DESIGN AND IMPLEMENTATION OF OFDM TRANSMITTER AND RECEIVER ON FPGA HARDWARE

KAMARU ADZHA BIN KADIRAN

UNIVERSITI TEKNOLOGI MALAYSIA

Universiti Teknologi Malaysia

BORANG PENGESAHAN STATUS TESIS◆

JUDUL: **DESIGN AND IMPLEMENTATION OF OFDM TRANSMITTER AND RECEIVER ON FPGA HARDWARE**

SESI PENGAJIAN: **2004/2005**

Saya **KAMARU ADZHA BIN KADIRAN**
(HURUF BESAR)

mengaku membenarkan tesis (PSM/Sarjana/Doktor Falsafah)* ini disimpan di Perpustakaan Universiti Teknologi Malaysia dengan syarat-syarat kegunaan seperti berikut:

1. Tesis adalah hakmilik Universiti Teknologi Malaysia.
2. Perpustakaan Universiti Teknologi Malaysia dibenarkan membuat salinan untuk tujuan pengajian sahaja.
3. Perpustakaan dibenarkan membuat salinan tesis ini sebagai bahan pertukaran antara institusi pengajian tinggi.
4. **Sila tandakan (✔)

☐	SULIT	(Mengandungi maklumat yang berdarjah keselamatan atau kepentingan Malaysia seperti yang termaktub di dalam AKTA RAHSIA RASMI 1972)
☐	TERHAD	(Mengandungi maklumat TERHAD yang telah ditentukan oleh organisasi/badan di mana penyelidikan dijalankan)
☑	TIDAK TERHAD	

Disahkan oleh

(TANDATANGAN PENULIS) (TANDATANGAN PENYELIA)

Alamat Tetap:

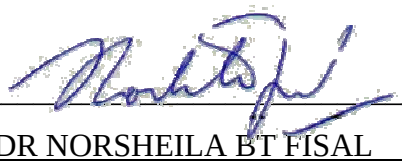
KG PT SINGGAHAN 4,
SERI MEDAN,
83400 BATU PAHAT,
JOHOR.

PROF. DR. NORSHEILA BT FISAL
Nama Penyelia

Tarikh: **10 NOVEMBER 2005** Tarikh: **10 NOVEMBER 2005**

CATATAN:
* Potong yang tidak berkenaan.
** Jika tesis ini SULIT atau TERHAD, sila lampirkan surat daripada pihak berkuasa/organisasi berkenaan dengan menyatakan sekali sebab dan tempoh tesis ini perlu dikelaskan sebagai SULIT atau TERHAD.
◆ Tesis dimaksudkan sebagai tesis bagi Ijazah Doktor Falsafah dan Sarjana secara penyelidikan, atau disertasi bagi pengajian secara kerja kursus dan penyelidikan, atau Laporan Projek Sarjana Muda (PSM).

"I hereby declare that I have read this thesis and in my opinion this thesis is sufficient in terms of scope and quality for the award of the degree of Master of Electrical Engineering (Electronics and Telecommunication)"

Signature : 

Supervisor : PROF DR NORSHEILA BT FISAL

Date : 10 November 2005

DESIGN AND IMPLEMENTATION OF OFDM TRANSMITTER AND RECEIVER ON FPGA HARDWARE

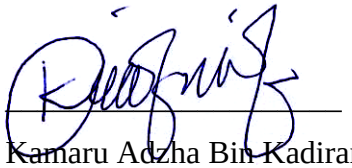
KAMARU ADZHA BIN KADIRAN

A project report submitted in partial fulfillment of the
requirement for the award of the degree of
Master of Electrical Engineering (Electronics and Telecommunication)

Faculty of Electrical Engineering
Universiti Teknologi Malaysia

NOVEMBER, 2005

I declare that this thesis entitled "*Design and Implementation of OFDM Transmitter and receiver on FPGA Hardware* " is the result of my own research except as cited in the references. The thesis has not been accepted for any degree and is not concurrently submitted in candidature of any other degree.

Signature : 

Name : Kamaru Adzha Bin Kadiran

Date : 10 November 2005

Special dedicated to

My Loving Family,
Father, Beloved Brothers,
All my friends and relatives,
And all my teachers and lecturers,
for the support and care.

ACKNOWLEDGEMENT

Praise to Allah S.W.T the Most Gracious, the Most Merciful, whose blessing and guidance have helped me through my thesis smoothly. There is no power no strength save in Allah, the Highest and the Greatest. Peace and blessing of Allah be upon our Prophet Muhammad S.A.W who has given light to mankind.

I would like to take this opportunity to express my deepest gratitude to my supervisor, Prof. Dr. Norsheila Bt Fisal for his guidance, help and encouragement throughout the period of completing my project.

I would like to thank to Mr Illyassak for all the help and guidance especially in using Apex development board and configuration of the related software.

I also would like to thank to Mr. Muladi for his kind assistant and advice in explaining the theory and concept of OFDM system.

I sincerely thank to all my friends and all those whoever has helped me either directly or indirectly in the completion of my final year project and thesis.

ABSTRACT

Orthogonal Frequency Division Multiplexing (OFDM) is a multi-carrier modulation technique which divides the available spectrum into many carriers. OFDM uses the spectrum efficiently compared to FDMA by spacing the channels much closer together and making all carriers orthogonal to one another to prevent interference between the closely spaced carriers. The main advantage of OFDM is their robustness to channel fading in wireless environment. The objective of this project is to design and implement a base band OFDM transmitter and receiver on FPGA hardware. This project concentrates on developing Fast Fourier Transform (FFT) and Inverse Fast Fourier Transform (IFFT). The work also includes in designing a mapping module, serial to parallel and parallel to serial converter module. The design uses 8-point FFT and IFFT for the processing module which indicate that the processing block contain 8 inputs data. All modules are designed using VHDL programming language and implement using Apex 20KE board. The board is connected to computer through serial port and Nios development kit software is used to provide interface between user and the hardware. All processing is executed in Apex board and user only requires to give the inputs data to the hardware throughout Nios. Input and output data is displayed to computer and the results is compared using Matlab software. Software and tools which used in this project includes VHDLmg Design Entry, Synopsys FPGA Express, Altera Maxplus+II and Altera Quartus 3.0. Software tools are used to assist the design process and downloading process into FPGA board while Apex board is used to execute the designed module.

ABSTRAK

Orthogonal Frequency Division Multiplexing (OFDM) adalah salah satu teknik pemodulatan multi-pembawa yang membahagikan satu spektrum frekuensi kepada banyak spektrum pembawa. OFDM menggunakan spektrum dengan lebih effisien berbanding FDMA. OFDM meletakkan saluran berdekatan antara satu sama lain dengan membuatkan setiap pembawa orthogonal dengan yang lain untuk mengelakkan gangguan antara pembawa. Kelebihan OFDM adalah kekuatan signalnya terhadap masalah *channel fading* di dalam persekitaran wireless. Objektif projek ini adalah untuk mereka dan melaksanakan pemancar dan penerima base band OFDM dengan menggunakan perkakasan FPGA. Projek ini menumpukan dalam pembinaan modul *Fast Fourier Transform (FFT)* dan *Inverse Fast Fourier Transform (IFFT*). Selain itu, kerja-kerja merekabentuk modul untuk *mapping*, pengubah sesiri ke selari dan selari ke sesiri juga termasuk dalam skop projek. Semua modul direkabentuk menggunakan bahasa pengaturcaraan VHDL dan dilaksana menggunakan litar Apex 20KE. Litar ini akan disambungkan kepada komputer melalui liang sesiri dan kit perisian Nios digunapakai untuk menyediakan antaramuka kepada pengguna dan perkakasan. Kesemua pemprosessan dilaksanakan oleh litar Apex 20KE yang mana pengguna hanya perlu menmberikan input kepada peranti tersebut. Masukan dan keluaran data akan dipapar melalui skrin komputer dan hasil keputusan akan dibanding dengan perisian Matlab. Antara perisian-perisian yang digunapakai adalah *VHDLmg Design entry, Synopsys FPGA Express, Altera Max+Plus II* dan *Altera Quartus II 3.0*. Perisian yang digunapakai adalah untuk membantu dalam proses merekabentuk modul dan memuat turun program ke dalam peranti manakala litar Apex digunapakai untuk melaksanakan operasi.

TABLE OF CONTENTS

APPENDICES

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

CHAPTER I

INTRODUCTION

1.1 Introduction

This chapter covers the material on project background, project objectives, project scope and the thesis outline. Introduction on this chapter covers about the OFDM implementation method and description on the available hardware for implementation. The problem statement of the project will also be carried out in this chapter.

With the rapid growth of digital communication in recent years, the need for high-speed data transmission has been increased. The mobile telecommunications industry faces the problem of providing the technology that be able to support a variety of services ranging from voice communication with a bit rate of a few kbps to wireless multimedia in which bit rate up to 2 Mbps. Many systems have been proposed and OFDM system has gained much attention for different reasons. Although OFDM was first developed in the 1960s, only in recent years, it has been recognized as an outstanding method for high-speed cellular data communication where its implementation relies on very high-speed digital signal processing. This method has only recently become available with reasonable prices versus performance of hardware implementation.

Since OFDM is carried out in the digital domain, there are several methods to implement the system. One of the methods to implement the system is using ASIC (Application Specific Integrated Circuit). ASICs are the fastest, smallest, and lowest power way to implement OFDM into hardware. The main problem using this method is inflexibility of design process involved and the longer time to market period for the designed chip.

Another method that can be used to implement OFDM is general purpose Microprocessor or Micro Controller. Power PC 7400 and DSP Processor is an example of microprocessor that is capable to implement fast vector operations. This processor is highly programmable and flexible in term of changing the OFDM design into the system. The disadvantages of using this hardware are, it needs memory and other peripheral chips to support the operation. Beside that, it uses the most power usage and memory space, and would be the slowest in term of time to produce the output compared to other hardware.

Field-Programmable Gate Array (FPGA) is an example of VLSI circuit which consists of a “sea of NAND gates” whereby the function are customer provided in a “wire list”. This hardware is programmable and the designer has full control over the actual design implementation without the need (and delay) for any physical IC fabrication facility. An FPGA combines the speed, power, and density attributes of an ASIC with the programmability of a general purpose processor will give advantages to the OFDM system. An FPGA could be reprogrammed for new functions by a base station to meet future needs particularly when new design is going to fabricate into chip. This will be the best choice for OFDM implementation since it gives flexibility to the program design besides the low cost hardware component compared to others.

1.2 Project Background

This project is the continuation from the previous master student project which entitled “Design of an OFDM Transmitter and Receiver Using FPGA” by Loo Kah Cheng. The works involved from previous student is focused on the design of the core processing block using 8 point Fast Fourier Transform (FFT) for receiver and 8 point Inverse Fast Fourier Transform (IFFT) for transmitter part. The implementation of this design into FPGA hardware is to no avail for several reasons encountered during the integration process from software into FPGA hardware.

The project was done up to simulation level using Max+Plus II software and only consists FFT and IFFT processing module. Some of the problem encountered by this student is that the design of FFT and IFFT is not fit to FPGA hardware. The design used a large number of gates and causes this problem to arise. Logic gates are greatly consumed if the number of multiplier and divider are increase. One method to overcome this problem is by decreasing the number of multiplier and divider in the VHDL design.

Beside that, the design does not include control signal which cause difficulties in controlling the data processing in FFT or IFFT module. The control signal is use to select the process executed for each computation process during VHDL design. As a result, the design is not applicable for hardware implementation in the FPGA development board. New design is required to overcome this problem. Since the design is not possible to use, this project will concentrate on designing the FFT and IFFT module which can be implement in the dedicated FPGA board. To ensure that the program can be implemented, the number of gates used in the design must be small or at least less than the hardware can support. Otherwise the design module is not able to implement into the dedicated bord.

1.3 Project Objective

The aim for this project is to design a baseband OFDM processing including FFT (Fast Fourier Transform) and IFFT (Inverse Fast Fourier Transform), mapping (modulator), serial to parallel and parallel to serial converter using hardware programming language (VHDL). These designs were developed using VHDL programming language in design entry software.

The design is then implemented in the Apex 20k200EFC484-2X FPGA development board. Description on the development board will be carried out at methodology chapter.

In order to implement IFFT computation in the FPGA hardware, the knowledge on Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) programming is required. This is because FPGA chip is programmed using VHDL language where the core block diagram of the OFDM transmitter implements in this hardware. The transmitter and receiver are developed in one FPGA board, thus required both IFFT and FFT algorithm implemented in the system.

Several tool involved in the process of completing the design in real hardware which can be divided into two categories, software tools and hardware tool. The software which include in this project is using CAD tools software, VHDL module generator v.109, Synopsis FPGA Express v3.31.4719 and Altera Max+plus II. While the hardware use is UP1 board of ALTERA Flex 10K FPGA chip.

1.4 Project Scope

The work of the project will be focused on the design of the processing block which is 8 point IFFT and FFT function. The design also includes mapping block, serial to parallel and parallel to serial block set. All design need to be verified to ensure that no error in VHDL programming before being simulated. Design process will be described on the methodology chapter.

The second scope is to implement the design into FPGA hardware development board. This process is implemented if all designs are correctly verified and simulated using particular software. Implementation includes hardware programming on FPGA or downloading hardware design into FPGA and software programming.

Creating test vector program also include in the scope of the project. Test vector is a program developed using c programming and is intended as the input interface for user as well as to control data processing performed by the hardware. Creating this software required in understanding the operation of the FFT and IFFT computation process. Further chapter will discuss the method on developing the program from mathematical algorithm into behavioral synthesis.

The last works is to verify the result of the output for each module which has been developed. Test vector program is used to deliver the computation result if input value is provided by the user. These computation values should be verified and tested to ensure the correctness of the developed module. Appropriate software is used to compare the computation performed by the FPGA hardware with the software. There are several test performed to the design modules and the test process also will be discuss in the methodology chapter.

1.5 Project Outline.

The project is organized into six chapters, namely introduction, literature review, methodology, hardware design, software design, result, analysis and discussion, and conclusion.

Chapter 1 discusses the general idea of the project which covers the introduction, project objective, project background and scope of the project.

Chapter 2 shows the literature review of the OFDM system, basic principles of OFDM system, advantages and disadvantages of OFDM system, and lastly is the application of the OFDM in recent technology.

Chapter 3 describes the methodology of the project. The project is divided into several stages which basically include study relevant topics, design stage, implementation stage and testing stage. Further description will cover in this chapter.

Chapter 4 explains regarding the hardware design which is developed from mathematical equations. The chapter also includes on the theoretical part of FFT and IFFT and describes until the hardware design.

Chapter 5 enlightens the software design process involved in the project. This part basically discussed on the works involved to download the modules into FPGA board. Besides that, development of test vector which is used to test the modules will be carried out in this chapter.

Chapter 6 shows the results obtained from the FPGA hardware. The results obtained are captured and show in the figure as an examples. Further results will be shown in the tables.

Chapter 7 describes on the analysis and discussion of the result. Some results which gives error output will be discussed in this chapter and provide the reason behind the problem occurred.

Chapter 8 consists of the conclusion and proposed works to enhance the project in the future.

CHAPTER II

LITERATURE REVIEW

2.1 Introduction

With the rapid growth of digital communication in recent years, the need for high-speed data transmission has increased. The mobile telecommunications industry faces the problem of providing the technology that be able to support a variety of services ranging from voice communication with a bit rate of a few kbps to wireless multimedia in which bit rate up to 2 Mbps. Many systems have been proposed and OFDM system based has gained much attention for different reasons. Although OFDM was first developed in the 1960s, only recently has it been recognized as an outstanding method for high-speed cellular data communication where its implementation relies on very high-speed digital signal processing, and this has only recently become available with reasonable prices of hardware implementation.

2.2 Multichannel Transmission

OFDM started in the mid 60's, Chang [2] proposed a method to synthesis band limited signals for multi channel transmission. The idea is to transmit signals

simultaneously through a linear band limited channel without inter channel (ICI) an inter symbol interference (ISI).

After that, Saltzberg [3] performed an analysis based on Chang's work and he conclude that the focus to design a multi channel transmission must concentrate on reducing crosstalk between adjacent channels rather than on perfecting the individual signals.

In 1971, Weinstein and Ebert [4] made an important contribution to OFDM. Discrete Fourier transform (DFT) method was proposed to perform the base band modulation and demodulation. DFT is an efficient signal processing algorithm. It eliminates the banks of sub carrier oscillators. They used guard space between symbols to combat ICI and ISI problem. This system did not obtain perfect orthogonality between sub carriers over a dispersive channel.

It was Peled and Ruiz [5] in 1980 who introduced cyclic prefix (CP) that solves the orthogonality issue. They filled the guard space with a cyclic extension of the OFDM symbol. It is assume the CP is longer than impulse response of the channel.

2.3 Basic Principles of OFDM

Orthogonal Frequency Division Multiplexing (OFDM) is a multi-carrier transmission technique, which divides the available spectrum into many carriers, each one being modulated by a low rate data stream. OFDM is similar to FDMA in that the multiple user access is achieved by subdividing the available bandwidth into multiple channels that are then allocated to users. However, OFDM uses the

spectrum much more efficiently by spacing the channels much closer together. This is achieved by making all the carriers orthogonal to one another, preventing interference between the closely spaced carriers.

2.4 Orthogonality Defined

Orthogonality is defined for both real and complex valued functions. The functions $\varphi_m(t)$ and $\varphi_n(t)$ are said to be orthogonal with respect to each other over the interval $a < t < b$ if they satisfy the condition:

$$\int_a^b \varphi_m(t)\,\varphi_m^*(t)dt = 0, \quad \text{Where } n \neq m$$

OFDM splits the available bandwidth into many narrowband channels (typically 100-8000), each with its own sub-carrier. These sub-carriers are made orthogonal to one another, meaning that each one has an integer number of cycles over a symbol period. Thus the spectrum of each sub-carrier has a “null” at the centre frequency of each of the other sub-carriers in the system, as demonstrated in Figure 2.0 below. This results in no interference between the sub-carriers, allowing then to be spaced as close as theoretically possible. Because of this, there is no great need for users of the channel to be time-multiplexed, and there is no overhead associated with switching between users. This overcomes the problem of overhead carrier spacing required in FDMA.

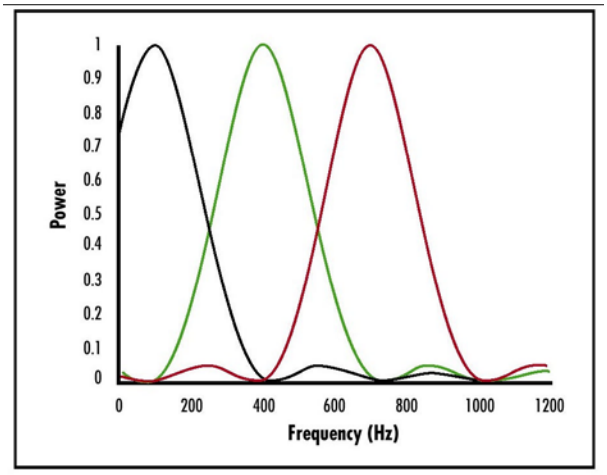
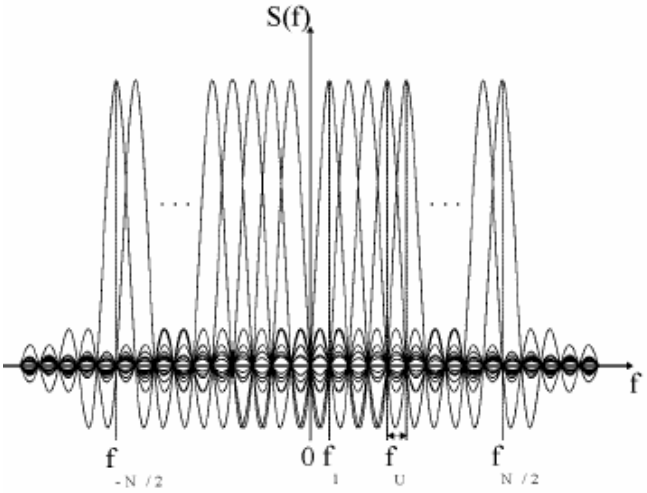


Figure 2.0**:** Orthogonality of sub-carriers

2.5 OFDM Carriers

As fore mentioned, OFDM is a special form of Multi Carrier Modulation (MCM) and the OFDM time domain waveforms are chosen such that mutual orthogonality is ensured even though sub-carrier spectra may over-lap. With respect to OFDM, it can be stated that orthogonality is an implication of a definite and fixed relationship between all carriers in the collection.

It means that each carrier is positioned such that it occurs at the zero energy frequency point of all other carriers. The sinc function, illustrated in Figure 2.1 exhibits this property and it is used as a carrier in an OFDM system.



f_u is the sub-carrier spacing

Figure 2.1: OFDM sub carriers in the frequency domain

2.6 Generation of OFDM Signals

To implement the OFDM transmission scheme, the message signal must first be digitally modulated. The carrier is then split into lower-frequency sub-carriers that are orthogonal to one another. This is achieved by making use of a series of digital signal processing operations.

The message signal is first modulated using a scheme such as BPSK, QPSK, or some form of QAM (16QAM or 64QAM for example). In BPSK, each data symbol modulates the phase of a higher frequency carrier. Figure 2.2 shows the time-domain representation of 8 symbols (01011101) modulated within a carrier using BPSK. In the frequency domain, the effect of the phase shifts in the carrier is to expand the bandwidth occupied by the BPSK signal to a *sinc* function. The zeros (or "nulls") of the *sinc* frequency occur at intervals of the symbol frequency.

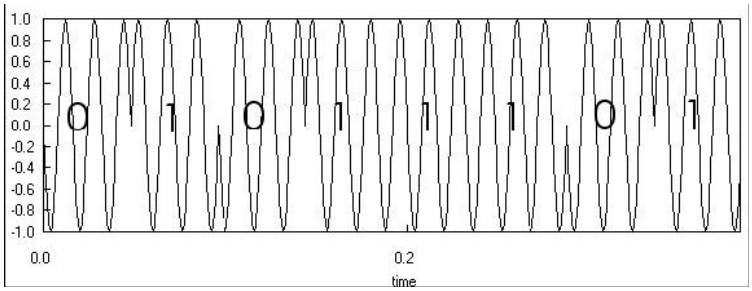


Figure 2.2**:** Binary Phase-Shift Key (BPSK) representation of "01011101"

Originally, multi-carrier systems were implemented through the use of separate local oscillators to generate each individual sub-carrier. This was both inefficient and costly. With the advent of cheap powerful processors, the sub-carriers can now be generated using Fast Fourier Transforms (FFT). The FFT is used to calculate the spectral content of the signal. It moves a signal from the time domain where it is expressed as a series of time events to the frequency domain where it is expressed as the amplitude and phase of a particular frequency. The inverse FFT (IFFT) performs the reciprocal operation.

The underlying principle here is that the FFT can keep tones orthogonal to one another if the tones have an integer number of cycles in a symbol period. In the example figure 2.3 below, we see signals with 1, 2, and 4 cycles respectively that form an orthogonal set.

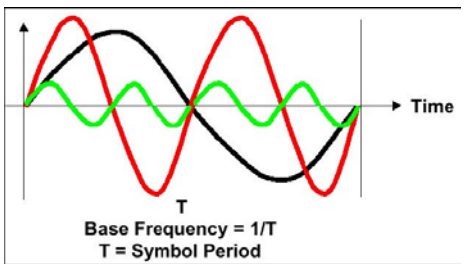


Figure 2.3**:** A set of orthogonal signals

To convert the sub-carriers to a set of orthogonal signals, the data is first combined into frames of a suitable size for an FFT or IFFT. A FFT should be always

in the length of 2N (where N is an integer). Next, an N-point IFFT is performed and the data stream is the output of the transmitter. Thus when the signals of the IFFT output are transmitted sequentially, each of the N channel bits appears at a different sub-carrier frequency.

By using an IFFT process, the spacing of the sub carriers is chosen in such a way that at the frequency where the received signal is evaluated, all other signals is zero. In order for this orthogonality, the receiver and the transmitter must be perfectly synchronized. This means they both must assume exactly the same modulation frequency and the same time-scale for transmission. At the receiver, the exact inverse operations are performed to recover the data. Since the FFT is performed in this stage, the data is back in the frequency domain. It is then demodulated according to the block diagram below.

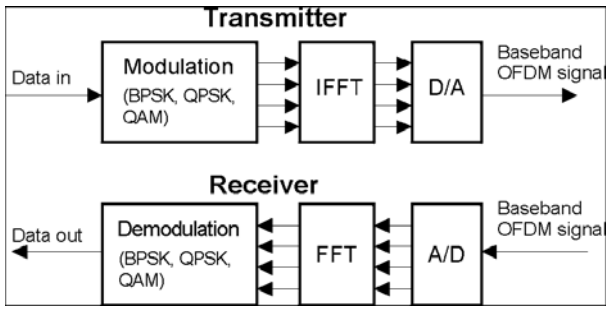


Figure 2.4**:** Block diagram for OFDM communications

2.7 Guard Period

One of the most important properties of OFDM transmission is its robustness against multi path delay. This is especially important if the signal's sub-carriers are to retain their orthogonality through the transmission process. The addition of a guard period between transmitted symbols can be used to accomplish this. The guard

period allows time for multipath signals from the previous symbol to dissipate before information from the current symbol is recorded.

The most effective guard period is a “cyclic prefix”, which is appended at the front of every OFDM symbol. The cyclic prefix is a copy of the last part of the OFDM symbol, and is of equal or greater length than the maximum delay spread of the channel (see Figure 2.5). Although the insertion of the cyclic prefix imposes a penalty on bandwidth efficiency, it is often the best compromise between performance and efficiency in the presence of inter-symbol interference.

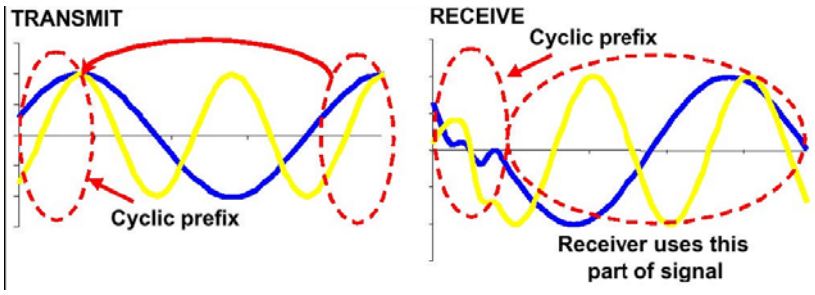


Figure 2.5: Implementation of cyclic prefix

2.8 Advantages of OFDM

OFDM has several advantages compared to other type of modulation technique implemented in wireless system. Below are some of the advantages that describe the uniqueness of OFDM compared to others:

2.8.1 Bandwidth Efficiency

A key aspect of all high-speed communications lies in bandwidth efficiency. This is especially important for wireless communications where all current and future devices are expected to share an already crowded range of carrier frequencies. In OFDM, the frequency band containing the message is divided up into parallel bit streams of lower-frequency carriers, or sub-carriers. These sub-carriers are designed to be orthogonal to one another, such that they can be separated out at the receiver without interference from neighboring carriers. In this manner, OFDM is able to space the channels much closer together, which allows for more efficient use of the spectrum than through simple frequency division multiplexing. The advantage of orthogonality in OFDM does not happen in FDMA where up to 50% of the total bandwidth is wasted due to the extra spacing between channels.

2.8.2 OFDM overcome the effect of ISI

The limitation of sending data in high bit rate is the effect of inter-symbol interference (ISI). As communication systems increase their information transfer speed, the time for each transmission becomes shorter. Since the delay time caused by multi-path remains constant, ISI becomes a limitation in sending high data rate communication. OFDM avoids this problem by sending many low speed transmissions simultaneously. For example figure 2.6 below shows two ways to transmit the same four pieces of binary data.

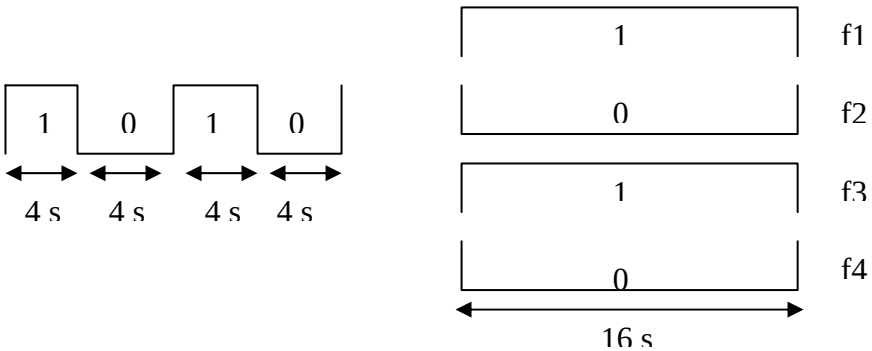


Figure 2.6: Two ways to transmit the same four pieces of binary data

Suppose that this transmission takes four seconds. Then, each piece of data in the left picture has duration of four second. When transmit these data, OFDM would send the four pieces simultaneously as shown on the right. In this case, each piece of data has duration of 16 seconds. This longer duration leads to fewer problems with ISI.

2.8.3 OFDM combats the effect of frequency selective fading and burst error

OFDM is used to spread out a frequency selective fade over many symbols. This effectively randomizes burst errors caused by a deep fade or impulse interference, so that instead of several adjacent symbols being completely destroyed, many symbols are only slightly distorted. This allows successful reconstruction of a majority of them even without forward error correction (FEC). Because of dividing an entire channel bandwidth into many narrow sub-bands, the frequency response over each individual sub-band is relatively flat. Since each sub-channel covers only a small fraction of original bandwidth, equalization is potentially simpler than in a serial system.

2.9 The weakness of OFDM

Although OFDM is excellent in combating fading effect, it does not mean that OFDM is free from any weaknesses. Below are some of the weaknesses for the OFDM system.

2.9.1 Peak-to-Mean Power Ratio

OFDM signal has varying amplitude as shown by figure 2.7. It is very important that the amplitude variations be kept intact as they define the content of the signal. If the amplitude is clipped or modified, then an FFT of the signal would no longer result in the original frequency characteristics, and the modulation may be lost.

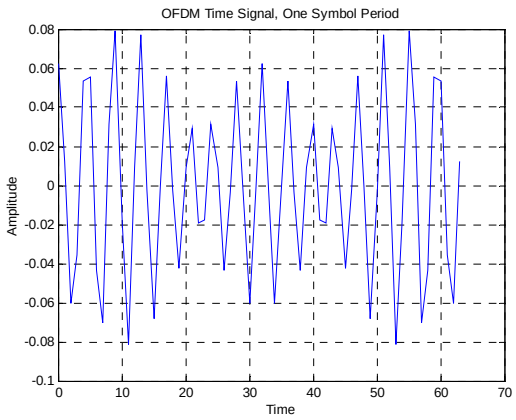


Figure2.7: Show amplitude varying in OFDM

This is one of the drawbacks of OFDM, the fact that it requires linear amplification. In addition, very large amplitude peaks may occur depending on how the sinusoids line up, so the peak-to-average power ratio is high. This means that the linear amplifier has to have a large dynamic range to avoid distorting the peaks. The result is a linear amplifier with a constant, high bias current resulting in very poor power efficiency.

2.9.2 Synchronization

The other limitation of OFDM in many applications is that it is very sensitive to frequency errors caused by frequency differences between the local oscillators in the transmitter and the receiver. Carrier frequency offset causes a number of impairments including attenuation and rotation of each of the sub carriers and inter-carrier interference (ICI) between sub carriers. In the mobile radio environment, the relative movement between transmitter and receiver causes Doppler frequency shifts, in addition, the carriers can never be perfectly synchronized. These random frequency errors in OFDM system distort orthogonality between sub carriers and thus inter-carrier interference (ICI) occurs.

To optimize the performance of an OFDM link, time and frequency synchronization between the transmitter and receiver is of absolute importance. This is achieved by using known pilot tones embedded in the OFDM signal or attach fine frequency timing tracking algorithms within the OFDM signal's cyclic extension (guard interval).

2.10 Application of OFDM.

OFDM has been chosen for several current and future communications systems all over the world. It is well suited for systems in which the channel characteristics make it difficult to maintain adequate communications link performance. In addition to high-speed wireless applications, wired systems such as asynchronous digital subscriber line (ADSL) and cable modem utilize OFDM as its underlying technology to provide a method of delivering high-speed data. Recently, OFDM has also been adopted into several European wireless communications applications such as the digital audio broadcast (DAB) and terrestrial digital video broadcast (DVB-T) systems.

2. 10.1 Digital Broadcasting

Standardized in 1995, Digital Audio Broadcasting (DAB) was the first standard to use OFDM. DAB uses a single frequency network, but the efficient handling of multi path delay spread results in improved CD quality sound, new data services, and higher spectrum efficiency. A broadcasting industry group also created Digital Video Broadcasting (DVB) in 1993. DVB produced a set of specifications for the delivery of digital television over cable, DSL and satellite. In 1997 the terrestrial network, Digital Terrestrial Television Broadcasting (DTTB), was standardized. DTTB utilizes OFDM in up to 2,000 and 8,000 sub-carrier modes.

2. 10.2 Terrestrial Digital Video Broadcasting

A pan-broadcasting-industry group created Digital Video Broadcasting (DVB) in 1993. DVB produced a set of specifications for the delivery of digital television over cable, DSL and satellite. In 1997 the terrestrial network, Digital Terrestrial Television Broadcasting (DTTB), was standardized. DTTB utilizes OFDM in the 2,000 and 8,000 sub carrier modes.

2.10.3 IEEE 802.11a/HiperLAN2 and MMAC Wireless LAN

OFDM in the new 5GHz band is comprised of 802.11a, HiperLAN2, and WLAN standards. In July 1998, IEEE selected OFDM as the basis for the new 802.11a 5GHz standard in the U.S. targeting a range of data rates up to 54 Mbps. In Europe, ETSI BRAN is now working on three extensions for OFDM in the HiperLAN standard: (i) HiperLAN2, a wireless indoor LAN with a QoS provision; (ii) HiperLink, a wireless indoor backbone; and (iii) HiperAccess, an outdoor, fixed

wireless network providing access to a wired infrastructure. In Japan, consumer electronics companies and service providers are cooperating in the MMAC project to define new wireless standards similar to those of IEEE and ETSI BRAN.

2.10.4 Mobile Wireless Communication.

OFDM's capability to work around interfering signals gives it potential to threaten existing CDMA (2.5G and 3G) wireless technology. This is what is allowing the technology to push forward in Europe. In densely populated areas where buildings, vehicles and people can scatter the path of a signal, broadcasters as well as high-speed data providers are anxious to eliminate multi-path effects. According to industry analysts, telecom providers may also be lured to OFDM technology because it could end up causing only a fraction of what it costs to implement 3G wireless technologies.

CHAPTER III

METHODOLOGY

3.1 Introduction

This chapter discusses the methodology of the project and tools that involved in the process to complete the design and implementation of OFDM tranciever in the FPGA hardware. The topic basically covers on the usage of the software and some explanation on the Altera APEX development board.

The methodology of the project is basically divided into four main stages. These stages is started with study the relevant topics and followed by the design process, implementation, test and analysis stages. All stages are subdivided into several small topics or sub-stages and explanation for each stage will be carried out in this chapter.

