Radar Signal Processing on Ambric Platform

Master's Thesis in Computer Systems Engineering

Yadagiri Pyaram & Md Mashiur Rahman

8-Point Input

Distributor Objects

Butterfly Operation

Assembler Objects

finalFFT

8-Point Input

D0

D1

B0

B4

B8

A1

Output

firstFFT

RealFFT

finalFFT

B1

B5

B9

A2

Distributor Objects

B2

B6

B10

B3

B7

B11

D2

D0
Radar Signal Processing on Ambric Platform
Master Thesis in Computer Systems Engineering

School of Information Science, Computer and Electrical Engineering
Halmstad University
Box 823, S-301 18 Halmstad, Sweden

September 2010
**Figure on cover page**: Design approach for one of algorithm FFT from page no.41, Figure 15: 8-Point FFT

Design Approach
Acknowledgements

This Master’s thesis is the part of Master’s Program for the fulfilment of Master’s Degree specialization in Computer Systems Engineering at Halmstad University. In this project the Ambric processor developed by Ambric Inc. has been used.

Behalf of completion of thesis, we are glad to our supervisors, Professor Bertil Svensson and Zain-ul-Abdin Ph.D. Student, Dept of IDE, Halmstad University for their valuable guidance and support throughout this thesis project and without them it would be difficult to complete our thesis. And we are very glad to the Jerker Bengtsson Dept of IDE, Halmstad University, who gave valuable suggestions in the meetings, which helped us.

And we would like to thank the Library personnel who helped some stuff in documentation and also to Eva Nestius Research Secretary, IDE Division University of Halmstad for providing access to Laboratory.

And we would like to thank our friends Mr. Albert, Husni, Sulayman, and Ajay Kumar who shared their ideas with us, which helped us to achieve our desired work.

Yadagiri Pyaram & Md Mashiur Rahman
Halmstad University, September 2010
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 with efficient algorithms.

The aim of this thesis was to evaluate the use of parallel processor architecture in Radar signal processing applications where the processor has to compute complex calculations. This has been done by implementing demanding algorithms on Ambric Am2000 family Massively Parallel Processor Array (MPPA). The Ambric platform evaluated in terms of Latency, Cycle Count per output sample and Efficiency of the development tools.

The two algorithms chosen for implementation are Fast Fourier Transform (FFT) and Finite Impulse Response (FIR) algorithms. We have implemented parameterized versions of FFT and FIR. The FFT algorithm implemented for N-point input for the range of 8 point to 32 point for complex input variables. It works for any given number of inputs within the range for given parameter values and mapped on Ambric processor with fixed point radix - 2. Another one is FIR algorithm for the range of 12 Taps to 64 Taps for complex input variables. The Implementation of algorithms shows that high level of parallelism can be achieved in Massively Parallel Processing Arrays (MPPA) especially on complex algorithms like FFT and FIR.
List of figures

Figure 1: Block diagram of Radar Signal Processing .................................................17
Figure 2: Signal Processing Chain ..........................................................................21
Figure 3: Structure of Object programming model ....................................................26
Figure 4: Ambric Channels and Registers .................................................................27
Figure 5: Processor Architecture ..............................................................................28
Figure 6: Ambric Chip ...............................................................................................29
Figure 7: CU-RU pair cluster of four Ambric processors .........................................30
Figure 8: Brics and their interconnection .................................................................32
Figure 9: SR Processor .............................................................................................33
Figure 10: SRD Processor .........................................................................................34
Figure 11: Butterfly Computation of Radix-2 with decimation in time .......................36
Figure 12: 8-point FFT Butterfly using Radix-2 decimation in time ..........................36
Figure 13: Schematic diagram for FIR .................................................................38
Figure 14: N-point FFT bit reversal sorting of distribute object ...............................40
Figure 15: 8-point FFT design approach ...............................................................41
Figure 16: Design approach for FIR algorithm for 12 Taps .....................................43
Figure 17: Working of PerfHarness tools ................................................................44
Figure 18: Screenshot of performance analyzer ......................................................47
Contents

ACKNOWLEDGEMENTS ....................................................................................................................................................... 7

ABSTRACT ........................................................................................................................................................................... 9

1 INTRODUCTION ............................................................................................................................................................... 15

2 RADAR SIGNAL PROCESSING ......................................................................................................................................... 17

2.1 DESCRIPTION OF BLOCK DIAGRAM ...................................................................................................................... 17

2.2 CHALLENGES IN RADAR SIGNAL PROCESSING .................................................................................................. 20

2.3 SIGNAL PROCESSING CHAIN .................................................................................................................................. 20

2.3.1 Pulse compression ..................................................................................................................................................... 21

2.3.2 Velocity Compensation ............................................................................................................................................. 22

2.3.3 MTI (Moving Target Indicator) filter .................................................................................................................... 22

2.3.4 Doppler filter ........................................................................................................................................................... 22

2.3.5 Envelope Detector .................................................................................................................................................. 22

2.3.6 Detection (CFAR) ................................................................................................................................................... 22

2.3.7 Resolving ................................................................................................................................................................ 23

3 INTRODUCTION TO AMBRIC PLATFORM ..................................................................................................................... 25

3.1 AMBRIC ARCHITECTURE ........................................................................................................................................... 25

3.2 STRUCTURAL OBJECT PROGRAMMING MODEL .................................................................................................. 25

3.3 AMBRIC REGISTERS AND CHANNELS .................................................................................................................. 26

3.4 HARDWARE OBJECTS AND CLOCKING ............................................................................................................. 27

3.5 PROCESSOR ARCHITECTURE .................................................................................................................................. 28

3.6 AMBRIC CHIP .............................................................................................................................................................. 29

3.7 AMBRIC COMPUTE UNIT AND RAM UNIT ........................................................................................................... 30

3.8 BRICS AND INTERCONNECTIONS ............................................................................................................................... 31

3.9 SR AND SRD PROCESSORS ....................................................................................................................................... 32

4 ALGORITHMS OVERVIEW ............................................................................................................................................... 35

4.1 FAST FOURIER TRANSFORM (FFT) ALGORITHM ...................................................................................................... 35

4.1.1 Radix-2 FFT .......................................................................................................................................................... 36

4.1.2 Complexity analysis of radix-2 FFT .................................................................................................................... 36

4.2 FINITE IMPULSE RESPONSE (FIR) ALGORITHM ........................................................................................................ 37

5 ALGORITHMS IMPLEMENTATION ................................................................................................................................ 39

5.1 FAST FOURIER TRANSFORM .................................................................................................................................. 39

5.2 FINITE IMPULSE RESPONSE ................................................................................................................................. 42

5.3 DEVELOPMENT TOOLS .............................................................................................................................................. 44
5.4 PERFORMANCE TESTING TOOLS ................................................................................................................. 44

6 RESULTS AND ANALYSIS OF SOLUTIONS ................................................................................................. 45
  6.1 FFT RESULTS ANALYSIS .......................................................................................................................... 45
  6.2 FIR RESULTS ANALYSIS .......................................................................................................................... 48

7 CONCLUSIONS AND FUTURE WORK ................................................................................................. 51

8 REFERENCES ............................................................................................................................................... 53

9 APPENDIX - SOURCE CODE ..................................................................................................................... 55
  9.1 APPENDIX A ............................................................................................................................................ 55
    9.1.1 Fixed Point ....................................................................................................................................... 55
  9.2 APPENDIX B ............................................................................................................................................. 56
    9.2.1 Source code for FFT ......................................................................................................................... 57
  9.3 APPENDIX C ............................................................................................................................................. 68
    9.3.1 Source code for FIR Algorithm ...................................................................................................... 68
Nowadays, the communication systems are playing a main role in our daily life. There is rapid development in modern communication systems and proportionally, system demands are also increasing. Due to faster development in the digital signal processing techniques, the communication system design is becoming easier. Digital signal processing is the analysis, interpretation, and manipulation of signals. Signal processing is used in many 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. Signal specification plays a key role both in selecting the appropriate system architecture and determining the necessary computational speed of all involved algorithms.

In this thesis, we mainly considered about Radar signal processing applications. Radar signal processing consists of many sub-stages. After receiving the electromagnetic signal by the receiver antenna, it is processed through the signal processor. It consists of different stages like Pulse Compression, Velocity Compression, MTI (Moving Target Identification) filter, Doppler filter, Envelop Creation, Detection and Resolving. In our thesis, we worked at two stages, one is Pulse Compression stage where the Finite Impulse Response algorithm is used and another stage is Doppler Filter where Fast Fourier Transform algorithm is used for filtering the received pulses.

In Radar signal processing, the processor works on many complex algorithms. And because of it being a real-time system, there should be efficient, timely communication and processing systems for quick update of data and there should not be delay in communication between target and transceiver. To overcome this, there is a need for efficient algorithms. The main aim of this thesis is to implement two efficient algorithms used in Radar Signal Processing. One is the Fast Fourier Transform (FFT) algorithm and another one is Finite Impulse Response (FIR) algorithm. These algorithms were mapped on Ambric Processor. Here the Ambric processor is Massively Parallel Processing Arrays (MPPA), mostly used in embedded applications where there is a need for more parallel processing of data.

We developed two parameterized versions of algorithms. One is parameterized Fast Fourier Transform algorithm; it can work for the range of 8 points to 32 points for a given parameter value in the program, so that it can be used for different ranges of input values. Another one is parameterized Finite Impulse Response algorithm; it can work for different ranges of taps from 12 taps to 64 taps. In this thesis work, we used aDesigner tools provided by Ambric Inc., and the programming language is java which is a subset of Java. Then we have evaluated the results in terms of latency, cycle count per output sample and efficiency of the development tools.

The thesis report is organized in several chapters. In chapter 2, we discussed clearly about Radar Signal Processing theoretical background like definitions of different important equipments used, and Radar Signal Processing Block diagram and how it is processing signals. Here important section is the block diagram of Signal Processing Chain, where we explained about internal blocks and its functions which play a main role in whole radar signal processing.
Introduction

The chapter 3 explains clearly about Ambric platform on which FFT and FIR algorithms are mapped. In this section we explained about Ambric programming model, its architectural design, chip configuration, registers used in Ambric, Channels in Ambric and Performance testing tools.

And chapter 4 describes about algorithms overview. It contains design approach for FFT and FIR algorithms and its mathematical descriptions which gives more insight in to the algorithms.

Similar way in chapter 5 we discussed about implementation of algorithms. That is, programmatic concepts for implementing FFT and FIR algorithms, its programming model and explanation.

