DESIGN AND SIMULATION OF 32-POINT FFT BY RADIX-2 ALGORITHM USING XILINX

A Project report submitted in partial fulfillment of the requirements for

the award of the degree of

BACHELOR OF TECHNOLOGY

IN

ELECTRONICS AND COMMUNICATION ENGINEERING

Submitted by

S.S.Ravi Teja (317126512112)

K.Gayatri Sirisha (317126512082)

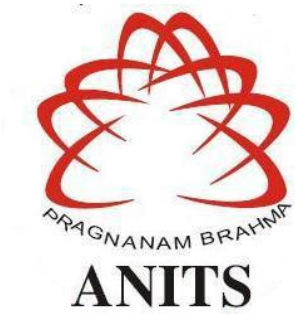
K.Sneha (317126512081)

Pantho.B(317126512096)

Under the guidance of

Mr. N. SRINIVASA NAIDU M.Tech, AMIETE (Ph.D)

Assistant Professor, Department of ECE



DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

ANIL NEERUKONDA INSTITUTE OF TECHNOLOGY AND SCIENCES
(UGC AUTONOMOUS)

(*Permanently Affiliated to AU, Approved by AICTE and Accredited by NBA & NAAC with 'A' Grade*)

Sangivalasa, Bheemili Mandal, Visakhapatnam dist. (A.P)

2020-2021

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

ANIL NEERUKONDA INSTITUTE OF TECHNOLOGY AND SCIENCES
(UGC AUTONOMOUS)

(Permanently Affiliated to AU, Approved by AICTE and Accredited by NBA & NAAC with 'A' Grade)
Sangivalasa, Bheemili Mandal, Visakhapatnam dist. (A.P)

ANITS

CERTIFICATE

This is to certify that the project report entitled **"DESIGN AND SIMULATION OF 32-POINT FFT BY RADIX-2 ALGORITHM USING XILINX"** submitted by **S.S.Ravi Teja (317126512112), K.Gayatri Sirisha (317126512082), K.Sneha (317126512081), Pantho.B (317126512096)** in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology** in **Electronics & Communication Engineering** of Andhra University, Visakhapatnam is a record of bonafide work carried out under my guidance and supervision.

Project Guide

17/7/2021

Mr. N. Srinivasa Naidu M. Tech., AMIETE (Ph.D)
Assistant Professor
Department of ECE, ANITS

Assistant Professor
Department of E.C.E.
Anil Neerukonda
Institute of Technology & Sciences
Sangivalasa, Visakhapatnam-531 162

Head of the Department

Dr. V. Rajya Lakshmi M.E, Ph.D., MHRM,MIEEE,MIMIET
Professor and HOD
Department of ECE, ANITS

Head of the Department
Department of E C E
Anil Neerukonda Institute of Technology & Sciences
Sangivalasa - 531 162

ACKNOWLEDGEMENT

We would like to express our deep gratitude to our project guide **Dr.N.Srinivasa Naidu,** Assistant Professor, M.Tech,AMIETE(Ph.D) Department of Electronics and Communication Engineering, ANITS, for his guidance with unsurpassed knowledge and immense encouragement. We are grateful to **Dr. V. Rajya Lakshmi**, Professor and Head of the Department, Electronics and Communication Engineering, for providing us with the required facilities for the completion of the project work.

We are very much thankful to the **Principal and Management, ANITS, Sangivalasa,** for their encouragement and cooperation to carry out this work.

We express our thanks to all **teaching faculty** of Department of ECE, whose suggestions during reviews helped us in accomplishment of our project. We would like to thank **all non-teaching staff** of the Department of ECE, ANITS for providing great assistance in accomplishment of our project.

We would like to thank our parents, friends, and classmates for their encouragement throughout our project period. Finally we thank everyone for supporting us directly or indirectly in completing this project successfully.

