Matrix Multiplication

In matrix multiplication algorithms we have targeted the algorithm scalability, I/O mapping and reduced power consumption on the Ambric architecture. We have evaluated different approaches of matrix multiplication and observed that due to the algorithm mapping in design approach 2 total cycle counts and processors stalls are reduced. So design approach 2 is more suitable for Ambric architecture.

The difference between these two results is because of the input and output streams provided. The design approach 1 requires eight *splitter* objects to provide data streams to four *multiplier* objects while design approach 2 requires only two *splitter* objects to provide data streams to four *multiplier* objects. The design approach 1 reads input matrices in two parallel streams and passes the data to other objects in serial order while in approach 2 input streams provided to *multiplier* objects are more parallelized. So the I/O provided to the design approach 2 is mapped more efficiently.

The algorithm design of approach 1 is scalable in such a way that we can add more composite objects by changing only one compile time parameter. For instance, if we change the order of matrix from 4 to 8 then this design will start multiplying 8x8 matrices. We can use this design where the order of matrices is not fixed and change frequently.

We can also build reusable design in *astruct* design language. This reusability is given by the leaf and composite interfaces. As in design approach 1 each composite object (represented by the C0, C1 and so on) is used again and again for calculating one row of the resultant matrix. But creating the reusable design as a whole is a very difficult task, to create a separate algorithm for the application design is very time consuming process. The *astruct* language is used to create application design only. It is not a full fledged programming language and provides only limited features so creating a reusable design will be very hard and time consuming.

As design approach 2 always requires less number of processors and less time than the design approach 1, it will certainly consume less energy.

7.3 Block Interleaver

We also have seen from the evaluations that applications like block interleaver are not suitable for this kind of architectures. In case of larger matrices the communication among processors will be increased very much than the computational power. As in block interleaver when size of matrix increases the total cycle counts and number of stalls also increases and the result will be very close to the serial implementation due to increase in channel communication between objects.

The design of both approaches is the same and number of processors used is equal, only technique used on the interleaver objects is different. As Matrix Interleaver requires less cycle counts and it will also requires less time to make the burst error into random error than the other technique, this algorithm mapping is energy efficient. The design of Block Interleaver shown in Figure 26 is more suitable for the Matrix Interleaver because it takes less cycle counts than the Helical Scan Interleaver.

7.4 Development Tools

A behavioural simulator, aSim, can execute and debug applications on a workstation without the chip. Java is compiled into assembly code by the standard Java compiler, and SR/SRD instruction set simulators execute assembly code. The simple combination of objects written in normal software code, combined in a hierarchy of block-diagram structures, makes high-performance design development much easier and cheaper.

The aSim simulator models the parallel object execution and channel behaviour. A simulated annealing placer maps primitive objects to resources on the target chip, and a PathFinder router assigns the structure's channels to specific hardware channels in the interconnect. Most applications are compiled in less than one minute. The most densely packed cases still compile in less than five minutes. As with software development, the design, debug, edit and rerun cycle is nearly interactive.

The object-based modularity of the *structural object programming model* facilitates the design reuse. For instance, in FFT design approach 2 we have created only one object for the butterfly computation and we are re-using it again and again. And in Matrix Multiplication design approach 1, one composite object (CO) is created and the same code of this composite object is running on each processor. In this way, divide and conquer technique will be very useful as there is no scheduling, no sharing and no multithreading. As channels are self-synchronized it means no interconnect scheduling is required. The inter-processor communication and synchronization is simple. Sending and receiving a word through a channel is so simple, just like reading or writing a processor register. This kind of development is much easier and cheaper and achieves long-term scalability, performance and power advantages of massive parallelism.

7.5 Limitations in Development Tools

There are some limitations in the current version of development tools (version 1.0) exist at the time of working on this thesis; these are highlighted in this section.

Sometimes it is required to create an array of properties to pass the information to objects at compile time, but we can not pass array of property to java objects. For instance, in FFT design approach 1 it is required to pass more than one twiddle factor value to objects which are computing more than one butterfly but we could not pass array of properties to *ajava* or assembly code. If array of property is provided then we can store twiddle factors within objects at compile time and some communication overhead could be reduced.

We can not create an array of input or output ports for the leaf objects; with this restriction we can not create easily a generalized application design. To understand this, consider a scenario: If we want to connect input or output ports at compile time within loop, or we want to decide at compile time whether one object is communicating with one or more objects then we can not do it currently in aDesigner. For example, in FFT design approach 2, each butterfly stage has four objects and the arrangement of communication channels is different among these stages. So we

have created channels statically among these objects, the orientation of channels can not be decided at runtime.

The *astruct* language is used to design applications but there is no mechanism for debugging this design. In case of complex application design it is required to have a debugger. For example, the *astruct* language provides a generate method with the help of which we can create in some way a scalable application design. We can use conditional statements and other programming loops within the *generate* method. We have created a scalable design for the matrix multiplication design approach 1. At the time of development we face some problems but can not debug this design, while manual debugging is very time consuming.