Several software are used throughout the stages which shown as Figure 3.1.. Each of the software function will be discussed in this chapter. For hardware part, an APEX 20K200E is used and some documentation regarding this hardware also will be shown.

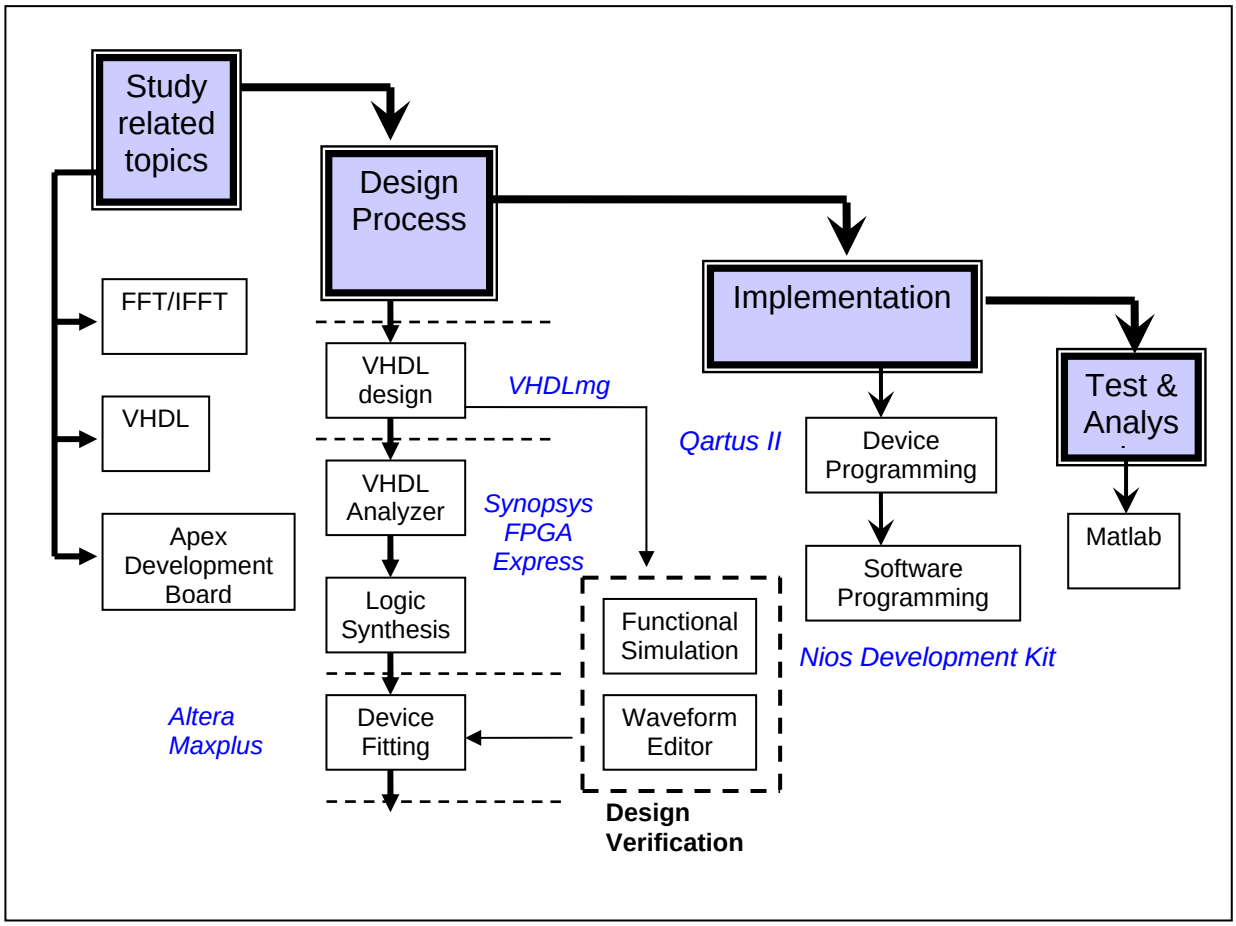


Figure 3.1: Flow chart of the projects methodology.

3.2 Study Relevant Topics

Figure 3.1 depicts the flow of the projects methodology. As mention before, methodology of the project is divided into three main stages. The first stage will cover on study the relevant topics. On this stage, the works is subdivided into three main topics which is FFT and IFFT, VHDL programming and Altera Apex development board. These are the topics that need to cover before moved into the design process. A study on FFT and IFFT is required to understand the computation process. This requirement is important especially during hardware development and software programming part. Bit representation in binary also is another issue which require to study in this stage. Bit representation is crucial when the multiplication or addition process involved point values such as twiddle factor. In VHDL topic, there are two topic need to cover up which is Register Transfer Logic (RTL) and

Behavioral Modeling and Synthesis. The last part in this stage is to study the Altera Apex development board. The description for this board will be carried out later in this chapter.

3.3 Design Process

After all preparation in theoretical part is covered, the works is continued into the design process stages. For this stage, the process is subdivided into several topics which are VHDL design, VHDL analyzer, Logic Synthesis, device fitting, and design verification. These topics actually are the process involved to complete the hardware design. Each of the process required different software to accomplish the design.

VHDL design is the first steps to perform in the design process. VHDLmg software is used as the design entry and programmed in VHDL language. Basically this process is to generate the VHDL source code. After generating the code, Synopsys FPGA Express software is used to verify the generated code. The software will perform two processes which is VHDL analyzer and logic synthesis. VHDL analyzer output is used as the logic synthesis and design verification. In logic synthesis, the netlist file which obtain from VHDL analyzer is synthesized base on the design constrain and technology library available in the software. The software will produce *.edif for output file. This file is then used in technology mapping which is performed by Altera Max+Plus II software. In technology mapping, a process called device fitting is executed to partition, fit, place and route the design base on the targeted device.

Device fitting process will produces three main output file which is *.pof, *.sof and *.snf file. The *.snf file is used for the design verification which also performed by Altera Max+Plus II software. There are two types of simulation at the design verification which is functional simulation and timing simulation. The

functional simulation is to simulate the hardware function and this process is not carried out since the software used is not available. But the timing simulation is perform using Max + Plus II software. The timing simulation is providing the timing function for the designed hardware.

3.4 Implemetation

The design is then proceed to the implementation stage. There are two processes in this stage which is device programming and software programming. Device programming is the process to program FPGA board using Quartus II software. This process basically will burn hardware design into FPGA board. Another task is to create test vector program in C. Creating this program is include as in the software programming process.

3.5 Tests and Analysis

Final stage involved is the test and analysis stage. During this stage, the output from hardware computation is compared with Matlab. This is to ensure that the design module is correctly works as performed by Matlab software.

3.6 VHDL and VHDLmg Software

VHDL is an acronym for VHSIC (Very high Speed Integrated Circuit) Hardware Description Language. It is a hardware description language that can be used to describe the structure and/or behavior of hardware designs and to model

digital systems at many levels of abstraction, ranging from the algorithmic level to gate level. The VHDL designs can be simulated and/or synthesized and permits the rapid creation of complex hardware designs.

```
ENTITY entity_name IS
        generics
        port declarations
END entity_name;

ARCHITECTURE arch_name OF entity_name IS
        enumerated data types
        internal signal declarations
        component declarations
BEGIN
        signal assignment statements
        process statements
        component instantiations
END arch_name;
```

Figure 3.2: Basic Modeling structure for VHDL.

Figure 3.2 show the structure of VHDL programming which basically describe the digital component's behavior in term programming language. A circuit or sub circuit described with VHDL code is called a design entity. General structure has two main parts which is entity declaration and architecture. For entity declaration, it specifies the input and output signals for the entity. The architecture part basically gives the details of the circuit in term of programming. Figure 3.3 shows the picture of VHDL design entity.

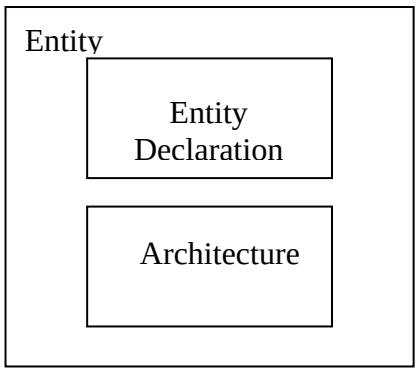


Figure 3.3: Show the VHDL design entity.

3.7 Synopsys FPGA Express Software

FPGA Express provides logic synthesis and optimization, so you can automatically convert a VHDL description to a gate-level implementation in a given technology. This methodology eliminates the former gate-level design bottleneck and reduces circuit design time and errors introduced when hand translating a VHDL specification to gates.

Before the synthesize process, the design entry from VHDL mg must be analyzed first. If the design entry is error-free, then the synthesize process can be done. But, if the design entry still has an error, it must be corrected and analyzed again. There are two steps in implementation of this synthesize process which are Create Implementation and Export Netlist. Create Implementation is where the circuit is created by FPGA Express. While Export Netlist is for the synthesized circuit from FPGA Express is sent to Max+Plus II.

3.8 Altera Max+Plus II Software

This is a testing stage using Max+Plus II software for the implemented system. During this simulation process, the circuit design will be simulated and the output can be seen at the timing diagram. Based on the output from timing diagram, the circuit design can be detected whether it is functional or not.

3.9 Matlab

Matlab is the multi purpose software which usually use for mathematical computation in the engineering field. In this project Matlab software is used for verification of the IFFT computation with the Max+Plus output simulation.

3.10 APEX 20K Devices board

The APEX™ device family ranges from 30,000 to over 1.5 million gates (113,000 to over 2.5 million system gates) and ships on 0.22-µm, 0.18-µm, and 0.15-µm processes. Introduced in 1999, the APEX device family extended Altera's leadership in embedded PLD architectures to new levels of efficiency and performance. APEX devices are uniquely suited for system-on-a-programmable-chip (SOPC) solutions, allowing designers to integrate a system efficiently and use it in a broad range of applications.

APEXTM 20K devices are the first PLDs designed with the Multi Core architecture, which combines the strengths of LUT-based and product term-based devices with an enhanced memory structure. LUT-based logic provides optimized performance and efficiency for data-path, register intensive, mathematical, or digital signal processing (DSP) designs. Product-term-based logic is optimized for complex combinatorial paths, such as complex state machines. LUT- and product-term-based logic combined with memory functions and a wide variety of Mega Core and AMPP functions make the APEX 20K device architecture uniquely suited for system-on-a-programmable-chip designs. Applications historically requiring a combination of LUT-, product-term-, and memory-based devices can now be integrated into one APEX 20K device.

APEX 20KE devices are a superset of APEX 20K devices and include additional features such as advanced I/O standard support, CAM, additional global clocks, and enhanced ClockLock clock circuitry. In addition, APEX 20KE devices extend the APEX 20K family to 1.5 million gates. APEX 20KE devices are denoted with an “E” suffix in the device name (e.g., the EP20K1000E device is an APEX 20KE device).

3.11 Quartus II 3.0

The Quartus® II development software provides a complete design environment for system-on-a-programmable-chip (SOPC) design. Regardless of whether you use a personal computer or a UNIX or Linux workstation, the Quartus II software ensures easy design entry, fast processing, and straightforward device programming.The Quartus II software is a fully integrated, architecture-independent package for designing logic with Altera® programmable logic devices, including ACEX® 1K, APEX™ 20K, APEX 20KC, APEX 20KE, APEX™ II, ARM®-based Excalibur™, Cyclone™, FLEX® 6000, FLEX 10K®, FLEX 10KA, FLEX 10KE, MAX® 3000A, MAX 7000AE, MAX 7000B, MAX 7000S, Mercury™, Stratix, and Stratix™ GX devices. The Quartus II software offers a full spectrum of logic design capabilities such as design entry using schematics, block diagrams, AHDL, VHDL, and Verilog HDL, floorplan editing, functional and timing simulation, timing analysis, combined compilation and software projects, device programming and verification and many more. The Quartus II software also reads standard EDIF netlist files, VHDL netlist files, and Verilog HDL netlist files, and generates VHDL and Verilog HDL netlist files, including VITAL-compliant files, for a convenient interface to other industry-standard EDA tools.

CHAPTER IV

HARDWARE DESIGN

4.1 Introduction

This chapter covers the material on the Fast Fourier Transform and Inverse Fast Fourier Transform theories, Mapping, Serial to Parallel and Parallel to Serial block, VHDL design modules and verification of the design modules. Behavioral synthesis is used to transfer the mathematical algorithm into register level process and this will be discussed further in this chapter.

There are various types of transmitter design for OFDM transmitter. Some of the design use DSP chip as the main part to implement the core-processing block, which is IFFT computation. This issue has been discussed in the previous chapter and as stated in that chapter, FPGA is the most cost effective to implement the design. As mentioned before, the OFDM transmitter consists of several block or modules to implement the system using the IFFT function. After consulting various books, white paper and journal, the proposed transmitter design is consist of serial to parallel converter, modulator bank, processing block, parallel to serial converter and cyclic prefix block module. This transmitter block diagram is close to the standard for all OFDM systems. It was in close accordance with the systems discussed in the

primary resource textbooks. These sources and several technical papers, served as useful tools to validate our design.

4.2 Simplified Transmitter Block Diagram

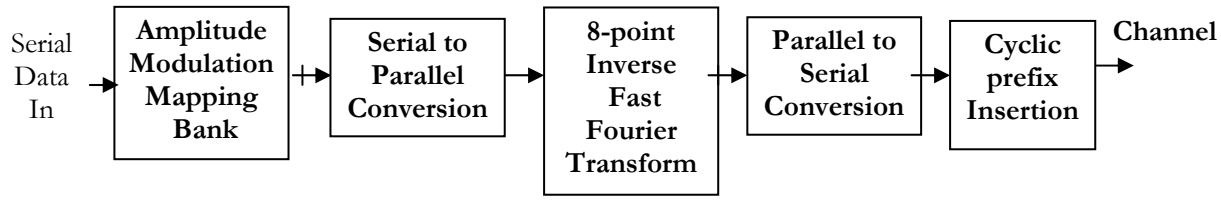


Figure 4.1: Simplified transmitter block diagram.

Figure above show the simplified block diagram of OFDM transmitter. It can be seen that the block is divided into several parts with each block function differently and this is to ensure that the system works effectively. Since the main component is processing block, so, the work is started from this part. All block set function is implemented in the FPGA development board. Cyclic prefix is a module, which is used to concatenate partial end of information bit and put at the beginning of the information frame. But in this project cyclic prefix is not included in this design because it is not the project scope. All module function will be discussed further in this chapter.

The generation of OFDM signal started from amplitude modulation mapping bank. The serial input data is mapped to appropriate symbol to represent the data bits. These symbols are in serial and need to convert into parallel format since IFFT module requires parallel input to process data. The serial to parallel module does the conversion. These parallel symbols are transformed from frequency domain into time domain using IFFT module. These signals are converted into serial format and add a cyclic prefix to data frame before being transmitted.

4.3 Simplified Receiver Block Diagram

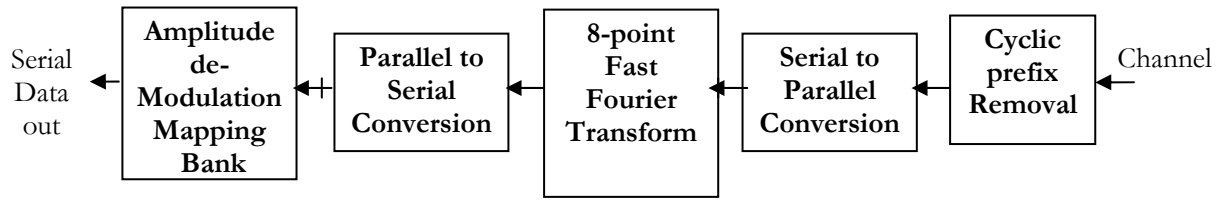


Figure 4.2: simplified receiver block diagram.

Figure above show the basic block diagram for receiver module. There are five modules in the receiver block and as mention before, cyclic prefix removal will not be included into the design. The received data is in serial format, thus, since FFT input is in parallel, a module which use to converts from serial to parallel is required. Output from FFT is converted back to serial format through parallel to serial converter. The conversion is required since the serial data need to be transmitted. Finally the serial output is demodulated using de-mapping module to get the transmitted data.

4.4 Mapping Module

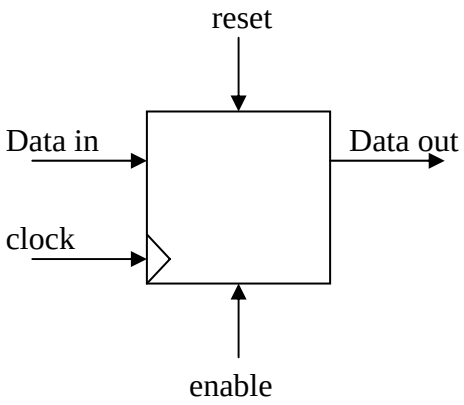


Figure 4.3: Mapping module.

Figure 4.3 show the mapping module for transmitter. The mapping module used is BPSK type of modulation. BPSK is used because module is much easier to design compared to QPSK or other modulation method. If the input is ‘1’ then the value is mapped with ‘1’ while if the input is ‘0’ the value is mapped with ‘0’. This type of modulation is monopodal type. The input passed through this module actually does not get any changes to the value, but it can be assumed that the input is modulated after pass through it.

4.5 Serial to Parallel module

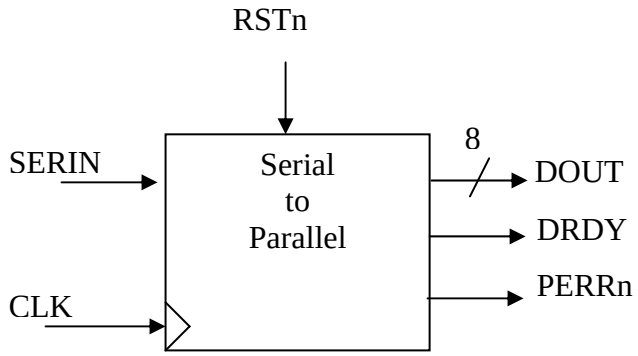


Figure 4.4: Block diagram for Serial to Parallel module.

A serial to parallel converter is somewhat the reverse of the operation of parallel to serial converter. The data comes serially from the input port SERIN. The parallel data is output from DOUT port. Output port DRDY is asserted ‘1’ when the start bit, 8 bit data and the parity bit is received. Output port PERRn is asserted ‘0’ when the parity bit received is different from the parity generated inside the serial to parallel circuit. When parity error is detected, the serial to parallel circuit would be reset before its normal operation can be performed. This is the operation for serial to parallel module. Source code is provided in the appendices chapter.

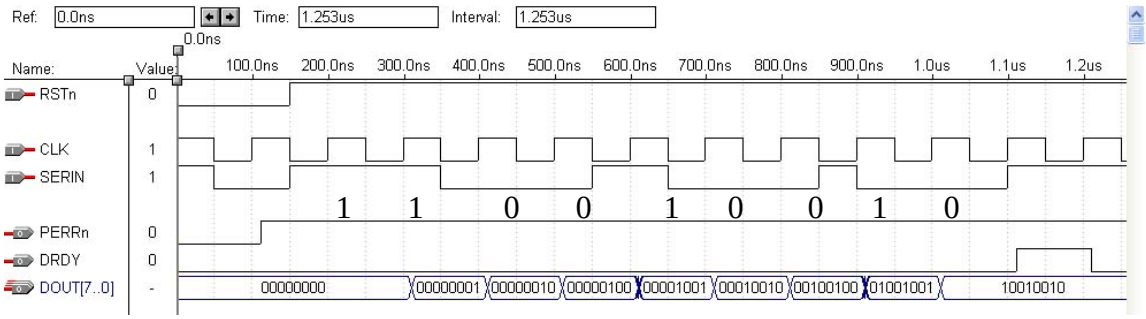


Figure 4.5: Simulation waveform of the serial to parallel.

The figure 4.5 shows a simulation waveform for an input data ‘11001001’. The input data is in serial format and the conversion is started with the start bit is being asserted ‘1’ in the SERIN input. Then, the SERIN input receives serial data ‘1’,’1’,’0’,’0’,’1’,’0’,’0’,’1’ followed by the even parity bit of value ‘0’. After the parity bit is received, the output signal DRDY is asserted ‘1’ in the next clock cycle. The DRDY signal is used to tell another circuit block to get the parallel data from DOUT right away. Otherwise the data may be lost when the next word comes. The DRDY and the start bit are allows to be asserted simultaneously and DOUT’s value is changed right after DRDY is disserted. The old data is shifted out bit by bit. Output PERRn is not asserted since the parity error is not detected. A source code in VHDL programming for serial to parallel is attached at the appendix pages.

4.6 Parallel to Serial module.

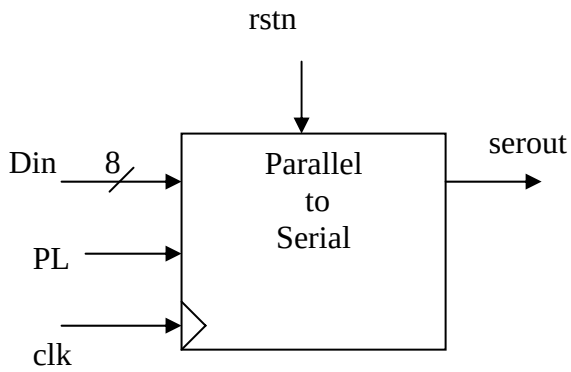


Figure 4.6: Block diagram of Parallel to Serial module.

A parallel to serial converter is a special function of shift register. The data is parallel loaded to the shift register and then shift out bit by bit also is bounded by a start bit and stop bit. In OFDM transmitter module, a parallel to serial converter is used to convert computation result which is in parallel to serial before being sent to other module for processing. This parallel to serial module is design such that the data to be transmit is first parallel loaded then transmitted bit by bit by a start bit of value ‘1’. This is followed by the 8-bit data with the left bit most bit first. The converter holds the output low when the transmission is completed.

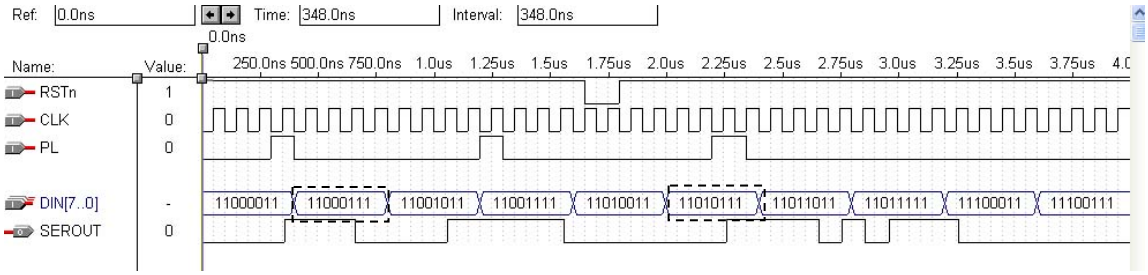


Figure 4.7: Simulation waveform for parallel to serial.

Figure 4.7 above show three example of data conversion from parallel to serial. When input signal PL is asserted ‘1’, the data DIN “11000111” is parallel loaded into the parallel to serial circuit. In the next clock cycle, a start bit of ‘1’ is outputted, followed by the data “11000111”, then completed with an even parity bit of value ‘1’. After that, the output stays at low until the PL input is asserted again. The second data is “11001111” and start bit value is ‘1’. But during data conversion RSTn signal is asserted to ‘0’ result that the output of SEROUT is ‘0’. The third data is “11010111”. The start bit is same followed by data and parity bit value is ‘1’. Further source code for this module is attached in the appendix pages.

4.7 Fast Fourier Transform

Before going further to discus on the FFT and IFFT design, it is good to explain a bit on the Fast Fourier Transform and Inverse Fast Fourier Transform operation. The Fast Fourier Transform (FFT) and Inverse Fast Fourier Transform (IFFT) are derived from the main function which is called Discrete Fourier Transform (DFT). The idea of using FFT/IFFT instead of DFT is that the computation of the function can be made faster where this is the main criteria for implementation in the digital signal processing. In DFT the computation for N-point of the DFT will calculate one by one for each point. While for FFT/IFFT, the computation is done simultaneously and this method saves quite a lot of time. Below is the equation showing the DFT and from here the equation is derived to get FFT/IFFT function.

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j2\pi k / N} \qquad (4.1)$$

X(k) represent the DFT frequency output at the *k*-the spectral point where *k* ranges from 0 to *N-1*. The quantity *N* represents the number of sample points in the DFT data frame. The quantity *x(n)* represents the *n*-th time sample, where *n* also ranges from 0 to *N-1*. In general equation, *x(n)* can be real or complex.

The DFT equation can be re-written into:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk} \qquad (4.2)$$

The quantity W_N^{nk} is defined as:

$$W_N^{nk} = e^{-j2\pi nk / N} \qquad (4.3)$$

Here is where the secret lies between DFT and FFT/IFFT where the function above is called Twiddle Factor. This factor is calculated and put in a table in order to

make the computation easier and can run simultaneously. The Twiddle Factor table is depending on the number of point use. During the computation of IFFT, the factor does not to recalculate since it can refer to the Twiddle factor table thus it save time since calculation is done concurrently. Below is the table for 8 point of FFT for twiddle factor.

Table 4.0: Twiddle Factor value for FFT

FFT (N = 8)		
nk	**W**	**Value**
1	W_8^0	*1*
2	W_8^1	*0.7071 – j0.7071*
3	W_8^2	*-j1*
4	W_8^3	*-0.7071 – j0.7071*
5	W_8^4	*-1*
6	W_8^5	*-0.7071 + j0.7071*
7	W_8^6	*j1*
8	W_8^7	*0.7071 + j0.7071*

For decimation in frequency radix-2, the input is separated into two halves which is:

$$x(0), x(1), ..., x\left(\frac{N}{2}-1\right) \qquad (4.4)$$

and

$$x\left(\frac{N}{2}\right), x\left(\frac{N}{2}+1\right), ..., x(N-1) \qquad (4.5)$$

Thus the DFT also can be separated into two summations:

$$X(k) = \sum_{n=0}^{(N/2)-1} x(n)W^{nk} + W^{kN/2} + \sum_{n=N/2}^{N-1} x(n)W^{nk} \qquad (4.6)$$

Substituting the input into equation above, the result is:

$$X(k) = \sum_{n=0}^{(N/2)-1} x(n)W^{nk} + W^{kN/2} \sum_{n=0}^{(N/2)-1} x\left(n+\frac{N}{2}\right)W^{nk} \qquad (4.7)$$

Substituting k = 2k for even and k = 2k + 1 for odd the equation become as:

$$X(2k) = \sum_{n=0}^{(N/2)-1}\left[x(n) + x\left(n+\frac{N}{2}\right)\right]W^{2nk}, \quad k = 0,1,.....,\left(\frac{N}{2}\right)-1 \qquad (4.8)$$

and

$$X(2k+1) = \sum_{n=0}^{(N/2)-1}\left[x(n) - x\left(n+\frac{N}{2}\right)\right]W^{n}W^{2nk}, \quad k = 0,1,.....,\left(\frac{N}{2}\right)-1 \qquad (4.9)$$

Furthermore, let:

$$a(n) = x(n) + x\left(n+\frac{N}{2}\right) \qquad (4.10)$$

$$b(n) = x(n) - x\left(n+\frac{N}{2}\right) \qquad (4.11)$$

To equation ….:

$$X(2k) = \sum_{n=0}^{(N/2)-1} a(n)W_{N/2}^{nk} \qquad (4.12)$$

$$X(2k+1) = \sum_{n=0}^{(N/2)-1} b(n)W_{N/2}^{nk} \qquad (4.13)$$

The equation above shows that for FFT decimation in frequency radix 2, the input can be grouped into odd and even number. Thus, graphically the operation can be view using FFT flow graph shown in figure.

4.7.1 FFT Signal Flow Graph.

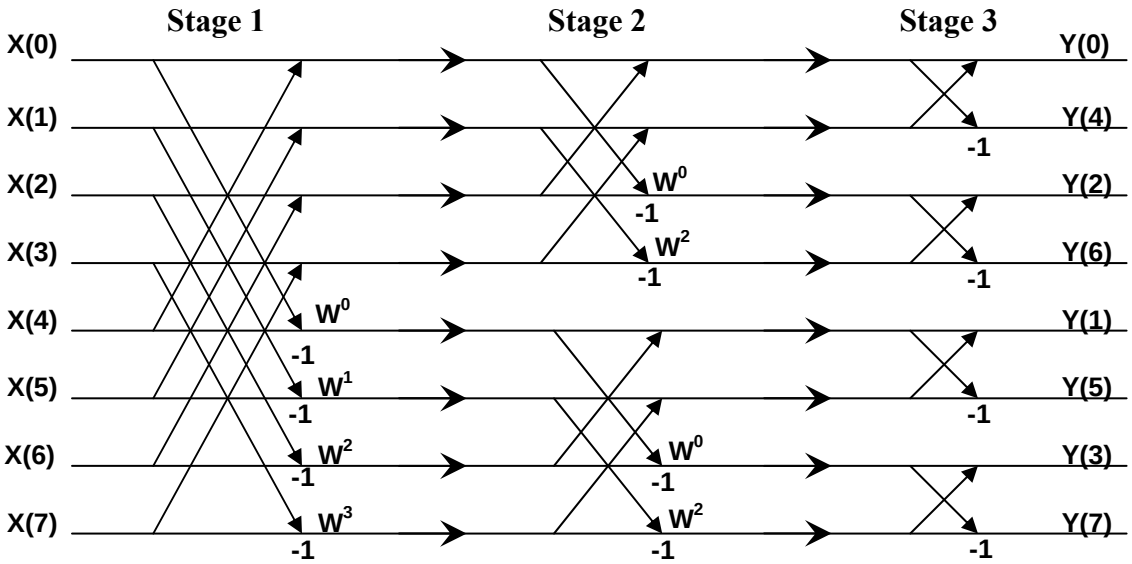


Figure 4.8: 8-point FFT flow graph using decimation-in-frequency (DIF).

From this figure, the FFT computation is accomplished in three stages. The X(0) until X(7) variable is denoted as the input value for FFT computation and Y(0) until Y(7) is denoted as the output. There are two operations to complete the computation in each stage. The upward arrow will execute addition operation while downward arrow will execute subtraction operation. The subtracted value is multiplied with twiddle factor value before being processed into the nest stage. This operation is done concurrently and is known as butterfly process. For second stage, there are two butterfly process with each process get reduced input variable. In the first stage the butterfly process get eight input variable while in the second stage, each butterfly process get four input variable that is from first stage computation. This process is continued until third stage. In third stage, there are four butterfly processes. Noted that each of the butterfly process is performed concurrently enable it to execute FFT computation process in a very fast technique.

Mathematically, the butterfly process for each stage can be derived as the equation stated in the next page.

FFT Stage 1

X(0) + X(4) => X’(0),
X(1) + X(5) => X’(1),
X(2) + X(6) => X’(2),
X(3) + X(7) => X’(3),
[X(0) – X(4)]W0 => X’(4),
[X(1) – X(5)]W1 => X’(5),
[X(2) – X(6)]W2 => X’(6),
[X(3) – X(7)]W3 => X’(7),

FFT Stage 2

X’(0) + X’(2) => X”(0),
X’(1) + X(3) => X”(1),
[X’(0) – X’(2)]W0 => X”(2),
[X’(1) – X’(3)]W0 => X”(3),
X’(4) + X’(2) => X”(4),
X’(5) + X(3) => X”(5),
[X’(4) – X’(6)]W0 => X”(6),
[X’(5) – X’(7)]W0 => X”(7),

FFT Stage 3

X”(0) + X”(1) => Y(0),
X”(1) – X”(5) => Y(1),
X”(2) + X”(3) => Y(2),
X”(2) – X”(3) => Y(3),
X”(4) + X”(5) => Y(4),
X”(4) – X”(5) => Y(5),
X”(6) + X”(7) => Y(6),
X”(6) – X”(7) => Y(7),

4.7.2 FFT Scheduling Diagram

In the stage three, final computation is done and the result is sent to the variable Y(0) to Y(7). Equation in each stage is used to construct scheduling diagram. Scheduling diagram is part of Behavioral Modeling and Synthesis steps to translate the algorithmic description into RTL (register transfer level) in VHDL design. The scheduling diagram for stage one computation is constructed as figure 4.9 below.

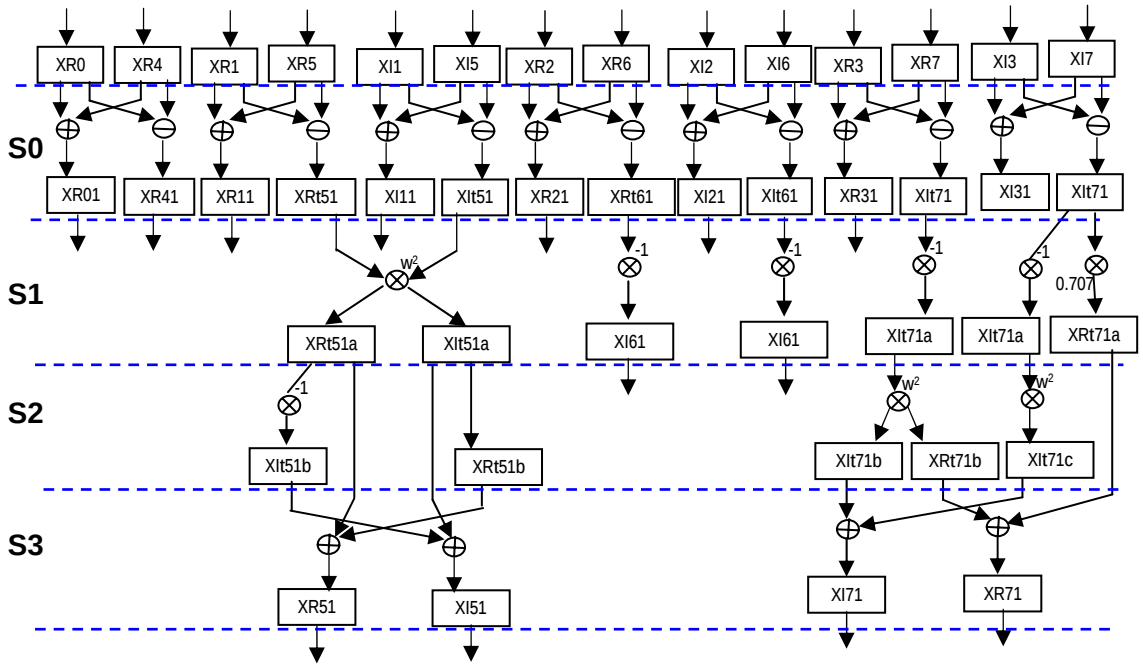


Figure 4.9: Scheduling diagram for stage one of 8 point FFT.

Base on figure 4.9., variable XR0, XR4, and the rest is denoted as the register in FPGA. The register is name as such to ensure that each register has its own unique name. During computation, these register will hold computation value, thus it is required to be unique for easy recalling the value when needed. S0 until S3 is denoted as clock cycle. Computation in stage one requires four clock cycle to complete before moves to the next stage. In this stage, the operation takes longer clock cycles because of the multiplication of twiddle factor value. Since twiddle

factor value is complex, computation need to separate real value and imaginary value. XR denoted as real value while XI is for imaginary.