PROJECT STUDENTS

S.S. Ravi Teja (317126512112)

K. Gayatri Sirisha (317126512082)

K. Sneha (317126512081)

Pantho.B (317126512096)

ABSTRACT

Fast Fourier Transform (FFT) is a highly efficient procedure for computing the DFT of a finite series. The FFT is based on decomposition and breaking the transform into smaller transforms and combining them to get the total transform. FFT reduces the computation time required to compute a DFT and improves the performance by a factor 100 or more over direct evaluation of the DFT. DFT converts a finite sequence of equally spaced samples of a function into a same length sequence of equally spaced samples of discrete time Fourier transform which is a complex valued function of frequency. Some of the applications of fast fourier transform include Sound filtering, Image filtering, fast large-integer and polynomial multiplication, solving difference equations, etc. The main advantage of FFT is speed, which it gets by reducing the number of calculations needed to analyze a waveform. This project mainly focuses on the development of Fast Fourier transform, based on decimation in time (DIT), radix-2 algorithm. Decimation in time(DIT) is the sequence for which we need the DFT is successively divided into smaller sequences and the DFTs of these subsequences are combined in a certain pattern to obtain the required DFT of the entire sequence.

The Fast Fourier Transform (FFT) is one of the rudimentary operations in field of digital signal and image processing. The input of Fast Fourier transform has been given by a PS2 KEYBOARD using a test bench and output has been displayed using the waveforms on the Xilinx Design Suite. The synthesis results show that the computation for calculating the 32-point Fast Fourier transform is efficient in terms of speed. Also, FFTs are used for fault analysis, quality control, and condition monitoring of machines or systems in a swift way. These are the basics in wide variety of software i.e. JPG , mp3 ...etc . Yes there are few techniques faster than FFT but it is a basic building block in every processing. And here in this project we will understand the processing of a signal using a very fundamental FFT method.

CONTENTS

LIST OF FIGURES **7**
LIST OF TABLES **9**
CHAPTER 1 INTRODUCTION **10**

1.1 Project Objective 10
1.2 Introduction to DFT 10
1.3 Visualization of DFT 11
1.4 Introduction to FFT 12

CHAPTER 2 FFT AND MATHEMATICAL CALCULATIONS **14**

2.1 Introduction 14
2.2 COOLEY-TUKEY FFT ALGORITHM 15
2.3 Implementation of FFT 16
2.4 32 point FFT 18
2.5 Block diagram of 32 point FFT 19
2.6 Mathematical calculations of 32 point FFT 20
2.7 Advantages of FFT over DFT 22
2.8 Disadvantages of FFT 22
2.9 Applications of FFT 22
2.10 Reliability 23