7.6 Summary

The aim of this thesis was to evaluate whether massively parallel processor architecture is suitable for the baseband signal processing in radio base stations. In the evaluation of different algorithms it is analyzed that high performance can be achieved by efficiently mapping the algorithms on the Ambric architecture. The Ambric chip is evaluated in terms of computational performance, efficiency of the development tools, algorithm and I/O mapping. Implementations of Matrix Multiplication, FFT and Block Interleaver were performed. Different mappings of the algorithms are compared to see which best fit the architecture.

7.7 Future work

Future work with this processor architecture would be interesting to implement algorithms on larger processor clusters to be able to study the true scaling of the architecture. It would be better to analyze the performance of the algorithms implemented in this thesis on the actual hardware device so that we can evaluate the hardware tools provided by the Ambric such as hardware debugger and on device performance analysis tools.

8 References

1. M. Butts, A. M. Jones, P. Wasson, Beaverton, Oregon, “A Structural Object Programming Model, Architecture, Chip and Tools for Reconfigurable Computing” 2007 International Symposium on Field-Programmable Custom Computing Machines.

2. “Ambric Technology Backgrounder” <http://www.ambric.com/technology/technology-overview.php> Date 02-05-2008.

3. M. Butts. “Synchronization through Communication in a Massively Parallel Processor Array”. IEEE Micro, vol. 27 no. 5, pp. 32-40, Sept./Oct. 2007.

4. White Paper Ericsson “Long Term Evolution (LTE): an introduction” October 2007. <http://www.ericsson.com/technology/whitepapers/lte_overview.pdf> Date 02-05-2008.

5. “AN 480: 1536-Point FFT for 3GPP Long Term Evolution”, ver.1.0 October 2007 <www.altera.com/literature/an/an480.pdf> Date 02-05-2008.

6. M. Rumney, “3GPP LTE: Introducing Single-Carrier FDMA”, Agilent Technologies, <http://cp.literature.agilent.com/litweb/pdf/5989-7898EN.pdf > Date 02-05-2008.

7. V. D. Nguyen, H. P. Kuchenbecker, ”Block interleaving for soft decisioln Viterbi decoding in OFDM systems” University of Hannover, Institut fur Allgemeine achrichtentechnik.

8. Á. Moravánszky, NovodeX AG, “Dense Matrix Algebra on the GPU” <http://www.shaderx2.com/shaderx.PDF > Date 02-05-2008.

9. S. Jenkins “MIMO/OFDM Implement OFDMA, MIMO for WiMAX, LTE” picoChip <http://www.eetasia.com/ARTICLES/2008MAR/PDF/EEOL_2008MAR17_RFD_NETD_TA.pdf?SOURCES=DOWNLOAD> Date 02-05-2008.

10. J. G. Proakis, D. G. Manolakis, "Digital Signal Processing, principles, algorithms and applications, 2nd ed.", Macmillan Publishing Company, 1992.

11. M. A. Kousa, “PERFORMANCE OF TURBO CODES WITH MATRIX INTERLEAVERS” Department of Electrical Engineering King Fahd University of Petroleum and Minerals Dhahran, Saudi Arabia. October 2003. <http://www.kfupm.edu.sa/publications/ajse/Articles/282B_07P.pdf> Date 02-05-2008.

12. B. Bylin and R. Karlsson, “Extreme processor for extreme processing”, Technical Report at Halmstad University, IDE0503, January 2005.

13. P. Söderstam, “STAP Signal Processing Algorithms on Ring and Torus SIMD Arrays”, Master thesis at Chalmers University of Technology. April 1998.

14. J. L. Tripp, J. Frigo, P. Graham, “A Survey of Multi-Core Coarse-Grained Reconfigurable Arrays for Embedded Applications”, Los Alamos National Labs <http://www.ll.mit.edu/HPEC/agendas/proc07/Day3/08_Tripp_Abstract.pdf> Date 02-05-2008.

9 Appendix - Source Code

This section contains the most important parts of the source code produced in this thesis project. Each implementation is divided into two parts; the *astruct* code is used for the designing of the overall structure of the application and the *ajava* code is used for the actual implementation of the computational kernels. We have provided source code for only one design approach from all implemented algorithms.

Appendix A contains the API implementation, Appendix B contains the implementation of Matrix Multiplication design approach 2, Appendix C contains the implementation of FFT design approach 2 and Appendix D contains the implementation of Block Interleaver design approach 2.