The chapter 6 gives information about simulation results of algorithms like, elaborated programming model that generated after simulation. Moreover this chapter shows the results for different versions of FFT and FIR in the tabulated form and these results have been analysed. Conclusions and suggestions for future work will be presented in chapter 7. After this references are provided and then at the end of the report, the Appendix is provided in which total program have been presented.
2 Radar signal processing

The Radar is defined as an object detection system that uses electromagnetic waves to identify the range, altitude, direction, or speed of both moving and fixed objects such as aircraft, ships, motor vehicles, weather formations, and terrain. The term RADAR was coined in 1940 by the U.S. Navy as an acronym for Radio Detection And Ranging [1].

The fundamental nature of signal processing is combination of its theory, efficient computational algorithms, and the implementation of these algorithms in hardware. The development of radar signal processing field were initially driven by the need to provide detected and processed signals for air and ballistic missile defence systems. The first processing work was on the Semi-Automatic Ground Environment (SAGE) air-defence system, which led to algorithms and techniques for detection of aircraft in the presence of clutter. This theoretical foundation was initially applied to programs in air defence. Soon, however, the rigid needs of ballistic missile defence required the application of both signal processing theory and practice. Later, signal processing requirements from fields as various as air traffic control, space surveillance, and tactical battlefield supervision. These stimulated the development and implementation of powerful new signal processing techniques and technology [1].

Block diagram of Radar Signal Processing

2.1 Description of block diagram

This section gives overview of Radar signal processing, its internal stages and the process of data. In its most elementary form, radar consists of five elements: a radio transmitter, a radio receiver tuned to the transmitter’s frequency, two antennas, and a display to detect the presence of an object (target). The transmitter generates radio waves, which are radiated by one of the antennas; meanwhile the receiver listens for the “echoes” (electromagnetic energy) of these waves, which are picked up by the other antenna. If a target is detected, a blip indicating its location appears on the display. As shown in the Figure 1, most objects—aircraft, ships, vehicles, buildings, features of the terrain, etc. reflect the radio waves much as they do light. Radio waves and light is, in fact the same thing - the flow of electromagnetic energy. This received signal will be send to the signal processor where it converted in to digital form by A/D converter. This signal will be processed for filtering and
Radar signal processing

compressing the data. Here the thesis focus on signal processor stage in which we developed parameterized version algorithms (we will discuss about this later in next section). Finally after processing the data it will be displayed on the screen where the human being could observe the altitude and position of target.

Antenna

In general radar applications an antenna consists of a radiator and a parabolic shaped reflector called as dish. In simple radars, the antenna generally consists of a radiator and a parabolic shaped reflector (dish), arranged on common support. The radiator is a horn shaped nozzle on the end of the wave guide coming from the duplexer. The horn directs the radio waves which are arriving from the transmitter onto the dish, which reflects the wave in the form of a narrow beam. Echoes intercepted by the dish are reflected into the horn and conveyed by the same waveguide back to the duplexer, hence to the receiver.

Modulator

For receipt of each timing pulse, the modulator produces a high power pulse of constant direct current (DC) energy and then it will be supplied to the transmitter.

Transmitter

The transmitter is an oscillator of high-power, like magnetron. For every duration of input pulse which is coming from the modulator, the magnetron generates a high-power radio-frequency wave. In this process the transmitter converts DC pulse to a radio-frequency energy wave. The wave length of the energy is typically around 3cm. The exact wave length may be fixed by the design of the magnetron or may be tuneable over a range of about 10% by the operator. The wave is radiated into a metal pipe called a waveguide, which conveys the wave it to the duplexer.

Receiver

Generally the receiver used is of heterodyne type, it translates the received signal to a lower frequency or higher band width at which they can be filtered and amplified more conveniently. Translation is accomplished by “beating” the received signals against the output of a low-power oscillator (called the local oscillator or LO) in a circuit called a mixer. The frequency of the resulting signal, called the intermediate frequency or IF is equals to the difference between the signal’s original frequency and the local oscillator frequency. The output of the mixer is amplified by a tuned circuit (IF amplifier). It filters out any interfering signals, as well as the electrical background noise lying outside the band of frequencies occupied by the received signal. Finally, the amplified signal is applied to a detector which produces an output voltage corresponding to the peak amplitude (or envelope) of the signal. It is similar to the signal that in a TV, which varies the intensity of the beam and paints the images on the picture tube. Consequently, the detector’s output is called a video signal. This signal is supplied to the indicator.

Indicator

The indicator displays the received echoes in a format that will satisfy the operator’s requirements. This helps to control the automatic searching, tracking functions and extract the desired target data. By using this data the target could be tracked. Here, any variety of display formats may be used. A video amplifier raises the receiver output to a level which is suitable for controlling the intensity of the cathode ray beam of the display tube. The operator generally sets the gain of the amplifier so that noise spikes make the beam barely visible. The target
Radar signal processing

Echoes are strong enough to be detected above the noise will then produce a bright spot, or “blip”. The vertical and horizontal positions of the beam are controlled as follow.

Each timing pulse from the synchronizer triggers the generation of a linearly increasing voltage that causes the beam to trace a vertical path from the bottom of the display to the top. Since the start of each trace is thus synchronized with the transmission of a radar pulse, if a target echo is received, the distance from the start of the trace to the point at which the target blip appears will correspond to the round-trip transit time for the echo, hence to the target’s range. For this reason the trace is called the range trace and the vertical motion of the beam, the range sweep. Meanwhile, the azimuth signal from the antenna is used to control the horizontal position of the range trace, and the elevation signal may be used to control the vertical position of a marker on the edge of the display, where an elevation scale is provided.

As the antenna executes its search scan, the range trace sweeps back and forth across the display in unison with the azimuth scan of the antenna. Each time the antenna beam sweeps across a target, a blip appears on the range trace, providing the operator with a plot of the range versus the azimuth of the target.

Signal Processor

The processor used in Radar signal processing is a digital computer. This processor specifically designed for radar applications for efficient performance on the huge number of repetitive additions, subtractions, and multiplications required for real-time signal processing. The data processor loads the program in to temporary memory for the currently selected mode of operation. As required by this program, the signal processor sorts the incoming numbers from the A/D converter by time of arrival, hence range and stores the numbers for each range interval in memory locations called range bins. After this it filters the bulk of the unwanted ground clutter on the basis of its Doppler frequency. By forming a bank of narrow band filters for each range bin, the processor then integrates the energy of successive echoes from the same target (i.e., echoes having the same Doppler frequency) and still further reduces the background of noise, clutter with which the target echoes must be compete. By examining the outputs of all the filters, the processor determines the level of the background noise and residual clutter, just as a human operator would by observing the range trace on a display. On the basis of increases in amplitude above this level, it automatically detects the target echoes. Rather than supplying the echoes directly to the display, the processor temporarily stores the target’s positions in its memory. Meanwhile, it continuously scans the memory at a rapid rate and provides the operator with a continuous bright TV-like display of the positions of all targets. This feature, called digital scan conversion, gets around the problem of target blips fading from the display during the comparatively long azimuth scan time. The target positions are indicated by synthetic blips of uniform brightness on a clear background, which making them extremely easy to see.

Data Processor

Data processor is general-purpose digital computer; it controls and performs routine computations for all units of the radar. Monitoring the positions of selector switches on the control panel, it schedules and carries out the selection of operating modes, e.g., long range search, track-while-scan, SAR mapping, close-in combat, etc. Receiving inputs from the aircraft’s inertial navigation system, it stabilizes and controls the antenna during search and track. On the basis of inputs from the signal processor, it controls target acquisition, making it necessary for the operator only to bracket the target to be tracked, with a symbol on the display. During automatic tracking, the data processor computes the tracking error signals in such a way as to anticipate the effects of all measurable and predictable variables - the velocity and acceleration of the radar bearing aircraft, the limits within which the target can reasonably be expected to change its velocity, the signal-to-noise ratio, and so on. This process yields extraordinarily smooth and accurate tracking. Throughout, the data processor monitors all operations of the radar, including its own. In the event of a malfunction, it alerts the operator to the
problem, and through built-in tests, isolates the failure to an assembly that can readily be replaced on the flight line.

### 2.2 Challenges in Radar Signal Processing

In this thesis we worked on signal processing stage, in that there are different sub stages as shown in the Figure 2. Earlier day’s many complex algorithms used to run only on few number of processors and with limited parallel execution. But now days due to number of applications increased, running these on limited number of processors is not efficient because of so many limitations like heat dissipation, less processing speed, etc, hence there is need of parallel processing techniques. In Radar signal processing, processor need to process complex data and there is need of high speed for quick identification of target and its altitude, position, etc. and also has to retransmit the data to the target within short time.

The main requirement in Radar signal processing is the processing speed; which plays important role in overall system performance, this is possible by processing the data in parallel that is, as possible on more number of processors simultaneously. This technique called SIMD (Single Instruction Multiple Data) processing.

Another problem in signal processing is that the processors could process only limited range of input data which limits the processors application areas, but in general there could be different range of input data. So, there is need of algorithm which could run for different range of input data like 8-point, 16-point, 32-point etc. Because the input range of data may not be constant in different Radar signal processing, so the algorithms should work for different range of input sample so that the speedup can be increased. Since the input variables are complex values it is challenging to write efficient algorithm. An algorithm works on more input samples means, it could perform more arithmetic and logic operations per cycle. When these algorithms mapped on to the processor there is increased system performance in terms of speed and use of memory on the system. The speedup depends on how efficiently we are programming the algorithms.

### 2.3 Signal processing chain

The reflected echo signals from land, sea and weather are regarded as clutter in air search radar or in other radars. The signal processor used to suppress the returned or reflected echo signals from land, sea and weather when the spectrum is narrowing compared with the radar’s pulse repetition rate (PRF). Filters that combine two or more returns from a single range cell are able to distinguish between the desired targets and the clutter. This allows the radar to detect targets with cross-sections smaller than that of the clutter. It also provides a system of preventing the clutter from causing false alarm. [3]
2.3.1 Pulse compression

When sending a pulse, the ideal scenario is that the pulse is short with high amplitude, to attain good range resolution. In reality, the amplitude is limited by the transmitter hardware. Therefore the pulse energy has to be distributed over time, but then the range resolution becomes poor. To compensate for this, the signal pulse is modulated and the receiver uses a compression filter matched to the modulation, which makes it possible to separate objects with overlapping echoes. In the compression filter FIR algorithm is used to match the signal modulation.