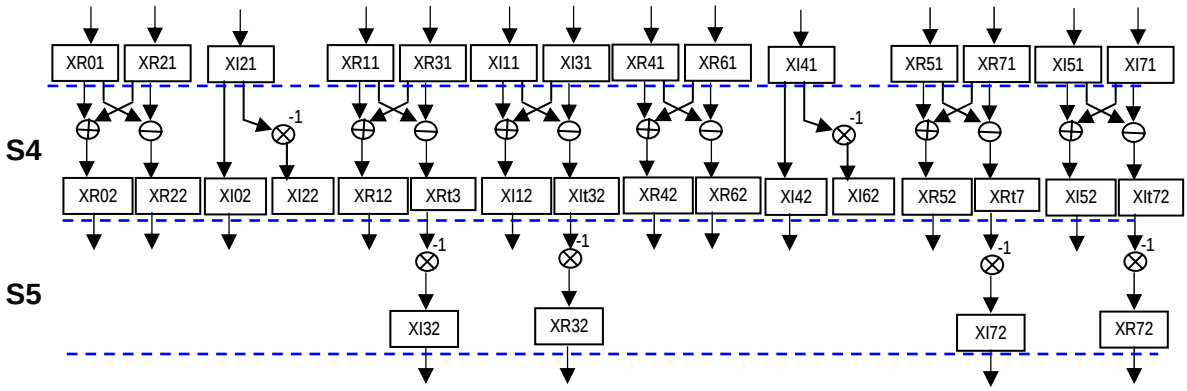


Figure 4.10: Scheduling diagrams for stage two of 8 point FFT.

Figure 4.10 show the scheduling diagram for stage two. The value from stage one computation is sent to this stage as input. The number of register to store computed values from stage one is same because it is already allocated to receive real and imaginary values. The situation is different for IFFT operation. This will be discussed later on IFFT topic. FFT commonly is used at the receiver to convert time domain signal into frequency domain. XR01 until XI71 denoted as the register name in stage two.

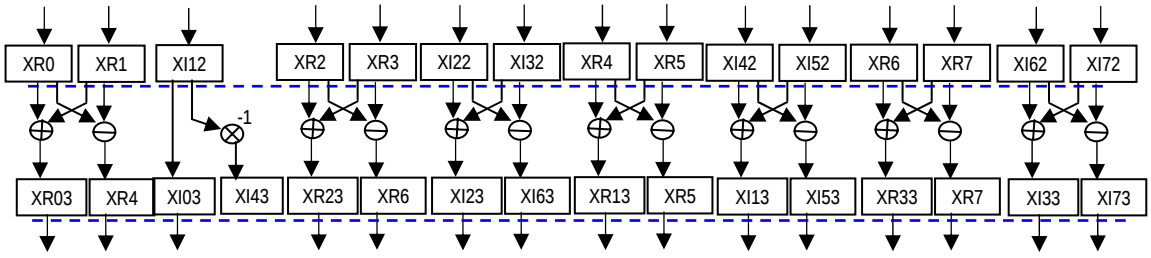


Figure 4.11: Scheduling diagrams for stage three of 8 point FFT.

Figure 4.11 show the last stage of FFT computation. The register XR03 until XI73 holds output values for FFT. These register will be call upon when displaying the result during software programming.

4.8 Inverse Fast Fourier Transform

As mention in previous chapter, Inverse Fast Fourier Transform (IFFT) is used to generate OFDM symbols. The data bits is represent as the frequency domain and since IFFT convert signal from frequency domain to time domain, it is used in transmitter to handle the process. IFFT is defined as the equation below:

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) W^{-nk}, \; k = 0,1.. \qquad (4.14)$$

Comparing this equation with the equation (1)., it is shown that the same FFT algorithm can be used to find IFFT function with the changes in certain properties. The changes that implement is by adding a scaling factor of 1/N and replacing twiddle factor value (W^{nk}) with the complex conjugate W^{-nk} to the equation (1). With these changes, the same FFT flow graph also can be used for the Inverse fast Fourier Transform. Below is the table show the value of twiddle factor for IFFT.

Table 4.1: Twiddle factor for 8 point Inverse Fast Fourier Transform.

IFFT (N = 8)		
nk	**W**	**Value**
1	W_8^{-0}	*1*
2	W_8^{-1}	*0.7071 + j0.7071*
3	W_8^{-2}	*j1*
4	W_8^{-3}	*-0.7071 + j0.7071*
5	W_8^{-4}	*-1*
6	W_8^{-5}	*-0.7071 - j0.7071*
7	W_8^{-6}	*-j1*
8	W_8^{-7}	*0.7071 - j0.7071*

Base on the equation obtain from signal flow graph, the scheduling diagram is developed. Eight registers is required to store input value from user. These registers only accept real value as the input for IFFT operation. Below is the scheduling diagram for IFFT stage 1.

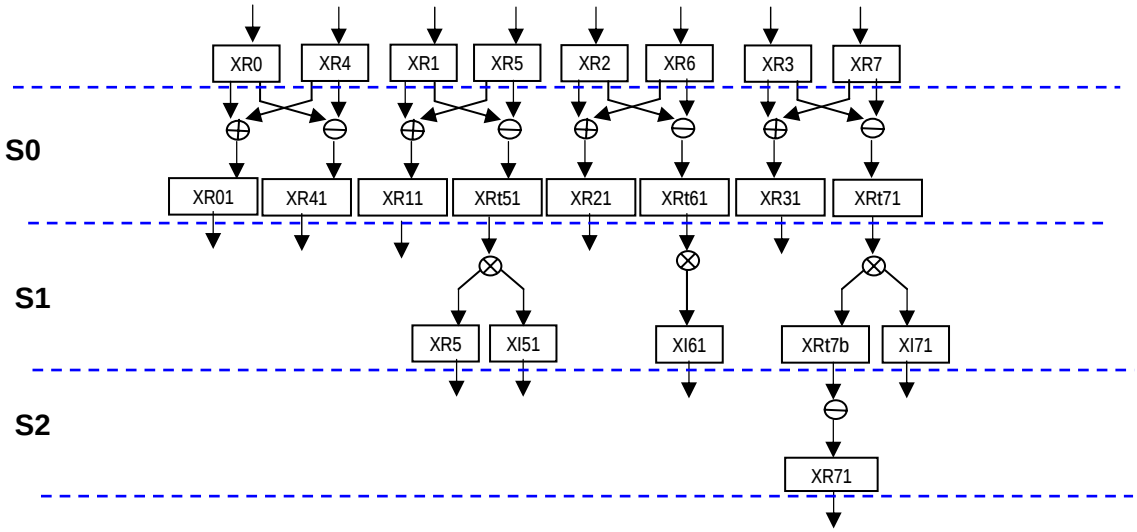


Figure 4.12: Scheduling diagrams for stage one of 8 point IFFT.

For stage one, computation is accomplished in three clock cycle denoted as S0 to S2. The operation is much simpler compared with the FFT. This is because FFT processed both real and imaginary value while IFFT only real. The result from IFFT is represented in real and imaginary value because of the multiplication of twiddle factor. Twiddle factor is a constant defined by the number of point used in this transform. This scheduling diagram is derived from the equation obtain in FFT signal flow graph.

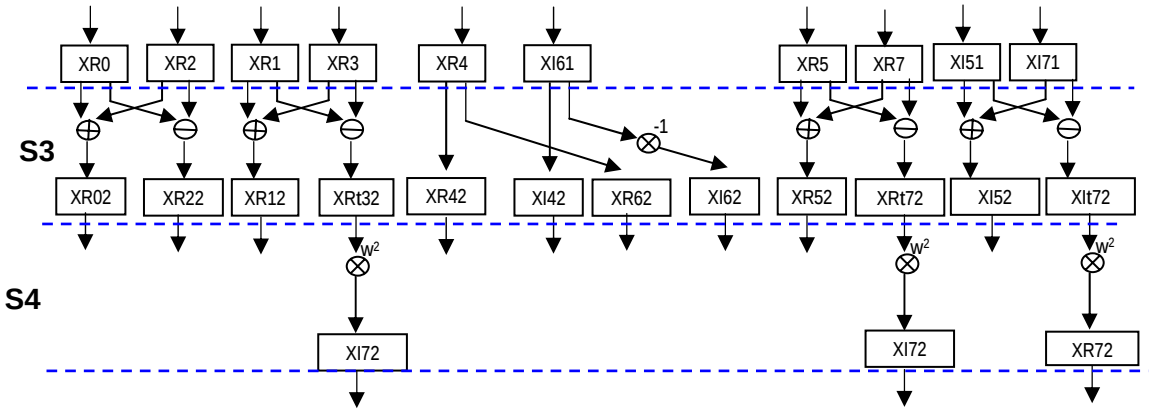


Figure 4.13: Scheduling diagrams for stage two of 8 point IFFT.

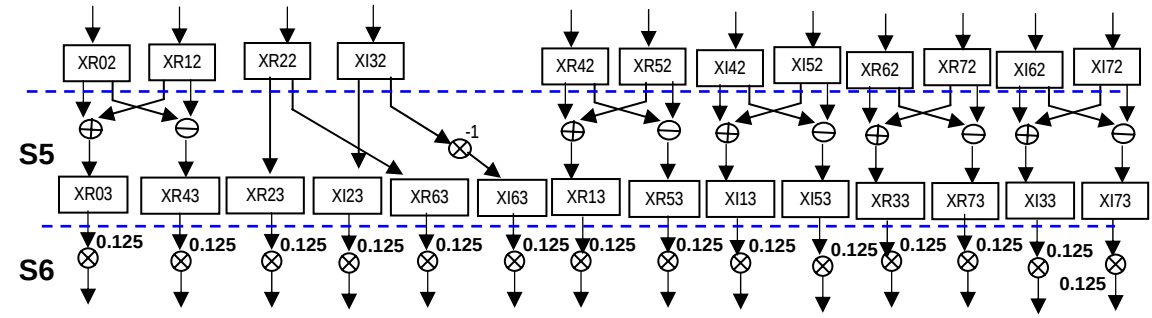


Figure 4.14: Scheduling diagrams for stage three of 8 point IFFT.

Figure 4.13 and figure 4.14 shows the scheduling diagram for stage two and three respectively. The same notation with FFT scheduling diagram is used for IFFT process. For example, referring to figure 4.13, in clock S3, the addition and subtraction operation is performed. As mention before, each clock cycle, the addition and subtraction is executed concurrently and the result is stored in the next register. In clock S4, only three operations is performed, that is multiplication of twiddle factor value.

In figure 4.14, the resultant value is multiplied with the number 0.125. This number actually same as the division with the N point value. In this case N value is 8 for 8 point IFFT. The final result is stored in the memory and will be called upon

when it is required to display the result at user interface. Signal flow graph is very important as the guided to understand the computation process especially during software programming whereby to create test vector program.

4.9 Hardware Module

Hardware module is developed using VHDL language. The modules which developed include FFT/IFFT, serial to parallel and parallel to serial and mapping block. Each of this module function is describe as in paragraph below.

4.9.1 Fast Fourier Transform (FFT)

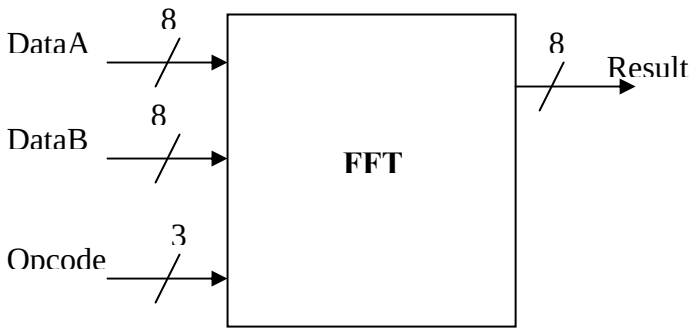


Figure 4.15: FFT module

Figure 4.15 show the block diagram for FFT module. This basic module consists of only two inputs which is DataA and DataB. Opcode is used to select the operation performed by the module. Result will be delivered through Result port. Several operations are performed by this hardware where each operation executed in one clock cycle. Each operation is assigned to the unique opcode value. Referring to the source code in appendix, FFT module has eight operations involved such as addition, subtraction, multiplication, pass module and conversion from positive number to negative. A complete code is available in appendix pages.

4.9.2 Inverse Fast Fourier Transform (FFT)

The same block diagram as FFT is used to develop IFFT module. Input port such as DataA, DataB and Opcode is also used as well as Result for output port. The different between FFT and IFFT is that the IFFT module needs to divide with eight at the end of the result. Additional operation to handle this process is inserted at this module.

4.9.3 Hardware Interfacing

Both FFT and IFFT need to connect to Avalon bus for data processing performed by the standard 32 type CPU module. CPU module which is call NIOS CPU is provided by Altera to manage the data processing performed by the FFT or IFFT module.

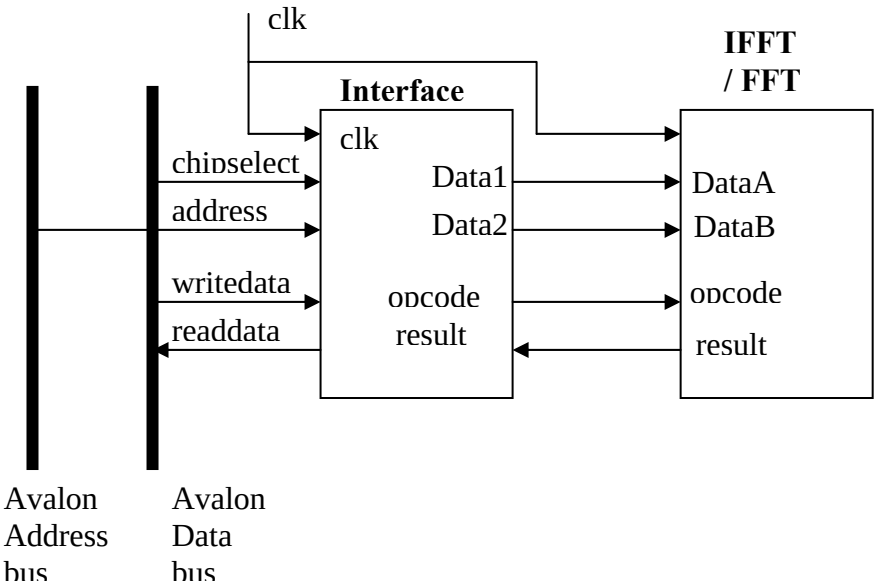


Figure 4.16: Block diagram of the connection between IFFT or FFT module with Avalon bus system.

Figure 4.16 shows the connection between FFT or IFFT module with the Avalon bus system. Interface module is responsible to manage the communication between buses with the FFT or IFFT module. Data is inputted through *writedata* port and buffered in the interface module before it is sent to the FFT or IFFT. The result of computation is delivered to the *wiredata* port and display to user through appropriate interface.

CHAPTER V

SOFTWARE DESIGN

5.1 Introduction

This chapter covers on the material related to the implementation stage as discussed in the methodology chapter. In this part, there are two work required to carry out which is hardware programming and software programming. The details will be discussed further in this chapter.

As mention in methodology chapter, the works required to complete this stage consists of the hardware and software programming. Hardware programming is the process where the designed hardware is programmed into the FPGA board while software programming consists of creating a test vector program in c language to test the operation of the designed module.

5.2 Hardware Programming

In hardware programming, Quartus II v 3.0 software is used to perform the works on this stage. Nios CPU also need to be installed together along with Quartus II software. Basically, Quartus II software handles the process during hardware connection between design module and Avalon bus interface. Beside that, compiling the system which includes the designed module and interface module also perform using this software.

A tool called SOPC builder (System on a programmable chip) is used to perform the module integration. This tool will be installed together during Quartus II software installation. SOPC is an automated system development tool that accelerates many phases of system-on-a-programmable-chip (SOPC) design including system definition and customization, component integration, system verification and software generation for the custom hardware. SOPC Builder enables the combination of components such as embedded processors, standard peripherals, IP cores, on-chip memory, interfaces to off-chip memory, and user-defined logic into a custom system module. SOPC Builder generates a single system module that instantiates these components, and automatically generates the necessary bus logic to connect them together.

5.2.1 Adding the User Defined Logic

To add a user defined logic (designed hardware) to the system module, several steps is required to carry out. The details for these steps will be provided in the appendix pages. Figure 5.1 show the interface to user logic where a process to add FFT module is performed. It can be seen that from this interface, we can see the port name used in the FFT module and the number of bit utilize in each ports. Basically this process will read every port available in the designed module.

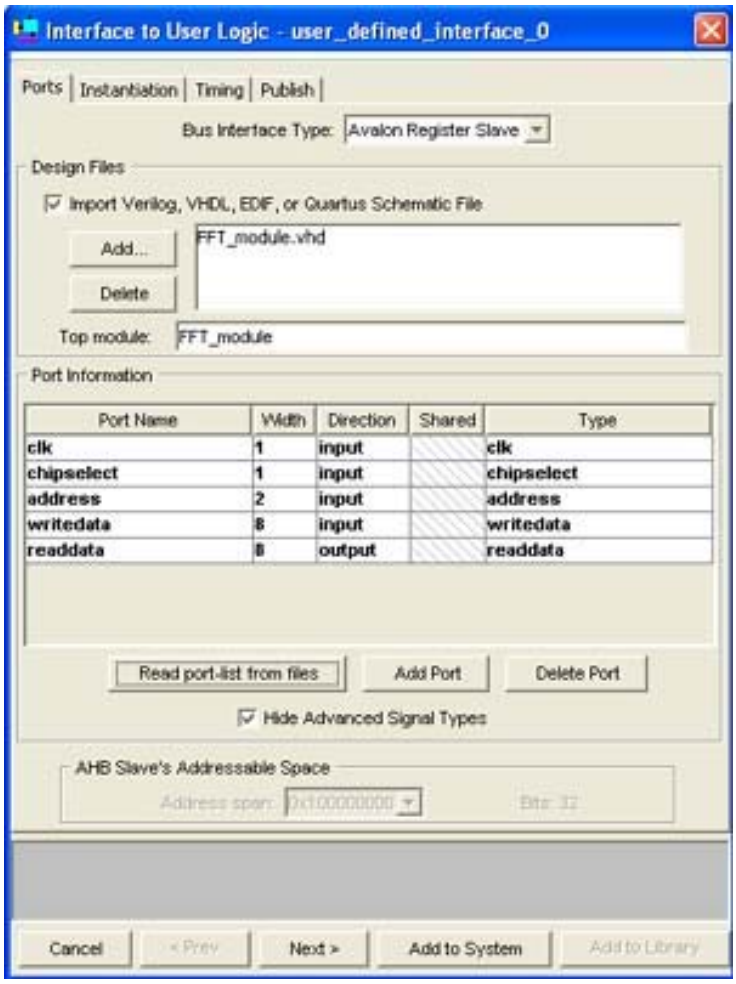


Figure 5.1: Interface to user logic setting on Port Tab

5.2.2 Setting the Module Properties

Figure 5.2 provided the interface for user to modify the properties of the designed module such as timing properties. User can change the instantiations and timing properties according to the desired value. Timing properties for example provided the user to set the time required for data bus or address bus to hold data or address value before another instruction is executed.

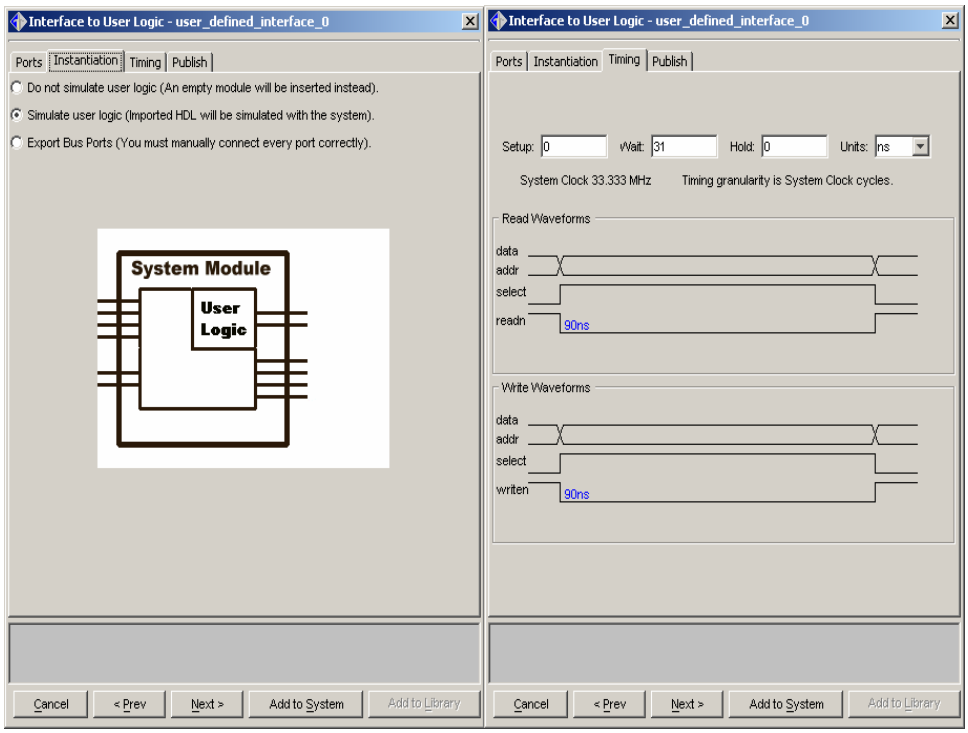


Figure 5.2: Interface to user logic setting on Instantiation and Timing tabs.

5.2.3 Complete Module Integration

Figure 5.3 is the example of the complete system integrated together between FFT module, Ram module and System modules. A System modules is a Nios embedded processor which provided by Altera as mention in previous chapter. From this interface, we can set any properties provided by this software and it is reminded that each property must be set properly according to the designed module capability. For example the targeted device used is APEX 20KE with the clock speed of 33.33 Mhz. All these properties setting can be acquired in the manual sheet.

Each of the modules in the system has its own unique address. These addresses are given by the SOPC Builder tool and it shows the location of the devices in the address of memory location. It is important especially during software

programming to send the input value and direct it to appropriate module. Beside that the result of the computation also can be read from the address in the specific module.

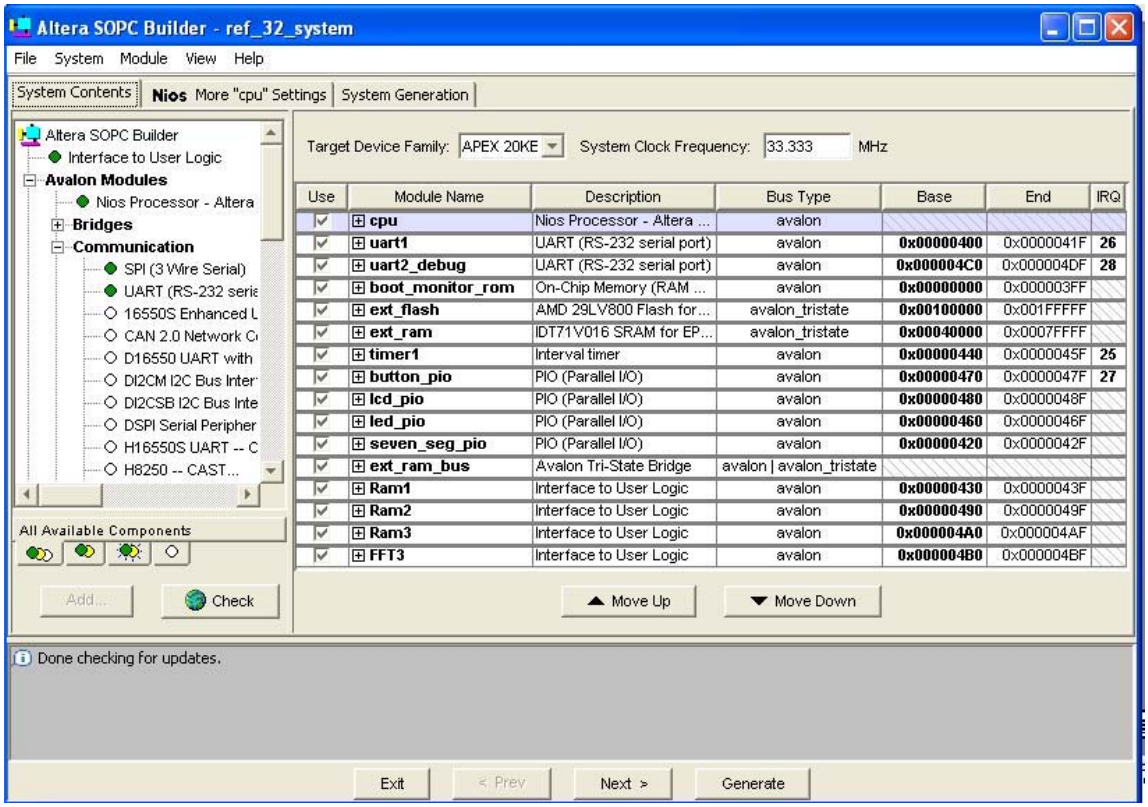


Figure 5.3 SOPC Builder with modules.

5.2.4 Generating the Files

After all setting is properly set, generate the module by clicking the generate button. The SOPC builder will generate the necessary files required by the Nios software to execute the test vector program. The process required around five minutes to complete. If the generation of the files is successful, a message show that the generation is completed will be displayed. Otherwise, the error message will be

displayed and if there is an error to the design, it is required to correct the error before continue to generate. Figure 5.4 shows the generation process.

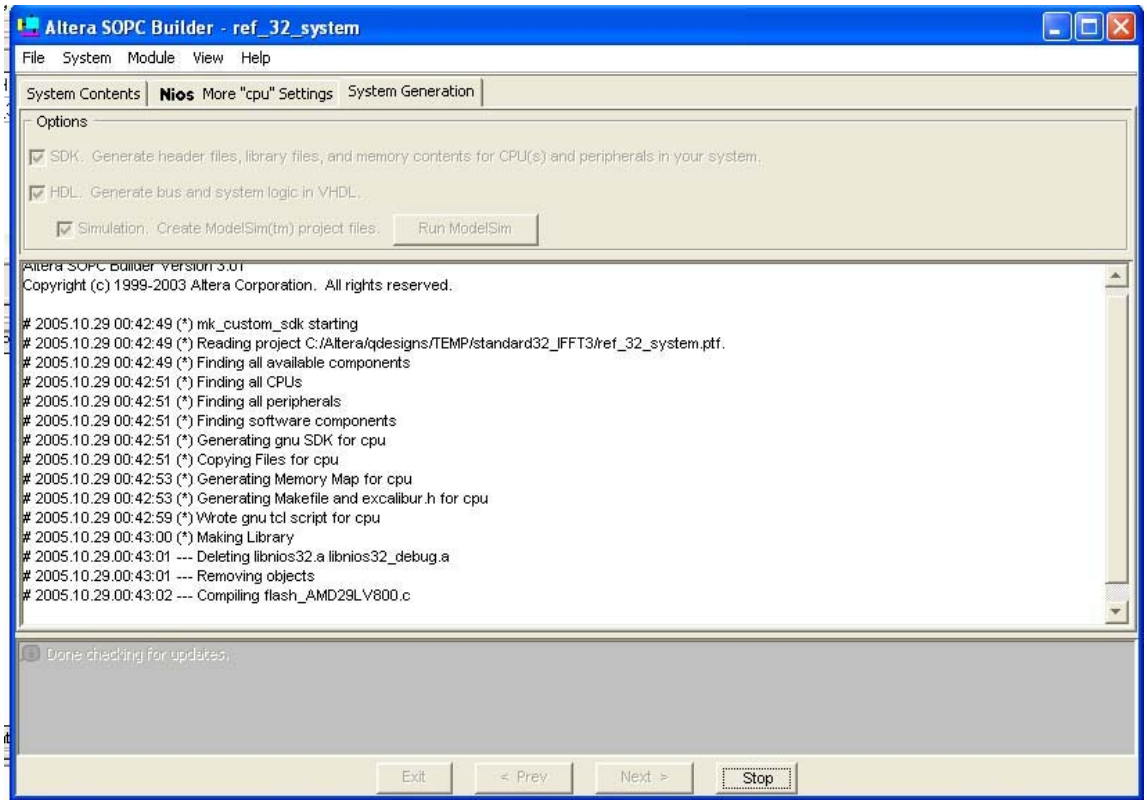


Figure 5.4: Generate the appropriate files for each module.

5.2.5 Compiling the System Module

If the generation of files is successful, the overall design must be saved and re-compiled by clicking the compile button. This recompilation is required since the standard 32 module setting is change due to the generation process. Beside that, it allows the QuartusII software to identify which port in the module is changed. Some changes may require the user to rename the default name of the Nios embedded processor pin code.

5.2.6 Download the design into FPGA board

When the compilation is successful, the design is ready to be downloaded into FPGA board. These steps are provided in the Apex board manual in appendix pages.

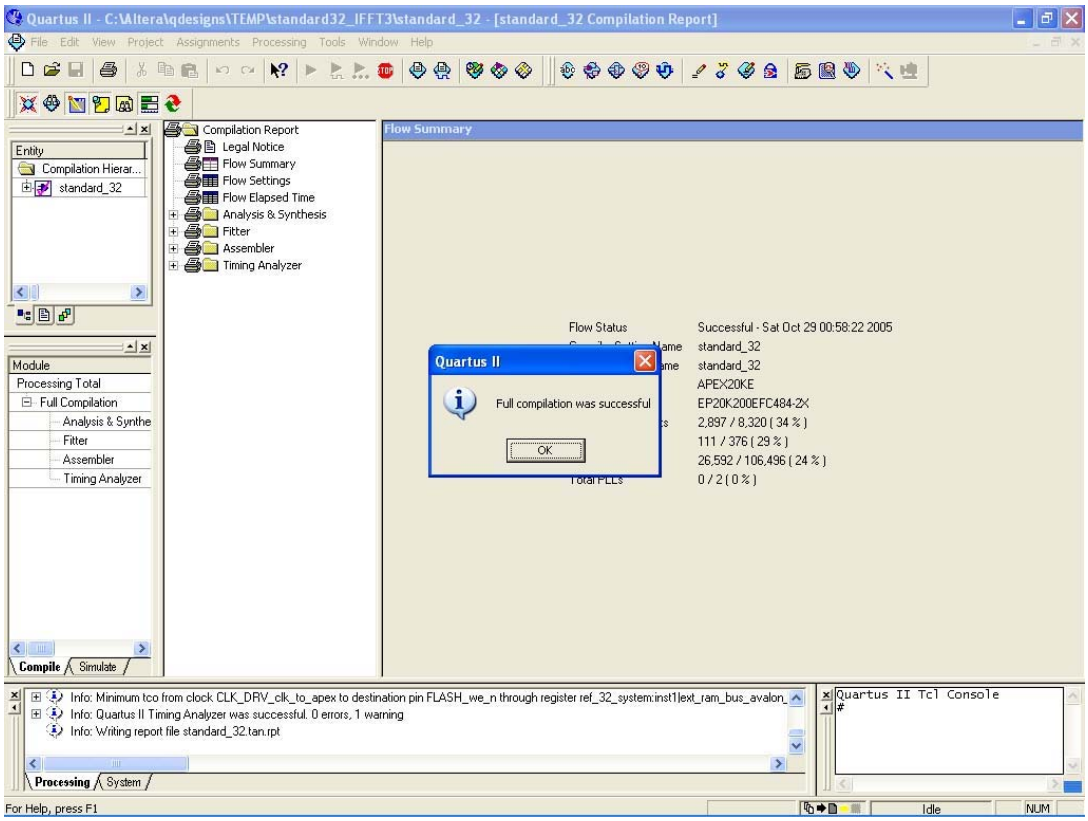


Figure 5.5: Compiling the system module.

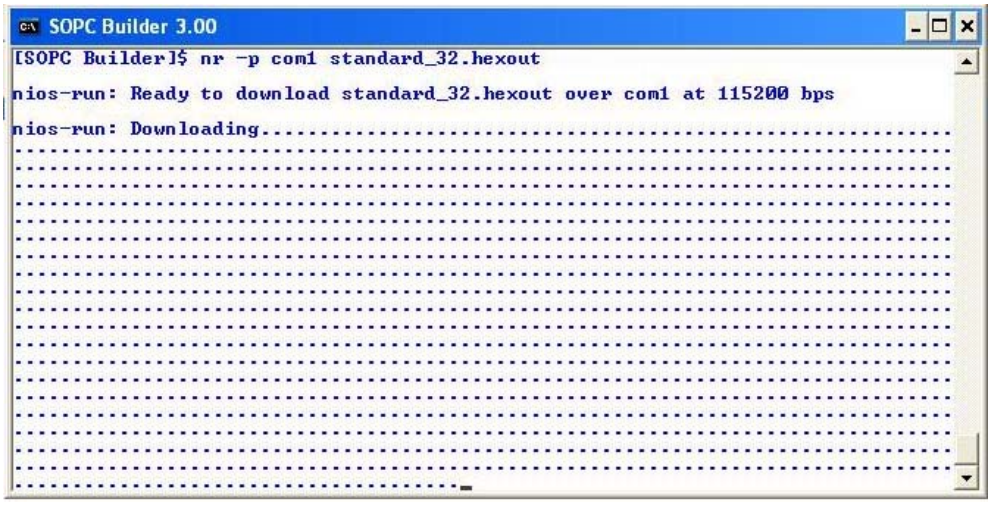


Figure 5.6: Download the design into development board.

5.3 Software Design

After the hardware design is successfully compiled and corrects all the errors, a programme in C programming language is required to verify your design. The most important things to know during this stage is how to communicate hardware design with the software design.

While inserting the user-defined logic design to the Nios system module using the SOPC Builder, there is information which provides the base address and end address of each module.