9.1 Appendix A

9.1.1 Fixed point

```
/*************************************************************
 *
 * File: FixedPoint.java
 *
 * Description: This class deals with the operation on fixed point
 * numbers. The format of Fixed point numbers considered in this class
 * is Q8.24 (8-bits for signed integral part and 24-bits hold
 * fractional part) within 32-bit signed integer.
 * word length(WL) = QI(including 1 signed bit) + QF
 * The range of QI => -128 to 127
 * The range of QF => 1/(2^24) => 0.000000059604644775390625
 *
 *************************************************************/

public class FixedPoint {

            private int nbInt;  // no of bits in integral part
            private int nbFrac; // no of bits in fractional part
            private Math math = new Math();

            /**
             * Constructor creates FixedPoint object and it takes two
             * parameters to specify fixed point number formate eg; Q8.24
             * where nbInt=8 and nbFrac=24
             *
             * @param nbInt - no of bits in integral part
             * @param nbFrac - no of bits in fractional part
             */

            public FixedPoint(int nbInt, int nbFrac){
                        this.nbInt = nbInt;
                        this.nbFrac = nbFrac;
```

```
        }

        /**
         * add function adds two signed fixed point numbers and
         * returns their sum.
         * @param a - Fixed point number of the formate Q8.24
         * @param b - Fixed point number of the formate Q8.24
         * @return - Fixed point number of the formate Q8.24
         */

        public int add(int a, int b){
                  // add and store the result in aacumulator
                  math.addacc(a, b, Marker.FIRST_LAST);
                  // read accumulator for result and returns it.
                  return math.rdacc_sum(Marker.LAST);
        }


        /**
         * subtract function subtracts two signed fixed point numbers,
         * the 2nd parameter from 1st one and returns their difference.
         * @param a - Fixed point number of the formate Q8.24
         * @param b - Fixed point number of the formate Q8.24
         * @return - Fixed point number of the formate Q8.24
         */

        public int subtract(int a, int b){
                  // subtract and store the result in aacumulator
                  math.subacc(a, b, Marker.FIRST_LAST);
                  // read accumulator for result and returns it.
                  return math.rdacc_sum(Marker.LAST);
        }

        /**
         * multiply_32 function multiplies two signed fixed point 32-bit
         * numbers and returns the result.
         *
         * @param a - Fixed point number of the formate Q8.24
         * @param b - Fixed point number of the formate Q8.24
         * @return - Fixed point number of the formate Q8.24
         */

        public int multiply_32(int a, int b){
                  math.mult_32_32(a, b, Marker.LAST);
                  // reads high and low part of accumulator
                  int lo = math.rdacc_lo(Marker.MORE);
                  int hi = math.rdacc_hi(Marker.LAST);
                  hi = ((hi << nbInt) | (lo >>> nbFrac));
                  return hi;
        }

}
```

9.1.2 Binary log

```
/**
 * log2(n) computes the log base 2 of any number by assuming the
 * floor value

 * @param n - input number to calculate log base 2
 * @return - returns the floor value of log2 of n
 */

public int log2(int n) {
          int pos = 0;
          if (n >= 1<<16) { n >>= 16; pos += 16; }
          if (n >= 1<< 8) { n >>=  8; pos +=  8; }
          if (n >= 1<< 4) { n >>=  4; pos +=  4; }
          if (n >= 1<< 2) { n >>=  2; pos +=  2; }
          if (n >= 1<< 1) {           pos +=  1; }
          return ((n == 0) ? (-1) : pos);
}
```

9.1.3 Binary Power

```
/**
 * powerOf2(n) calculates the power of two (2^n)
 *
 * @param n - input to find power of two
 * @return - returns the value of 2^n
 */
public int powerOf2(int n) {
          if(n < 0 || n > 31)
                    return -1;
          if(n == 0)
                    return 1;
          return 2 << (n-1);
}
```

9.2 Appendix B

Source code for the Matrix Multiplication is provided in this section. We only have provided the source code for the 4x4 matrix multiplication of design approach 2 because the code logic is quite similar for the 8x8 matrix multiplication.