The pulse compression processing can be treated as most advantageous filter (matched filter) reception for various modulated signals. For practical transmitted signals, linear chirp modulation is frequently used [2]. Finite impulse response filters have the advantages of linear phase, fixed stability, and efficient implementation. The finite impulse response filters have been used in signal processing as ghost cancellation and channel equalization [16]. In modern digital communication systems, pulse-shaping filters allow the transmission of pulses with negligible inter symbol interference. This means that pulse-shaping digital filter is a useful means to shape the signal spectrum and avoid interference of ultra wideband (UWB) to other inheritance narrow band signal. However, due to increasing demand for video signal processing and transmission, high speed and high order FIR filter frequently have been applied for performing adaptive pulse-shaping and signal equalization on the received data in the real time system. Hence minimizing the computational complexity and the system cost in terms of power consumption and memory storage needed is a major target in digital filter design task. The computational complexity is a function of the multipliers and adders used in filter realization [15].

An important step in radar signal processing involves cancelling unwanted interference and improving the information signal-to-noise ratio. Prior to the computation of the adaptive interference suppression weights, the input sensor data must be conditioned to meet the interference cancellation requirements (typically 50 to 60 dB). This data conditioning is performed by filtering the incoming signals using FIR filters.
2.3.2 Velocity Compensation

High range resolution can be achieved by using stepped-frequency waveform while still retaining the advantages of lower instantaneous receiver bandwidth and lower analogue-to-digital sampling rate. However, the relative radial motion between the target and the stepped-frequency radar will result in performance degradations, like range error, loss in signal-to-noise ratio, and degraded range resolution. The solution to this problem is to apply velocity compensation to the received signal, which can eliminate the degradations due to doppler effects. When the aircraft moves, the ground clutter is doppler modulated. The velocity compensation moves the clutter down to zero-frequency. [5]

2.3.3 MTI (Moving Target Indicator) filter

MTI filter is radar’s filter that extracts the doppler frequency shift and rejects the clutter frequency. Some MTI radar filters can extract the moving target echo from a clutter echo that is 70-90 dB larger [3]. In MTI radars, the doppler shift in frequency is used to distinguish moving targets even when the echo signal from fixed targets is orders of magnitude greater. Fixed-target echoes or clutter are included within the same radar pulse-packet as the target, but the signals from fixed targets are not shifted in frequency. Thus, any target moving with a relative velocity larger than zero produces signals shifted in frequency by a certain predictable amount. A vital process in such systems is filtering. The main attribute of the MTI radar is its delay line canceller.

2.3.4 Doppler filter

The pulse Doppler technique is most often used in either airborne or land based target tracking radars, where a high ambiguous pulse repetition frequency (PRF) can be used, thus providing an unambiguous range of doppler frequencies. This filter is introduced to improve the signal-to-noise ratio and to measure the object’s speed relative to the radar. By using Fast Fourier Transform (FFT), the doppler frequency can be extracted, and used to estimate the object’s speed. This pulse doppler process is class of clutter filter where the returns in each range resolution cell are gated and put into a bank of doppler filters. The number of filters in the bank approximately equals the number of pulse returns combined. Each filter is tuned to a different frequency and the pass bands contiguously positioned between zero frequency and prf. [6]

2.3.5 Envelope Detector

The envelope detector gets the input signal from receiver and converts it in to a video signal which is supplied to the display. After the doppler filter stage, the phase information of the signal is no longer needed. Envelope detection is introduced to remove this information, which is done by a simple absolute value calculation. The logarithm of the data may also be calculated in order to decrease the dynamic range.

2.3.6 Detection (CFAR)

Constant False Alarm Rate (CFAR) detection keeps all but short spikes of jamming from being detected, hence in a jamming environment makes targets easier to see. Though CFAR keeps jamming strobes from being detected when it is employed, a separate jamming detector must be provided for the ECCM system. The completely random noise alone will occasionally exceed the threshold, and the detector will falsely indicate that a target has been detected. This is called a false alarm. The chance of its occurring is called the false alarm
Radar signal processing

probability. The higher the detection threshold relative to the mean level of the noise energy, the lower the false-alarm probability will be, and vice versa.

Clearly the setting of the threshold is crucial. If it is too low, too many false alarms will occur. The optimum setting is just enough higher than the mean level of the noise to keep the false-alarm probability from exceeding an acceptable value. The mean level of the noise, as well as the system gain may vary over a wide range. Consequently, the output of the radar's doppler filters must be continuously monitored to maintain the optimum threshold setting. Generally, the threshold for each detector is individually set on the basis of both the probable noise level in the filter whose output is being detected (the “local” noise level) and the average noise level in all of the filters (the “global” noise level). However, the thresholds are set so as to maintain the false-alarm rate for each detector at the optimum value. If the rate is too high, the thresholds are raised; if it is too low, the thresholds are lowered. For this reason, the automatic detectors are called constant false-alarm-rate (CFAR) detectors.

### 2.3.7 Resolving

At this stage, speed and range of the target will be calculated by varying PRF during successive intervals. And multi targets are resolved and fed to the display therefore all the targets would be displayed on the display. And more over, resolving ability is a significant parameter for radar. Usual coherent radar has bad angle resolution, the direct result is that it cannot perfectly work in multi-target surroundings, for example the radar will detect out only one target if multi-target moving in the same resolution unit. Thus, multi-target resolving is an important problem for usual radar. Although target’s range and azimuth are nearly the same, target’s moving character (such as radial velocity and acceleration) will be different in a few radar observing time, which has been proved by radar’s measure data. So it is multi-target’s movement resolving for conventional coherent radar’s multi-target resolving.
3 Introduction to Ambric Platform

In this section we discussed about Ambric chip and its capabilities and whole hardware structure of Ambric platform with figures.

3.1 Ambric Architecture

The Am2000 family fines a new platform for embedded and accelerated computing, based on a new object-based programming model. This platform is aimed at application developers using software languages and methodologies. Its objectives are massive performance, long-term scalability, and easy development.

Massively Parallel Processor Array (MPPA) designed using more than 100 million logic transistors (Gates) on a single chip. In this hundreds of processors and memory units interconnected in general purpose way, with flexibility, high performance, and low power and low cost.

Initially Ambric decided to choose a programming model to solve the parallel development problem first, then created a processing and interconnect architecture, silicon, and tools to realize the desired model. In the Structural Object Programming Model (SOPM), objects are strictly encapsulated software programs running concurrently on an asynchronous array of processors and memories [12]. Objects are interconnected with Ambric channels to communicate and synchronize each other.

3.2 Structural Object Programming Model

In Structural Object Programming Model of Ambric as shown in Figure 3, an array of processors and memories are programmed with conventional sequential code. A programmed processor or memory is called a leaf object. Objects run independently at their own speeds. They are strictly encapsulated, execute with no side effects on one other, and have no implicitly shared memory.

To intercommunicate between Objects and to carry both data and control tokens hardware Channels used. By the use of Channel hardware, Objects are synchronized at each end dynamically as needed at run time, but not at compile time. Processors, memories, and channel hardware handle their own synchronization transparently, dynamically, and locally, relieving the developer and the tools from this difficult task [12].

Since Channels provide a common hardware-level interface between all objects, which makes it simple for objects to be assembled into higher-level composite objects. Because objects are encapsulated and interact only through channels, composite objects work the same way as leaf objects. Programming is a combination of familiar techniques, so the application developer expresses object-level parallelism with block diagrams, in graphical or textual aStruct form. These define a hierarchical structure of composite and leaf objects, connected by channels that carry structured data and control messages. And the ordinary sequential software is written, in Java or assembly, to implement the leaf objects that aren't already available in a library.
3.3 Ambric Registers and Channels

 Registers

Ambric registers play a main role in the design of the programming model, because the programming model will be embedded in these registers. Ambric registers as shown in Figure 4, replace ordinary edge-triggered registers everywhere in the system. It's a clocked register with data in and data out. There are two control signals, valid and accept, that implement a hardware protocol for local forward and back-pressure. This protocol makes self-synchronizing and asynchronous using Ambric registers. The Ambric register is preemptive, in that its operation is self-initiated. When a register can accept an input, it asserts its accepting signal upstream; when it has output available, it asserts valid downstream. When two registers connected as in Figure 4 both see valid and accept are both true, they each know the transfer has occurred, without negotiation or acknowledgement. This protocol is simple and glues less, using only two signals and no intermediate logic. Unlike FIFOs, these dynamics are self-contained, so they become hidden and abstract.

Scalability calls for all signals to be local. When the output of a register chain stalls, the stall can propagate back only one stage per cycle. To handle this, all Ambric registers can hold two words, one for the stalled output, and one caught when necessary from the previous stage.
Channels

A Channel consists of a chain of registers like Figure 4. Channels are the fully encapsulated, fully scalable technology for passing control and data between objects called for by the programming model. A channel is word-wide, unidirectional, point-to-point, and strictly ordered (first-in-first-out). Local synchronization at each stage lets channels dynamically accommodate varying delays and workloads on their own. Changing the length of a channel does not affect functionality, only latency changes. It is a simple and universal abstraction that combines communication and synchronization.

Ambric channels are used throughout the chip architecture, inside processors and memories as well as throughout the interconnection. The two words of buffering in each stage of a channel act as a small FIFO to smooth out dynamic behavior between objects. Larger FIFOs, implemented in RAMs, can be inserted transparently in channels for larger flow control, but also to implement storage in the application. FIFOs often replace random-access memories in Ambric applications, because they are a streaming, self-synchronizing form of storage.

3.4 Hardware Objects and Clocking

A hardware object in Ambric can be combinational logic, state machine or RAM. An object is interconnected by Ambric registers. By using accept and valid signals, internal synchronization are acquired. By connecting objects, composite objects and complete applications are obtained. These objects are connected by Ambric channels, so by sending data, the synchronization between objects is acquired. Since the objects run independently and synchronized only through the channel interfaces, this system is called globally asynchronous and locally synchronous (GALS). The synchronization protocol enables clock crossing registers. A clock crossing register works with different, but related, clocks on input and output. So the objects that are connected to the ends of this register can run with their own speed. Objects can adjust their clock rates dynamically during an application. For instance when an object feels that it is stalling then it reduces the clock
rate in order not to consume much power. So each object runs at the lowest clock rate possible. The processors are also hardware objects. They are interconnected by Ambric channels and called leaf objects in programming model. They synchronize with each other by using transfer events. Since they are connected with Ambric registers they communicate as registers. If a processor wants to send a message to another, first it checks if the receiver processor (accept signal) is ready to get it. If the receiver processor is ready then the sender processor sends the message otherwise it stalls until the receiver gets ready. The same protocol is valid for a processor which wants to get a message. This protocol is useful for systolic array implementation, since the processor stalls until it gets the input, and will be mentioned in next chapters.