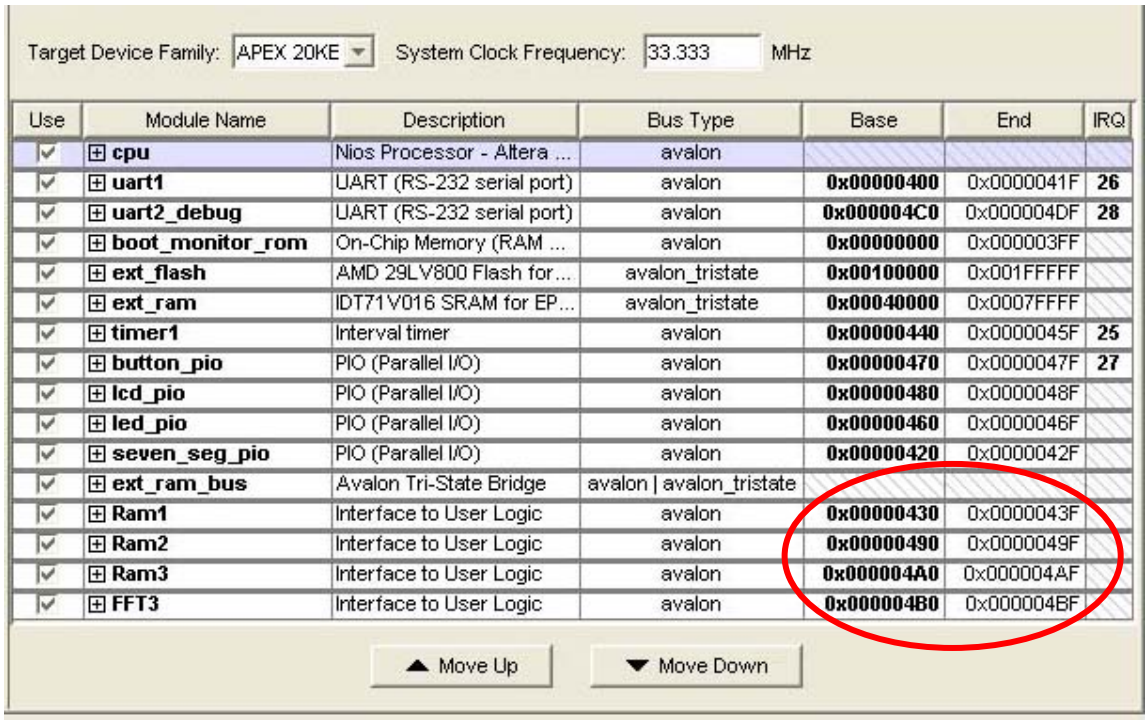


Figure 5.7: The SOPC Builder System Contents Page

This Base Address and the End Address will be use in the Address Mapping in the **Excalibur.h** file generated by the SOPC Builder. Figure 5.8 below shows the portion of Address Mapping in the Excalibur.h file. The highlighted line is address mapping specific for the user-defined logic.

```
#define na_Ram1 ((np_usersocket *) 0x00000430) // altera_avalon_user_defined_interface
#define na_Ram1_base                0x00000430
#define na_timer1   ((np_timer *)    0x00000440) // altera_avalon_timer
#define na_timer1_base              0x00000440
#define na_timer1_irq               25
#define na_led_pio      ((np_pio *)  0x00000460) // altera_avalon_pio
#define na_led_pio_base             0x00000460
#define na_button_pio   ((np_pio *)  0x00000470) // altera_avalon_pio
#define na_button_pio_base          0x00000470
#define na_button_pio_irq           27
#define na_lcd_pio      ((np_pio *)  0x00000480) // altera_avalon_pio
#define na_lcd_pio_base             0x00000480
#define na_Ram2  (np_usersocket *) 0x00000490) // altera_avalon_user_defined_interface
#define na_Ram2_base                0x00000490
#define na_Ram2  (np_usersocket *) 0x000004a0) // altera_avalon_user_defined_interface
#define na_Ram2_base                0x000004a0
#define na_FFT3   (np_usersocket *)  0x000004b0) // altera_avalon_user_defined_interface
#define na_FFT3_base                0x000004b0
```

Figure 5.8: The portion of Address Mapping in Excalibur.h file generated by SOPC Builder

If observed the Base Address and End Address, Ram1, Ram2, Ram3 and FFT3 just require two bits address, which is 00, 01, 10, and 11. So the address given by the SOPC Builder as expected should be as Table 5.1 below:

Table 5.1

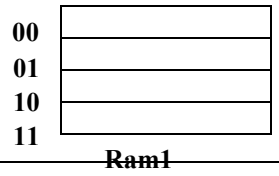
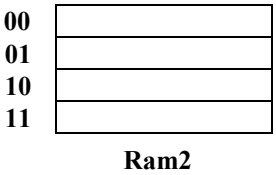
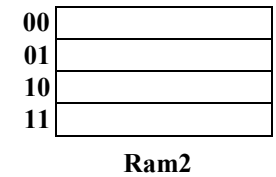
Module Name	**Base**	**End**
Ram1	0x00000430	0x00000433
Ram2	0x00000490	0x00000493
Ram3	0x000004A0	0x000004A3
FFT3	0x000004B0	0x000004B3

However, we found that the SOPC Builder gave extra two bits. As shown in Table 5.2 below: With the 4 bits that given to us, however, we just can utilize 2 bits from it. So, as the result, the last 2 bits are considered as don't care value. (We must assume the last 2 bits as don't care value instead of the first 2 bits!) So, to achieve the address to control your user-defined logic, we can achieve by the way that shown in Table 5.3.

Table 5.2

Module Name	**Base**	**End**
Ram1	0x00000430	0x0000043F
Ram2	0x00000490	0x0000049F
Ram3	0x000004A0	0x000004AF
FFT3	0x000004B0	0x000004B0

Table 5.3

Module Name	**Address(last 4 bits)**	**Example**	**Address Location in VHDL**	**Block Diagram illustration**
Ram1	00xx 01xx 10xx 11xx	0x00000430 0x00000434 0x00000438 0x0000043C	Address 00 in Ram1 memory Address 01 in Ram1 memory Address 10 in Ram1 memory Address 11 in Ram1 memory	
Ram2	00xx 01xx 10xx 11xx	0x00000491 0x00000495 0x00000499 0x0000049D	Address 00 in Ram2 memory Address 01 in Ram2 memory Address 10 in Ram2 memory Address 11 in Ram2 memory	
Ram3	00xx 01xx 10xx 11xx	0x000004A2 0x000004A6 0x000004AA 0x000004AE	Address 00 in Ram2 memory Address 01 in Ram2 memory Address 10 in Ram2 memory Address 11 in Ram2 memory	

FFT3	00xx
	01xx
	10xx
	11xx
		0x000004B3
		0x000004B7
		0x000004BB
		0x000004BF
			Case address is
			when “00” =>
			opcode<=writedata(1downto0);
			when “01” =>
			data1 <= writedata;
			when “10” =>
			data2 <= writedata;
			when others =>
			readdata <= result;
			end case;
				Please refer to Figure 20 on next page.

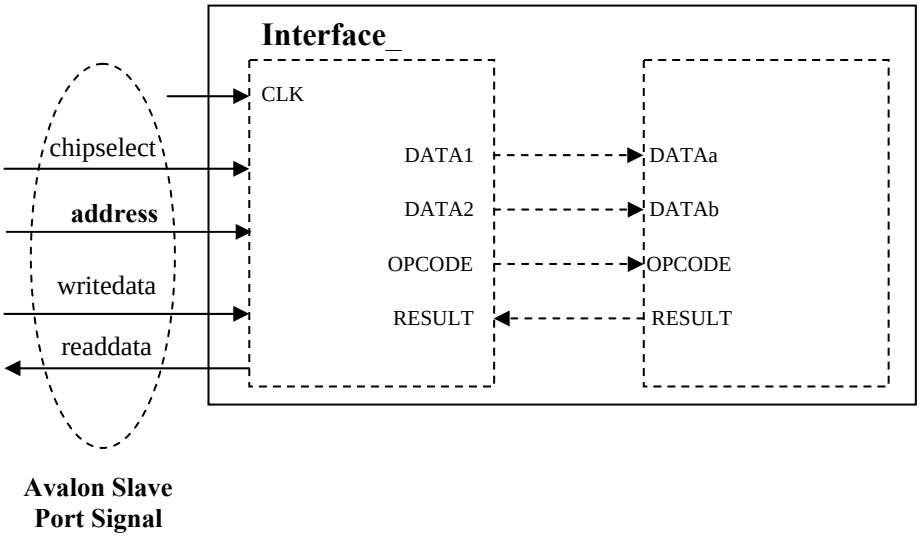


Figure 5.9: The simplified block diagram of Connection between ALU with Avalon Bus System

With the other words, for memory module such as RAM or ROM, the address signal from Avalon Slave bus means the memory location. However, for the processing module it is a slightly different.

For example, such a simple FFT in this case, the address indicates which data to be passed between the FFT module unit and the Avalon Bus System. In this FFT case, the FFT need 4 cycles to complete an operation. First clock cycle is used to fetch opcode (OPCODE) from Avalon bus to FFT, second clock cycle to fetch first operand (DATAa) from Avalon bus to FFT, third cycle to fetch second (DATAb) operand from Avalon bus to FFT, and the last clock cycle to fetch the operation result (RESULT) from the FFT to the Avalon Bus System.

After understand the architecture of the memory system, the works will be further carry out with the control vector programme in C language. The code for FFT and IFFT is provided in the appendix.

5.3.1 Compiling and Downloading the Control Vector programme

Figure 5.10: Compiling the C code for test vector program.

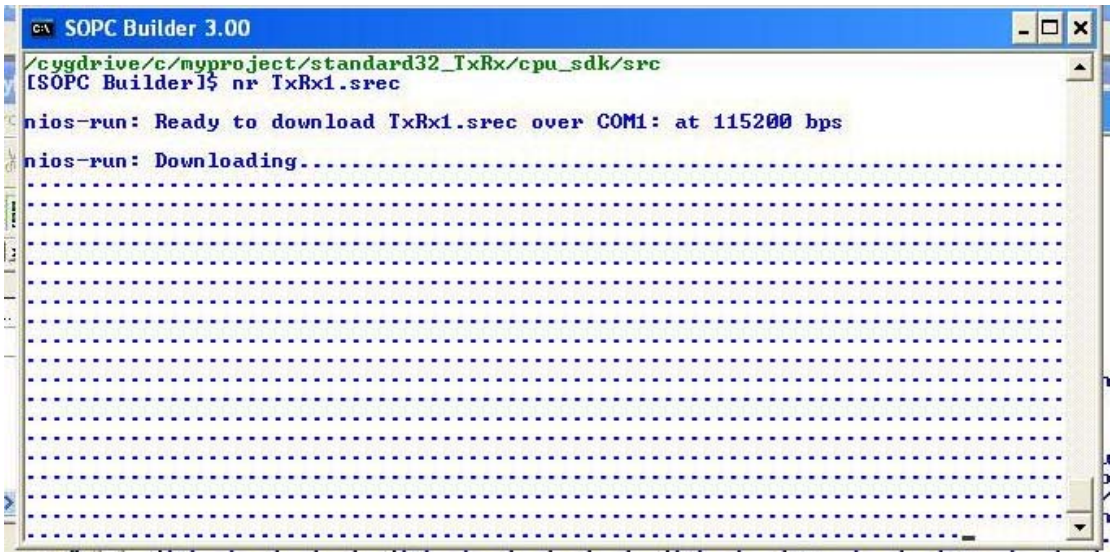


Figure 5.11: downloading C code for test vector program.

The test vector programmed is required to be compiled to ensure no error in the source code. If there is error during compilation, it has to be fixed before re-compilation is performed. Compilation for the programme will produce several files including *.srec files. This file is used to download into the Apex 20KE development board to test the hardware module operation. Figure 5.10 and 5.11 depicts the compilation process and download process respectively.

CHAPTER VI

RESULTS

6.1 Introduction

This chapter will discuss on the experiment set up and the test conducted to the FPGA module. All results from hardware module are recorded into Microsoft Excel to ensure the data is properly organized. The user interface for FPGA program is also been captured and shown in this chapter.

The Apex 20KE board requires to be set up properly before the program is downloaded into the board. The procedure on this step is available in the Apex 20KE manual data sheets. Basically the development board kits is shipped with one Apex 20KE board, DC power supply adapter, serial cable, LCD display and the ByteBlasterTM II parallel port download cable. Figure 6.1 and 6.2 depict the connection between Apex 20KE development boards with the computer.

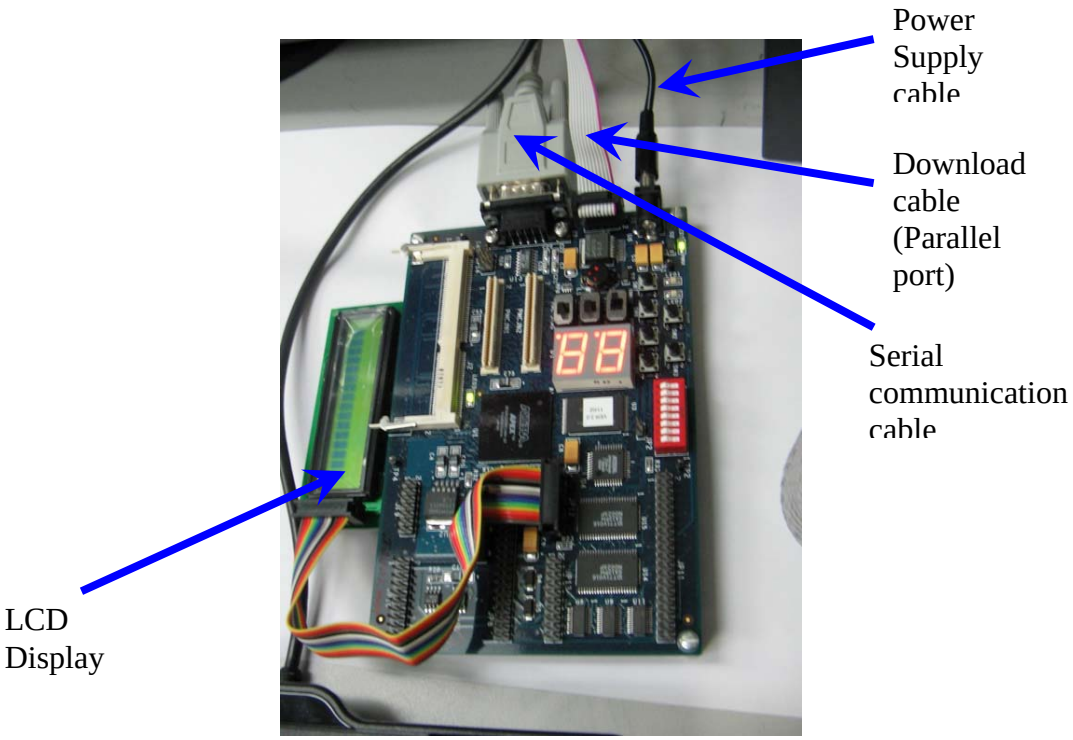


Figure 6.1: Apex 20KE development board.

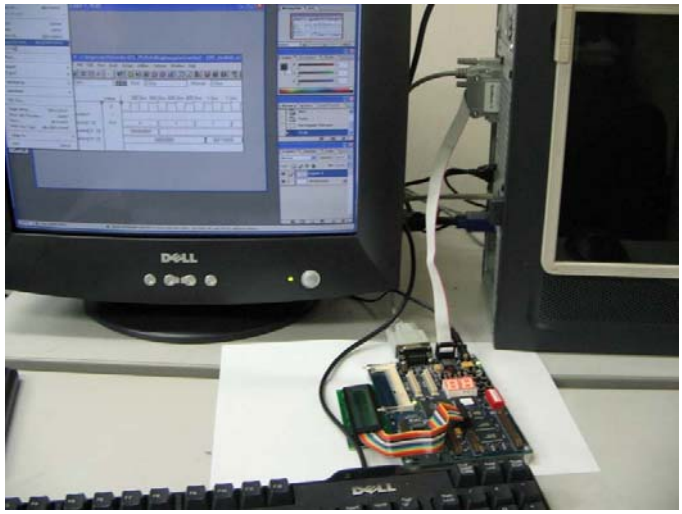


Figure 6.2: Apex 20KE connection with computer.

6.2 How to Conduct Test?

Several modules have been designed and testing is required to obtain the result. The most important module is FFT and IFFT since this module depict the processing technique performed in the receiver or transmitter. There are two methods to conduct the test to the designed module, which firstly is each FFT, and IFFT module is tested independently and the results is compared with the results computed by Matlab software. Connecting the output of the transmitter to the input of the receiver such that the input value will be same as the output value does the second test.

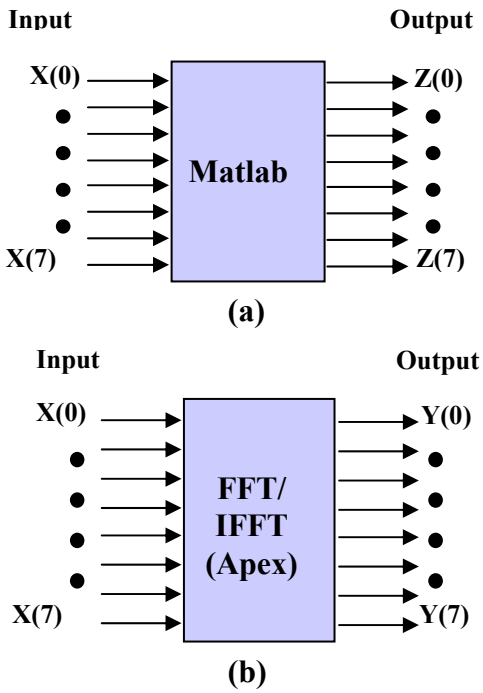


Figure 6.3: (a) Matlab FFT/IFFT module. (b) Apex 20KE FFT/IFFT module.

Figure 6.3 (a) and (b) show the illustration of the block diagram for Matlab module and designed module in Apex 20KE board. X(0) to X(7) denoted as the input for each module while Z(0) to Z(7) and Y(0) to Y(7) denoted as output for Matlab and Apex board respectively. Matlab output will be compared with the output obtain by the designed module computation in Apex 20KE.

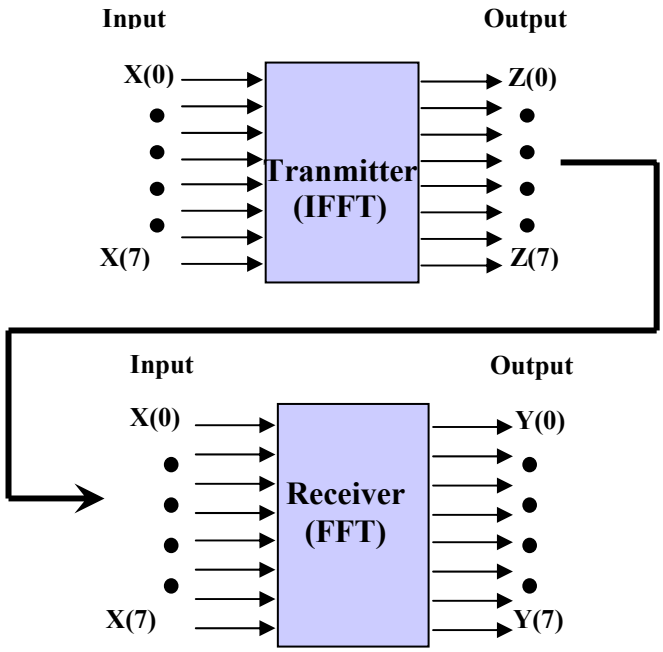


Figure 6.4: Transmitter module and receiver module.

Figure 6.4 depicts the illustration for the transmitter and receiver module. The output from transmitter module which is mainly consists of IFFT is used as the input to the receiver module. Generally if the input from transmitter is real value the computation of IFFT will result real and imaginary value. While if the input is imaginary, computation will result in real value. For the IFFT design, the input only accept real value, thus imaginary result is obtained. The receiver needs to have both real and imaginary at the input to convert back to the original value. So, FFT is design to have this feature in order to process the data and display the correct result.

6.3 Result obtained for IFFT

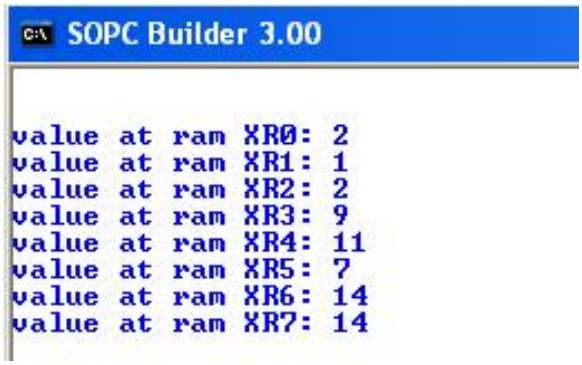


Figure 6.5: Input value to the IFFT module.

Figure 6.5 shows the captured result of the input value to the IFFT module. User keys the input in and the value is stored at the memory or external ram of the Apex 20KE board. XR0 to XR7 denoted as the variable to store the value in the memory. The program is run using SOPC Builder, which is available when Nios software is installed. SOPC builder provides an interface between user and the hardware development board.

Figure 6.6 depict the stage 2 and stage 3 of IFFT computation. The result of each stage computation is in integer value. Number is represented by the 8 bit of two's complement binary representation. Using this method, the allowed number to represent is from 0 to 255 values. Positive number is represent from 0 to 127 while negative number from 255 to 128 which represent -1 to -127 respectively.

Figure 6.6: Stage 1 and stage 2 operation.

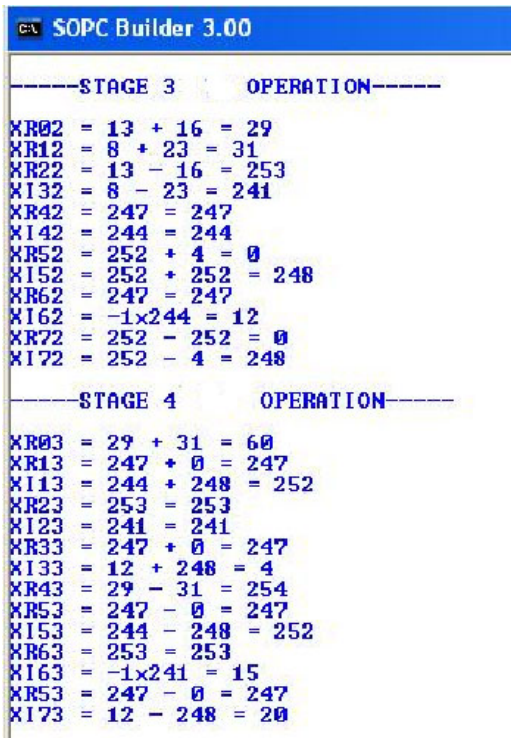


Figure 6.7: Stage 3 and stage 4 operation.

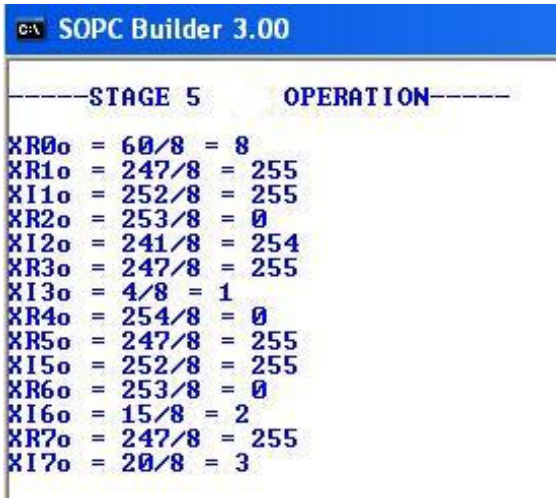


Figure 6.8: The final output from IFFT operation.

Figure 6.8 shows the final result of the IFFT computation. As explain in the theory final output of IFFT will be divided by 8. Overall computation results in both real and imaginary for each input variable except for XR0 and XR4.

6.4 Result obtained for FFT

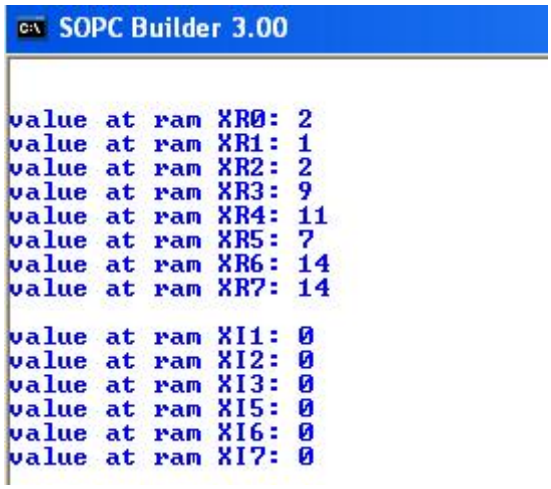


Figure 6.9: Input value to FFT module.

Figure 6.9 show the input that is given to the FFT module. FFT module as mention before is able to process both real and imaginary value. Each input variable has it own imaginary value except for variable X0 and X4.

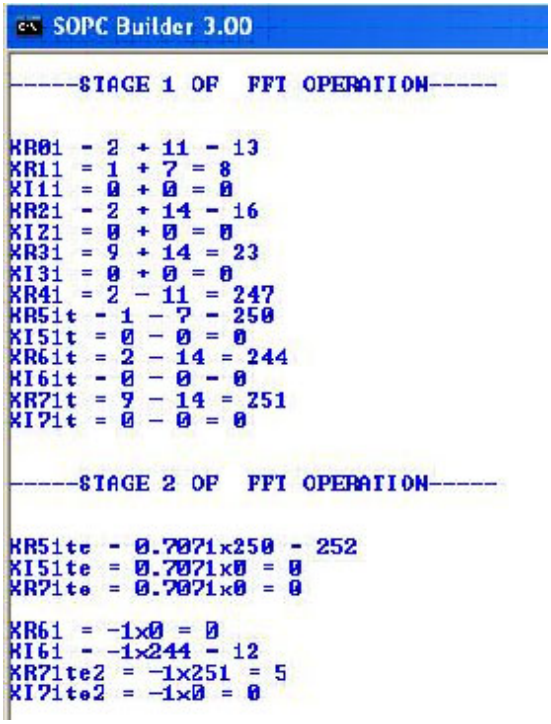


Figure 6.10: Stage 1 to stage 2 operations.

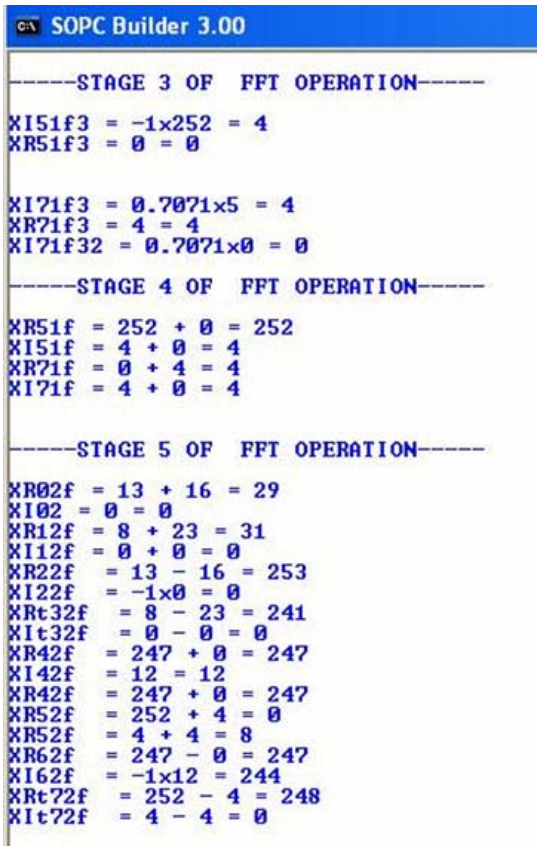


Figure 6.11: Stage 3 to stage 5 operation.

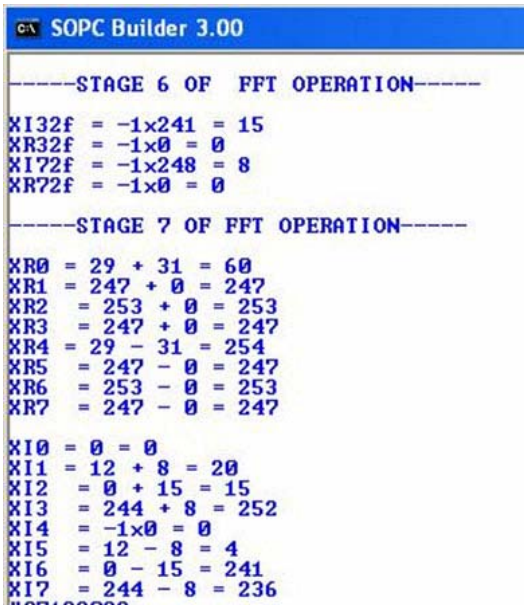


Figure 6.12: Stage 6 and final output of FFT computation.

Figure 6.12 shows the final result of FFT computation. Using FFT module in FPGA, the results obtained are integer value of 8 bit two's complement. FFT computation requires seven stages to complete until the result is obtained because it involves both real and imaginary at the input. Computation takes longer clock cycles since the twiddle factor value is not same as in IFFT. The FFT result will be compared with the Matlab result and the comparison is shown in the table 6.1 to 6.3.

6.5 Results for Transmitter and Receiver

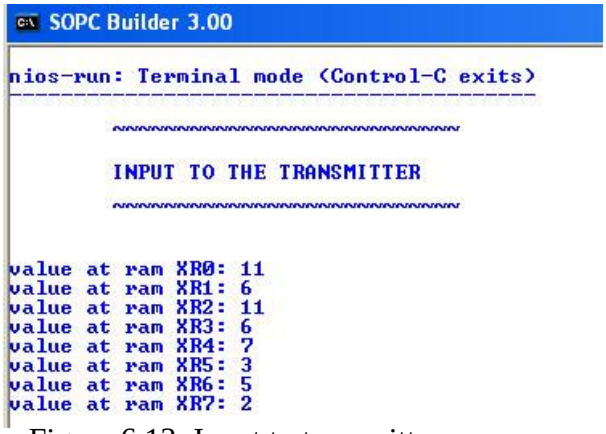


Figure 6.13: Input to transmitter.

Figure 6.12 shows the input value which stored in the memory for transmitter module. As we can see, the value for transmitter input is keyed in randomly. As mention before, there are some assumption is made for the transmitter and receiver module. The transmitter module only can accept real value with the bit length of 8 and two's complement. While the receiver module, it can accept both real and imaginary with the same bit length. The second condition is that, both of these modules are implemented in one board.

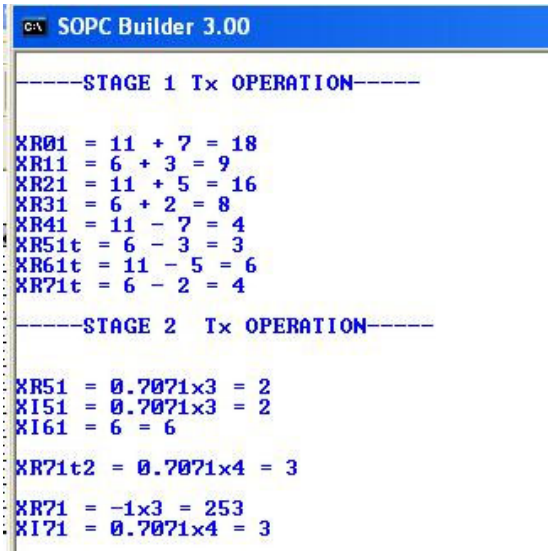


Figure 6.14: Transmitter processing stage 1 to 2.

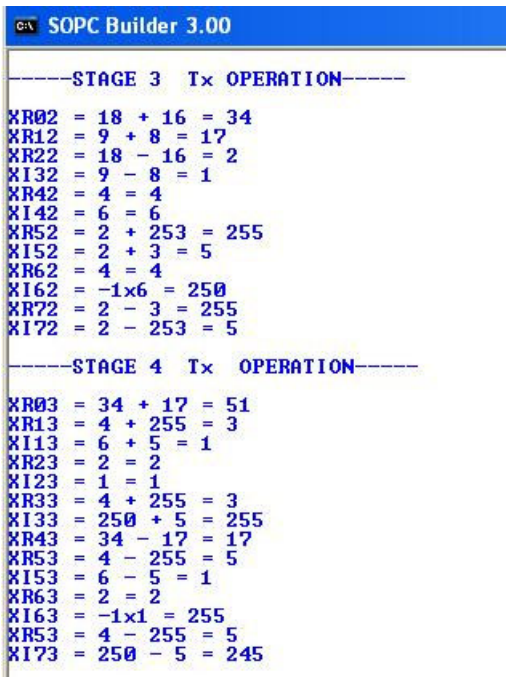


Figure 6.15: Transmitter processing for stage 3 to 4.

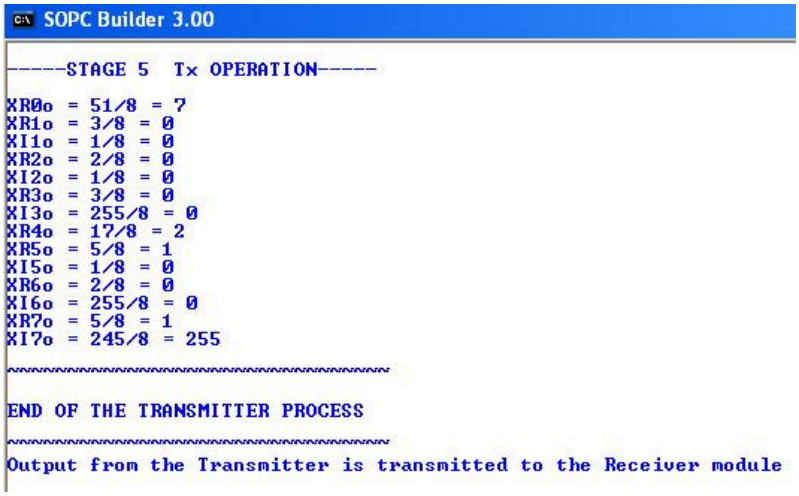


Figure 6.16: Output for transmitter module.

Figure 6.13 and 6.14 depicts the computation process occurred in the transmitter. With this, the computation can be easily checked in each input to ensure the addition, subtraction or multiplication is correctly executed. Figure 6.15 show the final output of the transmitter. The result of the transmitter is sent to the receiver module and buffered at the receiver memory buffer. The process is captured and the figure 6.16 depicts the buffering value in the receiver.

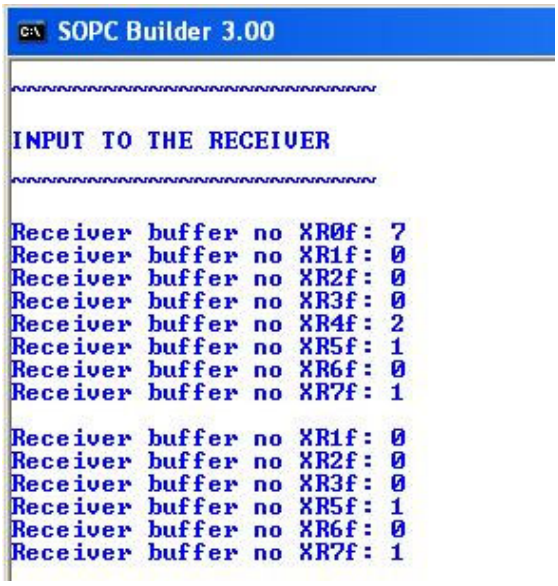


Figure 6.17: Receiver buffer

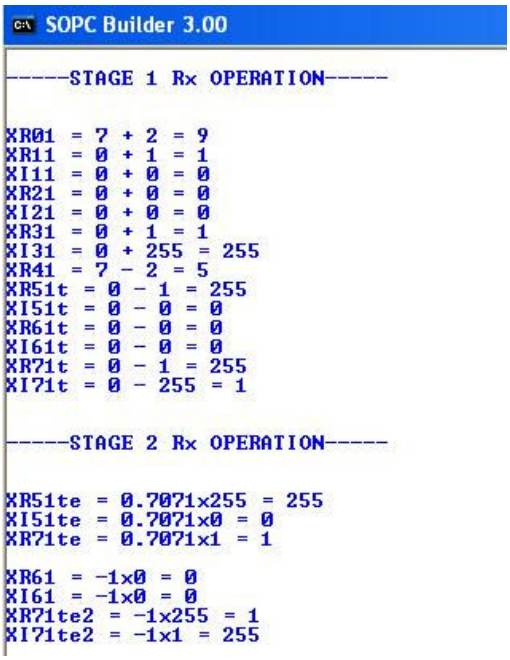


Figure 6.18: Receiver operation for stage 1 to 2.