9.2.1 Source code for Matrix Multiplication Design Approach 2

The design of the application is presented here.

```
/*********************************************************
 *
 * File: DesignApproach2.design
 *
 * Description: design file for the matrix multiplication approach 2
 *
 *********************************************************/

design DesignApproach2 {
           Top top;
}

interface Top {}

binding TopBinding implements Top {

           void generate(){

                      Vio io = {numSources=2, numSinks=1};

                      int i;
                      int dim = 4; // dimension of matrix

                      Multiplier m1 = {propDim = dim};
                      m1.name = "mul1";
                      Multiplier m2 = {propDim = dim};
                      m2.name = "mul2";
                      Multiplier m3 = {propDim = dim};
                      m3.name = "mul3";
                      Multiplier m4 = {propDim = dim};
                      m4.name = "mul4";

                      SplitterA sa = {propDim = dim};
                      sa.name = "splitA";
                      SplitterB sb = {propDim = dim};
                      sb.name = "splitB";

                      Join4 join;
                      join.name = "myJoin";

                      channel c0 = {io.out[0], sa.inA};
                      channel c1 = {io.out[1], sb.inB};
```

```
                        channel c2 = {sa.outA1, m1.inA};
                        channel c3 = {sa.outA2, m2.inA};
                        channel c4 = {sa.outA3, m3.inA};
                        channel c5 = {sa.outA4, m4.inA};

                        channel c6 = {sb.outB1, m1.inB};
                        channel c7 = {sb.outB2, m2.inB};
                        channel c8 = {sb.outB3, m3.inB};
                        channel c9 = {sb.outB4, m4.inB};

                        channel c10 = {m1.out, join.in_1};
                        channel c11 = {m2.out, join.in_2};
                        channel c12 = {m3.out, join.in_3};
                        channel c13 = {m4.out, join.in_4};

                        channel c14 = {join.out, io.in[0]};
            }

}


/*****************************************************
 *
 * File: SplitterA.astruct
 *
 * Description: a leaf interface for SplitterA and this interface
 * has one input and four output stream
 *
 *****************************************************/

interface SplitterA {

            inbound inA;
            outbound outA1, outA2, outA3, outA4;

            property int propDim; // dimension of matrix
}

binding SplitterABinding implements SplitterA {

            implementation "SplitterA.java";
            attribute CompilerOptions(targetSR = true) on SplitterABinding;

}


/*****************************************************
 *
 * File: SplitterB.astruct
 *
 * Description: a leaf interface for SplitterB and this interface
 * has one input and four output stream
 *
 *****************************************************/
```

```
interface SplitterB {

            inbound inB;
            outbound outB1, outB2, outB3, outB4;

            property int propDim; // dimension of matrix
}

binding SplitterBBinding implements SplitterB {

            implementation "SplitterB.java";
            attribute CompilerOptions(targetSR = true) on SplitterBBinding;

}


/*********************************************************
 *
 * File: Join4.astruct
 *
 * Description: a leaf interface for Join4 and this interface
 * joins the four input stream into one output stream
 *
 *********************************************************/

interface Join4 {

            inbound in_1, in_2, in_3, in_4;
            outbound out;
}

binding JoinBinding implements Join4 {

            implementation "Join4.java";
            attribute CompilerOptions(targetSR = true) on JoinBinding;

}
```

The source code for the java is presented in the section below.

```
/*********************************************************
 *
 * File: SplitterA.java
 *
 * Description: java object for SplitterA. It read each row
 * of matrix A and broadcast it to all Multiplier object
 *
 *********************************************************/

public class SplitterA {

            private int dim;

            public SplitterA(int propDim){
                        dim = propDim;
```

```
		}

		public void run(InputStream<Integer> inA,
				OutputStream<Integer> outA1,
				OutputStream<Integer> outA2,
				OutputStream<Integer> outA3,
				OutputStream<Integer> outA4) {

			int val;
			for(int i=0; i<dim; i++) {
				for(int j=0; j<dim; j++) {
					val = inA.readInt();
					outA1.writeInt(val);
					outA2.writeInt(val);
					outA3.writeInt(val);
					outA4.writeInt(val);
				}
			}
		}
}


/*************************************************************
 *
 * File: SplitterB.java
 *
 * Description: java object for SplitterB.
 *  read matrix B row-wise but distribute it coloumn-wise
 *  so that each Mulitplier object will have corresponding
 *  coloumn number e.g; Mulitiplier 1 will have coloumn#1
 *  and Mulitiplier 2 will have coloumn#2 and so on
 *
 *************************************************************/

public class SplitterB {

		private int dim;

		public SplitterB(int propDim){
			dim = propDim;
		}

		public void run(InputStream<Integer> inB,
				OutputStream<Integer> outB1,
				OutputStream<Integer> outB2,
				OutputStream<Integer> outB3,
				OutputStream<Integer> outB4) {

			for(int i=0; i<dim; i++) {
				outB1.writeInt(inB.readInt());
				outB2.writeInt(inB.readInt());
				outB3.writeInt(inB.readInt());
				outB4.writeInt(inB.readInt());
			}
		}

}
```

```
/*******************************************************
 *
 * File: Join4.java
 *
 * Description: java object for Join.
 *
 *******************************************************/

public class Join4 {

           public Join4(){
           }

           public void run(InputStream<Integer> in_1,
                           InputStream<Integer> in_2,
                           InputStream<Integer> in_3,
                           InputStream<Integer> in_4,
                           OutputStream<Integer> out) {


                     out.writeInt(in_1.readInt());
                     out.writeInt(in_2.readInt());
                     out.writeInt(in_3.readInt());
                     out.writeInt(in_4.readInt());

           }

}


/*******************************************************
 *
 * File: Multiplier.java
 *
 * Description: java object for the matrix multiplication
 *
 *******************************************************/

public class Multiplier {

           private int dim;
           private int[] col;

           public Multiplier(int propDim){
                     dim = propDim;
                     col = new int[dim];
           }

           public void run(InputStream<Integer>inA,
                     InputStream<Integer>inB,
                     OutputStream<Integer> out) {

                     int i,j;
                     int val = 0;

                     for(i=0; i<dim; i++){
                           val = 0;
```

```
                        for(j=0; j<dim; j++){

                             if(i == 0) {
                                       col[j] = inB.readInt();
                             }
                             val +=  col[j] * inA.readInt();
                        }
                        out.writeInt(val);
                   }
          }
}
```

9.3 Appendix C

Source code for the FFT is provided in this section. We have only provided the source code for the 8-point FFT of design approach 2.