### 3.5 Processor Architecture

Ambric processor architecture is designed to perform data processing and control through channels. The processor architecture is depicted in Figure 5. 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.

![Processor Architecture](Ref [12])

Ambric processor is very lightweight 32-bit streaming RISC (Reduced Instruction Set Computer) 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.
3.6 Ambric chip

The Ambric chip consists of 5x9 Array of Brics. Brics and their interconnection are shown in Figure 6. 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. Ambric chip consists of two types of processors, one is Streaming RISC with DSP extension (SRD) and another one is Streaming RISC (SR). 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.

SR processor is a 32 bit streaming RISC control processing unit. SRD is also 32 bit high performance and streaming RISC control processing unit with Digital Signal Processing extensions. This processor is used for mathematical manipulation of data. It has confined memory for 32 bit instruction and can execute further code from RAM unit directly.
3.7 Ambric Compute Unit and RAM unit

**Compute Unit**

The CU interconnect of Ambric channels joins two SRDs and two SRs with one another, and with CU input and output channels that connect with other CUs through the configurable interconnect. These interconnects establish channels between processors and memories according to the application’s structure.

**RAM Unit**

RAM (Random Access Memory) Units are the main on-chip memories for immediate use by the processors. Each RU has four independent single-port RAM banks (Figure 7), each with 512 words. It has six configurable memory access engines that turn RAM regions into objects that stream addresses and data over channels:
Two (marked RW in Figure 7 on above page) for SRD data by random access or FIFOs
Two (marked Inst) for SRD instructions by random access or FIFOs
Two (marked str) for channel-connected FIFOs, or random access over channels using packets

RAM banks are accessed by the engines on demand; through a dynamic, arbitrating channel interconnect. To get one word per cycle FIFO bandwidth from single-port RAMs, the engines can be configured to use two banks, striping even addresses to one and odds to the other.

3.8 Brics and Interconnections

One of Bric is highlighted in below Fig 8 and each bric has two CU-RU pairs, totalling eight CPUs and 21K bytes of SRAM. Inter-bric interconnects are arranged so brics connect by abutment. Cores of different sizes are constructed by stacking up rows and columns of brics. The chip interconnect is a configurable three-level hierarchy of channels, reflecting the fact that most channels are local, but routing long channels through local interconnects will congest them. At the base of the hierarchy are the local channels inside each CU, shown in. At the next level, blue channels directly connect CUs with neighbouring CUs, as shown in below figure. Also shown are split-FIFO channels that directly connect engines in neighbouring RUs, for implementing multi-RU FIFOs.

Blue channels directly connect the output crossbar of one CU to input crossbars of another CU. Each CU connects through two input blue channels and two output blue channels with four other CUs, north / south / east/west. Tools automatically assign channels in the application to specific CU blue channels, by configuring CU output crossbars and CU input mapping registers. Streaming and RW engines in adjacent RUs can interconnect through RU split-FIFO channels, shown in black in Fig 8. This makes it simple to implement larger FIFOs that span multiple RUs. At the top level, the green network for long connections is a 2-D circuit-switched interconnect of channels and configurable switches, shown in green in Fig 8. Each bric has a green network switch, with four channels in and four out connecting with each CU. Switches are interconnected in a grid with four channels each way. These bric-long channels are the longest signals in the core, except for the low-power clock tree and the reset. These repeated short connections make this interconnect physically scalable. Channel connections through the green switch are statically configured, establishing point-to-point channels between processors in distant CUs. Each green channel leaving a switch is registered, so there is one channel stage per hop. The green network always runs at the maximum clock rate, connecting to CUs through clock-crossing registers.
3.9 SR and SRD Processors

There are two types of processors in Ambric chip they are SRD processor and SR processor. These processors have local RAM memories they get the instructions through the channels. Processors execute the instructions streamed from local memory, by a program counter by using random access channels. In Ambric processors there is no interruption since each processor is dedicated to own task. SR processors are 32 bit streaming RISC processors. SRD processors are also 32 bit streaming RISC processors with DSP extension. In these two SRD is more powerful then SR since it has DSP extension for mathematical calculations. Thus SR processors are mainly used for small tasks, when SRD is not necessary, like forking. SRD is more like extension of SR. Both processors are programmable. The local memories are parity checked.

Each processor runs a leaf-level processor object that you program in Java or assembly code, or choose from a library. You can program SRs and SRDs in Java, without direct regard to their instruction sets and functional details. Nevertheless, to write the most effective code it is best to become familiar with them. Assembly-level programming in the native instruction sets, described here, is available for maximum performance and memory density, and for access to features not supported by a higher-level language.
**SR Processor**

SR is a simple 32-bit streaming RISC CPU as shown in Fig 9, it used mainly for small or fast tasks, such as forking and joining channels, generating complex address streams, and other utilities. It has a single-integer ALU with eight general registers, one input channel connected to the CU input crossbar, and one output channel connected to the CU output crossbar. 64 words of parity-checked local memory hold up to 128 16-bit instructions and data. SR’s data path is an Ambric channel, with Ambric registers on the ALU inputs and outputs, and its control path is a channel, because the Instruction Register (IR) and Program Counter (PC) are Ambric registers. This three-stage pipelined implementation means that nearly all SR instructions can complete in a single clock cycle.

![Fig 9. SR processor Ref [12]](image)

**SRD Processor**

SRD processor is a 32-bit streaming RISC processor with DSP extensions for math intensive processing and larger, more complex objects. SRD instructions are 32 bits wide, with multiple ALU fields to express instruction-level parallelism, for high performance with high code density. SRD can accept inputs from two channels and feed one output to a channel each cycle.
The differences between SRD and SR processors

<table>
<thead>
<tr>
<th>SRD Processor</th>
<th>SR Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRD Instructions are 32-bit wide, with Instruction level parallelism</td>
<td>SR instructions are 16 bits wide, for maximum code density on the chip</td>
</tr>
<tr>
<td>Accepts 2 inputs and feed 1 output</td>
<td>Accept 1 input and feed 1 output to a channel each cycle</td>
</tr>
<tr>
<td>3 ALUs, 2 in series and one in parallel. Its ALUs include full-word, Half-word, Quad byte logical, integer, fixed point operations including multiply and barrel shift.</td>
<td>SR has a single full-word, dual half-word, and quad byte integer ALU.</td>
</tr>
<tr>
<td>256 words of Local Memory for its 32-bit instructions and local memories are parity-checked</td>
<td>SR has 64 words of local memory, used for up to 128 16-bit SR instructions or data. Local memories are parity-checked.</td>
</tr>
<tr>
<td>SRD’s has pipelined implementation which execute all instructions in one cycle.</td>
<td>SR’s pipelined implementation means that nearly all instructions execute in one clock cycle</td>
</tr>
</tbody>
</table>

Fig 10. SRD processor ref [12]
Algorithms Overview

4 Algorithms Overview

In this chapter we described an overview of chosen algorithms which we are going to implement on the Ambric architecture in the next chapter.

4.1 Fast Fourier Transform (FFT) Algorithm

Discrete Fourier Transform (DFT) is a mathematical technique which is used for analyzing periodic digital series of observations: \( x(n) = 0 \ldots N - 1 \), here \( N \) will be a large number. DFTs main applications are in image processing, communication and radar signal processing. When analyzing series of digital signals by using DFT, the assumption looks like there are periodical repeating patterns that are hidden in the series of observations and also other phenomena that are not repeated in any discernible cyclic way, that is called “noise”. The DFT helps to identify the cyclic phenomena [7].

The mathematic definition of a DFT is,

\[
X_N[k] = \sum_{n=0}^{N-1} x[n] \times e^{-j\left(\frac{2\pi}{N}\right)kn} \quad k = 0 \ldots N - 1
\]  

Here the size of \( N \) in a DFT is often a factor of power of two like 8, 16, 32, 64, 128, 256, and so on. And \( n \) is the variable of samples.

To calculate the Discrete Fourier Transform (DFT), the Fast Fourier Transforms (FFT) is an optimized and fastest way that 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 [8]. Fast Fourier transform algorithms generally fall into two classes: Decimation-in-Time, and Decimation-in-Frequency. Both of the approaches require the same number of complex multiplications and additions. The main difference between two approaches is that decimation-in-time takes bit-reversed input and generates normal-order output; on the other hand decimation-in-frequency takes normal-order input and generates bit-reversed output [9].

An FFT algorithm, which is based on equation (1), uses easy way to calculate it. FFT technique divides the DFT with \( N \) points into \( N \) DFTs. First it divides \( N^{th} \) order equation (1) into two \( N/2 \) equations as shown in equation (2) where the equation is divided into a sum of even numbers \( (n = 2a) \) and odd numbers \( (n = 2a + 1) \) [10]. Equation (2) can then be divided into two \( N/4 \)s and so on until \( N \) DFTs.

\[
X_N[k] = \sum_{a=0}^{N/2-1} x[2a] \times e^{-j\left(\frac{2\pi}{N}\right)k2a} + \sum_{a=0}^{N/2-1} x[2a + 1] \times e^{-j\left(\frac{2\pi}{N}\right)k(2a+1)}
\]  

Where, \( a \) is an integer variable 0…..\( N \).

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.
4.1.1 Radix-2 FFT

Radix-2 algorithm is one of the popular techniques to calculate Fast Fourier Transform (FFT). In Figure 11 has shown radix-2 algorithm. It will be valid only to sequences of length $N=nm$, where $m$ will be a positive integer. In this method computational scheme will be regular and efficient. The basic computation in the radix-2 decimation-in-time algorithm is the butterfly computation. Butterfly Computation of radix-2 with decimation-in-time has been shown in Figure 12 that is applied in $\log_2 N$ following steps, $N/2$ times in each step giving the algorithm a complexity of $O(N \log_2 N)$. In Figure 12 shows the 8-point FFT computation using the radix-2 decimation-in-time algorithm. Examine the shuffled order of the input samples, the order is found by reversing the binary representation of a normally ordered sequence. [11]

\[
\begin{align*}
A &= a + bW(N,K) \\
B &= a - bW(N,K)
\end{align*}
\]

Fig 11. Butterfly Computation of radix-2 with decimation-in-time

4.1.2 Complexity analysis of radix-2 FFT

In each butterfly computation will need one complex multiplication and two additions. For $N$ point FFT, in each $\log_2 N$ steps with $N/2$ butterflies will have total $2N \log_2 N$ real multiplications and $3N \log_2 N$ real additions, so that $5N \log_2 N$ floating point operations needs in total. [11]

Fig 12. 8-point FFT butterfly using radix-2 decimation-in-time
Algorithms Overview

The complexity of different versions of algorithms is tabulated in below table.