CHAPTER 3 RADIX-2 BUTTERFLY DIAGRAM **24**

3.1 Butterfly Diagram 24
3.2 Radix-2 Butterfly 25
3.3 Radix-2 26
3.4 Other uses of butterfly diagram 27

CHAPTER 4 INTRODUCTION TO VERILOG **28**

4.1 Introduction 28
4.2 Features of VERILOG HDL 28
4.3 Module Declaration 29

CHAPTER 5 INTRODUCTION TO SOFTWARE TOOLS **32**

5.1 Software tools used 32

5.1.1 MATLAB 32

5.1.2 MODELSIM 33

5.1.3 XILINX VIVADO 35

5.2 XILINX VIVADO Design Suite (2020.2 version) 36

5.3 Project Navigator 37

5.4 Steps for Design entry 41

CHAPTER 6 SIMULATION AND RESULTS **48**

6.1 Simulation results of 32-point FFT 49

6.2 Synthesis Report 54

6.3 MATLAB Results 56

CONCLUSIONS **59**

FUTURE WORK **59**

REFERENCES **60**

LIST OF FIGURES

Figure no	Title	Page no
Fig. 2.1	Basic butterfly computation in the DIT FFT algorithm	16
Fig. 2.2	Block diagram of FFT	17
Fig. 2.3	The five stages of the 32-point DIF FFT architecture	19
Fig. 2.4	Block diagram of 32 point FFT	19
Fig. 2.5	Signal flow graph of a 32 point DIT-FFT with Radix-2	21
Fig. 3.1	Butterfly diagram	24
Fig. 3.2	DIT Radix-2 Algorithm	25
Fig. 3.3	32 point FFT	26
Fig. 5.2	XILINX VIVADO Project navigator window	37
Fig. 5.3	Creating a new project	39
Fig. 5.4	Guiding wizard for the project	39
Fig. 5.5	Creating a new project name	40
Fig. 5.6	Specifying the RTL project	40
Fig. 5.7	Choosing a board for project	41
Fig. 5.8	Project Summary	41
Fig. 5.9	Main window for the project	42
Fig5.11	Adding the source file	43
Fig5.12	wizard that shows to the design source	44
Fig5.13	Creating a new file name for new design source	44
Fig5.14	Selecting a type of file and location	45
Fig5.15	Module defining with ports	46
Fig5.16	Creating the simulation sources	47
Fig. 6.1	Internal architecture of butterfly component	48
Fig. 6.2	RTL analysis of 32 point FFT	55

Fig. 6.3 Magnitude of 32 point FFT 56
Fig. 6.4 Phase of 32 point FFT 56
Fig. 6.5 Signal view in MATLAB 57

LIST OF SYMBOLS

X and its subscritps - Sample points
Y and its subscritps - Sample points
W and its subscripts - Twiddle factor values

LIST OF TABLES

Table no	Title	Page no
Table 6.1	Summary of Virtex Ultrascale + HBM features used in the 32-point FFT	54
Table 6.2	Simulation results of 32-point FFT in MATLAB	58

LIST OF ABBREVATIONS

FFT	Fast Fourier Transform
DFT	Discrete Fourier Transform
ECG	electrocardiogram
EEG	Electroencephalography
IoMT	Internet of Medical Things
HDL	Hardware Description Language

CHAPTER-I
INTRODUCTION

1.1 Project Objective:

This project presents the design of 32-point FFT processing block. Design and implementation of FFT block and results are software simulated. This design computes 32-points FFT and all the numbers follow real and signed type format is used. In this project the coding is done in Verilog. The FPGA synthesis and logic simulation is done using Xilinx Vivado Design Suite 2020.2.

1.2 Introduction of DFT:

As the name implies, the **Discrete Fourier Transform** (DFT) is purely discrete: discrete-time data sets are converted into a discrete-frequency representation. This is in contrast to the DTFT that uses discrete time, but converts to continuous frequency. Since the resulting frequency information is discrete in nature, it is very common for computers to use DFT (Discrete fourier Transform) calculations when frequency information is needed.

Using a series of mathematical tricks and generalizations, there is an algorithm for computing the DFT that is very fast on modern computers. This algorithm is known as the **Fast Fourier Transform** (FFT), and produces the same results as the normal DFT, in a fraction of the computational time as ordinary DFT calculations.

The Discrete Fourier Transform is a numerical variant of the Fourier Transform. Specifically, given a vector of *n* input amplitudes such as $\{f_0, f_1, f_2, \ldots, f_{n-2}, f_{n-1}\}$, the Discrete Fourier Transform yields a set of *n* frequency magnitudes.

The DFT is defined as such:

$$x(k) = \sum_{n=0}^{N-1} x(n)\, e^{-j\left(\frac{2\pi}{N}\right)kn}$$

$$= \sum_{n=0}^{N-1} x(n)\, W_N^{kn} \qquad \left[\because W_N = e^{-j\left(\frac{2\pi}{N}\right)}\right]$$

$$= \sum_{n=0}^{N-1} \{Re[x(n)] + j\, Im[x(n)]\}\{Re[W_N^{kn}] + j\, Im[W_N^{kn}]\}$$