9.3.1 Source code for FFT Design approach 2

First, the design of application is presented here.

```
/*********************************************************
 *
 * File: FFT8Design.design
 *
 * Description: design file for the design approach two.
 * Each fft object will compute only one butterfly
 *
 *********************************************************/

design FFT8Design {
           Root root;
}

interface Root {}


binding RootBinding implements Root {

   void generate() {

           int N = 8;            // total no of points
           int nStages = 3;// total no of butterfly stages 2^(N)
           int nPoints = 2;// no of points calculated on each FFT object

           Vio io = {numSources=2, numSinks=2};

           /* each FFT object computes 2 points.
            * For this design we need {N/nPoints * log2(N)} FFT objects,
            * {N/nPoints - 1} Splitters and {N/nPoints - 1} Join
            */
           int i,k;
           int nSplitJoin = (N/nPoints-1);
           // no of fft objects to connect with splitter or join

           int nFft_onsides = N/nPoints;

           // no of inner fft objects in desgin
           int nFft = nStages*(N/nPoints)-N;

           // stage 1 objects
           FFT_2in4out fft24[nFft_onsides];
           // stage 4 objects
           FFT_4in2out fft42[nFft_onsides];
```

```
// stage 2 and 3 objects
FFT fft[nFft];
Splitter s[nSplitJoin];
Join j[nSplitJoin];

// stage 1 twiddle factors
int s1Cos = 16777216;
int s1Sin = 0;

// stage 2 twiddle factors - first value is for even
// points and second is for odd
int s2Cos[] = {16777216, 0};
int s2Sin[] = {0, -16777216};

// stage 3 twiddle factors
int s3Cos[] = {16777216, 11863683, 0,       -11863683};
int s3Sin[] = {0,       -11863683, -16777216, -11863683};

for(i=0; i<nFft_onsides; i++) {

            // objects for stage 1
            fft24[i].name = "FFT_2in4out"+i;
            fft24[i].propPoints=nPoints;
            fft24[i].propCosVal = s1Cos;
            fft24[i].propSinVal = s1Sin;

            // objects for stage 4
            fft42[i].name = "FFT_4in2out"+i;
            fft42[i].propPoints=nPoints;
            fft42[i].propCosVal = s3Cos[i];
            fft42[i].propSinVal = s3Sin[i];
}

int index=0;
for(i=0; i<nFft; i++) {

            fft[i].name = "FFT"+i;
            fft[i].propPoints=nPoints;
            fft[i].propCosVal = s2Cos[index];
            fft[i].propSinVal = s2Sin[index];
            index=index+1;

            if(index==2)
                index=0;
}

for(i=0; i<nSplitJoin; i++) {

            s[i].name = "split"+i;
            s[i].propPoints=N;
            j[i].name = "join"+i;
            j[i].propPoints=N;
}

s[0].propN = N;
s[1].propN = N/2;
s[2].propN = N/2;
```

```
j[0].propN = N/4;
j[1].propN = N/4;
j[2].propN = N/2;

// connect Vio object with first splitter and last join objects
channel c10 = {io.out[0], s[0].inReal};
channel c11 = {io.out[1], s[0].inImg};
channel c12 = {j[nSplitJoin-1].outReal, io.in[0]};
channel c13 = {j[nSplitJoin-1].outImg, io.in[1]};

// connecting splitters
channel c14 = {s[0].outReal1, s[1].inReal};
channel c15 = {s[0].outImg1, s[1].inImg};
channel c16 = {s[0].outReal2, s[2].inReal};
channel c17 = {s[0].outImg2, s[2].inImg};

// connecting joins
channel c114 = {j[0].outReal, j[2].inReal1};
channel c115 = {j[0].outImg, j[2].inImg1};
channel c116 = {j[1].outReal, j[2].inReal2};
channel c117 = {j[1].outImg, j[2].inImg2};

// connect splitters with FFT_2in4out
k = 1;
for(i=0; i<N/nPoints; i=i+2){

            channel c27 = {s[k].outReal1, fft24[i].inReal};
            channel c28 = {s[k].outImg1, fft24[i].inImg};
            channel c29 = {s[k].outReal2, fft24[i+1].inReal};
            channel c30 = {s[k].outImg2, fft24[i+1].inImg};

            // butterfly between stage 1 and 2
            channel c31 = {fft24[i].outReal1, fft[i].inReal1};
            channel c32 = {fft24[i].outImg1, fft[i].inImg1};
            channel c33 = {fft24[i].outReal2, fft[i+1].inReal2};
            channel c34 = {fft24[i].outImg2, fft[i+1].inImg2};

            channel c35 = {fft24[i+1].outReal1, fft[i+1].inReal1};
            channel c36 = {fft24[i+1].outImg1, fft[i+1].inImg1};
            channel c37 = {fft24[i+1].outReal2, fft[i].inReal2};
            channel c38 = {fft24[i+1].outImg2, fft[i].inImg2};

            k=k+1;
}

// connect join with FFT_4in2out
k = 0;
for(i=0; i<N/nPoints; i=i+2){
            channel c39 = {fft42[i].outReal, j[k].inReal1};
            channel c40 = {fft42[i].outImg, j[k].inImg1};
            channel c41 = {fft42[i+1].outReal, j[k].inReal2};
            channel c42 = {fft42[i+1].outImg, j[k].inImg2};