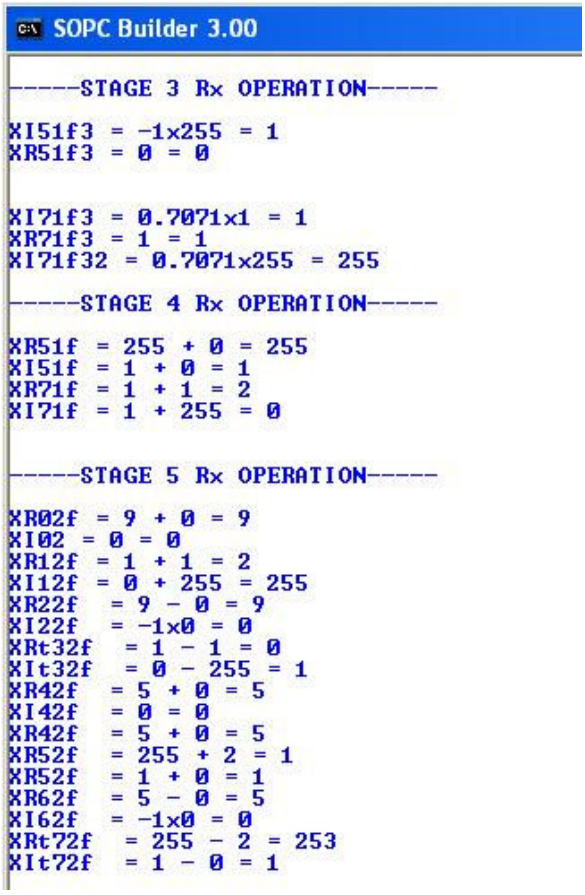


Figure 6.19: Receiver operation for stage 3 to 5.

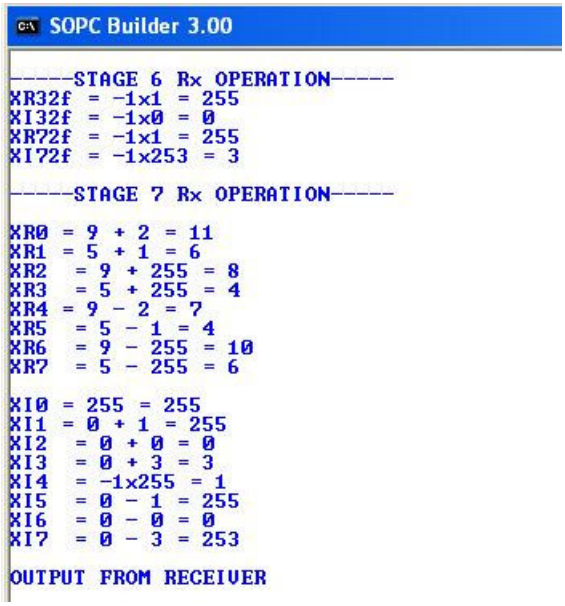


Figure 6.20: Receiver operation for stage 6 and the final output for receiver.

Figure 6.18 and 6.19 depict the process in the receiver. Some operation is not able to accomplish in one clock cycle which make the process longer to produce the result. For example suppose the twiddle factor value is -0.7071 -j0.7071 and the number to multiply is say 25. First step to do is to multiply the value with positive value (0.7071) which require one process and convert the result to negative requiring another process cycle. Figure 6.20 show the final result of the receiver process. The result obtained at receiver consists of real and imaginary value. Theoretically, if the input for FFT or IFFT is imaginary then the computation would result only real value. This condition actually is only true for the floating point number representation which used in Matlab software. For fix number representation, the point number is represented by approximation, thus the number is not correctly represented. This will be discussed more on the analysis and discussion chapter.

Table 6.1 to 6.3 shows the comparison between hardware computation and Matlab for FFT, IFFT and transmitter and receiver module. The output for hardware and Matlab is bold such that it gives easiness to compare the result. Briefly, output for FFT and IFFT modules gives the same result between hardware computation and Matlab. But for transmitter and receiver module, the output is slightly different. The reason will be discuss in the analysis and discussion chapter.

Table 6.1: Result for FFT

	Input		Output(matlab round)		Output(FPGA)			
					2's complement		Integer	
	Real	Imaginary	Real	Imaginary	Real	Imaginary	Real	Imaginary
x0	2	0	60	0	60	0	60	0
x1	1	0	-10	20	247	20	-9	20
x2	2	0	-3	15	253	15	-3	15
x3	9	0	-8	-4	247	252	-9	-4
x4	11	0	-2	0	254	0	-1	0
x5	7	0	-8	4	247	4	-9	4
x6	14	0	-3	-15	253	241	-3	-15
x7	14	0	-10	-20	247	236	-9	-20
x0	8	0	58	0	58	0	58	0
x1	0	0	7	12	7	12	7	12
x2	9	0	-7	17	249	17	-7	17
x3	5	0	-3	6	253	6	-3	6
x4	6	0	12	0	12	0	12	0
x5	3	0	-3	-6	253	250	-3	-6
x6	12	0	-7	-17	249	239	-7	-17
x7	15	0	7	-12	7	244	7	-12
x0	100	0	347	0	91	0	91	0
x1	25	0	66	145	66	145	66	-111
x2	250	0	35	- 18	35	238	35	-18
x3	2	0	26	- 117	26	139	26	-117
x4	54	0	199	0	199	0	-57	0
x5	21	0	26	117	26	117	26	117
x6	125	0	35	18	35	18	35	18
x7	26	0	66	- 145	66	111	66	111
x0	200	0	1434	0	154	0	-102	0
x1	156	0	-63	- 224	193	31	-63	31
x2	233	0	60	138	60	138	60	-118
x3	254	0	19	- 16	19	239	19	-17
x4	222	0	134	0	134	0	-122	0
x5	100	0	19	16	19	17	19	17
x6	129	0	60	-138	60	118	60	118
x7	140	0	-63	224	193	225	-63	-31
x0	2	0	33	27	33	27	33	27
x1	1	5	-5	4	249	0	-7	0
x2	9	6	-12	-8	0	248	0	-8
x3	2	4	-10	-3	251	4	-5	4
x4	8	0	19	-13	19	243	19	-13
x5	3	2	3	-8	241	252	-15	-4
x6	7	1	0	-6	244	250	-12	-6
x7	1	9	-12	7	3	0	3	0

Table 6.2: Result for IFFT

	Input		Output(matlab round)		Output(FPGA)			
					2's complement		Integer	
	Real	Imaginary	Real	Imaginary	Real	Imaginary	Real	Imaginary
x0	2	0	8	0	8	0	7	0
x1	1	0	-1	-2	255	254	-1	-2
x2	2	0	0	-2	0	255	0	-1
x3	9	0	-1	1	255	0	-1	0
x4	11	0	0	0	0	0	0	0
x5	7	0	-1	-1	255	0	-1	0
x6	14	0	0	2	0	1	0	1
x7	14	0	-1	2	255	2	-1	2
x0	8	0	7	0	7	0	7	0
x1	0	0	1	-2	0	255	0	-1
x2	9	0	-1	-2	0	254	0	-2
x3	5	0	0	-1	0	0	0	0
x4	6	0	2	2	1	0	1	0
x5	3	0	0	1	0	0	0	0
x6	12	0	-1	2	0	2	0	2
x7	15	0	1	2	0	1	0	1
x0	2	0	4	0	4	0	4	0
x1	1	0	-1	0	255	0	-1	0
x2	9	0	-1	0	255	0	-1	0
x3	2	0	0	0	255	0	-1	0
x4	8	0	2	0	2	0	2	0
x5	3	0	0	0	255	0	-1	0
x6	7	0	-1	0	255	0	-1	0
x7	1	0	-1	0	255	0	-1	0
x0	200	0	179	0	243	0	-13	0
x1	156	0	-8	28	248	254	-8	-2
x2	233	0	8	-17	8	15	8	15
x3	254	0	2	2	248	2	-8	2
x4	222	0	17	0	240	0	-16	0
x5	100	0	2	-2	2	254	2	-2
x6	129	0	8	17	8	241	8	-15
x7	140	0	-8	-28	2	4	2	4

Table 6.3: Results for transmitter and receiver

	Input(Transmitter)		Output(Transmitter)		Output(Receiver)			
					2's complement		Integer	
	Real	Imaginary	Real	Imaginary	Real	Imaginary	Real	Imaginary
x0	2	0	8	0	4	0	4	0
x1	1	0	255	253	8	6	8	6
x2	2	0	0	254	2	254	2	-2
x3	9	0	255	1	10	4	10	4
x4	11	0	0	0	8	248	8	-8
x5	7	0	25	255	4	2	4	2
x6	14	0	0	2	14	254	14	-2
x7	14	0	255	3	14	0	14	0
x0	8	0	7	0	9	0	9	0
x1	0	0	1	254	6	255	6	-1
x2	9	0	255	254	5	4	5	4
x3	5	0	1	255	6	3	6	3
x4	6	0	2	0	7	246	7	-10
x5	3	0	0	1	4	1	4	1
x6	12	0	255	2	11	254	11	-2
x7	15	0	0	2	8	5	8	5
x0	11	0	7	0	11	0	11	0
x1	6	0	2	0	8	0	8	0
x2	11	0	0	0	8	3	8	3
x3	6	0	0	0	6	0	6	0
x4	7	0	0	1	3	2	3	2
x5	3	0	1	0	6	254	6	-2
x6	5	0	0	0	6	255	6	-1
x7	2	0	1	255	8	254	8	-2
x0	1	0	5	0	254	0	-2	0
x1	2	0	255	255	5	1	5	1
x2	3	0	255	255	4	0	4	0
x3	4	0	255	0	7	1	7	1
x4	5	0	255	0	6	252	6	-4
x5	6	0	255	0	5	1	5	1
x6	7	0	255	1	8	0	8	0
x7	8	0	255	1	7	1	7	1
x0	-56	0	243	0	231	0	-25	0
x1	-100	0	248	252	5	255	5	-1
x2	-23	0	8	15	244	211	-12	-45
x3	-2	0	248	2	241	223	-14	-33
x4	-34	0	240	0	239	26	-17	26
x5	100	0	2	254	11	41	11	41
x6	-127	0	8	241	226	3	-30	3
x7	-116	0	2	4	235	9	-21	9

CHAPTER VII

ANALYSIS AND DISCUSSIONS

7.1 Introduction

This chapter covers on the analysis of the results obtained from the FFT and IFFT module plus the transmitter and receiver module. The comparison results between Matlab and these modules were shown in the Table 6.1, Table 6.2 and Table 6.3 in the previous chapter.

As notice from the comparison result, it can be concluded that the results obtained were not exactly correct as using Matlab software especially for transmitter and receiver module. This chapter will present some of the reason and discussed why this problem occurred and suggest the best solution for the problem encountered.

7.2 Why not Accurate?

The biggest problem when dealing with hardware in implementing mathematical computation is the accuracy. The main reason why computation using hardware module is not accurate as using software base is that the multiplication and division is using fix point number instead of floating point. The weakness of using fix point representation is, approximation made for number representation introduce error. For example decimal numbers for 0.7071 in binary is 010110101. If this binary number converted back into decimal, the result is equal to 0.70. From this example, it is proven that the twiddle factor is not represented accurately.

7.3 Multiplication of Twiddle Factor

Decimal	Binary
11	0000 1011
X 0.7071	X 01011 0101
7.7781	0 0000 0111 1100 0111

Figure 7.1: Example of twiddle multiplication.

Figure 7.1 show an example of integer number multiply with twiddle factor in decimal and binary representation. Result for decimal number can be shown up to 0.0001 point of accuracy. Compare to binary representation, the result obtained from computation only can be displayed in 8 bit representation which is 7.The 8 bit number multiply with 8 bit number will result 16 bit number. As we notice that, the result only can store 8 bit number thus, the register only can store 7 instead of 7.7781

and consequently it sacrifice the accuracy. This only involves one operation, since in FFT/IFFT requires many stages and many operations the final result will be totally different at some of the output. In Matlab computation is done all in floating point format and only during the final result, it is made rounded. Compared to FPGA module, each operation is already having an error, thus obviously some output will not given same value. Fix point number provide faster processing time, less circuit complexity and less usage of memory module compared to floating point representation. The result shown in this example is only for one multiplication. The result become worse if there are more than two twiddle multiplication.

7.4 Division by eight in IFFT module

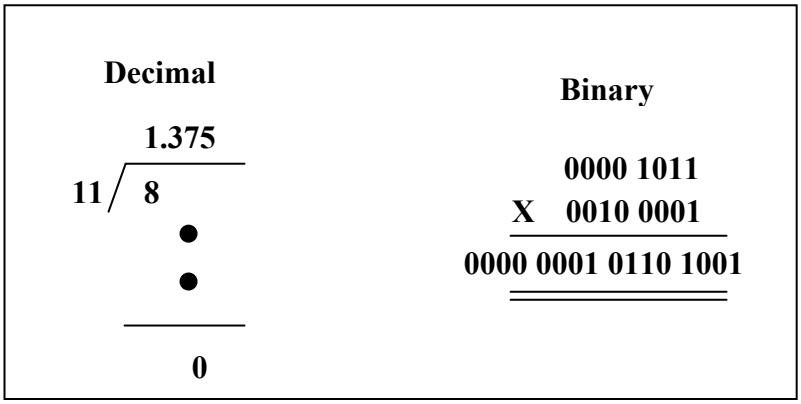


Figure 7.2: Example of twiddle division.

Figure 7.2 shows the example of division process at the IFFT module. In decimal, the result can be shown accurately until up to 0.001. But for binary representation, the result can be shown is 1. In binary, division process is represented as the multiplication process because it simplifies the programming code. Division of 8 can be shown as multiplication of number with 0.125.

7.5 Overflow

Overflow is the main problem in binary representation for arithmetic process. Basically, overflow is occurred when the value of carry in is not same as the value for carry out. Figure 7.3 depicts the example clearly.

Decimal	Binary
126	0111 1110
+ 126	+ 0111 1110
252	1111 1100

Figure 7.3: Addition of decimal number.

This example shows the addition of both positive decimal numbers. As mention before, maximum positive number representation for 8 bit two's complement is 127. Thus in decimal, additions of this number will results 252. But in binary two's complement, the value 252 is equal to -4. In this case, the result obtained for this addition will create an error result.

It can be concluded that the problem encountered are as discussed as above which leads to the reason on why FPGA and Matlab computation gives different result. Some suggestion is proposed and is stated at the conclusion chapter.

CHAPTER VIII

CONCLUSION

As mentioned in the objectives, a base band OFDM transmitter was successfully developed using Altera APEX 20k200EFC484-2X FPGA development board. The output from each module was tested using appropriate software to ensure the correctness of the output result. On the transmitter part there are four blocks which consists of mapper (modulation), serial to parallel, IFFT and parallel to serial block. Each of these blocks was tested using Altera Max+Plus II software during design process. This is to ensure that the hardware module was correctly working when implemented in the FPGA hardware. During the implementation stage, the operation for IFFT was tested using Matlab software. Since IFFT is base on mathematical operation, Matlab is the best platform to compare the computation result. The comparison result shows that IFFT module is working correctly as the Matlab computation. Some computation gives slightly different from Matlab especially in imaginary value and this problem has been discussed in the analysis and discussion chapter. Thus, base on the test result, it was concluded that IFFT module was viably used in transmitter part as processing module.

The same process was done at the receiver part whereby each of the modules was tested during design process. On the implementation stage, FFT operation was tested using Matlab software. From the result shown in the results chapter, FFT module was correctly operated as Matlab computation. The different was only that the result of the FFT computation was in decimal while Matlab provide in floating point value. Matlab result was rounded such that it can be equally compared with the

FFT computation using FPGA. FFT can accept real and imaginary at the input because the data received from transmitter is in real and imaginary format. The problem encountered by this module has been discussed in the analysis and discussion chapter. As mention in this chapter, FFT module was finely worked if the input only in real number format. If the input in both real and imaginary, the result was not fully worked as Matlab computation due to the problem discussed in previous chapter. Thus, this module was not fully viable to be used at the receiver unless the input from transmitter only given in real number format.

Other modules such as serial to parallel, parallel to serial and mapping module was correctly worked. Thus, this module can be used as part of the OFDM system. The waveform result for these modules was given in hardware design chapter and the discussion regarding the operation of these modules was also made in that chapter. The design can be further made to an improvement base on the suggestion discussed in this chapter.

8.1 Proposed Future Works

Some recommendations are suggested to overcome the problem encountered during development of this project. First is to use higher bit representation to represent the number. Instead of using 8 bit binary representation, use 16 bit or more to represent each number in binary. The reason of using this method is to make the number representation can represent in more wide range of number. Thus the risk of overflow problem will decrease. Beside that, selective code words also can be used at the input such as input is limited from 0 to 64 for positive value of 8 bit binary two’s complement.

Use higher fixes point representation for point value representation. Floating point format also can be considered as the solution to reduce error number representation especially for twiddle factor value which is 0.7071. Although floating point consume processing time and output latency, but it is an excellence method to overcome accuracy problem.

Beside that, it is suggested to create a circuit to detect the overflow by indicating the flag or whatever way to ensure that the user know that the input given creates error to the system. The user will notice this problem and will change the input value to ensure no error occurred.

In this design, the receiver module which is mainly using FFT is good at processing for the positive input value. Therefore, any imaginary value should be mapped into real value such that receiver can process the input data correctly.

For the future works, it is suggested to develop other modules such as interleaving, error correction, QAM or QPSK modulation, cyclic prefix module and RF part. These modules will make a complete set of OFDM system for transmitter and receiver.

REFERENCES

1. Dusan Matiae, “OFDM as a possible modulation technique for multimedia applications in the range of mm waves”, TUD-TVS, 1998.
2. R.W Chang, “Synthesis of Band limited Orthogonal Signals for Multichannel Data Transmission”, Bell System Tech. J., pp.1775-1776, Dec 1996.
3. B. R. Saltzberg, “Performance of an Efficient Parallel DataTransmission System”, IEEE Trans. Comm., pp. 805-811, Dec 1967.
4. S. B Weinstein and P.M. Ebert, “Data Transmission by Frequency Division Multiplexing Using the Discrete Fourier Transform”, IEEE Transactions on Communication Technology”, Vol. COM-19, pp. 628-634, October 1971.
5. A. Peled an A. Ruiz, “Frequency Domain Data Transmission using Reduced Computational Complexity Algorithms”, In Proc. IEEE Int. conf. Acoust., Speech, Signal Processing, pp. 964-967, Denver, CO, 1980.
6. Uwe Meyer-Baese, Digital Signal Processing with Field Programmable Gate Arrays, second edition, Springer, Berlin, 2004.
7. Paul A. Lynn, Wolfgang Fuerst, Introductory Digital Signal Processing with Computer Application, Second Edition, John Wiley & Sons, England, 1999.
8. Aseem Pandey,Shyam Ratan Agrawalla and Shrikant Manivannan, 2002, “VLSI Implementation of OFDM Modem”, White Paper, Wipro Technologies-Wipro Limited (NYSE:WIT).
9. “An Introduction to OFDM”, International Engineering Consortium (IEC), http://www.iec.org/online/tutorial/ofdm/topic04.html.
10. Brown, Stephen D Zvonko G. Vranesic (2000). “Fundamentalls of Digital Logic with VHDL Design”, McGraw-Hill Higher Education.
11. “Orthogonal Frequency Division Multiplexing (OFDM) Explained”, Magis Networks, Inc., February 8, 2001, www.magisnetworks.com.
12. Erich Cosby, 2001, “Orthogonal Frequency Division Multiplexing (OFDM): Tutorial and Analysis”, www.eng.jcu.edu.au/eric/thesis/Thesis.htm.

13. "Advantages of OFDM" http//pic.qcslink.com/introPLC/AdvOFDM.htm.
14. Fredrik Kristensen, Peter Nilsson and Anders Olsson, "Flexible baseband transmitter of OFDM", www.sciencedirect.com.
15. "OFDM Tutorial", Wave Report, www.wave-report.com/tutorials/ofdm.htm.
16. "Wide-Band Orthogonal Frequency Multiplexing (W-Band)", White Paper by Wireless Data Communications Inc., www.wi-lan.com.
17. Rulph Chassaing, "Digital Signal Processing with C and the TMS320C30", John Wiley & Sons, Inc., Canada, 1992.
18. Design of an OFDM Transmitter and Receiver using FPGA , Loo Kah Cheng, UTM, 2004.
19. Implementation of 8 point IFFT and FFT for OFDM system, Nor Hafizah Bt Abdul Satar, UTM, 2004.
20. Advanced Electronic Communication Systems, Wayne Tomasi, 5th edition, Prentice Hall 2001.
21. Communication Systems, Simon Haykin, 4th Edition, Wiley 2000.
22. Modulation and Coding for Wireless Communications, Alister Burr, Prentice Hall 2001.
23. VHDL for Designers, Stefan Sjoholm and Lennart Lindh, Prentice Hall 1997.
24. VHDL for Programmable Logic, Kevin Skahill, Addison Wesley 1996.
25. Single and Multi-Carrier Quadrature Amplitude Modulation: principle and Applications for Personal Communications, WLANs and Broadcasting, L.Hanzo, W.Webb and T.Keller, Wiley, 2000.

Appendices A

VHDL CODE FOR MAPPER

```
-- bpsk
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;

entity bpsk is
        port (
                CLK : in STD_LOGIC;
                d : in STD_LOGIC;
                q : out STD_LOGIC_VECTOR(1 downto 0)
        );
end bpsk;

architecture bpsk_arch of bpsk is
begin
        process (clk)
begin
        if (clk' event and clk='1') then
                if d='0' then
                        q<= "01";
                else
                        q<= "11";
                end if;
        end if;
end process;
        -- Enter concurrent statements here
end bpsk_arch;
```

Appendices B

- VHDL CODE FOR SERIAL TO PARALLEL

```
library IEEE;
library work;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use work.PACK.all;
entity Se2par is
	port (
		CLK : in STD_LOGIC;
		RSTn : in STD_LOGIC;
		SERIN : in STD_LOGIC;
		PERRn : out STD_LOGIC;
		DRDY : out STD_LOGIC;
		DOUT : out STD_LOGIC_VECTOR(7 downto 0)
	);
end Se2par;

architecture Se2par_arch of Se2par is
	signal DFF:		STD_LOGIC_VECTOR(7 downto 0);
	signal CNT		:	STD_LOGIC_VECTOR(2 downto 0);
	signal CNT7_FF, SL_EN, NORMAL :	std_logic;
	signal PERRn_FF, PAR_EN, CNT_EN,REDUCE_XOR:		std_logic;
begin
regs:process
begin
	wait until (CLK' event and CLK='1');
	if (RSTn = '0') then
		DFF	<=	(DFF' range => '0');
		NORMAL <=	'1';
		PERRn_FF		<=	'1';
		DRDY	<=	'0';
		PAR_EN	<=	'0';
		CNT7_FF	<=	'0';
		CNT			<=	"000";
	else
		if (SL_EN = '1') then
			DFF <= DFF (6 downto 0)&SERIN;
		end if;
		if (PERRn_FF = '0') then
			NORMAL <= '0';
		end if;
		if (CNT_EN ='1') then
			CNT <= CNT + 1;
		end if;
	if	(PAR_EN = '1') then
			PERRn_FF <= not (REDUCE_XOR xor SERIN);
	end if;
		DRDY	<= PAR_EN;
		CNT7_FF	<=	CNT(2) and CNT(1) and CNT (0);
		PAR_EN	<=	CNT7_FF;
	end if;
	end process;
	SL_EN	<=	NORMAL and (CNT7_FF or CNT(2) or CNT(1) or CNT(0));
	CNT_EN	<=	'1'	when	(NORMAL ='1') AND (PAR_EN ='0') AND
					(CNT7_FF ='0') AND (PERRn_FF ='1') AND
					((CNT ="000" AND SERIN ='1') OR
					(CNT /= "000")) ELSE '0';
	PERRN	<=	PERRn_FF;
	DOUT	<=	DFF;
		-- Enter concurrent statements here
end Se2par_arch;
```

source code for work.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;

package PACK is
	function REDUCE_AND(DIN:   in std_logic_vector) return std_logic;
	function REDUCE_OR(DIN:                in std_logic_vector) return std_logic;
	function REDUCE_XOR(DIN:   in std_logic_vector) return std_logic;
end PACK;

package body PACK is
	function REDUCE_AND(DIN:   in std_logic_vector) return std_logic is
		variable result: std_logic;
	begin
		result := '1';
		for i in DIN' range loop
			result := result and DIN(i);
		end loop;
	return result;
	end REDUCE_AND;
-------------------------------------------------------------------------
	function REDUCE_OR( DIN:     in std_logic_vector) return std_logic is
		variable result: std_logic;
	begin
		result := '0';
		for i in DIN' range loop
			result := result or DIN(i);
		end loop;
	return result;
	end REDUCE_OR;
-------------------------------------------------------------------------
	function REDUCE_XOR( DIN:  in std_logic_vector) return std_logic is
		variable result: std_logic;
	begin
		result := '0';
		for i in DIN' range loop
			result := result xor DIN(i);
		end loop;
	return result;
	end REDUCE_XOR;
end PACK;
```

Appendices C

```
-- VHDL CODE FOR PARALLEL TO SERIAL
library IEEE;
library work;
use IEEE.std_logic_1164.all;
use work.PACK.all;

entity par2ser is
	port (
		CLK : in STD_LOGIC;
		RSTn : in STD_LOGIC;
		PL : in STD_LOGIC;
		DIN : in STD_LOGIC_VECTOR(7 downto 0);
		SEROUT : out STD_LOGIC
	);
end par2ser;

architecture par2ser_arch of par2ser is

	signal DFF:		std_logic_vector (7 downto 0);
	signal START, parbit :		std_logic;
begin
	p0: process (RSTn ,CLK)
begin
	if (RSTn = '0') then
		START	<= '0';
		PARBIT	<= '0';
		DFF	<= (DFF' range => '0');
	elsif (CLK' event and CLK='1' ) then
		if (PL='1') then
			START	<= '1';
			DFF	<= DIN;
			PARBIT	<= REDUCE_XOR(DIN);
		else
			START	<= DFF(7);
			DFF		<= DFF(6 downto 0)&PARBIT;
			PARBIT	<= '0';
		END IF;
	end if;
end process;
SEROUT <= START;
	-- Enter concurrent statements here
end par2ser_arch;
```

Appendices D

```
-- VHDL CODE FOR IFFT_MODULE AND INTERFACE
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;

entity IFFT_stage is
	port (
		clk : in STD_LOGIC;
		opcode	: in std_logic_vector ( 2 downto 0);
		dataA	: in std_logic_vector ( 7 downto 0);
		dataB	: in std_logic_vector ( 7 downto 0);
		result	: out std_logic_vector ( 7 downto 0)
		-- Enter port list here
	);
end IFFT_stage;

architecture IFFT_stage_arch of IFFT_stage is
constant twiddle : std_logic_vector :="010110101";
constant divider : std_logic_vector :="000100001";
signal result1,result2 : std_logic_vector ( 16 downto 0);
begin
	process (clk,opcode, dataA,dataB)
begin
	if clk' event and clk='0' then
				case opcode is
					when "000" =>
						result <= dataA + dataB;			--operation for addition
					when "001" =>
						result <= dataA - dataB;			--operation for subtraction
					when "010" =>
						result1 <= twiddle*dataA;			--twiddle multiplication
						if result1(7) ='1' then
							result <= result1(15 downto 8) + '1';
						else
							result <= result1(15 downto 8);
						end if;
					when "011" =>
						result2 <= divider*dataA;

						if result2(7) ='1' then
						result <= result2(15 downto 8) + '1'; --if twiddle multiplication is -ve
						else
				--add 1 bit and take 8-bit of the MSB as a result
						result <= result2(15 downto 8);	--if twiddle multiplication is +ve
						end if;
				--just take 8-bit of the MSB

					when "100" =>
						result <= -dataA;			--convert data from +ve to -ve
					when "101" =>					--just take 8-bit of the MSB
						result <= (not dataA) + '1';		--convert data from +ve to -ve
					when others  =>
						result <= dataA;			--pass the data
				end case;
	end if;
end process;
	-- Enter concurrent statements here
end IFFT_stage_arch;
```

```
-- IFFT_Interface
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity IFFT_Interface is
	port (
		clk : in STD_LOGIC;
		chipselect : in STD_LOGIC;
		address : in STD_LOGIC_VECTOR(1 downto 0);
		writedata : in STD_LOGIC_VECTOR(7 downto 0);
		readdata : out STD_LOGIC_VECTOR(7 downto 0);
		result : in STD_LOGIC_VECTOR(7 downto 0);
		data1 : out STD_LOGIC_VECTOR(7 downto 0);
		data2 : out STD_LOGIC_VECTOR(7 downto 0);
		opcode : out STD_LOGIC_VECTOR(2 downto 0)
	);
end IFFT_Interface;

architecture IFFT_Interface_arch of IFFT_Interface is
begin
	process (clk, chipselect)
	begin
		if clk'event and clk = '1' then
			if chipselect = '1' then
				case address is
					when "00" =>
						opcode <= writedata (2 downto 0);
					when "01" =>
						data1 <= writedata;
					when "10" =>
						data2 <= writedata;
					when others =>
						readdata <= result;
				end case;

			else
				readdata <= (OTHERS => '0');
			end if;
		end if;
	end process;
	-- Enter concurrent statements here
end IFFT_Interface_arch;
```

```
-- IFFT1_module
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity IFFT1_module is
	port (
		clk : in STD_LOGIC;
		chipselect : in STD_LOGIC;
		address : in STD_LOGIC_VECTOR(1 downto 0);
		writedata : in STD_LOGIC_VECTOR(7 downto 0);
		readdata : out STD_LOGIC_VECTOR(7 downto 0)
	);
end IFFT1_module;

architecture IFFT1_module_arch of IFFT1_module is
signal opkod : STD_LOGIC_VECTOR(2 downto 0);
signal lineA,lineB : STD_LOGIC_VECTOR(7 downto 0);
signal result_IFFT : STD_LOGIC_VECTOR(7 downto 0);

	-- VHDL Module Generator component declarations
	component IFFT_stage
	port (
		clk : in STD_LOGIC;
		opcode	: in std_logic_vector ( 2 downto 0);
		dataA	: in std_logic_vector ( 7 downto 0);
		dataB	: in std_logic_vector ( 7 downto 0);
		result	: out std_logic_vector ( 7 downto 0)
		-- Enter port list here
	);
	end component;

	component IFFT_Interface
	port (
		clk : in STD_LOGIC;
		chipselect : in STD_LOGIC;
		address : in STD_LOGIC_VECTOR(1 downto 0);
		writedata : in STD_LOGIC_VECTOR(7 downto 0);
		readdata : out STD_LOGIC_VECTOR(7 downto 0);
		result : in STD_LOGIC_VECTOR(7 downto 0);
		data1 : out STD_LOGIC_VECTOR(7 downto 0);
		data2 : out STD_LOGIC_VECTOR(7 downto 0);
		opcode : out STD_LOGIC_VECTOR(2 downto 0)
	);
	end component;

begin
	-- VHDL Module Generator component instantiations
	U_IFFT_stage: IFFT_stage
		port map (clk => clk,
				opcode => opkod,
				dataA => lineA,
				dataB => lineB,
				result => result_IFFT);

	U_IFFT_Interface: IFFT_Interface
		port map (clk => clk,
				chipselect => chipselect,
				address => address,
				writedata => writedata,
				readdata => readdata,
				result => result_IFFT,
				data1 => lineA,
				data2 => lineB,
				opcode => opkod);

	-- Enter concurrent statements here
end IFFT1_module_arch;
```

Appendices E

-- VHDL CODE FOR FFT_AND INTERFACE MODULE

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;

entity FFT_stage is
	port (
		clk : in STD_LOGIC;
		opcode	: in std_logic_vector ( 2 downto 0);
		dataA	: in std_logic_vector ( 7 downto 0);
		dataB	: in std_logic_vector ( 7 downto 0);
		result	: out std_logic_vector ( 7 downto 0)
		-- Enter port list here
	);
end FFT_stage;

architecture FFT_stage_arch of FFT_stage is
constant twiddle : std_logic_vector :="010110101";
signal result1 : std_logic_vector ( 16 downto 0);
begin
	process (clk,opcode, dataA,dataB)
begin
	if clk' event and clk='0' then
				case opcode is
					when "000" =>
						result <= dataA + dataB;		--operation for addition
					when "001" =>
						result <= dataA - dataB;		--operation for subtraction
					when "010" =>
						result1 <= twiddle*dataA;		--twiddle multiplication

						if result1(7) ='1' then
							result <= result1(15 downto 8) + '1';
						else
							result <= result1(15 downto 8);
						end if;

					when "011" =>
					result <= result1(15 downto 8) + 1;--if twiddle multiplication is -ve
					when "100" =>
					--add 1 bit and take 8-bit of the MSB as a result
					result <= result1(15 downto 8);    --if twiddle multiplication is +ve
					when "101" =>					--just take 8-bit of the MSB
						result <= -dataA;			--convert data from +ve to -ve
					when "110" =>					--just take 8-bit of the MSB
						result <= (not dataA) + '1';		--convert data from +ve to -ve
					when others  =>
						result <= dataA;			--pass the data
				end case;
	end if;
end process;
	-- Enter concurrent statements here
end FFT_stage_arch;
```

```
-- FFT_Interface
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity FFT_Interface is
	port (
		clk : in STD_LOGIC;
		chipselect : in STD_LOGIC;
		address : in STD_LOGIC_VECTOR(1 downto 0);
		writedata : in STD_LOGIC_VECTOR(7 downto 0);
		readdata : out STD_LOGIC_VECTOR(7 downto 0);
		result : in STD_LOGIC_VECTOR(7 downto 0);
		data1 : out STD_LOGIC_VECTOR(7 downto 0);
		data2 : out STD_LOGIC_VECTOR(7 downto 0);
		opcode : out STD_LOGIC_VECTOR(2 downto 0)
	);
end FFT_Interface;

architecture FFT_Interface_arch of FFT_Interface is
begin
	process (clk, chipselect)
	begin
		if clk'event and clk = '1' then
			if chipselect = '1' then
				case address is
					when "00" =>
						opcode <= writedata (2 downto 0);
					when "01" =>
						data1 <= writedata;
					when "10" =>
						data2 <= writedata;
					when others =>
						readdata <= result;
				end case;

			else
				readdata <= (OTHERS => '0');
			end if;
		end if;
	end process;
	-- Enter concurrent statements here
end FFT_Interface_arch;
```

```
-- FFT_module
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity FFT_module is
        port (
                clk : in STD_LOGIC;
                chipselect : in STD_LOGIC;
                address : in STD_LOGIC_VECTOR(1 downto 0);
                writedata : in STD_LOGIC_VECTOR(7 downto 0);
                readdata : out STD_LOGIC_VECTOR(7 downto 0)
        );
end FFT_module;

architecture FFT_module_arch of FFT_module is
signal  lineA, lineB        : std_logic_vector (7 downto 0);
signal opkod                        : std_logic_vector (2 downto 0);
signal result_FFT           : std_logic_vector (7 downto 0);

        -- VHDL Module Generator component declarations
        component FFT_stage
        port (
                clk : in STD_LOGIC;
                opcode  : in std_logic_vector ( 2 downto 0);
                dataA   : in std_logic_vector ( 7 downto 0);
                dataB   : in std_logic_vector ( 7 downto 0);
                result  : out std_logic_vector ( 7 downto 0)
                -- Enter port list here
        );
        end component;

        component FFT_Interface
        port (
                clk : in STD_LOGIC;
                chipselect : in STD_LOGIC;
                address : in STD_LOGIC_VECTOR(1 downto 0);
                writedata : in STD_LOGIC_VECTOR(7 downto 0);
                readdata : out STD_LOGIC_VECTOR(7 downto 0);
                result : in STD_LOGIC_VECTOR(7 downto 0);
                data1 : out STD_LOGIC_VECTOR(7 downto 0);
                data2 : out STD_LOGIC_VECTOR(7 downto 0);
                opcode : out STD_LOGIC_VECTOR(2 downto 0)
        );
        end component;

begin
        -- VHDL Module Generator component instantiations
        U_FFT_stage: FFT_stage
                port map (clk => clk,
                                        opcode => opkod,
                                        dataA => lineA,
                                        dataB => lineB,
                                        result => result_FFT);

        U_FFT_Interface: FFT_Interface
                port map (clk => clk,
                                         chipselect => chipselect,
                                         address => address,
                                         writedata => writedata,
                                         readdata => readdata,
                                         result => result_FFT,
                                         data1 => lineA,
                                         data2 => lineB,
                                         opcode => opkod);

        -- Enter concurrent statements here
end FFT_module_arch;
```