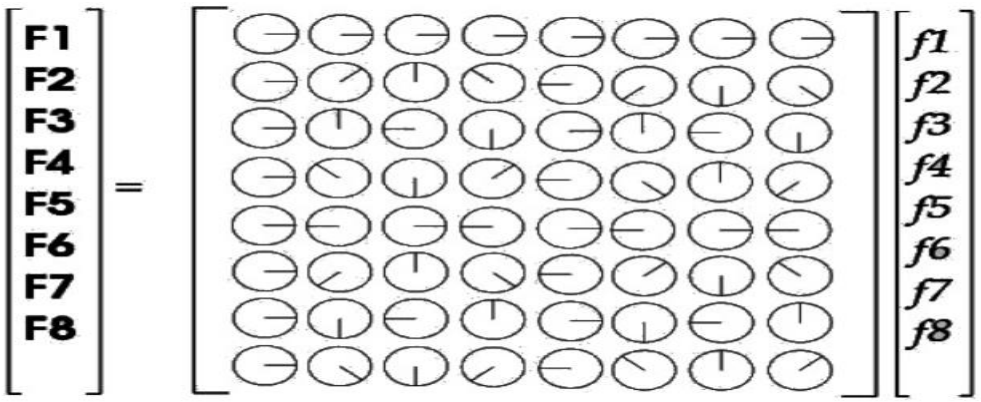
$$= \sum_{n=0}^{N-1} (\{Re[x(n)]\, Re[W_N^{kn}] - Im[x(n)]\, Im[W_N^{kn}]\} + j\{Im[x(n)]\, Re[W_N^{kn}] + Re[x(n)]\, Im[W_N^{kn}]\})$$

1.3 Visualization of Discrete Fourier Transform:

It may be easily seen that the term e^(-πt) represents a unit vector in the complex plane, for any value of j and k. The angle of the vector is initially 0 radians (along the real axis) for j=0 or k=0. As j and k increase, the angle is increased in units of $1/n^{th}$ of a circle. The total angle therefore becomes $2\pi*jk/n$ radians.

To understand the meaning of the Discrete Fourier Transform, it becomes effective to write the transform in matrix form, depicting the complex terms pictorally.

For example, with n=8 , the fourier transform can be written:



The top row (or the left-hand column) changes not at all; therefore, F0 is the resultant when the amplitudes augment each other. The second row changes by $1/n^{th}$ of a circle, going from left to right. The third row changes by $2/n^{th}$ s of a circle; the fourth row changes by $3/n^{th}$ s of a circle; the fifth row, here, changes by $4/n^{th}$ s = 4/8 = 1/2 a circle. Therefore, F4 is the resultant when the even terms are set at odds with the odd terms.

We therefore see from the square matrix that the DFT is the resultant of all 2-D vector combinations of the input amplitudes.

1.4 Introduction to FFT:

FFT are very important mathematical tools for carrying out these tasks. FFT procedure is followed to find DFTs of a complex signal with a less complex multiplication and less addition. FFT is an algorithm that computes the discrete Fourier transform of a sequence. Discrete Fourier Transform (DFT) is used to derive the frequency domain(spectral) representation of the signal. DFT is one of the most powerful tools in digital signal processing as it enables us to find the spectrum of a finite duration signal. The DFT provides a representation of finite duration sequence using a periodic sequence, where one period of this periodic sequence is the same as the finite duration sequence. As a result, we can use discrete Fourier series to derive the DFT equations. Fourier analysis converts a signal from its original domain to a representation in the frequency domain. The FFT operates by decomposing an N-point time domain signal into N time domain signal each composed of a single point. The second step is to calculate the N frequency spectra corresponding to these N time domain signals. Lastly, the N spectra are synthesized into a single frequency spectrum. The output of Fourier transform is a complex number and has a much greater range than the image in the spatial domain.