            k=k+1;
}

// connect FFT_4in2out with FFT between stage 2 and 3
channel c43 = {fft[0].outReal1, fft42[0].inReal1};
```

```
            channel c44 = {fft[0].outImg1, fft42[0].inImg1};
            channel c45 = {fft[0].outReal2, fft42[2].inReal2};
            channel c46 = {fft[0].outImg2, fft42[2].inImg2};

            channel c47 = {fft[1].outReal1, fft42[1].inReal1};
            channel c48 = {fft[1].outImg1, fft42[1].inImg1};
            channel c49 = {fft[1].outReal2, fft42[3].inReal2};
            channel c50 = {fft[1].outImg2, fft42[3].inImg2};

            channel c51 = {fft[2].outReal1, fft42[2].inReal1};
            channel c52 = {fft[2].outImg1, fft42[2].inImg1};
            channel c53 = {fft[2].outReal2, fft42[0].inReal2};
            channel c54 = {fft[2].outImg2, fft42[0].inImg2};

            channel c55 = {fft[3].outReal1, fft42[3].inReal1};
            channel c56 = {fft[3].outImg1, fft42[3].inImg1};
            channel c57 = {fft[3].outReal2, fft42[1].inReal2};
            channel c58 = {fft[3].outImg2, fft42[1].inImg2};

    }
}


/*********************************************************
 *
 * File: Splitter.astruct
 *
 * Description: Splitter interface and it contains two input
 *  ports and four output ports
 *
 *********************************************************/

interface Splitter {

            // input port for real signal
            inbound inReal;

            // input port for imaginary signal
            inbound inImg;

            outbound outReal1, outImg1;
            outbound outReal2, outImg2;

            // total no of point in FFT
            property int propPoints;

            // no of points to read from previous splitter
            property int propN;
}

binding SplitterBinding implements Splitter {

            implementation "Splitter.java";
            attribute CompilerOptions(targetSR = true) on SplitterBinding;

}
package com.fft;
```

```
/*********************************************************
 *
 * File: Join.astruct
 *
 * Description: Join interface and it contains four input
 *  ports and two output ports
 *
 *********************************************************/

interface Join {

           // input ports for real and imaginary signal
           inbound inReal1, inImg1;
           inbound inReal2, inImg2;

           // output ports for real and imaginary signal
           outbound outReal, outImg;

           // total no of points in FFT
           property int propPoints;

           // no of points to read from previous join
           property int propN;
}

binding JoinBinding implements Join {

           implementation "Join.java";
           attribute CompilerOptions(targetSR = true) on JoinBinding;

}


/*********************************************************
 *
 * File: FFT.astruct
 *
 * Description: FFT interface, it reads two points from two
 * different channels and writes two points to two different channels
 *
 *********************************************************/


interface FFT {

           inbound inReal1, inImg1;
           inbound inReal2, inImg2;
           outbound outReal1, outImg1;
           outbound outReal2, outImg2;

           property int propPoints;
           property int propCosVal;
           property int propSinVal;

}
```

```
binding FFTBinding implements FFT {

           implementation "FFT.java";

}


/*********************************************************
 *
 * File: FFT_4in2out.astruct
 *
 * Description: this interface reads two points from two
 * different channels and writes two points on one channels
 *
 *********************************************************/

interface FFT_4in2out {

           inbound inReal1, inImg1;
           inbound inReal2, inImg2;
           outbound outReal, outImg;

           property int propPoints;
           property int propCosVal;
           property int propSinVal;

}

binding FFT_4in2outBinding implements FFT_4in2out {

           implementation "FFT_4in2out.java";

}


/*********************************************************
 *
 * File: FFT_2in4out.astruct
 *
 * Description: this interface reads two points from one
 * channels and writes two points to two different channels
 *
 *********************************************************/

interface FFT_2in4out {

           inbound inReal, inImg;
           outbound outReal1, outImg1;
           outbound outReal2, outImg2;

           property int propPoints;
           property int propCosVal;
           property int propSinVal;

}
```

```
binding FFT_2in4outBinding implements FFT_2in4out {

            implementation "FFT_2in4out.java";

}
```