Appendices F

/*TEST VECTOR PROGRAM FOR IFFT IN C LABGUAGE*/

```
//The input to the ALU is from RAM and the result will store to RAM
#include "excalibur.h"
#include "stdio.h"


        int *opkod = (int*) 0x000004B3;          //memory for fft computation
        int *data1 = (int*) 0x000004B7;
        int *data2 = (int*) 0x000004BB;
        int *dataout = (int*) 0x000004BF;

        int *XR0 = (int*) 0x00000430;            //memory to hold data (input from keyboard)
        int *XR1 = (int*) 0x00000434;            //hold real data
        int *XR2 = (int*) 0x00000438;
        int *XR3 = (int*) 0x0000043C;

        int *XR4 = (int*) 0x00000490;
        int *XR5 = (int*) 0x00000494;
        int *XR6 = (int*) 0x00000498;
        int *XR7 = (int*) 0x0000049C;

        int *XR01 = (int*) 0x00040000;           //memory to hold data (1st stage of fft computation)
        int *XR11 = (int*) 0x00040004;           //hold real data
        int *XR21 = (int*) 0x00040008;
        int *XR31 = (int*) 0x0004000C;

        int *XR41 = (int*) 0x00040010;
        int *XR51t = (int*) 0x00040014;
        int *XR61t = (int*) 0x00040018;
        int *XR71t = (int*) 0x0004001C;

        int *XR51 = (int*) 0x00040020;           //memory to hold data (1st stage of fft computation)
        int *XI51 = (int*) 0x00040024;           //hold imaginary data
        int *XI61 = (int*) 0x00040028;
        int *XR71t2 = (int*) 0x0004002C;

        int *XR71 = (int*) 0x00040030;
        int *XI71 = (int*) 0x00040034;
        int *XR02 = (int*) 0x00040038;
        int *XR12 = (int*) 0x0004003C;

        int *XR22 = (int*) 0x00040040;
        int *XI32 = (int*) 0x00040044;
        int *XR42 = (int*) 0x00040048;
        int *XI42 = (int*) 0x0004004C;

        int *XR52 = (int*) 0x00040050;
        int *XI52 = (int*) 0x00040054;
        int *XR62 = (int*) 0x00040058;
        int *XI62 = (int*) 0x0004005C;

        int *XR72 = (int*) 0x00040060;
        int *XI72 = (int*) 0x00040064;
        int *XR03 = (int*) 0x00040068;
        int *XR13 = (int*) 0x0004006C;

        int *XI13 = (int*) 0x00040070;
        int *XR23 = (int*) 0x00040074;
        int *XI23 = (int*) 0x00040078;
        int *XR33 = (int*) 0x0004007C;

        int *XI33 = (int*) 0x00040080;
        int *XR43 = (int*) 0x00040084;
        int *XR53 = (int*) 0x00040088;
        int *XI53 = (int*) 0x0004008C;

        int *XR63 = (int*) 0x00040090;
        int *XI63 = (int*) 0x00040094;
        int *XR73 = (int*) 0x00040098;
        int *XI73 = (int*) 0x0004009C;
```

```
		int *XR0o = (int*) 0x000400A0;//
		int *XR1o = (int*) 0x000400A4;
		int *XI1o = (int*) 0x000400A8;
		int *XR2o = (int*) 0x000400AC;

		int *XI2o = (int*) 0x000400B0;
		int *XR3o = (int*) 0x000400B4;
		int *XI3o = (int*) 0x000400B8;
		int *XR4o = (int*) 0x000400BC;

		int *XR5o = (int*) 0x000400C0;
		int *XI5o = (int*) 0x000400C4;
		int *XR6o = (int*) 0x000400C8;
		int *XI6o = (int*) 0x000400CC;

		int *XR7o = (int*) 0x000400D0;
		int *XI7o = (int*) 0x000400D4;

		int a,b,c,d,e,f,g,h;			//input variable from keyboard
		int op,clear;

		void Stage_1_Fftopration();	// stage 1 function initialization
		void Stage_2_Fftopration(); // stage 2 function initialization
		void Stage_3_Fftopration();
		void Stage_4_Fftopration();
		void Stage_5_Fftopration();
/*		void Stage_6_Fftopration();
		void Stage_7_Fftopration();
		*/
main(void)
{

		/*clear = 0x00000000;	//to insert clear value to memory
		*XR0f = clear;					// clear all value in the ram
		*XR1f = clear;
		*XR2f = clear;
		*XR3f = clear;
		*XR4f = clear;
		*XR5f = clear;
		*XR6f = clear;
		*XR7f = clear;*/

		a = clear;				// clear all value in the ram
		b = clear;
		c = clear;
		d = clear;
		e = clear;
		f = clear;
		g = clear;
		h = clear;

		// get input from user Keyboard)
		printf ("\nInput 0 is: XR0 = ");
		scanf ("%d",&a);
		printf ("Input 1 is: XR1 = ");
		scanf ("%d",&b);
		printf ("Input 2 is: XR2 = ");
		scanf ("%d",&c);
		printf ("Input 3 is: XR3 = ");
		scanf ("%d",&d);
		printf ("Input 4 is: XR4 = ");
		scanf ("%d",&e);
		printf ("Input 5 is: XR5 = ");
		scanf ("%d",&f);
		printf ("Input 6 is: XR6 = ");
		scanf ("%d",&g);
		printf ("Input 7 is: XR7 = ");
		scanf ("%d",&h);

		// Set the Ram1 and Ram2 memory initial content

		/*a = 9;				// clear all value in the ram
		b = 8;
		c = 5;
		d = 11;
		e = 5;
```

```
	f = 11;
	g = 11;
	h = 6;*/

	*XR0 = a;
	*XR1 = b;
	*XR2 = c;
	*XR3 = d;
	*XR4 = e;
	*XR5 = f;
	*XR6 = g;
	*XR7 = h;

	/*XI1f = 0;
	*XI2f = 0;
	*XI3f = 0;
	*XI5f = 0;
	*XI6f = 5;
	*XI7f = 0;*/

	printf("\n\nvalue at ram XR0: %d\n", *XR0);
	printf("value at ram XR1: %d\n", *XR1);
	printf("value at ram XR2: %d\n", *XR2);
	printf("value at ram XR3: %d\n", *XR3);
	printf("value at ram XR4: %d\n", *XR4);
	printf("value at ram XR5: %d\n", *XR5);
	printf("value at ram XR6: %d\n", *XR6);
	printf("value at ram XR7: %d\n", *XR7);

	/*printf("\nvalue at ram XI1: %d\n", *XI1f);
	printf("value at ram XI2: %d\n", *XI2f);
	printf("value at ram XI3: %d\n", *XI3f);
	printf("value at ram XI5: %d\n", *XI5f);
	printf("value at ram XI6: %d\n", *XI6f);
	printf("value at ram XI7: %d\n", *XI7f);*/

	Stage_1_Fftopration();		//int,int,int,int,int,int,int,int
	Stage_2_Fftopration();
	Stage_3_Fftopration();
	Stage_4_Fftopration();
	Stage_5_Fftopration();

	printf("\n\nFINAL RESULTS OF IFFT\n\n");
	printf("\nXR0o = %d/8 = %d\n",*XR03,*XR0o);
printf("XR1o = %d/8 = %d\n",*XR13,*XR1o);
printf("XR2o = %d/8 = %d\n",*XR23,*XR2o);
printf("XR3o = %d/8 = %d\n",*XR33,*XR3o);
printf("XR4o = %d/8 = %d\n",*XR43,*XR4o);
printf("XR5o = %d/8 = %d\n",*XR53,*XR5o);
printf("XR6o = %d/8 = %d\n",*XR63,*XR6o);
printf("XR7o = %d/8 = %d\n",*XR73,*XR7o);

printf("\nXI0o = 0\n");
printf("XI1o = %d/8 = %d\n",*XI13,*XI1o);
printf("XI2o = %d/8 = %d\n",*XI23,*XI2o);
printf("XI3o = %d/8 = %d\n",*XI33,*XI3o);
printf("XI4o = 0\n");
printf("XI5o = %d/8 = %d\n",*XI53,*XI5o);
printf("XI6o = %d/8 = %d\n",*XI63,*XI6o);
printf("XI7o = %d/8 = %d\n",*XI73,*XI7o);


	return 0;
}


void Stage_1_Fftopration()
{
	printf ("\n-----STAGE 1 OF IFFT OPERATION-----\n");

	op = 0;			//opcode for hardware to execute addition operation

	*opkod = op;					//process input
	*data1 = *XR0;
	*data2 = *XR4;
	*XR01 = *dataout;			//0th output
```

```
		printf("\n\nXR01 = %d + %d = %d\n",*XR0,*XR4,*XR01); // XR01 = XR0 + XR4

		*opkod = op;
		*data1 = *XR1;
		*data2 = *XR5;
		*XR11 = *dataout;			//1st output real

		printf("XR11 = %d + %d = %d\n",*XR1,*XR5,*XR11); // XR11 = XR1 + XR5

		*opkod = op;
		*data1 = *XR2;
		*data2 = *XR6;
		*XR21 = *dataout;			//2nd output real

		printf("XR21 = %d + %d = %d\n",*XR2,*XR6,*XR21); // XR21 = XR2 + XR6

		*opkod = op;
		*data1 = *XR3;
		*data2 = *XR7;
		*XR31 = *dataout;			//3rd output real

		printf("XR31 = %d + %d = %d\n",*XR3,*XR7,*XR31); // XR31 = XR3 + XR7

		op = 1;
		*opkod = op;
		*data1 = *XR0;
		*data2 = *XR4;
		*XR41 = *dataout;			//4th output real

		printf("XR41 = %d - %d = %d\n", *XR0,*XR4,*XR41); // XR41 = XR0 - XR4

		*opkod = op;
		*data1 = *XR1;
		*data2 = *XR5;
		*XR51t = *dataout;			//5th output real

		printf("XR51t = %d - %d = %d\n",*XR1,*XR5,*XR51t); // XR51t = XR1 - XR5

		*opkod = op;
		*data1 = *XR2;
		*data2 = *XR6;
		*XR61t = *dataout;			//6th output real

		printf("XR61t = %d - %d = %d\n",*XR2,*XR6, *XR61t); // XR61t = XR2 - XR6

		*opkod = op;
		*data1 = *XR3;
		*data2 = *XR7;
		*XR71t = *dataout;			//7th output real

		printf("XR71t = %d - %d = %d\n",*XR3,*XR7,*XR71t); // XR71t = XR3 - XR7


}

void Stage_2_Fftopration()
{
		printf ("\n-----STAGE 2 OF IFFT OPERATION-----\n");

		op = 2;			//opcode for hardware to execute addition operation

		*opkod = op;					//process input
		*data1 = *XR51t;
		//*data2 = *XR4f;
		*XR51 = *dataout;

		printf("\n\nXR51 = 0.7071x%d = %d\n",*XR51t,*XR51); // XR51 = 0.7071*XR51t

		*opkod = op;					//process input
		*data1 = *XR51t;
		//*data2 = *XR4f;
		*XI51 = *dataout;
		printf("XI51 = 0.7071x%d = %d\n",*XR51t,*XI51); // XI51 = 0.7071*XI51t

		*XI61 = *XR61t;
```

```
		printf("XI61 = %d = %d\n",*XR61t,*XI61); // XI61 = XR61t

		*opkod = op;					//process input
		*data1 = *XR71t;
		//*data2 = *XR4f;
		*XR71t2 = *dataout;
		printf("\nXR71t2 = 0.7071x%d = %d\n",*XR71t,*XR71t2); // XR71t2 = 0.7071*XI71t

		op = 5;
		*opkod = op;					//process input
		*data1 = *XR71t2;
		*XR71 = *dataout;

		printf("\nXR71 = -1x%d = %d\n",*XR71t2,*XR71); // XI71 = -1*XI71t2

		op = 2;
		*opkod = op;					//process input
		*data1 = *XR71t;
		//*data2 = *XR4f;
		*XI71 = *dataout;
		printf("XI71 = 0.7071x%d = %d\n",*XR71t,*XI71); // XR71t2 = 0.7071*XI71t

}

void Stage_3_Fftopration()
{
		printf ("\n-----STAGE 3 OF IFFT OPERATION-----\n");

		op = 0;
		*opkod = op;					//process input
		*data1 = *XR01;
		*data2 = *XR21;
		*XR02 = *dataout;
		printf("\nXR02 = %d + %d = %d\n",*XR01,*XR21,*XR02); // XR02 = XR01+XR21

		*opkod = op;					//process input
		*data1 = *XR11;
		*data2 = *XR31;
		*XR12 = *dataout;
		printf("XR12 = %d + %d = %d\n",*XR11,*XR31,*XR12); // XR12 = XR11+XR31

		op = 1;
		*opkod = op;					//process input
		*data1 = *XR01;
		*data2 = *XR21;
		*XR22 = *dataout;
		printf("XR22 = %d - %d = %d\n",*XR01,*XR21,*XR22); // XR22 = XR01-XR21

		*opkod = op;					//process input
		*data1 = *XR11;
		*data2 = *XR31;
		*XI32 = *dataout;
		printf("XI32 = %d - %d = %d\n",*XR11,*XR31,*XI32); // XI32 = XR11-XR31

		*XR42 = *XR41;
		printf("XR42 = %d = %d\n",*XR41,*XR42);

		*XI42 = *XI61;
		printf("XI42 = %d = %d\n",*XI61,*XI42);

		op = 0;
		*opkod = op;					//process input
		*data1 = *XR51;
		*data2 = *XR71;
		*XR52 = *dataout;
		printf("XR52 = %d + %d = %d\n",*XR51,*XR71,*XR52); // XR52 = XR51+XR71

		*opkod = op;					//process input
		*data1 = *XI51;
		*data2 = *XI71;
		*XI52 = *dataout;
		printf("XI52 = %d + %d = %d\n",*XI51,*XI71,*XI52); // XI52 = XI51+XI71

		*XR62 = *XR41;
		printf("XR62 = %d = %d\n",*XR41,*XR62);
```

```
	op = 5;
	*opkod = op;				//process input
	*data1 = *XI61;
	//*data2 = *XI71;
	*XI62 = *dataout;
	printf("XI62 = -1x%d = %d\n",*XI61,*XI62); // XI52 = -1xXI62

	op = 1;
	*opkod = op;				//process input
	*data1 = *XI51;
	*data2 = *XI71;
	*XR72 = *dataout;
	printf("XR72 = %d - %d = %d\n",*XI51,*XI71,*XR72); // XR72 = XI51-XI71

	*opkod = op;				//process input
	*data1 = *XR51;
	*data2 = *XR71;
	*XI72 = *dataout;
	printf("XI72 = %d - %d = %d\n",*XR51,*XR71,*XI72); // XI72 = XR51-XR71


}


void Stage_4_Fftopration()
{
	printf ("\n-----STAGE 4 OF IFFT OPERATION-----\n");
	op = 0;
	*opkod = op;				//process input
	*data1 = *XR02;
	*data2 = *XR12;
	*XR03 = *dataout;
	printf("\nXR03 = %d + %d = %d\n",*XR02,*XR12,*XR03); // XR03 = XR02+XR12

	*opkod = op;				//process input
	*data1 = *XR42;
	*data2 = *XR52;
	*XR13 = *dataout;
	printf("XR13 = %d + %d = %d\n",*XR42,*XR52,*XR13); // XR13 = XR42+XR52

	op = 1; //subtraction operation
	*opkod = op;				//process input
	*data1 = *XI42;
	*data2 = *XI52;
	*XI13 = *dataout;
	printf("XI13 = %d + %d = %d\n",*XI42,*XI52,*XI13); // XI13 = XI42+XI52

	*XR23 = *XR22;
	printf("XR23 = %d = %d\n",*XR22,*XR23);

	*XI23 = *XI32;
	printf("XI23 = %d = %d\n",*XI32,*XI23);

	op = 0;
	*opkod = op;				//process input
	*data1 = *XR62;
	*data2 = *XR72;
	*XR33 = *dataout;
	printf("XR33 = %d + %d = %d\n",*XR62,*XR72,*XR33); // XR33 = XR62+XR72

	*opkod = op;				//process input
	*data1 = *XI62;
	*data2 = *XI72;
	*XI33 = *dataout;
	printf("XI33 = %d + %d = %d\n",*XI62,*XI72,*XI33); // XI33 = XI62+XI72

	op = 1;
	*opkod = op;				//process input
	*data1 = *XR02;
	*data2 = *XR12;
	*XR43 = *dataout;
	printf("XR43 = %d - %d = %d\n",*XR02,*XR12,*XR43); // XR43 = XR02-XR12

	op = 1;
	*opkod = op;				//process input
	*data1 = *XR42;
```

```
	*data2 = *XR52;
	*XR53 = *dataout;
	printf("XR53 = %d - %d = %d\n",*XR42,*XR52,*XR53); // XR53 = XR42-XR52

	*opkod = op;					//process input
	*data1 = *XI42;
	*data2 = *XI52;
	*XI53 = *dataout;
	printf("XI53 = %d - %d = %d\n",*XI42,*XI52,*XI53); // XI53 = XI42-XI52


	*XR63 = *XR22;
	printf("XR63 = %d = %d\n",*XR22,*XR63);

	op = 5;
	*opkod = op;					//process input
	*data1 = *XI32;
	//*data2 = *XI71;
	*XI63 = *dataout;
	printf("XI63 = -1x%d = %d\n",*XI32,*XI63); // XI63 = -1xXI32

	op = 1;
	*opkod = op;					//process input
	*data1 = *XR62;
	*data2 = *XR72;
	*XR73 = *dataout;
	printf("XR53 = %d - %d = %d\n",*XR42,*XR52,*XR53); // XR53 = XR42-XR52

	*opkod = op;					//process input
	*data1 = *XI62;
	*data2 = *XI72;
	*XI73 = *dataout;
	printf("XI73 = %d - %d = %d\n",*XI62,*XI72,*XI73); // XI73 = XI62-XI72



}

void Stage_5_Fftopration()
{
	printf ("\n-----STAGE 5 OF IFFT OPERATION-----\n");

	op = 3;				//opcode for hardware to execute addition operation
	*opkod = op;					//process input
	*data1 = *XR03;
	//*data2 = *XR21f;
	*XR0o = *dataout;
	printf("\nXR0o = %d/8 = %d\n",*XR03,*XR0o);

	*opkod = op;					//process input
	*data1 = *XR13;
	//*data2 = *XR31f;
	*XR1o = *dataout;
	printf("XR1o = %d/8 = %d\n",*XR13,*XR1o);

	*opkod = op;					//process input
	*data1 = *XI13;
	//*data2 = *XR31f;
	*XI1o = *dataout;
	printf("XI1o = %d/8 = %d\n",*XI13,*XI1o);

	*opkod = op;					//process input
	*data1 = *XR23;
	//*data2 = *XR31f;
	*XR2o = *dataout;
	printf("XR2o = %d/8 = %d\n",*XR23,*XR2o);

	*opkod = op;					//process input
	*data1 = *XI23;
	//*data2 = *XR31f;
	*XI2o = *dataout;
	printf("XI2o = %d/8 = %d\n",*XI23,*XI2o);

	*opkod = op;					//process input
	*data1 = *XR33;
```

```
		//*data2 = *XR31f;
		*XR3o = *dataout;
		printf("XR3o = %d/8 = %d\n",*XR33,*XR3o);

		*opkod = op;				//process input
		*data1 = *XI33;
		//*data2 = *XR31f;
		*XI3o = *dataout;
		printf("XI3o = %d/8 = %d\n",*XI33,*XI3o);

		*opkod = op;				//process input
		*data1 = *XR43;
		//*data2 = *XR31f;
		*XR4o = *dataout;
		printf("XR4o = %d/8 = %d\n",*XR43,*XR4o);

		*opkod = op;				//process input
		*data1 = *XR53;
		//*data2 = *XR31f;
		*XR5o = *dataout;
		printf("XR5o = %d/8 = %d\n",*XR53,*XR5o);

		*opkod = op;				//process input
		*data1 = *XI53;
		//*data2 = *XR31f;
		*XI5o = *dataout;
		printf("XI5o = %d/8 = %d\n",*XI53,*XI5o);

		*opkod = op;				//process input
		*data1 = *XR63;
		//*data2 = *XR31f;
		*XR6o = *dataout;
		printf("XR6o = %d/8 = %d\n",*XR63,*XR6o);

		*opkod = op;				//process input
		*data1 = *XI63;
		//*data2 = *XR31f;
		*XI6o = *dataout;
		printf("XI6o = %d/8 = %d\n",*XI63,*XI6o);

		*opkod = op;				//process input
		*data1 = *XR73;
		//*data2 = *XR31f;
		*XR7o = *dataout;
		printf("XR7o = %d/8 = %d\n",*XR73,*XR7o);

		*opkod = op;				//process input
		*data1 = *XI73;
		//*data2 = *XR31f;
		*XI7o = *dataout;
		printf("XI7o = %d/8 = %d\n",*XI73,*XI7o);

}
```