<table>
<thead>
<tr>
<th>Complexity (Nlog&lt;sub&gt;2&lt;/sub&gt;N)</th>
<th>8 point</th>
<th>16 point</th>
<th>32 point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Butterfly Stages (log&lt;sub&gt;2&lt;/sub&gt;N)</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>No. of Additions (3Nlog&lt;sub&gt;2&lt;/sub&gt;N)</td>
<td>72</td>
<td>192</td>
<td>480</td>
</tr>
<tr>
<td>No. of Multiplications (2Nlog&lt;sub&gt;2&lt;/sub&gt;N)</td>
<td>54</td>
<td>128</td>
<td>320</td>
</tr>
</tbody>
</table>

Table 1: Algorithm Complexity variation

4.2 Finite Impulse Response (FIR) algorithm

Finite Impulse Response (FIR) filter is widely used in many applications involving signal processing algorithms and introduces one of many computationally demanding signal processing tasks. The new generation of telecommunication often require the use of high order FIR filters for the implementation of the new modulation schemes. Moreover, the telecommunication market demands for high speed and low power consumption for the new portable multimedia terminals. FIR filtering is realized by a large number of adders, multipliers and delay elements. The difference equation that defines the output of a Finite Impulse Response filter in terms of its input variable is as follow,

\[ Y(n) = h_0 X(n) + h_1 X(n-1) + \ldots + h_N X(n-N) \]  \hspace{1cm} (3)

Where,

- \( X(n) \) is the input variable
- \( Y(n) \) is the output variable
- \( h_0, h_1, h_2, \ldots, h_N \) are the filter coefficients, and

\( N \) is the filter order i.e. \( N+1 \) terms on right hand side, these are usually referred as Taps.

Thus, each entry in the output vector is accumulated product of \( N \) multiplications of a coefficient by the corresponding delayed data sample.

In general form can also expressed as,

\[ Y(n) = \sum_{i=0}^{N} h_i \times X(n-i) \]  \hspace{1cm} (4)

Where \( n \) is the time index and \( i \) is the delay or shift variable respectively.
Algorithms Overview

The equation shows that the filter output is weighted sum of the current and finite number of previous values of the input. The basic design of FIR algorithm is shown below figure 13.

In the above figure $X(n)$ is the input and $Y(n)$ is the output of FIR and delay box indicates the delay between sampled inputs, cross marked circle indicates multiplier and $h_0$, $h_1$ etc. are coefficients and circles indicating with plus sign are adders in which two inputs from two different multipliers will be added.

Fig 13. Schematic diagram for FIR
5 Algorithms Implementation

For the implementation and evaluation of Ambric architecture we have chosen two different algorithms. One of them is Fast Fourier Transform (FFT) and another is Finite Impulse Response (FIR). In previous chapter we have presented an overview of these algorithms. Now in this chapter we will describe our practical work that we have done within our thesis with detailed description of the algorithm implementation and mapping on the Ambric architecture.

The parallel processing is the ability to carry out multiple operations or tasks simultaneously. Our target will be optimal utilization of parallel processing capabilities. We are going to implement FFT and FIR algorithm on Am2045 processor. In this processor there has only four input and output ports in one processor. So we are able to read maximum four input streams from the chip. If we want to run an algorithm on more than four processors we have to distribute a single stream for parallel processing elements. According to the Streaming RISC with DSP (SRD) processor’s instruction set an object can have maximum five input ports and six output ports. Principally Streaming RISC (SR) processors are used for streaming and Streaming RISC with DSP (SRD) processors are used for math exhaustive operations.

5.1 Fast Fourier Transform

We have described in previous Chapter about radix-2 FFT, here we will discuss about design of FFT Algorithm that will support on Ambric platform then implement this design using aDesign environment and finally mapping on Ambric architecture.

In this thesis work, our main goal is to develop a design approach for algorithms that will work for different range of input points. Point will be the format of power of two like 8, 16, 32, and 64 and so on. Providing pre-calculated twiddle factors to the application is a better technique without computing them on run-time because of complexity. If we want to compute twiddle factors on run-time, it will be more complex. We can store twiddle factors to the application in several ways, we can use lookup table to store these or we can store twiddle factors in external memories. We can also provide twiddle factors 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. [7]

Since our design approach is parameterized version, we can not provide twiddle factor for specific objects or processors because, we do not know how many objects we have. It will depend upon number of points of FFT that we will consider. The no of twiddle factors required will be depends on the complexity of FFT algorithm that is for 8 point we required only 8 twiddle factors and each object needs only one value of twiddle factor. We store the maximum number of twiddle factors required for the maximum range of algorithm in the design file. Then these will be accessed by the algorithm during compile time by using index. In our program we store 32 twiddle factors for using different versions of algorithm for the range 8 point to 32 point FFT. Here, twiddle factors will be the complex variables, so we store 32 real and 32 imaginary values in the design file.

In our program we have used decimation-in-time technique for the implementation of radix-2 operation. In this technique at first it uses bit-reversal mechanism for distributing input point and then it performs the butterfly calculations.
There has no need to reserve one or more processors for the bit-reversal sorting, so by using this mechanism we have saved some execution time. *Distribute* objects divides the even and odd elements of the input stream. *Distribute* object will send set of even points to its left object and odd points to its right object of the input stream. The final stage will get totally separated even and odd points. That is shown in Figure 14. In this Figure, we have shown that for N-point FFT at first it divided by N/2 point and next step divided by N/4 point. It will divide until every processor gets exactly two points. In our design we have worked two points on one processor for butterfly calculation.

In Figure 15 we have shown an example of mapping and communication through channels for 8 point FFT. For N point FFT the same mapping mechanism will be used. Here each circle represents an object or processor which is running and arrow represents the flow of data. Input stream consists of both real and imaginary parts of the time domain signal.

In our design approach, each object will compute only 2-point butterfly calculations. By this all the objects will get the same work load. Because of all objects have same work load, processor stalls are reduced in this design. This design does not require any change in the clock frequency of objects or processors. Figure 15 shows three butterfly stages where each object gets 2 point from distribute objects and perform the butterfly computations throughout the butterfly stages. The twiddle factors are used in butterfly computations which have to supplied either through design file or compute at runtime by using another algorithm. In our implementations we have been sent twiddle factors through design file at run time. After butterfly computations the output samples collected and assembled by assembler objects $A_1$, $A_2$, $A_0$ then the final result will be send to the output file.
Since our design is parameterized version, channel connection between two processors has to be established automatically during compile time and make a proper design for a specific version of FFT. The distribute objects and assembler objects have the same way of connecting one processor to another. The distribute object $D_0$ will distribute the samples in two ways and send to the next stage distribute objects $D_1$, $D_2$ for further process. So, every distribute object will have one input channel and two output channels. But, assembler object’s function is exactly opposite to the distribute objects. The assembler objects will have two input channels and one output channels. But in butterfly stage it is difficult to make dynamic channel connection, for that we consider three stages inside butterfly operation. We consider the name of object of first stage of butterfly operation is $firstFFT$ and last stage of butterfly operation is $finalFFT$. The object of $firstFFT$ is connecting between distribute object and $RealFFT$. At last the final stage objects are connected between $RealFFT$ and assembler object.

According to our design if we consider 8-point FFT, we need totally 18 objects, here 3 objects for distribute, 3 objects for assemble and 12 objects for butterfly computation, similarly for 16-point FFT we need 46 total objects.

In general for N-point FFT,
\[
\begin{align*}
\text{Numbers of SR processors} &= 2 \left( \frac{N}{m} - 1 \right) \hspace{2cm} \text{.......... (5)} \\
\text{Number of SRD processors} &= \left( \frac{N}{m} \log_2 N \right) \hspace{2cm} \text{.......... (6)}
\end{align*}
\]

Then,
\[
\text{The Total number of Processor} = 2 \left( \frac{N}{m} - 1 \right) + \left( \frac{N}{m} \times \log_2 N \right) \hspace{2cm} \text{.......... (7)}
\]
Algorithms Implementation

Where \( m \) is the number of points calculating on each object and it should be in the power of 2. \( N \) is the total number of points in the FFT and \( \log_2 (N) \) represents number of stages in the FFT.

If we want to calculate larger number of FFT points, the total number of processors required also increases gradually. The Am2045 processor has a total of 336 processors and half of them are SR (Streaming RISC) processor. So, we have only 168 SRD processors available to calculate butterfly operations, since we can not use SR processors for the multiplication. In our design we consider two points calculation in one object. So we can only implement this design for the range of 8 to 32-point FFT. But if we want calculating larger FFT point by using this design then we have to increase the number of point calculation on each object that means the value of \( m \) in the equation (7). So by following this technique we will be able to use this design for calculating any number of FFT points with in the design constraints.

5.2 Finite Impulse Response

Let consider 12 taps FIR design first, in this case the number of multipliers will be equals to number of taps that is, for 12 taps the number of multipliers required are 12, and always the number of adders will be less than that of taps. Here number of adders required 11 for 12 taps FIR design, mathematically

In FIR, the number of Multipliers will be equal to number of taps that is 12. Then the number of Adders required will be 12-1 =11.

In general, let say

Number of Multipliers = \( M \)

Number of Adders (\( A \)) = Number of Taps-1 = \( T-1 \)

Number of Coefficients required = Number of Taps (\( T \))

Then, Number of Processors required (\( P \)) = Number of Multipliers (\( M \)) + Adders (\( A \))

\[ = Number of Taps + T-1 \]

Number of Processors required (\( P \)) = \( T+ (T-1) = 2T-1 \)

Therefore the Number of Processors required = \( 2T-1 \) \( ....................... \) (8)

So, for 12 Taps, Processors required \( P=23 \), in that 11 used as Adders and 12 used as Multipliers, similarly for 32 Taps, \( P= 63 \), \( A = 31 \) and so on.

As shown in the below figure 16, in our FIR design approach we use processors for two different purposes, one is for multiplying and another is for Adding. The two processors which are used in this design are SRDs, because as we mentioned in earlier sections SRDs have three ALU units, so which can perform intensive math operations than SR processors. In general, to use a particular processor as target one, we have to set \( \text{CompilerOptions(targetSR=true)} \) in the corresponding aStruct file of an object file. If we do not mention any thing in CompilerOption then compiler considers SRD processor as a target processor by default. Now the
Algorithms Implementation

compiler can understand which files to be mapped on which processor. This can understand by looking in the below example.