The source code for the java is presented in the section below.

```
/*********************************************************
 *
 * File: Splitter.java
 *
 * Description: this java object distribute points in even
 *  and odd order for bit-reversal sorting
 *
 *********************************************************/

public class Splitter {

            // no of points to read from previous splitter object
            private int N;

            public Splitter(int propN){
                        this.N = propN;
            }

            public void run(InputStream<Integer> inReal,
                                    InputStream<Integer> inImg,
                                    OutputStream<Integer> outReal1,
                                    OutputStream<Integer> outImg1,
                                    OutputStream<Integer> outReal2,
                                    OutputStream<Integer> outImg2) {


                        for(int i=0; i<N; i+=2) {
                                    // send even point to left output stream
                                    outReal1.writeInt( inReal.readInt() );
                                    outImg1.writeInt( inImg.readInt() );
                                    // send odd point to left output stream
                                    outReal2.writeInt( inReal.readInt() );
                                    outImg2.writeInt( inImg.readInt() );
                        }
            }

}



/*********************************************************
 *
 * File: Join.java
 *
 * Description: this java object combines total no of points
```

```
 *         and finally write to output
 *
 ************************************************************/

public class Join {

           // no of points to read from previous splitter object
           private int N;

           public Join(int propN){
                      this.N = propN;
           }

           public void run(InputStream<Integer> inReal1,
                                 InputStream<Integer> inImg1,
                                 InputStream<Integer> inReal2,
                                 InputStream<Integer> inImg2,
                                 OutputStream<Integer> outReal,
                                 OutputStream<Integer> outImg) {


                      for(int i=0; i<N; i++) {
                                 outReal.writeInt( inReal1.readInt() );
                                 outImg.writeInt( inImg1.readInt() );
                      }
                      for(int i=0; i<N; i++) {
                                 outReal.writeInt( inReal2.readInt() );
                                 outImg.writeInt( inImg2.readInt() );
                      }
           }

}


/************************************************************
 *
 * File: FFT.java
 *
 * Description: this java object associated with FFT.astruct
 *  interface. This object performs one butterfly computation
 *
 ************************************************************/

public class FFT {

           private int N;          // no of points

           // array of complex numbers
           private Complex[] x;

           // twiddle factors
           private final int cos;
           private final int sin;

           // for fixed point calculations
           private FixedPoint fp = new FixedPoint(8,24);
```

```
            public FFT(int propPoints, int propCosVal, int propSinVal){
                        N = propPoints;
                        x = new Complex[N];
                        cos = propCosVal;
                        sin = propSinVal;
            }


        public void run(InputStream<Integer> inReal1,
                        InputStream<Integer> inImg1,
                        InputStream<Integer> inReal2,
                        InputStream<Integer> inImg2,
                        OutputStream<Integer> outReal1,
                        OutputStream<Integer> outImg1,
                        OutputStream<Integer> outReal2,
                        OutputStream<Integer> outImg2) {


            int i;

            for(i=0; i<N; i+=2){ // read real and imaginary signal
                        x[i].real = inReal1.readInt();
                        x[i].img = inImg1.readInt();
                        x[i+1].real = inReal2.readInt();
                        x[i+1].img = inImg2.readInt();
            }

            // Butterfly calculation using fixed point numerics
            int TR = fp.subtract(fp.multiply_32(x[1].real, cos),
                                                fp.multiply_32(x[1].img, sin));

            int TI = fp.add(fp.multiply_32(x[1].real, sin),
                                                fp.multiply_32(x[1].img, cos));

            x[1].real = fp.subtract(x[0].real, TR);
            x[1].img = fp.subtract(x[0].img, TI);
            x[0].real = fp.add(x[0].real, TR);
            x[0].img = fp.add(x[0].img, TI);

            for(i=0; i<N; i+=2) {
                        outReal1.writeInt(x[i].real);
                        outImg1.writeInt(x[i].img);
                        outReal2.writeInt(x[i+1].real);
                        outImg2.writeInt(x[i+1].img);
            }
      }

}
```

9.4 Appendix D

Source code for the Block Interleaver is provided in this section. We have only provided the source code for the Helical Scan Interleaver for the input size of 4x4.