Appendices G

/* TEST VECTOR SOURCE CODE FOR FFT IN C LANGUAGE*/

```
//The input to the ALU is from RAM and the result will store to RAM
#include "excalibur.h"
#include "stdio.h"


        int *opkod = (int*) 0x000004B3; //memory for fft computation
        int *data1 = (int*) 0x000004B7;
        int *data2 = (int*) 0x000004BB;
        int *dataout = (int*) 0x000004BF;

        int *XR0f = (int*) 0x00000430;//memory to hold data (input from keyboard)
        int *XR1f = (int*) 0x00000434;  //hold real data
        int *XR2f = (int*) 0x00000438;
        int *XR3f = (int*) 0x0000043C;

        int *XR4f = (int*) 0x00000490;
        int *XR5f = (int*) 0x00000494;
        int *XR6f = (int*) 0x00000498;
        int *XR7f = (int*) 0x0000049C;

        int *XI0f = (int*) 0x00040000;
        int *XI1f = (int*) 0x00040004;              //hold imaginary data
        int *XI2f = (int*) 0x00040008;
        int *XI3f = (int*) 0x0004000C;

        int *XI4f = (int*) 0x00040010;
        int *XI5f = (int*) 0x00040014;
        int *XI6f = (int*) 0x00040018;
        int *XI7f = (int*) 0x0004001C;

        int *XR01f = (int*) 0x00040020;//memory to hold data (1st stage of fft computation)
        int *XR11f = (int*) 0x00040024;             //hold real data
        int *XR21f = (int*) 0x00040028;
        int *XR31f = (int*) 0x0004002C;

        int *XR41f = (int*) 0x00040030;
        int *XR51tf = (int*) 0x00040034;
        int *XR61tf = (int*) 0x00040038;
        int *XR71tf = (int*) 0x0004003C;

        int *XI01f = (int*) 0x00040040;             //memory to hold data (1st stage of fft computation)
        int *XI11f = (int*) 0x00040044;             //hold imaginary data
        int *XI21f = (int*) 0x00040048;
        int *XI31f = (int*) 0x0004004C;

        int *XI41f = (int*) 0x00040050;
        int *XI51tf = (int*) 0x00040054;
        int *XI61tf = (int*) 0x00040058;
        int *XI71tf = (int*) 0x0004005C;

        int *XR51taf = (int*) 0x00040060;
        int *XI51taf = (int*) 0x00040064;
        int *XR51tbf = (int*) 0x00040068;
        int *XI51tbf = (int*) 0x0004006C;

        int *XR61f = (int*) 0x00040070;
        int *XI61f = (int*) 0x00040074;
        int *XR71taf = (int*) 0x00040078;
        int *XI71taf = (int*) 0x0004007C;

        int *XI71tbf = (int*) 0x00040080;
        int *XR71tbf = (int*) 0x00040084;
        int *XI51taaf = (int*) 0x00040088;
        int *XI51tbbf = (int*) 0x0004008C;

        int *XI71taaf = (int*) 0x00040090;
        int *XR71taaf = (int*) 0x00040094;
        int *XI71tbbf = (int*) 0x00040098;
        int *XR71tbbf = (int*) 0x0004009C;

        int *XR51f = (int*) 0x000400A0;
```

```
int *XI51f = (int*) 0x000400A4;
int *XR71f = (int*) 0x000400A8;
int *XI71f = (int*) 0x000400AC;
/*
int *XR01f = (int*) 0x000400B0;
int *XI01f = (int*) 0x000400B4;
int *XR21f = (int*) 0x000400B8;
int *XI21f = (int*) 0x000400BC;

int *XR11f = (int*) 0x000400C0;
int *XI11f = (int*) 0x000400C4;
int *XR31f = (int*) 0x000400C8;
int *XI31f = (int*) 0x000400CC;

int *XR41f = (int*) 0x000400D0;
int *XI41f = (int*) 0x000400D4;
int *XR61f = (int*) 0x000400D8;
int *XI61f = (int*) 0x000400DC;

int *XR51f = (int*) 0x000400E0;
int *XI51f = (int*) 0x000400E4;
int *XR71f = (int*) 0x000400E8;
int *XI71f = (int*) 0x000400EC;*/

int *XR02f = (int*) 0x000400F0;
int *XR22f = (int*) 0x000400F4;
int *XI02f = (int*) 0x000400F8;
int *XI22f = (int*) 0x000400FC;

int *XR12f = (int*) 0x00040100;
int *XR32tf = (int*) 0x00040104;
int *XI12f = (int*) 0x00040108;
int *XI32tf = (int*) 0x0004010C;

int *XR42f = (int*) 0x00040120;
int *XR62f = (int*) 0x00040124;
int *XI42f = (int*) 0x00040128;
int *XI62f = (int*) 0x0004012C;

int *XR52f = (int*) 0x00040130;
int *XR72tf = (int*) 0x00040134;
int *XI52f = (int*) 0x00040138;
int *XI72tf = (int*) 0x0004013C;

int *XI32f = (int*) 0x00040140;
int *XR32f = (int*) 0x00040144;
int *XI72f = (int*) 0x00040148;
int *XR72f = (int*) 0x0004014C;

int *XR03f = (int*) 0x00040150;
int *XR43f = (int*) 0x00040154;
int *XI03f = (int*) 0x00040158;
int *XI43f = (int*) 0x0004015C;

int *XR23f = (int*) 0x00040160;
int *XR63f = (int*) 0x00040164;
int *XI23f = (int*) 0x00040168;
int *XI63f = (int*) 0x0004016C;

int *XR13f = (int*) 0x00040170;
int *XR53f = (int*) 0x00040174;
int *XI13f = (int*) 0x00040178;
int *XI53f = (int*) 0x0004017C;

int *XR33f = (int*) 0x00040180;
int *XR73f = (int*) 0x00040184;
int *XI33f = (int*) 0x00040188;
int *XI73f = (int*) 0x0004018C;


int a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p;                    //input variable from keyboard
int op,clear;

void Stage_1_Fftopration();         // stage 1 function initialization
void Stage_2_Fftopration(); // stage 2 function initialization
void Stage_3_Fftopration();
```

```
		void Stage_4_Fftopration();
		void Stage_5_Fftopration();
		void Stage_6_Fftopration();
		void Stage_7_Fftopration();

main(void)
{

		/*clear = 0x00000000;	//to insert clear value to memory
		*XR0f = clear;					// clear all value in the ram
		*XR1f = clear;
		*XR2f = clear;
		*XR3f = clear;
		*XR4f = clear;
		*XR5f = clear;
		*XR6f = clear;
		*XR7f = clear;*/

		a = 0;					// clear all value in the ram
		b = 0;
		c = 0;
		d = 0;
		e = 0;
		f = 0;
		g = 0;
		h = 0;

		*XI1f = 0;
		*XI2f = 0;
		*XI3f = 0;
		*XI5f = 0;
		*XI6f = 0;
		*XI7f = 0;

		// get input from user Keyboard)
		printf ("\nInput 0 is: XR0 = ");
		scanf ("%d",&a);
		printf ("Input 1 is: XR1 = ");
		scanf ("%d",&b);
		printf ("Input 2 is: XR2 = ");
		scanf ("%d",&c);
		printf ("Input 3 is: XR3 = ");
		scanf ("%d",&d);
		printf ("Input 4 is: XR4 = ");
		scanf ("%d",&e);
		printf ("Input 5 is: XR5 = ");
		scanf ("%d",&f);
		printf ("Input 6 is: XR6 = ");
		scanf ("%d",&g);
		printf ("Input 7 is: XR7 = ");
		scanf ("%d",&h);



		printf ("\nInput 0 is: XI0 = ");
		scanf ("%d",&i);
		printf ("Input 1 is: XI1 = ");
		scanf ("%d",&j);
		printf ("Input 2 is: XI2 = ");
		scanf ("%d",&k);
		printf ("Input 3 is: XI3 = ");
		scanf ("%d",&l);
		printf ("Input 4 is: XI4 = ");
		scanf ("%d",&m);
		printf ("Input 5 is: XI5 = ");
		scanf ("%d",&n);
		printf ("Input 6 is: XI6 = ");
		scanf ("%d",&o);
		printf ("Input 7 is: XI7 = ");
		scanf ("%d",&p);

		// Set the Ram1 and Ram2 memory initial content

		/*a = 5;					// clear all value in the ram
		b = 0;
		c = 0;
```

```
d = 0;
e = 0;
f = 0;
g = 0;
h = 0;
*/
*XR0f = a;
*XR1f = b;
*XR2f = c;
*XR3f = d;
*XR4f = e;
*XR5f = f;
*XR6f = g;
*XR7f = h;

*XI0f = 0;
*XI1f = j;
*XI2f = k;
*XI3f = l;
*XI4f = 0;
*XI5f = n;
*XI6f = o;
*XI7f = p;

printf("\n\nvalue at ram XR0: %d\n", *XR0f);
printf("value at ram XR1: %d\n", *XR1f);
printf("value at ram XR2: %d\n", *XR2f);
printf("value at ram XR3: %d\n", *XR3f);
printf("value at ram XR4: %d\n", *XR4f);
printf("value at ram XR5: %d\n", *XR5f);
printf("value at ram XR6: %d\n", *XR6f);
printf("value at ram XR7: %d\n", *XR7f);

printf("\nvalue at ram XI0: %d\n",*XI0f);
printf("value at ram XI1: %d\n", *XI1f);
printf("value at ram XI2: %d\n", *XI2f);
printf("value at ram XI3: %d\n", *XI3f);
printf("value at ram XI1: %d\n", *XI4f);
printf("value at ram XI5: %d\n", *XI5f);
printf("value at ram XI6: %d\n", *XI6f);
printf("value at ram XI7: %d\n", *XI7f);

Stage_1_Fftopration();
Stage_2_Fftopration();
Stage_3_Fftopration();
Stage_4_Fftopration();
Stage_5_Fftopration();
Stage_6_Fftopration();
Stage_7_Fftopration();

printf("\n\nOutput of XR03: %d\n", *XR03f);
printf("Output of XR13: %d\n", *XR13f);
printf("Output of XR23: %d\n", *XR23f);
printf("Output of XR33: %d\n", *XR33f);
printf("Output of XR43: %d\n", *XR43f);
printf("Output of XR53: %d\n", *XR53f);
printf("Output of XR63: %d\n", *XR63f);
printf("Output of XR73: %d\n", *XR73f);

printf("\nOutput of XI03: %d\n",*XI03f);
printf("Output of XI13: %d\n", *XI13f);
printf("Output of XI23: %d\n", *XI23f);
printf("Output of XI33: %d\n", *XI33f);
printf("Output of XI43: %d\n", *XI43f);
printf("Output of XI53: %d\n", *XI53f);
printf("Output of XI63: %d\n", *XI63f);
printf("Output of XI73: %d\n", *XI73f);

/*
printf("\n\nOutput of XR03: %d\n", *XR03f);
printf("Output of XR13: %d\n",*XI73f);
printf("Output of XR23: %d\n",*XR63f );
printf("Output of XR33: %d\n",*XI53f );
printf("Output of XR43: %d\n",*XR43f );
printf("Output of XR53: %d\n",*XI33f);
printf("Output of XR63: %d\n",*XR23f );
```

```
	printf("Output of XR73: %d\n",*XI13f );

	printf("\nOutput of XI03: %d\n",*XI03f);
	printf("Output of XI13: %d\n",*XR73f );
	printf("Output of XI23: %d\n",*XI63f );
	printf("Output of XI33: %d\n",*XR53f );
	printf("Output of XI43: %d\n",*XI43f );
	printf("Output of XI53: %d\n",*XR33f );
	printf("Output of XI63: %d\n",*XI23f );
	printf("Output of XI73: %d\n",*XR13f );
	*/
	return 0;
}


void Stage_1_Fftopration()
{
	printf ("\n-----STAGE 1 OF IFFT OPERATION-----\n");

	op = 0;			//opcode for hardware to execute addition operation

	*opkod = op;					//process input
	*data1 = *XR0f;
	*data2 = *XR4f;
	*XR01f = *dataout;			//0th output
	printf("\n\nXR01 = %d + %d = %d\n",*XR0f,*XR4f,*XR01f); // XR01 = XR0 + XR4

	*opkod = op;					//process input
	*data1 = *XI0f;
	*data2 = *XI4f;
	*XI01f = *dataout;			//0th output
	printf("XI01 = %d + %d = %d\n",*XI0f,*XI4f,*XI01f); // XI01 = XI0 + XI4

	*opkod = op;
	*data1 = *XR1f;
	*data2 = *XR5f;
	*XR11f = *dataout;			//1st output real
	printf("XR11 = %d + %d = %d\n",*XR1f,*XR5f,*XR11f); // XR11 = XR1 + XR5

	*opkod = op;
	*data1 = *XI1f;
	*data2 = *XI5f;
	*XI11f = *dataout;			//1st output imag
	printf("XI11 = %d + %d = %d\n",*XI1f,*XI5f,*XI11f); // XI11 = XI1 + XI5

	*opkod = op;
	*data1 = *XR2f;
	*data2 = *XR6f;
	*XR21f = *dataout;			//2nd output real
	printf("XR21 = %d + %d = %d\n",*XR2f,*XR6f,*XR21f); // XR21 = XR2 + XR6

	*opkod = op;
	*data1 = *XI2f;
	*data2 = *XI6f;
	*XI21f = *dataout;			//2nd output imag
	printf("XI21 = %d + %d = %d\n",*XI2f,*XI6f,*XI21f); // XI11 = XI2 + XI5

	*opkod = op;
	*data1 = *XR3f;
	*data2 = *XR7f;
	*XR31f = *dataout;			//3rd output real
	printf("XR31 = %d + %d = %d\n",*XR3f,*XR7f,*XR31f); // XR31 = XR3 + XR7

	*opkod = op;
	*data1 = *XI3f;
	*data2 = *XI7f;
	*XI31f = *dataout;			//3rd output imag
	printf("XI31 = %d + %d = %d\n",*XI3f,*XI7f,*XI31f); // XI31 = XI3 + XI7


	op = 1;
	*opkod = op;
	*data1 = *XR0f;
	*data2 = *XR4f;
	*XR41f = *dataout;
	printf("XR41 = %d - %d = %d\n", *XR0f,*XR4f,*XR41f); // XR41 = XR0 - XR4
```

```
		*opkod = op;
		*data1 = *XI0f;
		*data2 = *XI4f;
		*XI41f = *dataout;
		printf("XI41 = %d - %d = %d\n", *XI0f,*XI4f,*XI41f); // XI41 = XI0 - XI4

		*opkod = op;
		*data1 = *XR1f;
		*data2 = *XR5f;
		*XR51tf = *dataout;
		printf("XR51t = %d - %d = %d\n",*XR1f,*XR5f,*XR51tf); // XR51 = XR1 - XR5

		*opkod = op;
		*data1 = *XI1f;
		*data2 = *XI5f;
		*XI51tf = *dataout;			//5th output imag
		printf("XI51t = %d - %d = %d\n",*XI1f,*XI5f, *XI51tf); // XI51t = XI1 - XI5

		*opkod = op;
		*data1 = *XR2f;
		*data2 = *XR6f;
		*XR61tf = *dataout;			//6th output real
		printf("XR61t = %d - %d = %d\n",*XR2f,*XR6f, *XR61tf); // XR61t = XR2 - XR6

		*opkod = op;
		*data1 = *XI2f;
		*data2 = *XI6f;
		*XI61tf = *dataout;			//6th output imag
		printf("XI61t = %d - %d = %d\n",*XI2f,*XI6f,*XI61tf); // XI61t = XI2 - XI6

		*opkod = op;
		*data1 = *XR3f;
		*data2 = *XR7f;
		*XR71tf = *dataout;			//7th output real
		printf("XR71t = %d - %d = %d\n",*XR3f,*XR7f,*XR71tf); // XR71t = XR3 - XR7

		*opkod = op;
		*data1 = *XI3f;
		*data2 = *XI7f;
		*XI71tf = *dataout;			//7th output imag
		printf("XI71t = %d - %d = %d\n\n",*XI3f,*XI7f,*XI71tf);  // XI71t = XI3 - XI7

}

void Stage_2_Fftopration()
{
		printf ("\n-----STAGE 2 OF FFT OPERATION-----\n");

		op = 2;			//opcode for hardware to execute addition operation

		*opkod = op;					//process input
		*data1 = *XR51tf;
		//*data2 = *XR4f;
		*XR51taf = *dataout;
		printf("\nXR51ta = 0.7071x%d = %d\n",*XR51tf,*XR51taf);

		*opkod = op;					//process input
		*data1 = *XR51tf;
		//*data2 = *XR4f;
		*XI51taf = *dataout;
		printf("XI51ta = 0.7071x%d = %d\n",*XR51tf,*XI51taf);

		*opkod = op;					//process input
		*data1 = *XI51tf;
		//*data2 = *XR4f;
		*XR51tbf = *dataout;
		printf("\nXR51tb = 0.7071x%d = %d\n",*XI51tf,*XR51tbf);

		*opkod = op;					//process input
		*data1 = *XI51tf;
		//*data2 = *XR4f;
		*XI51tbf = *dataout;
		printf("XI51tb = 0.7071x%d = %d\n",*XI51tf,*XI51tbf);

		op = 6;
```

```
		*opkod = op;					//process input
		*data1 = *XR61tf;
		//*data2 = *XR4f;
		*XI61f = *dataout;			//6th output
		printf("\nXI61 = -1x%d = %d\n",*XR61tf,*XI61f); // XI61 = -1*XR61

		*opkod = op;					//process input
		*data1 = *XI61tf;
		//*data2 = *XR4f;
		*XR61f = *dataout;			//6th output
		printf("\nXR61 = -1x%d = %d\n",*XI61tf,*XR61f); // XR61 = -1*XI61


		op = 2;
		*opkod = op;					//process input
		*data1 = *XR71tf;
		//*data2 = *XR4f;
		*XR71taf = *dataout;
		printf("\nXR71ta = 0.7071x%d = %d\n",*XR71tf,*XR71taf);

		*opkod = op;					//process input
		*data1 = *XR71tf;
		//*data2 = *XR4f;
		*XI71taf = *dataout;
		printf("\nXI71ta = 0.7071x%d = %d\n",*XR71tf,*XI71taf);

		*opkod = op;					//process input
		*data1 = *XI71tf;
		//*data2 = *XR4f;
		*XR71tbf = *dataout;
		printf("\nXR71tb = 0.7071x%d = %d\n",*XI71tf,*XR71tbf);

		*opkod = op;					//process input
		*data1 = *XI71tf;
		//*data2 = *XR4f;
		*XI71tbf = *dataout;
		printf("\nXR71tb = 0.7071x%d = %d\n",*XI71tf,*XI71tbf);



}

void Stage_3_Fftopration()
{
		printf ("\n-----STAGE 3 OF FFT OPERATION-----\n");


		op = 6;					//opcode for hardware to execute addition operation
		*opkod = op;					//process input
		*data1 = *XI51taf;
		//*data2 = *XR4f;
		*XI51taaf = *dataout;
		printf("\nXI51taa = -1x%d = %d\n",*XI51taf,*XI51taaf); //

		op = 6;					//opcode for hardware to execute addition operation
		*opkod = op;					//process input
		*data1 = *XI51tbf;
		//*data2 = *XR4f;
		*XI51tbbf = *dataout;
		printf("\nXI51tbb = -1x%d = %d\n",*XI51tbf,*XI51tbbf); //

		*opkod = op;					//process input
		*data1 = *XI71taf;
		//*data2 = *XR4f;
		*XI71taaf = *dataout;
		printf("\nXI71taa = -1x%d = %d\n",*XI71taf,*XI71taaf); //

		*opkod = op;					//process input
		*data1 = *XR71taf;
		//*data2 = *XR4f;
		*XR71taaf = *dataout;
		printf("\nXR71taa = -1x%d = %d\n",*XR71taf,*XR71taaf); //

		*opkod = op;					//process input
		*data1 = *XI71tbf;
		//*data2 = *XR4f;
```

```
		*XI71tbbf = *dataout;
		printf("\nXI71tbb = -1x%d = %d\n",*XI71tbf,*XI71tbbf); //

		*opkod = op;					//process input
		*data1 = *XR71tbf;
		//*data2 = *XR4f;
		*XR71tbbf = *dataout;
		printf("\nXR71tbb = -1x%d = %d\n",*XR71tbf,*XR71tbbf); //



}


void Stage_4_Fftopration()
{
		printf ("\n-----STAGE 4 OF FFT OPERATION-----\n");

		op = 0;					//opcode for hardware to execute addition operation
		*opkod = op;					//process input
		*data1 = *XR51taf;
		*data2 = *XR51tbf;
		*XR51f = *dataout;
		printf("\nXR51 = %d + %d = %d\n",*XR51taf,*XR51tbf,*XR51f);

		*opkod = op;					//process input
		*data1 = *XI51taaf;
		*data2 = *XI51tbbf;
		*XI51f = *dataout;
		printf("XI51 = %d + %d = %d\n",*XI51taaf,*XI51tbbf,*XI51f);

		*opkod = op;					//process input
		*data1 = *XR71taaf;
		*data2 = *XR71tbbf;
		*XR71f = *dataout;
		printf("XR71 = %d + %d = %d\n",*XR71taaf,*XR71tbbf,*XR71f);

		*opkod = op;					//process input
		*data1 = *XI71taaf;
		*data2 = *XI71tbbf;
		*XI71f = *dataout;
		printf("XI71 = %d + %d = %d\n\n",*XI71taaf,*XI71tbbf,*XI71f);

}

void Stage_5_Fftopration()
{
		printf ("\n-----STAGE 5 OF FFT OPERATION-----\n");

		op = 0;					//opcode for hardware to execute addition operation
		*opkod = op;					//process input
		*data1 = *XR01f;
		*data2 = *XR21f;
		*XR02f = *dataout;
		printf("\nXR02 = %d + %d = %d\n",*XR01f,*XR21f,*XR02f);

		*opkod = op;					//process input
		*data1 = *XI01f;
		*data2 = *XI21f;
		*XI02f = *dataout;
		printf("XI02 = %d + %d = %d\n",*XI01f,*XI21f,*XI02f);

		*opkod = op;					//process input
		*data1 = *XR11f;
		*data2 = *XR31f;
		*XR12f = *dataout;
		printf("XR12 = %d + %d = %d\n",*XR11f,*XR31f,*XR12f);

		*opkod = op;					//process input
		*data1 = *XI11f;
		*data2 = *XI31f;
		*XI12f = *dataout;
		printf("XI12 = %d + %d = %d\n",*XI11f,*XI31f,*XI12f);

		*opkod = op;					//process input
		*data1 = *XR41f;
```

```
*data2 = *XR61f;
*XR42f = *dataout;
printf("XR42 = %d + %d = %d\n",*XR41f,*XR61f,*XR42f);

*opkod = op;                                    //process input
*data1 = *XI41f;
*data2 = *XI61f;
*XI42f = *dataout;
printf("XI42 = %d + %d = %d\n",*XI41f,*XI61f,*XI42f);

*opkod = op;                                    //process input
*data1 = *XR51f;
*data2 = *XR71f;
*XR52f = *dataout;
printf("XR52 = %d + %d = %d\n",*XR51f,*XR71f,*XR52f);

*opkod = op;                                    //process input
*data1 = *XI51f;
*data2 = *XI71f;
*XI52f = *dataout;
printf("XI52 = %d + %d = %d\n",*XI51f,*XI71f,*XI52f);

op = 01;                                //subtraction
*opkod = op;
*data1 = *XR01f;
*data2 = *XR21f;
*XR22f = *dataout;
printf("XR22 = %d - %d = %d\n",*XR01f,*XR21f,*XR22f);

*opkod = op;                                    //process input
*data1 = *XI01f;
*data2 = *XI21f;
*XI22f = *dataout;
printf("XI22 = %d - %d = %d\n",*XI01f,*XI21f,*XI22f);

*opkod = op;
*data1 = *XR11f;
*data2 = *XR31f;
*XR32tf = *dataout;
printf("XR32t = %d - %d = %d\n",*XR11f,*XR31f,*XR32tf);

*opkod = op;                                    //process input
*data1 = *XI11f;
*data2 = *XI31f;
*XI32tf = *dataout;
printf("XI32t = %d - %d = %d\n",*XI11f,*XI31f,*XI32tf);

*opkod = op;
*data1 = *XR41f;
*data2 = *XR61f;
*XR62f = *dataout;
printf("XR62 = %d - %d = %d\n",*XR41f,*XR61f,*XR62f);

*opkod = op;                                    //process input
*data1 = *XI41f;
*data2 = *XI61f;
*XI62f = *dataout;
printf("XI62 = %d - %d = %d\n",*XI41f,*XI61f,*XI62f);

*opkod = op;
*data1 = *XR51f;
*data2 = *XR71f;
*XR72tf = *dataout;
printf("XR72t = %d - %d = %d\n",*XR51f,*XR71f,*XR72tf);

*opkod = op;                                    //process input
*data1 = *XI51f;
*data2 = *XI71f;
*XI72tf = *dataout;
printf("XI72t = %d - %d = %d\n",*XI51f,*XI71f,*XI72tf);


}

void Stage_6_Fftopration()
{
```

```
		printf ("\n-----STAGE 6 OF FFT OPERATION-----\n");

		op = 6;				//opcode for hardware to execute addition operation
		*opkod = op;				//process input
		*data1 = *XR32tf;
		//*data2 = *XR21f;
		*XI32f = *dataout;
		printf("\nXI32 = -1x%d = %d\n",*XR32tf,*XI32f);

		*opkod = op;				//process input
		*data1 = *XI32tf;
		//*data2 = *XR21f;
		*XR32f = *dataout;
		printf("XR32 = -1x%d = %d\n",*XI32tf,*XR32f);

		*opkod = op;				//process input
		*data1 = *XR72tf;
		//*data2 = *XR21f;
		*XI72f = *dataout;
		printf("XI72 = -1x%d = %d\n",*XR72tf,*XI72f);

		*opkod = op;				//process input
		*data1 = *XI72tf;
		//*data2 = *XR21f;
		*XR72f = *dataout;
		printf("XR72 = -1x%d = %d\n",*XI72tf,*XR72f);

}

//////////////////////////-----stage 7------///////////////

void Stage_7_Fftopration()
{
		printf ("\n-----STAGE 7 OF FFT OPERATION-----\n");

		op = 0;				//opcode for hardware to execute addition operation
		*opkod = op;				//process input
		*data1 = *XR02f;
		*data2 = *XR12f;
		*XR03f = *dataout;
		printf("\nXR0 = %d + %d = %d\n",*XR02f,*XR12f,*XR03f);

		*opkod = op;				//process input
		*data1 = *XI02f;
		*data2 = *XI12f;
		*XI03f = *dataout;
		printf("XI0 = %d + %d = %d\n",*XI02f,*XI12f,*XI03f);

		*opkod = op;				//process input
		*data1 = *XR22f;
		*data2 = *XR32f;
		*XR23f = *dataout;
		printf("XR23 = %d + %d = %d\n",*XR22f,*XR32f,*XR23f);

		*opkod = op;				//process input
		*data1 = *XI22f;
		*data2 = *XI32f;
		*XI23f = *dataout;
		printf("XI23 = %d + %d = %d\n",*XI22f,*XI32f,*XI23f);

		*opkod = op;				//process input
		*data1 = *XR42f;
		*data2 = *XR52f;
		*XR13f = *dataout;
		printf("XR13 = %d + %d = %d\n",*XR42f,*XR52f,*XR13f);

		*opkod = op;				//process input
		*data1 = *XI42f;
		*data2 = *XI52f;
		*XI13f = *dataout;
		printf("XI13 = %d + %d = %d\n",*XI42f,*XI52f,*XI13f);

		*opkod = op;				//process input
		*data1 = *XR62f;
		*data2 = *XR72f;
		*XR73f = *dataout;
```

```
printf("XR73 = %d + %d = %d\n",*XR62f,*XR72f,*XR73f);

*opkod = op;                                    //process input
*data1 = *XI62f;
*data2 = *XI72f;
*XI33f = *dataout;
printf("XI33 = %d + %d = %d\n",*XI62f,*XI72f,*XI33f);

op = 01;                                //opcode for hardware to execute subtraction operation
*opkod = op;                                    //process input
*data1 = *XR02f;
*data2 = *XR12f;
*XR43f = *dataout;
printf("\nXR43 = %d - %d = %d\n",*XR02f,*XR12f,*XR43f);

*opkod = op;                                    //process input
*data1 = *XI02f;
*data2 = *XI12f;
*XI43f = *dataout;
printf("XI43 = %d - %d = %d\n",*XI02f,*XI12f,*XI43f);

*opkod = op;                                    //process input
*data1 = *XR22f;
*data2 = *XR32f;
*XR63f = *dataout;
printf("XR63 = %d - %d = %d\n",*XR22f,*XR32f,*XR63f);

*opkod = op;                                    //process input
*data1 = *XI22f;
*data2 = *XI32f;
*XI63f = *dataout;
printf("XI63 = %d - %d = %d\n",*XI02f,*XI12f,*XI63f);

*opkod = op;                                    //process input
*data1 = *XR42f;
*data2 = *XR52f;
*XR53f = *dataout;
printf("XR53 = %d - %d = %d\n",*XR42f,*XR52f,*XR53f);

*opkod = op;                                    //process input
*data1 = *XI42f;
*data2 = *XI52f;
*XI53f = *dataout;
printf("XI53 = %d - %d = %d\n",*XI42f,*XI52f,*XI53f);

*opkod = op;                                    //process input
*data1 = *XR62f;
*data2 = *XR72f;
*XR33f = *dataout;
printf("XR33 = %d - %d = %d\n",*XR62f,*XR72f,*XR33f);

*opkod = op;                                    //process input
*data1 = *XI62f;
*data2 = *XI72f;
*XI73f = *dataout;
printf("XI73 = %d - %d = %d\n",*XI62f,*XI72f,*XI73f);



}
```