Example:

```java
File name: AssemblerTwo.astruct

binding JAssemblerTwo implements AssemblerTwo{
    implementation "AssemblerTwo.java";

    attribute CompilerOptions(targetSR=true) on JAssemblerTwo;
}
```

When application starts running, Coef0 Object or processor (see below fig) collects one input complex number sample from input text file and multiplies with a complex coefficient value which is supplied in the design file then the result will be send to first adder0 Object. And whatever the input sample has been read will be send to the Coef1. And in the Coef1 Object, the received sample from Coef0 will be multiplied by another new coefficient which is supplied through design file in the program and result will be send to adder0, and the collected input from Coef0 will sent to the next Coef2 Object. And adder0 adds two results which received from Coef0 and Coef1 and the final result will be sent to the next adder1 object. In Coef2 object input sample will be multiplied with new coefficient and send the result to adder2 object, in this way the process continues until the number of taps and adders are finished. Finally, the last Adder10 Object sends the total accumulated result to the capture file or output file which is provided in aDesigner. This whole process will do continue until input number of sample finished. In every Coef objects (Coef0, Coef1…) is used different complex coefficient value for multiplying with input sample that we have stored in design file and Coef object collect the value of coefficient during compile time.

Fig.16. Design approach for FIR algorithm for 12 taps
5.3 Development Tools

The aSim simulator can execute and debug applications on a host computer. Java program 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 implementation we have created only one object per stage in butterfly computations and we are re-using it again and again. In case of 12 taps-FIR implementation also, we have been used only one Coef object out of twelve and one adder object out eleven then we reused them. 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 of massive parallelism.

5.4 Performance Testing Tools

The PerfHarness tools released in July, 2010 by the Ambric providers Nethra Imaging Inc. By using these tools we found the performance in terms of latency and cycle count per output sample for different versions of FIR and FFT algorithms. Here Latency means the response time for the first output for a given first input. And the cycle count per output sample indicates the clock cycles between two workload results.

In this test, the ddrLoader Object reads the input samples and stores in DDR memory from input file. Then the PerfHarness Object streams the input samples from the DDR memory and calculates the performance. After that in the next stage the full program will be loaded into the dut Object. After completing algorithm’s execution then again the output will be sent to the PerfHarness Object where it finds the latency and cycle count per output sample as shown in the below Fig 17.

![Fig. 17.Working of PerfHarness tools](image-url)
6 Results and Analysis of Solutions

The main objective of this thesis is to design and implement parameterized version of two complex algorithms named FFT and FIR and mapped them on parallel (Ambric) processor architecture. In this chapter we will investigate the results of implemented algorithms by using performance analyzer which is attached on the aDesigner simulator and using PerfHarness tools.

The results from the implemented algorithms are presented in terms of latency and cycle count per output sample. For finding latency we have to consider cycle counts and processor stalls for every single processor or object. 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.

\[ \text{Total Cycle Counts} = \text{No. of Instructions} + \text{Processor Stalls} \]

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.

We will find out latency by calculating response time (cycles) between first input and first output of a sample for given application. There are some processors running in sequence while some other processors are running in parallel. So we needed to find out when they run in parallel and when they run in serial. For the processors which are running in parallel we have picked the one which has the maximum number of execution cycles and then we add up cycles of processors which are running in sequence. Cycle count per output sample is defined as, the clock cycles between two workload results; it is the non uniform time period between two consecutive output results.

6.1 FFT Results Analysis

The table 1 shows the results of FFT where we displayed latency, cycle count and number of processors required in different cases like 8 point, 16 point and 32 point. We collect the values for required SRD and SR processors and cycle count per output sample by using PerfHarness tools. For calculating latency of FFT algorithm we have been used performance analyzer which is attached on the aDesigner simulator.
As shown in the table 1, our design for 8 point FFT requires 12 SRD processors that we can verify from equation (6) in previous chapter, but it utilizes 18 SRDs. That means 6 SRDs are used by PerfHarness tools and this will remain the same for all the other design cases. According to our design in 8 point FFT every SRD object will work on two input points. For this purpose we have to distribute 8 points until every processor gets two input samples. For distribution purpose we require three SR processors since they do not perform any arithmetic operations. After distributing the input samples they have to be sent to the next processors where they are performing butterfly calculations. Here we require three stages with four set of SRD processors which can perform arithmetic operations, because they can execute DSP instructions that we have clearly described in third chapter. Therefore we use 12 SR for 8 point FFT in total. After performing the arithmetic operations the results have to be assembled to be sent out to the design output. At this Assembler stage, we require three more SR processors for assembling the results which are coming from the butterfly (FFT) stage. Therefore totally 12 SRD and 6 SR processors are required for 8 point FFT algorithm mapping. However there are 30 SR processors are utilized by the design, out of which 6 SR processors are used in FFT design and the remaining 24 SR processors are used by the PerfHarness tools while doing performance test.

When the complexity of algorithm increases the number of processors required will be more, because we consider two points butterfly calculation for each and every SRD processor. So for 16 point FFT it requires 32 SRD and 14 SR processors. Similarly for 32 point also more processors requires than 16 point. This happen because when number of point increases, the design requires more SR and SRD processors. In that the SR processors are used for distributing input samples and assembling the output samples and the SRD processors are used for performing butterfly operations.

The latency as shown in the table 1 for 8 point FFT is 979cycles, which means the time taken for first input and to get first output. This can found by using performance analyzer tool which in embedded with the aDesigner simulator. But this analyzer is mainly used to analyze code efficiency rather than analyzing the application run time. It allows the programmer to analyze only single processors by giving some cycle counts as total cycle count. The total number of cycle counts is equal to the number of instructions executed plus the number of processor stalls. Whenever a processor is waiting for an input or output (waiting for other processor to get input
Results and Analysis of Solutions

from it) is known as stall. This analyzer can not be able to analyze both composite objects and applications. In addition to the total cycle counts, the analyzer gives also the data in the registers like inputs, outputs and program counter with the first and last cycle numbers of duration time of the data, for example between 5th cycle and 13th cycle there is a Hexadecimal number 0x10000 in the first input and output. So, when finding latency individually for each processor the first processor distribute_00 of 8 point FFT, we got 8 cycles. In the next stage there two processors in parallel, but we consider the processor which having max latency, this added with previous stage serial processor. Similarly in further stages also we have to find out the execution time for all parallel connected objects and we will consider that processor which has the maximum number of latency. This maximum latency would be added to the previously calculated execution time values and this cumulative addition of execution time of individual objects gives the total latency of the entire application.

Fig. 18. Screenshot of performance analyzer

Here the analyzer ignores the channel stalls which mean that it does not count the cycles when the processor waits for an input or when it waits to send an output. The length of a cycle in seconds can change from a processor to another, because the processors can run with their own different frequencies. So when a processor counts two cycles, the other one may count more than two for three cycles for the same time, depending on the frequencies of the processors. When there is any negative value in any register, the performance analyzer shows it as 0x7fffffff. This becomes a serious problem for analyzing in an accurate way. Because of the drawbacks and constraints, the performance analyzer is not reliable, but we used this performance analyser particularly for finding latency for FFT algorithm.

As shown in the table, the latency increasing with algorithm complexity because, when we increased FFT points we need more number of processors for distribute the input samples, butterfly computations and assembling all the output samples. For more butterfly computations the instructions to be executed by the processor obviously will be more and also processor stall will be more, so the overall latency will also be more. For 16 point FFT the latency is more than double compared 8 point FFT. In case of 32 point FFT the latency is
Results and Analysis of Solutions

approximately less than double compared to 16 point, this is due to the less number of processor stalls. If we compare the complexity with respective latency, we can observe that 32 point FFT is running more efficiently.

The cycle count per output sample for different point of FFT is shown in the table 1. As we discussed above, the cycle count for 8 point FFT is 377 cycles. So, every 377 cycles it can receives one output sample for the case of 8 point FFT. When the complexity increases the cycle count will be decreased because, when we increase FFT points the work load on each processor will be increases which keeps all processors busy so that the processor stalls decreases as shown in table 1.

### 6.2 FIR Results Analysis

The FIR results are tabulated below. We recorded the number of SRD and SR processors which used for execution of FIR algorithm, latency of the FIR algorithm for different range and cycle count per output sample by the help of PerfHarness tools.

<table>
<thead>
<tr>
<th>FIR</th>
<th>12 Taps</th>
<th>16 Taps</th>
<th>32 Taps</th>
<th>64 Taps</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRD Processor</td>
<td>By Design</td>
<td>Perf Harness</td>
<td>By Design</td>
<td>Perf Harness</td>
</tr>
<tr>
<td>23</td>
<td>6</td>
<td>31</td>
<td>6</td>
<td>127</td>
</tr>
<tr>
<td>SR Processor</td>
<td>0</td>
<td>24</td>
<td>0</td>
<td>24</td>
</tr>
<tr>
<td>Latency(cycles)</td>
<td>1757</td>
<td>1951</td>
<td>2725</td>
<td>4281</td>
</tr>
<tr>
<td>Cycle Count/Output Sample</td>
<td>121</td>
<td>121</td>
<td>121</td>
<td>121</td>
</tr>
</tbody>
</table>

Table 2: Results for FIR by using PerfHarness Tools

As shown in table 2, the number of SRD and SR processor are used by FIR algorithm are 29 and 24 respectively for 12 taps. According to our design approach it requires only 23 SRD (Eqn. 8) processors and we do not need of SR processors, but still it using 24 SR processors and 6 more SRD processors. These extra number of SR and SRD processors used by PerfHarness testing tools, for its own use to perform test on given algorithm.

FIR design is based on pipeline architecture in which the objects are interconnected serially and they depends on other objects or processors to process the data, known as data dependency between objects. Here we use processors for two different requirements, one for multiplication purpose and another for addition purpose. The processors that are used for multiplication purpose is connecting serially among them and parallel with the processors that are used for addition purpose. Here, SRD is used for arithmetic operations like addition and
Results and Analysis of Solutions

multiplication, there is no case where the data distribution and data assembling, so we do not require SR processor in FIR algorithm. The PerfHarness tools always uses constant number of processors, that is 24 SR and 6 SRD processors for it’s own requirement, irrespective of number of taps been used. When we increase number of taps the number of processors also increases, because we need more multiplier objects to multiply input sample with coefficient and need more adder objects to add results coming from multiplier objects.