9.4.1 Source code for Helical Scan Interleaver

First, the design of application is presented here.

```
/*********************************************************
 *
 * File: Helical4.design
 *
 * Description: design file for the Helical Scan Interleaver
 *  for the input size of 4x4 matrix
 *
 *********************************************************/

design Helical4 {
           Top top;
}

interface Top {}

binding TopBinding implements Top {

           void generate(){

                     Vio io = {numSources=1, numSinks=1};

                     int row = 6; // no of rows for input matrix

                     Interleaver m1 = {propId = 0, propRows = row};
                     m1.name = "BI1";
                     Interleaver m2 = {propId = 1, propRows = row};
                     m2.name = "BI2";
                     Interleaver m3 = {propId = 2, propRows = row};
                     m3.name = "BI3";
                     Interleaver m4 = {propId = 3, propRows = row};
                     m4.name = "BI4";

                     SplitterA sa = {propRows = row};
                     sa.name = "splitA";

                     Join4 join = {propRows = row};
                     join.name = "join";

                     channel c0 = {io.out[0], sa.inA};

                     channel c2 = {sa.outA1, m1.in};
                     channel c3 = {sa.outA2, m2.in};
                     channel c4 = {sa.outA3, m3.in};
```

```
                        channel c5 = {sa.outA4, m4.in};

                        channel c10 = {m1.out, join.in_1};
                        channel c11 = {m2.out, join.in_2};
                        channel c12 = {m3.out, join.in_3};
                        channel c13 = {m4.out, join.in_4};

                        channel c14 = {join.out, io.in[0]};
            }

}


/*************************************************************
 *
 * File: SplitterA.astruct
 *
 * Description: this splitter interface reads a single input
 *  stream and divides it into four output streams
 *
 *************************************************************/

interface SplitterA {

            inbound inA;
            outbound outA1, outA2, outA3, outA4;

            property int propRows;
}

binding SplitterABinding implements SplitterA {

            implementation "SplitterA.java";
            attribute CompilerOptions(targetSR = true) on SplitterABinding;

}


/*************************************************************
 *
 * File: Join4.astruct
 *
 * Description: this join interface joins the four input
 *  streams into a single output stream
 *
 *************************************************************/

interface Join4 {

            inbound in_1, in_2, in_3, in_4;
            outbound out;

            property int propRows;

}
```

```
binding JoinBinding implements Join4 {

            implementation "Join4.java";
            attribute CompilerOptions(targetSR = true) on JoinBinding;

}


/*********************************************************
 *
 * File: Interleaver.astruct
 *
 * Description: this interface reads single input stream
 *  re-order it and sends to output.
 *
 *********************************************************/

interface Interleaver {

            inbound in;
            outbound out;

            property int propId;
            property int propRows;

}

binding BInterleaver implements Interleaver {

            implementation "Interleaver.java";
            attribute CompilerOptions(targetSR = true) on BInterleaver;

}
```

The source code for the java is presented in the section below.

```
/*********************************************************
 *
 * File: SplitterA.java
 *
 * Description: this java object reads rows of matrix one by
 * one and distribute elements of each row to the
 * corresponding interleaver
 *
 *********************************************************/

public class SplitterA {

            private int nRows;

            public SplitterA(int propRows){
                        nRows = propRows;
            }
```

```
            public void run(InputStream<Integer> inA,
                            OutputStream<Integer> outA1,
                            OutputStream<Integer> outA2,
                            OutputStream<Integer> outA3,
                            OutputStream<Integer> outA4) {


                        for(int i=0; i<nRows; i++) {
                            outA1.writeInt(inA.readInt());
                            outA2.writeInt(inA.readInt());
                            outA3.writeInt(inA.readInt());
                            outA4.writeInt(inA.readInt());
                        }
            }

}


/*********************************************************
 *
 * File: Join4.java
 *
 * Description: this java object collects the data from the
 *  interleaver objects
 *
 *********************************************************/

public class Join4 {

            private int nRows;

            public Join4(int propRows){
                        nRows = propRows;
            }

            public void run(InputStream<Integer> in_1,
                            InputStream<Integer> in_2,
                            InputStream<Integer> in_3,
                            InputStream<Integer> in_4,
                            OutputStream<Integer> out) {

                        for(int i=0; i<nRows; i++) {
                            out.writeInt(in_1.readInt());
                            out.writeInt(in_2.readInt());
                            out.writeInt(in_3.readInt());
                            out.writeInt(in_4.readInt());
                        }
            }

}


/*********************************************************
 *
 * File: Interleaver.java
```

```
 *
 * Description: this java object performs the re-ordering
 *  on single row of the input matrix in a helical fashion.
 *
 ********************************************************/

public class Interleaver {

            private int nRows;
            private int id;
            private int m[];

            public Interleaver(int propId, int propRows){
                        id = propId;
                        nRows = propRows;
                        m = new int[nRows];
            }

            public void run(InputStream<Integer>in, OutputStream<Integer> out)
{
                        int i,j;
                        for(i=0; i<nRows; i++){
                                    m[i] = in.readInt();
                        }

                        i = id;
                        for(j=0; j<nRows; j++){
                                    out.writeInt(m[i]);
                                    i++;
                                    if(i==nRows)
                                                i=0;
                        }
            }

}
```