Appendices H

/*TEST VECTOR PROGRAM FOR TRANSMITTER AND RECEIVER IN C LANGUAGE*/

```
#include "excalibur.h"
#include "stdio.h"


//////////--------- Memory Allocation for IFFT Process------//////////

        int *opkodA = (int*) 0x00000430;            //address of the transmitter
        int *data1A = (int*) 0x00000434;            //contain IFFT module
        int *data2A = (int*) 0x00000438;
        int *dataoutA = (int*) 0x0000043C;

        int *XR0 = (int*) 0x00000491;          //ram module
        int *XR1 = (int*) 0x00000495;
        int *XR2 = (int*) 0x00000499;
        int *XR3 = (int*) 0x0000049D;

        int *XR4 = (int*) 0x000004A2;
        int *XR5 = (int*) 0x000004A6;
        int *XR6 = (int*) 0x000004AA;
        int *XR7 = (int*) 0x000004AE;

        int *opkodB = (int*) 0x000004B3;            //address of the receiver
        int *data1B = (int*) 0x000004B7;            //contain FFT module
        int *data2B = (int*) 0x000004BB;
        int *dataoutB = (int*) 0x000004BF;

        int *XR01 = (int*) 0x00040000;         //memory to hold data (1st stage of ifft computation)
        int *XR11 = (int*) 0x00040004;         //hold real data
        int *XR21 = (int*) 0x00040008;
        int *XR31 = (int*) 0x0004000C;

        int *XR41 = (int*) 0x00040010;
        int *XR51t = (int*) 0x00040014;
        int *XR61t = (int*) 0x00040018;
        int *XR71t = (int*) 0x0004001C;

        int *XR51 = (int*) 0x00040020;         //memory to hold data (1st stage of fft computation)
        int *XI51 = (int*) 0x00040024;         //hold imaginary data
        int *XI61 = (int*) 0x00040028;
        int *XR71t2 = (int*) 0x0004002C;

        int *XR71 = (int*) 0x00040030;
        int *XI71 = (int*) 0x00040034;
        int *XR02 = (int*) 0x00040038;
        int *XR12 = (int*) 0x0004003C;

        int *XR22 = (int*) 0x00040040;
        int *XI32 = (int*) 0x00040044;
        int *XR42 = (int*) 0x00040048;
        int *XI42 = (int*) 0x0004004C;

        int *XR52 = (int*) 0x00040050;
        int *XI52 = (int*) 0x00040054;
        int *XR62 = (int*) 0x00040058;
        int *XI62 = (int*) 0x0004005C;

        int *XR72 = (int*) 0x00040060;
        int *XI72 = (int*) 0x00040064;
        int *XR03 = (int*) 0x00040068;
        int *XR13 = (int*) 0x0004006C;

        int *XI13 = (int*) 0x00040070;
        int *XR23 = (int*) 0x00040074;
        int *XI23 = (int*) 0x00040078;
        int *XR33 = (int*) 0x0004007C;

        int *XI33 = (int*) 0x00040080;
        int *XR43 = (int*) 0x00040084;
        int *XR53 = (int*) 0x00040088;
        int *XI53 = (int*) 0x0004008C;
```

```
int *XR63 = (int*) 0x00040090;
int *XI63 = (int*) 0x00040094;
int *XR73 = (int*) 0x00040098;
int *XI73 = (int*) 0x0004009C;

int *XR0o = (int*) 0x000400A0;//
int *XR1o = (int*) 0x000400A4;
int *XI1o = (int*) 0x000400A8;
int *XR2o = (int*) 0x000400AC;

int *XI2o = (int*) 0x000400B0;
int *XR3o = (int*) 0x000400B4;
int *XI3o = (int*) 0x000400B8;
int *XR4o = (int*) 0x000400BC;

int *XR5o = (int*) 0x000400C0;
int *XI5o = (int*) 0x000400C4;
int *XR6o = (int*) 0x000400C8;
int *XI6o = (int*) 0x000400CC;

int *XR7o = (int*) 0x000400D0;
int *XI7o = (int*) 0x000400D4;

//////////////-------- Memory Allocation for FFT Process------//////////

int *XR0f = (int*) 0x000400E0;          //memory to hold data (input from keyboard)
int *XR1f = (int*) 0x000400E4;          //hold real data
int *XR2f = (int*) 0x000400E8;
int *XR3f = (int*) 0x000400EC;

int *XR4f = (int*) 0x000400F0;
int *XR5f = (int*) 0x000400F4;
int *XR6f = (int*) 0x000400F8;
int *XR7f = (int*) 0x000400FC;

int *XI0f = (int*) 0x00040100;
int *XI1f = (int*) 0x00040104;          //hold imaginary data
int *XI2f = (int*) 0x00040108;
int *XI3f = (int*) 0x0004010C;

int *XI4f = (int*) 0x00040110;
int *XI5f = (int*) 0x00040114;
int *XI6f = (int*) 0x00040118;
int *XI7f = (int*) 0x0004011C;

int *XR01f = (int*) 0x00040120;         //memory to hold data (1st stage of fft computation)
int *XR11f = (int*) 0x00040124;         //hold real data
int *XR21f = (int*) 0x00040128;
int *XR31f = (int*) 0x0004012C;

int *XR41f = (int*) 0x00040130;
int *XR51tf = (int*) 0x00040134;
int *XR61tf = (int*) 0x00040138;
int *XR71tf = (int*) 0x0004013C;

int *XI01f = (int*) 0x00040140;         //memory to hold data (1st stage of fft computation)
int *XI11f = (int*) 0x00040144;         //hold imaginary data
int *XI21f = (int*) 0x00040148;
int *XI31f = (int*) 0x0004014C;

int *XI41f = (int*) 0x00040150;
int *XI51tf = (int*) 0x00040154;
int *XI61tf = (int*) 0x00040158;
int *XI71tf = (int*) 0x0004015C;

int *XR51taf = (int*) 0x00040160;
int *XI51taf = (int*) 0x00040164;
int *XR51tbf = (int*) 0x00040168;
int *XI51tbf = (int*) 0x0004016C;

int *XR61f = (int*) 0x00040170;
int *XI61f = (int*) 0x00040174;
int *XR71taf = (int*) 0x00040178;
int *XI71taf = (int*) 0x0004017C;

int *XI71tbf = (int*) 0x00040180;
```

```
int *XR71tbf = (int*) 0x00040184;
int *XI51taaf = (int*) 0x00040188;
int *XI51tbbf = (int*) 0x0004018C;

int *XI71taaf = (int*) 0x00040190;
int *XR71taaf = (int*) 0x00040194;
int *XI71tbbf = (int*) 0x00040198;
int *XR71tbbf = (int*) 0x0004019C;

int *XR51f = (int*) 0x000401A0;
int *XI51f = (int*) 0x000401A4;
int *XR71f = (int*) 0x000401A8;
int *XI71f = (int*) 0x000401AC;
/*
int *XR01f = (int*) 0x000401B0;
int *XI01f = (int*) 0x000401B4;
int *XR21f = (int*) 0x000401B8;
int *XI21f = (int*) 0x000401BC;

int *XR11f = (int*) 0x000401C0;
int *XI11f = (int*) 0x000401C4;
int *XR31f = (int*) 0x000401C8;
int *XI31f = (int*) 0x000401CC;

int *XR41f = (int*) 0x000401D0;
int *XI41f = (int*) 0x000401D4;
int *XR61f = (int*) 0x000401D8;
int *XI61f = (int*) 0x000401DC;

int *XR51f = (int*) 0x000401E0;
int *XI51f = (int*) 0x000401E4;
int *XR71f = (int*) 0x000401E8;
int *XI71f = (int*) 0x000401EC;*/

int *XR02f = (int*) 0x000401F0;
int *XR22f = (int*) 0x000401F4;
int *XI02f = (int*) 0x000401F8;
int *XI22f = (int*) 0x000401FC;

int *XR12f = (int*) 0x00040200;
int *XR32tf = (int*) 0x00040204;
int *XI12f = (int*) 0x00040208;
int *XI32tf = (int*) 0x0004020C;

int *XR42f = (int*) 0x00040220;
int *XR62f = (int*) 0x00040224;
int *XI42f = (int*) 0x00040228;
int *XI62f = (int*) 0x0004022C;

int *XR52f = (int*) 0x00040230;
int *XR72tf = (int*) 0x00040234;
int *XI52f = (int*) 0x00040238;
int *XI72tf = (int*) 0x0004023C;

int *XI32f = (int*) 0x00040240;
int *XR32f = (int*) 0x00040244;
int *XI72f = (int*) 0x00040248;
int *XR72f = (int*) 0x0004024C;

int *XR03f = (int*) 0x00040250;
int *XR43f = (int*) 0x00040254;
int *XI03f = (int*) 0x00040258;
int *XI43f = (int*) 0x0004025C;

int *XR23f = (int*) 0x00040260;
int *XR63f = (int*) 0x00040264;
int *XI23f = (int*) 0x00040268;
int *XI63f = (int*) 0x0004026C;

int *XR13f = (int*) 0x00040270;
int *XR53f = (int*) 0x00040274;
int *XI13f = (int*) 0x00040278;
int *XI53f = (int*) 0x0004027C;

int *XR33f = (int*) 0x00040280;
int *XR73f = (int*) 0x00040284;
```

```
int *XI33f = (int*) 0x00040288;
int *XI73f = (int*) 0x0004028C;

int a,b,c,d,e,f,g,h;                    //input variable from keyboard
int op;

////---------ifft function initialization-------///////
void Stage_1_IFFTopration();
void Stage_2_IFFTopration();
void Stage_2a_IFFTopration();
void Stage_3_IFFTopration();
void Stage_4_IFFTopration();
void Stage_5_IFFTopration();
////---------fft function initialization-------///////
void Stage_1_FFTopration();
void Stage_2_FFTopration();
void Stage_3_FFTopration();
void Stage_4_FFTopration();
void Stage_5_FFTopration();
void Stage_6_FFTopration();
void Stage_7_FFTopration();

main(void)
{
a = 0;                                  // clear all value in the ram
b = 0;
c = 0;
d = 0;
e = 0;
f = 0;
g = 0;
h = 0;

/*XI1f = 0;
*XI2f = 0;
*XI3f = 0;
*XI5f = 0;
*XI6f = 0;
*XI7f = 0;*/

// get input from user Keyboard)
printf ("\nInput 0 is: XR0 = ");
scanf ("%d",&a);
printf ("Input 1 is: XR1 = ");
scanf ("%d",&b);
printf ("Input 2 is: XR2 = ");
scanf ("%d",&c);
printf ("Input 3 is: XR3 = ");
scanf ("%d",&d);
printf ("Input 4 is: XR4 = ");
scanf ("%d",&e);
printf ("Input 5 is: XR5 = ");
scanf ("%d",&f);
printf ("Input 6 is: XR6 = ");
scanf ("%d",&g);
printf ("Input 7 is: XR7 = ");
scanf ("%d",&h);


/*a = 2;                                // clear all value in the ram
b = 1;
c = 2;
d = 9;
e = 11;
f = 7;
g = 14;
h = 14;
*/
*XR0 = a;
*XR1 = b;
*XR2 = c;
*XR3 = d;
*XR4 = e;
*XR5 = f;
*XR6 = g;
*XR7 = h;
```

```
printf ("\n\t~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
printf ("\n\tINPUT TO THE TRANSMITTER\n");
printf ("\n\t~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");

printf("\n\nvalue at ram XR0: %d\n",*XR0);
printf("value at ram XR1: %d\n", *XR1);
printf("value at ram XR2: %d\n", *XR2);
printf("value at ram XR3: %d\n", *XR3);
printf("value at ram XR4: %d\n", *XR4);
printf("value at ram XR5: %d\n", *XR5);
printf("value at ram XR6: %d\n", *XR6);
printf("value at ram XR7: %d\n", *XR7);

Stage_1_IFFTopration();
Stage_2_IFFTopration();
Stage_2a_IFFTopration();
Stage_3_IFFTopration();
Stage_4_IFFTopration();
Stage_5_IFFTopration();

*XR0f = 0;
*XR1f = 0;
*XI1f = 0;
*XR2f = 0;
*XI2f = 0;
*XR3f = 0;
*XI3f = 0;
*XR4f = 0;
*XR5f = 0;
*XI5f = 0;
*XR6f = 0;
*XI6f = 0;
*XR7f = 0;
*XI7f = 0;
printf ("\nOUTPUT OF THE TRANSMITTER \n");
printf("\nOutput of XR03: %d\n", *XR0o);
printf("Output of XR1o: %d\n", *XR1o);
printf("Output of XR2o: %d\n", *XR2o);
printf("Output of XR3o: %d\n", *XR3o);
printf("Output of XR4o: %d\n", *XR4o);
printf("Output of XR5o: %d\n", *XR5o);
printf("Output of XR6o: %d\n", *XR6o);
printf("Output of XR7o: %d\n", *XR7o);

//printf("\nOutput of XI0o: %d\n",*XI0o);
printf("Output of XI1o: %d\n", *XI1o);
printf("Output of XI2o: %d\n", *XI2o);
printf("Output of XI3o: %d\n", *XI3o);
//printf("Output of XI4o: %d\n", *XI4o);
printf("Output of XI5o: %d\n", *XI5o);
printf("Output of XI6o: %d\n", *XI6o);
printf("Output of XI7o: %d\n", *XI7o);

printf ("\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
printf ("\nEND OF THE TRANSMITTER PROCESS\n");
printf ("\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
printf ("Output from the Transmitter is transmitted to the Receiver module\n\n");
printf ("\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
printf ("\nINPUT TO THE RECEIVER\n");
printf ("\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");

/*XR0f = *XR0o;       //output Tx mapping to input Rx (in sequence)
*XI0f = 0;
*XR1f = *XR1o;
*XI1f = *XI1o;
*XR2f = *XR2o;
*XI2f = *XI2o;
*XR3f = *XR3o;
*XI3f = *XI3o;
*XR4f = *XR4o;
*XI4f = 0;
*XR5f = *XR5o;
*XI5f = *XI5o;
*XR6f = *XR6o;
*XI6f = *XI6o;
```

```
*XR7f = *XR7o;
*XI7f = *XI7o;
*/
*XR0f = *XR0o;        //output Tx mapping to input Rx (reverse order)
*XI0f = 0;
*XR1f = *XR4o;
*XI1f = 0;
*XR2f = *XR2o;
*XI2f = *XI2o;
*XR3f = *XR6o;
*XI3f = *XI6o;
*XR4f = *XR1o ;
*XI4f = *XI1o ;
*XR5f = *XR5o;
*XI5f = *XI5o;
*XR6f = *XR3o ;
*XI6f = *XI3o ;
*XR7f = *XR7o;
*XI7f = *XI7o;

/*XR0f = *XR0o;       //imaginary value = 0
*XI0f = 0;                          //output Tx mapping to input Rx (reverse order)
*XR1f = *XR4o;
*XI1f = 0;
*XR2f = *XR2o;
*XI2f = 0;
*XR3f = *XR6o;
*XI3f = 0;
*XR4f = *XR1o ;
*XI4f = 0 ;
*XR5f = *XR5o;
*XI5f = 0;
*XR6f = *XR3o ;
*XI6f = 0 ;
*XR7f = *XR7o;
*XI7f = 0;
*/
/*XR0f = *XR0o;                     //output Tx mapping to input Rx (in sequence)
*XI0f = 0;                          //imaginary value = 0
*XR1f = *XR1o;
*XI1f = 0;
*XR2f = *XR2o;
*XI2f = 0;
*XR3f = *XR3o;
*XI3f = 0;
*XR4f = *XR4o;
*XI4f = 0;
*XR5f = *XR5o;
*XI5f = 0;
*XR6f = *XR6o;
*XI6f = 0;
*XR7f = *XR7o;
*XI7f = 0;
*/
printf("\nReceiver buffer no XR0f: %d\n",*XR0f);
printf("Receiver buffer no XR1f: %d\n", *XR1f);
printf("Receiver buffer no XR2f: %d\n", *XR2f);
printf("Receiver buffer no XR3f: %d\n", *XR3f);
printf("Receiver buffer no XR4f: %d\n", *XR4f);
printf("Receiver buffer no XR5f: %d\n", *XR5f);
printf("Receiver buffer no XR6f: %d\n", *XR6f);
printf("Receiver buffer no XR7f: %d\n", *XR7f);

printf("\nReceiver buffer no XI0f: %d\n",*XI0f);
printf("Receiver buffer no XI1f: %d\n", *XI1f);
printf("Receiver buffer no XI2f: %d\n", *XI2f);
printf("Receiver buffer no XI3f: %d\n", *XI3f);
printf("Receiver buffer no XI4f: %d\n", *XI4f);
printf("Receiver buffer no XI5f: %d\n", *XI5f);
printf("Receiver buffer no XI6f: %d\n", *XI6f);
printf("Receiver buffer no XI7f: %d\n", *XI7f);

Stage_1_FFTopration();
Stage_2_FFTopration();
Stage_3_FFTopration();
Stage_4_FFTopration();
```

```
	Stage_5_FFTopration();
	Stage_6_FFTopration();
	Stage_7_FFTopration();

	printf ("\nOUTPUT FROM RECEIVER\n");
	/*
	printf("\n\nOutput of XR03: %d\n", *XR03f); //output in sequence order
	printf("Output of XR13: %d\n", *XR13f);
	printf("Output of XR23: %d\n", *XR23f);
	printf("Output of XR33: %d\n", *XR33f);
	printf("Output of XR43: %d\n", *XR43f);
	printf("Output of XR53: %d\n", *XR53f);
	printf("Output of XR63: %d\n", *XR63f);
	printf("Output of XR73: %d\n", *XR73f);

	printf("\nOutput of XI03: %d\n",*XI03f);
	printf("Output of XI13: %d\n", *XI13f);
	printf("Output of XI23: %d\n", *XI23f);
	printf("Output of XI33: %d\n", *XI33f);
	printf("Output of XI43: %d\n", *XI43f);
	printf("Output of XI53: %d\n", *XI53f);
	printf("Output of XI63: %d\n", *XI63f);
	printf("Output of XI73: %d\n", *XI73f);
	*/
	printf("\n\nOutput of XR03: %d\n", *XR03f);				//output as matlab order
	printf("Output of XR13: %d\n", *XR73f);
	printf("Output of XR23: %d\n", *XR63f);
	printf("Output of XR33: %d\n", *XR53f);
	printf("Output of XR43: %d\n", *XR43f);
	printf("Output of XR53: %d\n", *XR33f);
	printf("Output of XR63: %d\n", *XR23f);
	printf("Output of XR73: %d\n", *XR13f);

	printf("\nOutput of XI03: %d\n",*XI03f);
	printf("Output of XI13: %d\n", *XI73f);
	printf("Output of XI23: %d\n", *XI63f);
	printf("Output of XI33: %d\n", *XI53f);
	printf("Output of XI43: %d\n", *XI43f);
	printf("Output of XI53: %d\n", *XI33f);
	printf("Output of XI63: %d\n", *XI23f);
	printf("Output of XI73: %d\n", *XI13f);
	/*
	printf("\n\nOutput of XR03: %d\n", *XR03f); //output reverse order
	printf("Output of XR13: %d\n",*XI73f);
	printf("Output of XR23: %d\n",*XR63f );
	printf("Output of XR33: %d\n",*XI53f );
	printf("Output of XR43: %d\n",*XR43f );
	printf("Output of XR53: %d\n",*XI33f);
	printf("Output of XR63: %d\n",*XR23f );
	printf("Output of XR73: %d\n",*XI13f );

	printf("\nOutput of XI03: %d\n",*XI03f);
	printf("Output of XI13: %d\n",*XR73f );
	printf("Output of XI23: %d\n",*XI63f );
	printf("Output of XI33: %d\n",*XR53f );
	printf("Output of XI43: %d\n",*XI43f );
	printf("Output of XI53: %d\n",*XR33f );
	printf("Output of XI63: %d\n",*XI23f );
	printf("Output of XI73: %d\n",*XR13f );
	*/

	printf ("\n\n\n");

	return 0;
}

void Stage_1_IFFTopration()
{
	printf ("\n-----STAGE 1 Tx OPERATION-----\n");

	op = 0;			//opcode for hardware to execute addition operation
	*opkodA = op;					//process input
	*data1A = *XR0;
	*data2A = *XR4;
	*XR01 = *dataoutA;		//0th output
	printf("\n\nXR01 = XR0 + XR4 = %d + %d = %d\n",*XR0,*XR4,*XR01); // XR01 = XR0 + XR4
```

```
		*opkodA = op;
		*data1A = *XR1;
		*data2A = *XR5;
		*XR11 = *dataoutA;		//1st output real
		printf("XR11 =  XR1 + XR5 = %d + %d = %d\n",*XR1,*XR5,*XR11); // XR11 = XR1 + XR5

		*opkodA = op;
		*data1A = *XR2;
		*data2A = *XR6;
		*XR21 = *dataoutA;		//2nd output real
		printf("XR21 =  XR2 + XR6 = %d + %d = %d\n",*XR2,*XR6,*XR21); // XR21 = XR2 + XR6

		*opkodA = op;
		*data1A = *XR3;
		*data2A = *XR7;
		*XR31 = *dataoutA;		//3rd output real
		printf("XR31 =  XR3 + XR7 = %d + %d = %d\n",*XR3,*XR7,*XR31); // XR31 = XR3 + XR7

		op = 1;
		*opkodA = op;
		*data1A = *XR0;
		*data2A = *XR4;
		*XR41 = *dataoutA;		//4th output real
		printf("XR41 =  XR0 - XR4 = %d - %d = %d\n", *XR0,*XR4,*XR41); // XR41 = XR0 - XR4

		*opkodA = op;
		*data1A = *XR1;
		*data2A = *XR5;
		*XR51t = *dataoutA;		//5th output real
		printf("XR51t =  XR1 - XR5 = %d - %d = %d\n",*XR1,*XR5,*XR51t); // XR51t = XR1 - XR5

		*opkodA = op;
		*data1A = *XR2;
		*data2A = *XR6;
		*XR61t = *dataoutA;		//6th output real
		printf("XR61t =  XR2 - XR6 = %d - %d = %d\n",*XR2,*XR6, *XR61t); // XR61t = XR2 - XR6

		*opkodA = op;
		*data1A = *XR3;
		*data2A = *XR7;
		*XR71t = *dataoutA;		//7th output real
		printf("XR71t =  XR3 - XR7 = %d - %d = %d\n",*XR3,*XR7,*XR71t); // XR71t = XR3 - XR7

}

void Stage_2_IFFTopration()
{
		printf ("\n-----STAGE 2  Tx OPERATION-----\n");

		op = 2;			//opcode for hardware to execute addition operation

		*opkodA = op;				//process input
		*data1A = *XR51t;
		//*data2 = *XR4f;
		*XR51 = *dataoutA;

		printf("\n\nXR51 = 0.7071xXR51t = 0.7071x%d = %d\n",*XR51t,*XR51); // XR51 = 0.7071*XR51t

		*opkodA = op;				//process input
		*data1A = *XR51t;
		//*data2 = *XR4f;
		*XI51 = *dataoutA;
		printf("XI51 = 0.7071xXR51t = 0.7071x%d = %d\n",*XR51t,*XI51); // XI51 = 0.7071*XI51t

		*XI61 = *XR61t;
		printf("XI61 = XR61t = %d = %d\n",*XR61t,*XI61); // XI61 = XR61t

		*opkodA = op;				//process input
		*data1A = *XR71t;
		//*data2 = *XR4f;
		*XR71t2 = *dataoutA;
		printf("\nXR71t2 = 0.7071xXR71t = 0.7071x%d = %d\n",*XR71t,*XR71t2); // XR71t2 = 0.7071*XI71t
```

```
		op = 2;
		*opkodA = op;				//process input
		*data1A = *XR71t;
		//*data2 = *XR4f;
		*XI71 = *dataoutA;
		printf("XI71 = 0.7071xXR71t = 0.7071x%d = %d\n",*XR71t,*XI71); // XR71t2 = 0.7071*XI71t

}
void Stage_2a_IFFTopration()
{
		printf ("\n-----STAGE 2a  Tx OPERATION-----\n");
		op = 5;
		*opkodA = op;				//process input
		*data1A = *XR71t2;
		*XR71 = *dataoutA;
		printf("\nXR71 = -1xXR71t2 = -1x%d = %d\n",*XR71t2,*XR71); // XI71 = -1*XI71t2
}
void Stage_3_IFFTopration()
{
		printf ("\n-----STAGE 3  Tx OPERATION-----\n");

		op = 0;
		*opkodA = op;				//process input
		*data1A = *XR01;
		*data2A = *XR21;
		*XR02 = *dataoutA;
		printf("\nXR02 = XR01 + XR21 = %d + %d = %d\n",*XR01,*XR21,*XR02); // XR02 = XR01+XR21

		*opkodA = op;				//process input
		*data1A = *XR11;
		*data2A = *XR31;
		*XR12 = *dataoutA;
		printf("XR12 = XR11 + XR31 = %d + %d = %d\n",*XR11,*XR31,*XR12); // XR12 = XR11+XR31

		op = 1;
		*opkodA = op;				//process input
		*data1A = *XR01;
		*data2A = *XR21;
		*XR22 = *dataoutA;
		printf("XR22 = XR01 - XR21 = %d - %d = %d\n",*XR01,*XR21,*XR22); // XR22 = XR01-XR21

		*opkodA = op;				//process input
		*data1A = *XR11;
		*data2A = *XR31;
		*XI32 = *dataoutA;
		printf("XI32 = XR11 - XR31 = %d - %d = %d\n",*XR11,*XR31,*XI32); // XI32 = XR11-XR31 --XI32 =betulkan

		*XR42 = *XR41;
		printf("XR42 = XR41 = %d = %d\n",*XR41,*XR42);

		*XI42 = *XI61;
		printf("XI42 = XI61 = %d = %d\n",*XI61,*XI42);

		op = 0;
		*opkodA = op;				//process input
		*data1A = *XR51;
		*data2A = *XR71;
		*XR52 = *dataoutA;
		printf("XR52 = XR51 + XR71 = %d + %d = %d\n",*XR51,*XR71,*XR52); // XR52 = XR51+XR71

		*opkodA = op;				//process input
		*data1A = *XI51;
		*data2A = *XI71;
		*XI52 = *dataoutA;
		printf("XI52 = XI51 + XI71 = %d + %d = %d\n",*XI51,*XI71,*XI52); // XI52 = XI51+XI71

		*XR62 = *XR41;
		printf("XR62 = XR41 = %d = %d\n",*XR41,*XR62);

		op = 5;
		*opkodA = op;				//process input
		*data1A = *XI61;
		//*data2 = *XI71;
		*XI62 = *dataoutA;
		printf("XI62 = -1xXI61 = -1x%d = %d\n",*XI61,*XI62); // XI52 = -1xXI62
```

```
		op = 1;
		*opkodA = op;				//process input
		*data1A = *XI51;
		*data2A = *XI71;
		*XR72 = *dataoutA;
		printf("XR72 = XI51 - XI71 = %d - %d = %d\n",*XI51,*XI71,*XR72); // XR72 = XI51-XI71

		*opkodA = op;				//process input
		*data1A = *XR51;
		*data2A = *XR71;
		*XI72 = *dataoutA;
		printf("XI72 = XR51 - XR71 = %d - %d = %d\n",*XR51,*XR71,*XI72); // XI72 = XR51-XR71

}

void Stage_4_IFFTopration()
{
		printf ("\n-----STAGE 4  Tx  OPERATION-----\n");
		op = 0;
		*opkodA = op;				//process input
		*data1A = *XR02;
		*data2A = *XR12;
		*XR03 = *dataoutA;
		printf("\nXR03 = XR02 + XR12 = %d + %d = %d\n",*XR02,*XR12,*XR03); // XR03 = XR02+XR12

		*opkodA = op;				//process input
		*data1A = *XR42;
		*data2A = *XR52;
		*XR13 = *dataoutA;
		printf("XR13 = XR42 + XR52 = %d + %d = %d\n",*XR42,*XR52,*XR13); // XR13 = XR42+XR52

		//op = 1; //subtraction operation
		*opkodA = op;				//process input
		*data1A = *XI42;
		*data2A = *XI52;
		*XI13 = *dataoutA;
		printf("XI13 = XI42 + XI52 = %d + %d = %d\n",*XI42,*XI52,*XI13); // XI13 = XI42+XI52

		*XR23 = *XR22;
		printf("XR23 = XR22 = %d = %d\n",*XR22,*XR23);

		*XI23 = *XI32;
		printf("XI23 = XI32 = %d = %d\n",*XI32,*XI23);

		op = 0;
		*opkodA = op;				//process input
		*data1A = *XR62;
		*data2A = *XR72;
		*XR33 = *dataoutA;
		printf("XR33 = XR62 + XR72 = %d + %d = %d\n",*XR62,*XR72,*XR33); // XR33 = XR62+XR72

		*opkodA = op;				//process input
		*data1A = *XI62;
		*data2A = *XI72;
		*XI33 = *dataoutA;
		printf("XI33 = XI62 + XI72 = %d + %d = %d\n",*XI62,*XI72,*XI33); // XI33 = XI62+XI72

		op = 1;
		*opkodA = op;				//process input
		*data1A = *XR02;
		*data2A = *XR12;
		*XR43 = *dataoutA;
		printf("XR43 = XR02 + XR12 = %d - %d = %d\n",*XR02,*XR12,*XR43); // XR43 = XR02-XR12

		op = 1;
		*opkodA = op;				//process input
		*data1A = *XR42;
		*data2A = *XR52;
		*XR53 = *dataoutA;
		printf("XR53 = XR42 + XR52 = %d - %d = %d\n",*XR42,*XR52,*XR53); // XR53 = XR42-XR52

		*opkodA = op;				//process input
		*data1A = *XI42;
		*data2A = *XI52;
		*XI53 = *dataoutA;
		printf("XI53 = XI42 + XI52 = %d - %d = %d\n",*XI42,*XI52,*XI53); // XI53 = XI42-XI52
```

```
	*XR63 = *XR22;
	printf("XR63 = XR22 = %d = %d\n",*XR22,*XR63);

	op = 5;
	*opkodA = op;				//process input
	*data1A = *XI32;
	//*data2 = *XI71;
	*XI63 = *dataoutA;
	printf("XI63 = -1xXI32 = -1x%d = %d\n",*XI32,*XI63); // XI63 = -1xXI32

	op = 1;
	*opkodA = op;				//process input
	*data1A = *XR62;
	*data2A = *XR72;
	*XR73 = *dataoutA;
	printf("XR73 = XR62 - XR72 = %d - %d = %d\n",*XR62,*XR72,*XR73); // XR53 = XR42-XR52

	*opkodA = op;				//process input
	*data1A = *XI62;
	*data2A = *XI72;
	*XI73 = *dataoutA;
	printf("XI73 = XI62 - XI72 = %d - %d = %d\n",*XI62,*XI72,*XI73); // XI73 = XI62-XI72


}

void Stage_5_IFFTopration()
{
	printf ("\n-----STAGE 5  Tx OPERATION-----\n");
	//printf ("\nFinal output of the IFFT\n");
	op = 3;				//opcode for hardware to execute addition operation
	*opkodA = op;				//process input
	*data1A = *XR03;
	//*data2 = *XR21f;
	*XR0o = *dataoutA;
	printf("\nXR0o = XR03/8 = %d/8 = %d\n",*XR03,*XR0o);

	*opkodA = op;				//process input
	*data1A = *XR13;
	//*data2 = *XR31f;
	*XR1o = *dataoutA;
	printf("XR1o = XR13/8 = %d/8 = %d\n",*XR13,*XR1o);

	*opkodA = op;				//process input
	*data1A = *XI13;
	//*data2 = *XR31f;
	*XI1o = *dataoutA;
	printf("XI1o = XI13/8 = %d/8 = %d\n",*XI13,*XI1o);

	*opkodA = op;				//process input
	*data1A = *XR23;
	//*data2 = *XR31f;
	*XR2o = *dataoutA;
	printf("XR2o = XR23/8 = %d/8 = %d\n",*XR23,*XR2o);

	*opkodA = op;				//process input
	*data1A = *XI23;
	//*data2 = *XR31f;
	*XI2o = *dataoutA;
	printf("XI2o = XI23/8 = %d/8 = %d\n",*XI23,*XI2o);

	*opkodA = op;				//process input
	*data1A = *XR33;
	//*data2 = *XR31f;
	*XR3o = *dataoutA;
	printf("XR3o = XR33/8 = %d/8 = %d\n",*XR33,*XR3o);

	*opkodA = op;				//process input
	*data1A = *XI33;
	//*data2 = *XR31f;
	*XI3o = *dataoutA;
	printf("XI3o = XI33/8 = %d/8 = %d\n",*XI33,*XI3o);

	*opkodA = op;				//process input
```

```
		*data1A = *XR43;
		//*data2 = *XR31f;
		*XR4o = *dataoutA;
		printf("XR4o = XR43/8 = %d/8 = %d\n",*XR43,*XR4o);

		*opkodA = op;					//process input
		*data1A = *XR53;
		//*data2 = *XR31f;
		*XR5o = *dataoutA;
		printf("XR5o = XR53/8 = %d/8 = %d\n",*XR53,*XR5o);

		*opkodA = op;					//process input
		*data1A = *XI53;
		//*data2 = *XR31f;
		*XI5o = *dataoutA;
		printf("XI5o = XI53/8 = %d/8 = %d\n",*XI53,*XI5o);

		*opkodA = op;					//process input
		*data1A = *XR63;
		//*data2 = *XR31f;
		*XR6o = *dataoutA;
		printf("XR6o = XR63/8 = %d/8 = %d\n",*XR63,*XR6o);

		*opkodA = op;					//process input
		*data1A = *XI63;
		//*data2 = *XR31f;
		*XI6o = *dataoutA;
		printf("XI6o = XI63/8 = %d/8 = %d\n",*XI63,*XI6o);

		*opkodA = op;					//process input
		*data1A = *XR73;
		//*data2 = *XR31f;
		*XR7o = *dataoutA;
		printf("XR7o = XR73/8 = %d/8 = %d\n",*XR73,*XR7o);

		*opkodA = op;					//process input
		*data1A = *XI73;
		//*data2 = *XR31f;
		*XI7o = *dataoutA;
		printf("XI7o = XI73/8 = %d/8 = %d\n",*XI73,*XI7o);

}

////////////////// -------PROGRAM FOR FFT COMPUTATION---------/////////////

void Stage_1_FFTopration()
{
		printf ("\n-----STAGE 1 OF IFFT OPERATION-----\n");

		op = 0;			//opcode for hardware to execute addition operation

		*opkodB = op;					//process input
		*data1B = *XR0f;
		*data2B = *XR4f;
		*XR01f = *dataoutB;		//0th output
		printf("\n\nXR01 = %d + %d = %d\n",*XR0f,*XR4f,*XR01f); // XR01 = XR0 + XR4

		*opkodB = op;					//process input
		*data1B = *XI0f;
		*data2B = *XI4f;
		*XI01f = *dataoutB;		//0th output
		printf("XI01 = %d + %d = %d\n",*XI0f,*XI4f,*XI01f); // XI01 = XI0 + XI4

		*opkodB = op;
		*data1B = *XR1f;
		*data2B = *XR5f;
		*XR11f = *dataoutB;		//1st output real
		printf("XR11 = %d + %d = %d\n",*XR1f,*XR5f,*XR11f); // XR11 = XR1 + XR5

		*opkodB = op;
		*data1B = *XI1f;
		*data2B = *XI5f;
		*XI11f = *dataoutB;		//1st output imag
		printf("XI11 = %d + %d = %d\n",*XI1f,*XI5f,*XI11f); // XI11 = XI1 + XI5

		*opkodB = op;
```

```
		*data1B = *XR2f;
		*data2B = *XR6f;
		*XR21f = *dataoutB;		//2nd output real
		printf("XR21 = %d + %d = %d\n",*XR2f,*XR6f,*XR21f); // XR21 = XR2 + XR6

		*opkodB = op;
		*data1B = *XI2f;
		*data2B = *XI6f;
		*XI21f = *dataoutB;		//2nd output imag
		printf("XI21 = %d + %d = %d\n",*XI2f,*XI6f,*XI21f); // XI11 = XI2 + XI5

		*opkodB = op;
		*data1B = *XR3f;
		*data2B = *XR7f;
		*XR31f = *dataoutB;		//3rd output real
		printf("XR31 = %d + %d = %d\n",*XR3f,*XR7f,*XR31f); // XR31 = XR3 + XR7

		*opkodB = op;
		*data1B = *XI3f;
		*data2B = *XI7f;
		*XI31f = *dataoutB;		//3rd output imag
		printf("XI31 = %d + %d = %d\n",*XI3f,*XI7f,*XI31f); // XI31 = XI3 + XI7


		op = 1;
		*opkodB = op;
		*data1B = *XR0f;
		*data2B = *XR4f;
		*XR41f = *dataoutB;
		printf("XR41 = %d - %d = %d\n", *XR0f,*XR4f,*XR41f); // XR41 = XR0 - XR4

		*opkodB = op;
		*data1B = *XI0f;
		*data2B = *XI4f;
		*XI41f = *dataoutB;
		printf("XI41 = %d - %d = %d\n", *XI0f,*XI4f,*XI41f); // XI41 = XI0 - XI4

		*opkodB = op;
		*data1B = *XR1f;
		*data2B = *XR5f;
		*XR51tf = *dataoutB;
		printf("XR51t = %d - %d = %d\n",*XR1f,*XR5f,*XR51tf); // XR51 = XR1 - XR5

		*opkodB = op;
		*data1B = *XI1f;
		*data2B = *XI5f;
		*XI51tf = *dataoutB;		//5th output imag
		printf("XI51t = %d - %d = %d\n",*XI1f,*XI5f, *XI51tf); // XI51t = XI1 - XI5

		*opkodB = op;
		*data1B = *XR2f;
		*data2B = *XR6f;
		*XR61tf = *dataoutB;		//6th output real
		printf("XR61t = %d - %d = %d\n",*XR2f,*XR6f, *XR61tf); // XR61t = XR2 - XR6

		*opkodB = op;
		*data1B = *XI2f;
		*data2B = *XI6f;
		*XI61tf = *dataoutB;		//6th output imag
		printf("XI61t = %d - %d = %d\n",*XI2f,*XI6f,*XI61tf); // XI61t = XI2 - XI6

		*opkodB = op;
		*data1B = *XR3f;
		*data2B = *XR7f;
		*XR71tf = *dataoutB;		//7th output real
		printf("XR71t = %d - %d = %d\n",*XR3f,*XR7f,*XR71tf); // XR71t = XR3 - XR7

		*opkodB = op;
		*data1B = *XI3f;
		*data2B = *XI7f;
		*XI71tf = *dataoutB;		//7th output imag
		printf("XI71t = %d - %d = %d\n\n",*XI3f,*XI7f,*XI71tf);  // XI71t = XI3 - XI7

}

void Stage_2_FFTopration()
```

```
{
	printf ("\n-----STAGE 2 OF FFT OPERATION-----\n");

	op = 2;			//opcode for hardware to execute addition operation

	*opkodB = op;					//process input
	*data1B = *XR51tf;
	//*data2 = *XR4f;
	*XR51taf = *dataoutB;
	printf("\nXR51ta = 0.7071x%d = %d\n",*XR51tf,*XR51taf);

	*opkodB = op;					//process input
	*data1B = *XR51tf;
	//*data2 = *XR4f;
	*XI51taf = *dataoutB;
	printf("XI51ta = 0.7071x%d = %d\n",*XR51tf,*XI51taf);

	*opkodB = op;					//process input
	*data1B = *XI51tf;
	//*data2 = *XR4f;
	*XR51tbf = *dataoutB;
	printf("\nXR51tb = 0.7071x%d = %d\n",*XI51tf,*XR51tbf);

	*opkodB = op;					//process input
	*data1B = *XI51tf;
	//*data2 = *XR4f;
	*XI51tbf = *dataoutB;
	printf("XI51tb = 0.7071x%d = %d\n",*XI51tf,*XI51tbf);

	op = 6;
	*opkodB = op;					//process input
	*data1B = *XR61tf;
	//*data2 = *XR4f;
	*XI61f = *dataoutB;		//6th output
	printf("\nXI61 = -1x%d = %d\n",*XR61tf,*XI61f); // XI61 = -1*XR61

	*opkodB = op;					//process input
	*data1B = *XI61tf;
	//*data2 = *XR4f;
	*XR61f = *dataoutB;		//6th output
	printf("\nXR61 = -1x%d = %d\n",*XI61tf,*XR61f); // XR61 = -1*XI61


	op = 2;
	*opkodB = op;					//process input
	*data1B = *XR71tf;
	//*data2 = *XR4f;
	*XR71taf = *dataoutB;
	printf("\nXR71ta = 0.7071x%d = %d\n",*XR71tf,*XR71taf);

	*opkodB = op;					//process input
	*data1B = *XR71tf;
	//*data2 = *XR4f;
	*XI71taf = *dataoutB;
	printf("\nXI71ta = 0.7071x%d = %d\n",*XR71tf,*XI71taf);

	*opkodB = op;					//process input
	*data1B = *XI71tf;
	//*data2 = *XR4f;
	*XR71tbf = *dataoutB;
	printf("\nXR71tb = 0.7071x%d = %d\n",*XI71tf,*XR71tbf);

	*opkodB = op;					//process input
	*data1B = *XI71tf;
	//*data2 = *XR4f;
	*XI71tbf = *dataoutB;
	printf("\nXR71tb = 0.7071x%d = %d\n",*XI71tf,*XI71tbf);



}

void Stage_3_FFTopration()
{
	printf ("\n-----STAGE 3 OF FFT OPERATION-----\n");
```

```
	op = 6;				//opcode for hardware to execute addition operation
	*opkodB = op;				//process input
	*data1B = *XI51taf;
	//*data2 = *XR4f;
	*XI51taaf = *dataoutB;
	printf("\nXI51taa = -1x%d = %d\n",*XI51taf,*XI51taaf); //

	op = 6;				//opcode for hardware to execute addition operation
	*opkodB = op;				//process input
	*data1B = *XI51tbf;
	//*data2 = *XR4f;
	*XI51tbbf = *dataoutB;
	printf("\nXI51tbb = -1x%d = %d\n",*XI51tbf,*XI51tbbf); //

	*opkodB = op;				//process input
	*data1B = *XI71taf;
	//*data2 = *XR4f;
	*XI71taaf = *dataoutB;
	printf("\nXI71taa = -1x%d = %d\n",*XI71taf,*XI71taaf); //

	*opkodB = op;				//process input
	*data1B = *XR71taf;
	//*data2 = *XR4f;
	*XR71taaf = *dataoutB;
	printf("\nXR71taa = -1x%d = %d\n",*XR71taf,*XR71taaf); //

	*opkodB = op;				//process input
	*data1B = *XI71tbf;
	//*data2 = *XR4f;
	*XI71tbbf = *dataoutB;
	printf("\nXI71tbb = -1x%d = %d\n",*XI71tbf,*XI71tbbf); //

	*opkodB = op;				//process input
	*data1B = *XR71tbf;
	//*data2 = *XR4f;
	*XR71tbbf = *dataoutB;
	printf("\nXR71tbb = -1x%d = %d\n",*XR71tbf,*XR71tbbf); //



}


void Stage_4_FFTopration()
{
	printf ("\n-----STAGE 4 OF FFT OPERATION-----\n");

	op = 0;				//opcode for hardware to execute addition operation
	*opkodB = op;				//process input
	*data1B = *XR51taf;
	*data2B = *XR51tbf;
	*XR51f = *dataoutB;
	printf("\nXR51 = %d + %d = %d\n",*XR51taf,*XR51tbf,*XR51f);

	*opkodB = op;				//process input
	*data1B = *XI51taaf;
	*data2B = *XI51tbbf;
	*XI51f = *dataoutB;
	printf("XI51 = %d + %d = %d\n",*XI51taaf,*XI51tbbf,*XI51f);

	*opkodB = op;				//process input
	*data1B = *XR71taaf;
	*data2B = *XR71tbbf;
	*XR71f = *dataoutB;
	printf("XR71 = %d + %d = %d\n",*XR71taaf,*XR71tbbf,*XR71f);

	*opkodB = op;				//process input
	*data1B = *XI71taaf;
	*data2B = *XI71tbbf;
	*XI71f = *dataoutB;
	printf("XI71 = %d + %d = %d\n\n",*XI71taaf,*XI71tbbf,*XI71f);

}

void Stage_5_FFTopration()
```

```
{
        printf ("\n-----STAGE 5 OF FFT OPERATION-----\n");

        op = 0;                         //opcode for hardware to execute addition operation
        *opkodB = op;                   //process input
        *data1B = *XR01f;
        *data2B = *XR21f;
        *XR02f = *dataoutB;
        printf("\nXR02 = %d + %d = %d\n",*XR01f,*XR21f,*XR02f);

        *opkodB = op;                   //process input
        *data1B = *XI01f;
        *data2B = *XI21f;
        *XI02f = *dataoutB;
        printf("XI02 = %d + %d = %d\n",*XI01f,*XI21f,*XI02f);

        *opkodB = op;                   //process input
        *data1B = *XR11f;
        *data2B = *XR31f;
        *XR12f = *dataoutB;
        printf("XR12 = %d + %d = %d\n",*XR11f,*XR31f,*XR12f);

        *opkodB = op;                   //process input
        *data1B = *XI11f;
        *data2B = *XI31f;
        *XI12f = *dataoutB;
        printf("XI12 = %d + %d = %d\n",*XI11f,*XI31f,*XI12f);

        *opkodB = op;                   //process input
        *data1B = *XR41f;
        *data2B = *XR61f;
        *XR42f = *dataoutB;
        printf("XR42 = %d + %d = %d\n",*XR41f,*XR61f,*XR42f);

        *opkodB = op;                   //process input
        *data1B = *XI41f;
        *data2B = *XI61f;
        *XI42f = *dataoutB;
        printf("XI42 = %d + %d = %d\n",*XI41f,*XI61f,*XI42f);

        *opkodB = op;                   //process input
        *data1B = *XR51f;
        *data2B = *XR71f;
        *XR52f = *dataoutB;
        printf("XR52 = %d + %d = %d\n",*XR51f,*XR71f,*XR52f);

        *opkodB = op;                   //process input
        *data1B = *XI51f;
        *data2B = *XI71f;
        *XI52f = *dataoutB;
        printf("XI52 = %d + %d = %d\n",*XI51f,*XI71f,*XI52f);

        op = 01;                        //subtraction
        *opkodB = op;
        *data1B = *XR01f;
        *data2B = *XR21f;
        *XR22f = *dataoutB;
        printf("XR22 = %d - %d = %d\n",*XR01f,*XR21f,*XR22f);

        *opkodB = op;                   //process input
        *data1B = *XI01f;
        *data2B = *XI21f;
        *XI22f = *dataoutB;
        printf("XI22 = %d - %d = %d\n",*XI01f,*XI21f,*XI22f);

        *opkodB = op;
        *data1B = *XR11f;
        *data2B = *XR31f;
        *XR32tf = *dataoutB;
        printf("XR32t = %d - %d = %d\n",*XR11f,*XR31f,*XR32tf);

        *opkodB = op;                   //process input
        *data1B = *XI11f;
        *data2B = *XI31f;
        *XI32tf = *dataoutB;
        printf("XI32t = %d - %d = %d\n",*XI11f,*XI31f,*XI32tf);
```

```
		*opkodB = op;
		*data1B = *XR41f;
		*data2B = *XR61f;
		*XR62f = *dataoutB;
		printf("XR62 = %d - %d = %d\n",*XR41f,*XR61f,*XR62f);

		*opkodB = op;					//process input
		*data1B = *XI41f;
		*data2B = *XI61f;
		*XI62f = *dataoutB;
		printf("XI62 = %d - %d = %d\n",*XI41f,*XI61f,*XI62f);

		*opkodB = op;
		*data1B = *XR51f;
		*data2B = *XR71f;
		*XR72tf = *dataoutB;
		printf("XR72t = %d - %d = %d\n",*XR51f,*XR71f,*XR72tf);

		*opkodB = op;					//process input
		*data1B = *XI51f;
		*data2B = *XI71f;
		*XI72tf = *dataoutB;
		printf("XI72t = %d - %d = %d\n",*XI51f,*XI71f,*XI72tf);


}

void Stage_6_FFTopration()
{
		printf ("\n-----STAGE 6 OF FFT OPERATION-----\n");

		op = 6;				//opcode for hardware to execute addition operation
		*opkodB = op;					//process input
		*data1B = *XR32tf;
		//*data2 = *XR21f;
		*XI32f = *dataoutB;
		printf("\nXI32 = -1x%d = %d\n",*XR32tf,*XI32f);

		*opkodB = op;					//process input
		*data1B = *XI32tf;
		//*data2 = *XR21f;
		*XR32f = *dataoutB;
		printf("XR32 = -1x%d = %d\n",*XI32tf,*XR32f);

		*opkodB = op;					//process input
		*data1B = *XR72tf;
		//*data2 = *XR21f;
		*XI72f = *dataoutB;
		printf("XI72 = -1x%d = %d\n",*XR72tf,*XI72f);

		*opkodB = op;					//process input
		*data1B = *XI72tf;
		//*data2 = *XR21f;
		*XR72f = *dataoutB;
		printf("XR72 = -1x%d = %d\n",*XI72tf,*XR72f);

}

//////////////////////////-----stage 7------///////////////

void Stage_7_FFTopration()
{
		printf ("\n-----STAGE 7 OF FFT OPERATION-----\n");

		op = 0;				//opcode for hardware to execute addition operation
		*opkodB = op;					//process input
		*data1B = *XR02f;
		*data2B = *XR12f;
		*XR03f = *dataoutB;
		printf("\nXR0 = %d + %d = %d\n",*XR02f,*XR12f,*XR03f);

		*opkodB = op;					//process input
		*data1B = *XI02f;
		*data2B = *XI12f;
		*XI03f = *dataoutB;
```

```
printf("XI0 = %d + %d = %d\n",*XI02f,*XI12f,*XI03f);

*opkodB = op;                                   //process input
*data1B = *XR22f;
*data2B = *XR32f;
*XR23f = *dataoutB;
printf("XR23 = %d + %d = %d\n",*XR22f,*XR32f,*XR23f);

*opkodB = op;                                   //process input
*data1B = *XI22f;
*data2B = *XI32f;
*XI23f = *dataoutB;
printf("XI23 = %d + %d = %d\n",*XI22f,*XI32f,*XI23f);

*opkodB = op;                                   //process input
*data1B = *XR42f;
*data2B = *XR52f;
*XR13f = *dataoutB;
printf("XR13 = %d + %d = %d\n",*XR42f,*XR52f,*XR13f);

*opkodB = op;                                   //process input
*data1B = *XI42f;
*data2B = *XI52f;
*XI13f = *dataoutB;
printf("XI13 = %d + %d = %d\n",*XI42f,*XI52f,*XI13f);

*opkodB = op;                                   //process input
*data1B = *XR62f;
*data2B = *XR72f;
*XR73f = *dataoutB;
printf("XR73 = %d + %d = %d\n",*XR62f,*XR72f,*XR73f);

*opkodB = op;                                   //process input
*data1B = *XI62f;
*data2B = *XI72f;
*XI33f = *dataoutB;
printf("XI33 = %d + %d = %d\n",*XI62f,*XI72f,*XI33f);

op = 01;                                //opcode for hardware to execute subtraction operation
*opkodB = op;                                   //process input
*data1B = *XR02f;
*data2B = *XR12f;
*XR43f = *dataoutB;
printf("\nXR43 = %d - %d = %d\n",*XR02f,*XR12f,*XR43f);

*opkodB = op;                                   //process input
*data1B = *XI02f;
*data2B = *XI12f;
*XI43f = *dataoutB;
printf("XI43 = %d - %d = %d\n",*XI02f,*XI12f,*XI43f);

*opkodB = op;                                   //process input
*data1B = *XR22f;
*data2B = *XR32f;
*XR63f = *dataoutB;
printf("XR63 = %d - %d = %d\n",*XR22f,*XR32f,*XR63f);

*opkodB = op;                                   //process input
*data1B = *XI22f;
*data2B = *XI32f;
*XI63f = *dataoutB;
printf("XI63 = %d - %d = %d\n",*XI02f,*XI12f,*XI63f);

*opkodB = op;                                   //process input
*data1B = *XR42f;
*data2B = *XR52f;
*XR53f = *dataoutB;
printf("XR53 = %d - %d = %d\n",*XR42f,*XR52f,*XR53f);

*opkodB = op;                                   //process input
*data1B = *XI42f;
*data2B = *XI52f;
*XI53f = *dataoutB;
printf("XI53 = %d - %d = %d\n",*XI42f,*XI52f,*XI53f);

*opkodB = op;                                   //process input
```

```
*data1B = *XR62f;
*data2B = *XR72f;
*XR33f = *dataoutB;
printf("XR33 = %d - %d = %d\n",*XR62f,*XR72f,*XR33f);

*opkodB = op;                                   //process input
*data1B = *XI62f;
*data2B = *XI72f;
*XI73f = *dataoutB;
printf("XI73 = %d - %d = %d\n",*XI62f,*XI72f,*XI73f);



}
```