The latency for 12 taps is 1757 cycles, which is nothing but the execution time. So, when we send first input sample to FIR algorithm of 12 taps we will receive the first output sample after 1757 cycles. The latency is increasing gradually with complexity of FIR algorithm or the number of taps as shown in the table 2. This happens because, when the number of taps increases the number of processors will be increases for performing ALU operations. For this reason, the execution time will be more between input and output. In other words every input sample will go through all the objects between input and output which makes to increase execution time or latency. Hence latency increases with complexity of algorithm.

The clock cycles between two workload results are 121 cycles for 12 taps FIR algorithm. So, for every 121 cycles FIR algorithm filters one sample and sends to output. The cycle count per output sample for FIR algorithm is constant for all other designs and even for more complex FIR algorithms. This is due to the constant work load on each processor or at every stage of FIR design the stall time is identical. In FIR the data processed through serially connected objects. So, every object of next stage has to wait for results to be get from previous stage, this is known as data dependency. This waiting time will be identical at all the stages throughout the design from input to output. This is indicated in the table 2.
Conclusions and Future Work

7 Conclusions and Future Work

The main aim of our thesis Radar Signal Processing on Ambric Platform is to develop two different parameterized version algorithms namely Fast Fourier Transform Algorithm and Finite Impulse Response Algorithm for complex input variables such that the algorithms should work for different range of complexities of algorithms.

Here, we implemented the parameterized version FFT algorithm which works from 8 point to 32 point for given parameter values in the program within the range of design, and implemented algorithms are specifically for Radar Signal Processing applications where the input variable would be the complex variables. This algorithm useful in multi input’s range applications in Radar Signal Processing. We used aDesigner tools for writing and mapping FFT and FIR algorithms provided by Ambric, since The aDesigner do not support Floating point operations, we have overcome this problem by converting Floating point to fixed point operation. The programming language used is aJava which is subset of Java.

In our implementation of algorithms, there is need to provide coefficients. We can provide these coefficients in different ways like using lookup tables, storing coefficients in external memory provide to processor on run time through input streaming and also could pass to the objects at compile time. In our implementation we provided these coefficients which are used in FFT and FIR algorithms in design file so that the algorithm can access the coefficients at run time. And FIR algorithm is designed by using pipeline architecture approach. This algorithm works for different number of taps, from 12 taps to 64 taps for given parameter values within the range of design.

Higher order FIR and FFT filters could sample the data at higher rate, so which permit the user to specify larger order filters without compromising maximum attainable clock speed. We performed the test or simulation of FFT and FIR algorithms by using PerfHarness tools which provided by Nethra Inc. Using these tools we collected the simulated output in terms of latency and cycle count per output sample. This thesis shows that the architecture of MPPA is suitable for parameterized versioned algorithms FFT and FIR. Since Ambric platform is reconfigurable processing array (RPA), these algorithms running for different ranges or orders of algorithms instantly.

Future work

We suggest that, even we can implement more complex FFT algorithm and run on the same chip. But here we should map the algorithm efficiently; we can achieve this by utilizing all the processors keeping them busy without wasting it’s time. In other words due to limited number of processors on Ambric chip it is better to design a good approach such that we can map the algorithm more and more efficiently for larger range like 64 point, 128 point, 256 points, so that we can process more data in short time. Even for FIR also it is possible to write algorithm for more number of taps.
8 References


References
Appendix - Source Code

## 9 Appendix - Source Code

In this chapter included the most important parts of the source code that we have developed in our thesis project. We have implemented two different complex algorithms that are Fast Fourier Transform (FFT) and Finite Impulse Response (FIR). Both are parameterized version implementations. Each implementation is divided into three parts; the design code is used for the designing of the overall structure of the application, astruct code is for define input and output channel and the ajava code is used for the real implementation of the computational kernels. Here we have provided source code for FFT and FIR algorithms implementation design approach.

In Appendix A contains the API implementation like Fixed Point then Appendix B contains the implementation of the FFT algorithm and finally Appendix C contains the implementation of FIR algorithm.

### 9.1 Appendix A

#### 9.1.1 Fixed Point

As previous we have mentioned that aDesigner does not support floating point so we need source code for convert floating point to fixed point.

```java
import ajava.lang.Math;
import ajava.lang.Math.Marker;

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

    // Methods for operation on fixed point numbers
    public double toDouble() {
        // Convert FixedPoint to double
        return (double) nbInt / (1 << nbFrac);
    }
    public int toInt() {
        // Convert FixedPoint to int
        return nbInt;
    }
    public FixedPoint multiply(FixedPoint other) {
        // Multiply two FixedPoint numbers
        return new FixedPoint(nbInt * other.nbInt, nbFrac + other.nbFrac);
    }
}
```

The range of QI = -128 to 127
The range of QF = 1/(2^24) => 0.000000059604644775390625
public FixedPoint(int nbInt, int nbFrac)
{
    this.nbInt = nbInt;
    this.nbFrac = nbFrac;
}

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

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

public int multiply(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;
}

9.2 Appendix B
In this section we include source code for the Fast Fourier Transform. We only have provided the source code of design file for the 8 point FFT and related java file.
First, the design of application is presented here.

interface Root{}
binding JRoot implements Root{

    void generate() {

        /*There are several variables: points, distribute_stage, 
        * distribute_number, stage.
        * The variable "points" is given by the user, and
        * distribute_stage = the logarithm of (points / 4) to the base 2 plus 1
        * distribute_number = points / 2 minus 1
        * distribute_connection = points / 2 minus 2
        * distribute_connection = distribute_number minus 1
        * stage = logarithm of points/4 to the base 2
        * assembler_number=distribute_number
        * assembler_stage=distribute_stage
        * assume that each processor deals with 2 points
        */

        int points = 8;                                      //Point should be: 8/16/32
        int distribute_stage = 2;
        int distribute_number = 3;
        int stage = 1;
        int distribute_connection = distribute_number - 1;
        int x, y;
        int tmp, l;

        int group, member, interval;
        int tmpName;
        int twiddle0[32];
        twiddle0[0] = 16777216;
Appendix - Source Code

twiddle0[1]=16696429;
......
twiddle0[31]=-16696429;
int twiddle1[32];
twiddle1[0]=0;
twiddle1[1]=-1644455;
......
twiddle1[31]=-1644455;
int i,j,k;
int channelCount;
int count;
Vio io={
    numSources=1,
    numSinks=1
};

Distribute distribute[distribute_number];
//Distribute distribute[4];
count=0;
for(i=0;i<distribute_stage;i++){
    tmp=1;
    for(l=0;l<i;l++){
        tmp=tmp*2;
    }
    for(j=0;j<tmp;j++){
        distribute[count].name="distribute_"+i+j;
        count=count+1;
    }
}
channel c_0={io.out[0], distribute[0].in};
for(i=0;i<distribute_stage-1;i++){
    tmp=1;
    for(l=0;l<i;l++){
        tmp=tmp*2;
    }
    //tmp is equal to power(2,i)
for (j=0; j<tmp; j++) {
    channel c_1=distribute[(tmp-1)+j].out_1, distribute[(2*tmp-1)+(2*j)].in;
    channel c_2=distribute[(tmp-1)+j].out_2, distribute[(2*tmp-1)+(2*j+1)].in;
}

FirstFFT firstFFT[points/2];
for (i=0; i<points/2; i++) {
    firstFFT[i].name="firstFFT_"+i;
} tmp=1;
for (l=0; l<distribute_stage-1; l++) {
    tmp=tmp*2;
} //tmp is equal to power(2, distribute_stage-1)
for (j=0; j<points/4; j++) {
    channel c_3=distribute[(tmp-1)+j].out_1, firstFFT[2*j].in;
    channel c_4=distribute[(tmp-1)+j].out_2, firstFFT[2*j+1].in;
}

RealFFT fft[stage*points/2];
//assign twiddles to processors
for (i=0; i<stage; i++) {
    tmp=1;
    for (l=0; l<stage-i; l++) {
        tmp=tmp*2;
    } //tmp is equal to power(2, stage-i)
    group=tmp;
    member=(points/2)/group;
    //member*2 is the number of twiddles we need in this stage
    interval=16/member;
    for (j=0; j<group; j++) {
        for (k=0; k<member; k++) {
            tmpName=j*member+k;
        }
    }
}
fft[(points/2)*i+tmpName].name="fft_"+i+tmpName;
y=2*k*interval;
fft[(points/2)*i+tmpName].a1=twiddle0[y];
fft[(points/2)*i+tmpName].a2=twiddle1[y];
y=2*k*interval+interval;
fft[(points/2)*i+tmpName].b1=twiddle0[y];
fft[(points/2)*i+tmpName].b2=twiddle1[y];
}
}

//channel between the firstFFT and the first stage of FFT
for(i=0;i<points/4;i++){
channel c_5={firstFFT[2*i].out_1, fft[2*i].in_1};
channel c_6={firstFFT[2*i].out_2, fft[2*i+1].in_1};
channel c_7={firstFFT[2*i+1].out_1, fft[2*i].in_2};
channel c_8={firstFFT[2*i+1].out_2, fft[2*i+1].in_2};
}
for(i=1;i<stage;i++){
tmp=1;
for(l=0;l<stage-i;l++){
    tmp=tmp*2;
}
//tmp is equal to power(2, stage-i)
group=tmp;
member=(points/2)/group;
for(j=0;j<group;j++){
    for(k=0;k<member/2;k++){

//the coordinates of each processor is used to calculate its index
channel c_9={ fft[(i-1)*points/2+j*member+k].out_1, fft[i*points/2+j*member+k].in_1};
channel c_10={ fft[(i-1)*points/2+j*member+k+member/2].out_1, fft[i*points/2+j*member+k].in_2};
channel c_11={ fft[(i-1)*points/2+j*member+k].out_2, fft[i*points/2+j*member+k+member/2].in_1};
channel c_12={ fft[(i-1)*points/2+j*member+k+member/2].out_2, fft[i*points/2+j*member+k+member/2].in_2};
Appendix - Source Code

```c
FinalFFT finalFFT[points/2];
interval=32/points;
for (i=0; i<points/2; i++){
    finalFFT[i].name="finalFFT_"+i;
    y=2*i*interval;
    finalFFT[i].a1=twiddle0[y];
    finalFFT[i].a2=twiddle1[y];
    y=2*i*interval+interval;
    finalFFT[i].b1=twiddle0[y];
    finalFFT[i].b2=twiddle1[y];
}

//channel between the fft on the last stage and the finalFFT
for (i=0; i<points/4; i++){
    channel c_13={fft[(stage-1)*points/2+i].out_1,finalFFT[i].in_1};
    channel c_14={fft[(stage-1)*points/2+i+points/4].out_1,finalFFT[i].in_2};
    channel c_15={ fft[(stage-1)*points/2+i].out_2,finalFFT[i+points/4].in_1};
    channel c_16={ fft[(stage-1)*points/2+i+points/4].out_2,finalFFT[i+points/4].in_2};
}

//the number of assembler is the same to distribute_number
//assembler collect from back site.
AssemblerTwo assembler2[distribute_number];
for (i=0; i<distribute_stage; i++){
    tmp=1;
    for (l=0; l<i; l++){
        tmp=tmp*2;
    }
    //tmp = power(2, i)
    for (j=0; j<tmp; j++){
```

61
assembler2[(tmp-1)+j].name="assembler_"+i+j;
assembler2[(tmp-1)+j].loop=(points/2)/tmp;
}

channel c_17=(assembler2[0].out, io.in[0]);
//interconnection between assemblers
//the stage of assembler is the same to that of distribute
for (i=0; i<distribute_stage-1; i++){
tmp=1;
for (l=0; l<i; l++){
tmp=tmp*2;
}

//tmp is equal to power(2, i)
for (j=0; j<tmp; j++){
channel c_18=(assembler2[(2*tmp-1)+(2*j)].out, assembler2[(tmp-1)+j].in_1 ];
channel c_19=(assembler2[(2*tmp-1)+(2*j+1)].out, assembler2[(tmp-1)+j].in_2 ];
}
}

//connection between assembler and finalFFT
tmp=1;
for (l=0; l<distribute_stage-1; l++){
tmp=tmp*2;
}

//tmp is equal to power(2, distribute_stage-1)
for (j=0; j<points/4; j++){
channel c_20=(finalFFT[2*j].out, assembler2[(tmp-1)+j].in_1];
channel c_21=(finalFFT[2*j+1].out, assembler2[(tmp-1)+j].in_2 ];
}
}
design fft{
    Root root;
}
Appendix - Source Code

/*********** File: Distribute.astruct **********/
* Description: Distribute interface and it contains one input
* and two output ports

interface Distribute{
    inbound in;

    outbound out_1;
    outbound out_2;
}

binding JDistributeBegin implements Distribute{
    implementation "Distribute.java";
    attribute CompilerOptions(targetSR=true) on JDistributeBegin;
}

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

interface RealFFT{
    inbound in_1;
    inbound in_2;
    outbound out_1;
    outbound out_2;
    property int a1;
    property int a2;
    property int b1;
    property int b2;
}

binding JRealFFT implements RealFFT{
    implementation "FFT.java";
}
interface FinalFFT{
    inbound in_1;
    inbound in_2;
    outbound out;
    property int a1;
    property int a2;
    property int b1;
    property int b2;
}

binding JFinalFFT implements FinalFFT{
    implementation "FinalFFT.java";
}

interface AssemblerTwo{
    inbound in_1;
    inbound in_2;
    outbound out;
    property int loop;
}

binding JAssemblerTwo implements AssemblerTwo{
    implementation "AssemblerTwo.java";
    attribute CompilerOptions(targetSR=true) on JAssemblerTwo;
}
Appendix - Source Code

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

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

class Distribute{
    int tmp00;
    int tmp01;
    public void run(InputStream<Integer> in, OutputStream<Integer> out_1,
        OutputStream<Integer> out_2){
        tmp00=in.readInt(); //read real value then store
        tmp01=in.readInt(); //read Image value then store
        out_1.writeInt(tmp00);
        out_1.writeInt(tmp01);
        tmp00=in.readInt();
        tmp01=in.readInt();
        out_2.writeInt(tmp00);
        out_2.writeInt(tmp01);
    }
}

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

class FFT{
    // number of values & pairs
    int pairs00;
    int pairs01;
    int pairs10;
    int pairs11;
}
int tmp00;
int tmp01;
int twiddle00;
int twiddle01;
int twiddle10;
int twiddle11;
int result00;
int result01;

private FixedPoint arithmetic=new FixedPoint(8,24);

public FFT(int a1, int a2, int b1, int b2){
    twiddle00=a1;
    twiddle01=a2;
    twiddle10=b1;
    twiddle11=b2;
}

private void multiply(int a1, int a2, int b1, int b2){
    tmp00=arithmetic.subtract((arithmetic.multiply(a1,b1)),(arithmetic.multiply(a2,b2)));
    tmp01=arithmetic.add((arithmetic.multiply(a1,b2)),(arithmetic.multiply(a2,b1)));
}

public void run(InputStream<Integer> in_1, InputStream<Integer> in_2,
    OutputStream<Integer> out_1, OutputStream<Integer> out_2){
    pairs00=in_1.readInt();
    pairs01=in_1.readInt();
    pairs10=in_2.readInt();
    pairs11=in_2.readInt();
multiply(twiddle00,twiddle01,pairs10,pairs11);
    result00=arithmetic.add(pairs00,tmp00);
    result01=arithmetic.add(pairs01,tmp01);
    out_1.writeInt(result00);
    out_1.writeInt(result01);
    out_2.writeInt(result00);
    out_2.writeInt(result01);
}

    //Calculate the second point
pairs00=in_1.readInt();
pairs01=in_1.readInt();
pairs10=in_2.readInt();
pairs11=in_2.readInt();
multiply(twiddle10, twiddle11, pairs10, pairs11);
result00=arithmetic.add(pairs00,tmp00);
result01=arithmetic.add(pairs01,tmp01);
out_1.writeInt(result00);
out_1.writeInt(result01);
out_2.writeInt(result00);
out_2.writeInt(result01);
}
}

/**********************************************************
* File: AssemblerTwo.java
* Description: this java object combines total no of points and finally
* write to output
**********************************************************/
9.3 Appendix C

In this section we have provide source code for the Finite Impulse Response.

9.3.1 Source code for FIR Algorithm

First, the design file for PerfHarness test of application is presented here.

/**********************************************************
* File: PerfHarnessTestFifoLoader.design
* Description: design file for the Finite Impulse Response
***********************************************************/

design PerfHarnessTestWithFifoLoader {
    Root Root;
}

interface Root {}

binding Root_bind implements Root {
    Vio io = { numSources = 1, numSinks = 1 };

    // run time playback into DUT
    FifoPreloader ddrLoader = { words = 128 };
}
// Each measured unit of work consists of ten input words and ten output words
// PerfHarness perfHarness = { mode="timing", wlWords=10, resWords=10 }; 
PerfHarness perfHarness = { mode="latency", wlWords=10, resWords=10 }; 

// The device under performance testing 
DUT dut; 

cchannel c10 = {io.out[0], ddrLoader.in};
cchannel c20 = {ddrLoader.out, perfHarness.wlIn};
cchannel c30 = {perfHarness.wlOut, dut.in};
cchannel c40 = {dut.out, perfHarness.resIn};
cchannel c50 = {perfHarness.resOut, io.in[0]};

interface DUT { 
    inbound in;
    outbound out;
}

binding DUTBinding implements DUT { 
    
    void generate(){
        int i;
        int taps=64;

        int coef_real[64];
        coef_real[0]=16777216;
        coef_real[1]=16696429;
        ...... 
        coef_real[63]=12968963;

        int coef_img[64];
appendix - source code

c_coef_img[0]=0;
c_coef_img[1]=-1644455;
c_coef_img[2]=-3273072;
......
c_coef_img[63]=-4870169;

c_adder adder[taps-1];
c_coef last coeflast[1];
c_coef coef[taps-1];

for(i=0;i<taps-1;i++){  
  coef[i].name="coef"+i;
  coef[i].a = coef_real[i];
  coef[i].b = coef_img[i];
}

for(i=0;i<taps-1;i++){
  adder[i].name="adder"+i;
}

for(i=0;i<1;i++){
  coeflast[i].name="coeflast"+i;
  coeflast[i].a = coef_real[11];
  coeflast[i].b = coef_img[11];
}

channel c_0=(in, coef[0].in);

for(i=0;i<taps-2;i++){
  channel c_1=(coef[i].o1, coef[i+1].in);
}

channel c_2=(coef[0].o2, adder[0].in_1);

for(i=1;i<taps-1;i++){
  channel c_3=(coef[i].o2, adder[i-1].in_2);
for (i=0; i<taps-2; i++) {
    channel c_4 = {adder[i].out, adder[i+1].in_1};
}

channel c_5 = {adder[taps-2].out, out};

channel c_6 = {coef[taps-2].o1, coeflast[0].in};

channel c_7 = {coeflast[0].out, adder[taps-2].in_2};

}
package fir;

interface coef {
    inbound in;
    outbound o1;
    outbound o2;

    property int a;
    property int b;
}

binding coefBinding implements coef {
    implementation "coef.java";
}

/**
* File: coef.java
* Description: this java object associated with coef.astruct
* interface. This object performs multiplication input sample with
* coefficient.
*/

public class coef {
    int tmp00,tmp01;
    int input_real;
    int input_img;
    int co_real;
    int co_img;

    private FixedPoint arithmetic = new FixedPoint(8,24);

    public coef(int a, int b){
        co_real = a;
        co_img = b;
    }
}
private void multiply(int a1, int a2, int b1, int b2) {

tmp00 = arithmetic.Multiply(a1, b1) - arithmetic.Multiply(a2, b2);
    tmp01 = arithmetic.Multiply(a1, b2) + arithmetic.Multiply(a2, b1);
}

public void run(InputStream<Integer> in, OutputStream<Integer> o1, OutputStream<Integer> o2) {

    input_real = in.readInt();
    input_img = in.readInt();
    o1.writeInt(input_real);
    o1.writeInt(input_img);
    multiply(co_real, co_img, input_real, input_img);
    o2.writeInt(tmp00);
    o2.writeInt(tmp01);
}
/* File: adder.java
* Description: this java object associated with adder.astruct
* interface. This object performs addition two output that come from coef * * objects.
*******************************************************************************************/

public class adder {

    int tmp00, tmp01;
    int tmp10, tmp11;
    int result00;
    int result01;
    private FixedPoint arithmetic = new FixedPoint(8, 24);

    private void add(int a1, int a2, int b1, int b2) {
        result00 = arithmetic.add(a1, b1);
    }
result01 = arithmetic.add(a2,b2);
}

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

tmp00=in_1.readInt();
tmp01=in_1.readInt();
tmp10=in_2.readInt();
tmp11=in_2.readInt();
add(tmp00,tmp01,tmp10,tmp11);
out.writeInt(result00);
out.writeInt(result01);
}
}