FFT is an efficient implementation of DFT, and is used, apart from other fields, in digital image processing. Fast Fourier Transform is applied to convert an image from the spatial domain to the frequency domain. Applying filters to images in frequency domain is

computationally faster than to do the same in the image domain. FFT turns the complicated convolution operations into simple multiplications. FFT is the core of the signal and image processing in this modern era. There are a few techniques that are faster than DFT but FFT is the basic building block in each and every processing. The calculation required to find DFT in FFT is very less as compared to formula-based method or by a matrix-based method to find the DFT. The computation efficiency is achieved if we adopt divide and conquer approach. This approach is based on the decomposition of N-point DFT into smaller DFT. This basic approach leads to a family of efficient computational algorithm known as FFT algorithm. So, any software use FFT algorithm to find DFT like MATLAB, octave, etc. FFT also reduces the number of computations needed for a problem of size N from O(N^2) to O(N logN). DFT is widely employed in signal processing and related fields to analyze frequencies contained in a sample signal, to solve partial differential equations, and to perform other operations such as convolutions. Some other applications of FFT are used in digital recording, sampling, additive synthesis, vibration analysis, optics, quantum mechanics, etc.

CHAPTER – 2

FFT AND ITS MATHEMATICAL CALCULATIONS

2.1 Introduction:

The DFT converts a time-domain sequence into an equivalent frequency-domain sequence. The inverse DFT performs the reverse operation and converts a frequency-domain sequence into an equivalent time-domain sequence. The FFT is a very efficient algorithm technique based on the DFT but with fewer computations required. The FFT is one of the most commonly used operations in digital signal processing to provide a frequency spectrum analysis. Two different procedures are introduced to compute an FFT: the decimation-in-frequency and the decimation-in-time. Several variants of the FFT have been used, such as the Winograd transform, the discrete cosine transform (DCT), the discrete Hartley transform and the fast Hartley transform (FHT). Transform methods such as the DCT have become increasingly popular in recent years, especially for real-time systems. They provide a large compression ratio.

Fourier Transform decomposes an image into its real and imaginary components which is a representation of the image in the frequency domain. If the input signal is an image then the number of frequencies in the frequency domain is equal to the number of pixels in the image or spatial domain. The inverse transform re-transforms the frequencies to the image in the spatial domain. One of the most important things to understand about the Discrete Fourier Transform is the tradeoff between time and frequency resolution.

The sampling rate determines the maximum frequency we can represent in our data, which is half the sampling rate, known as the Nyquist rate. But the frequency resolution is largely independent of sample rate. If we transform one second of data, the frequency resolution we get is one Hertz.

For example at 44,100 samples per second, one second of data would require a 44,100 length DFT, whose 44,100 outputs are the amplitude and phase of each frequency in 1 Hertz steps.

In order to get such an accurate frequency resolution we needed a relatively long sample. In the case of something like music, the sound is going to change a fair amount over that one second. If we want to model how the frequency content changes with time, we have

to analyze successive shorter snippets. We might analyze 50 msec long clips. The DFT of those will have a resolution of 1/.050 = 20 Hertz.

In other words, if we're willing to know our time only to the nearest second, we can know our frequency content to a one Hertz resolution. When we focus on a more concentrated time interval of 50 milliseconds, we can only get 20 Hertz resolution.

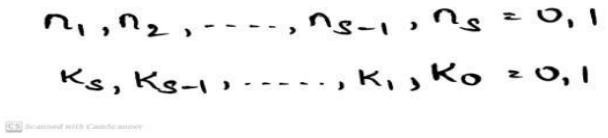
So we decide the length of our FFT based on the sample rate and the frequency resolution we desire. We trade frequency resolution for time resolution by taking multiple FFTs, hopping along a longer signal. That's sometimes called the Short Time Fourier Transform.

Direct DFT calculation requires a computational complexity of O (N^2). By using The Cooley-Turkey FFT algorithm, the complexity can be reduced to O ($N\log_2 N$).

2.2 COOLEY-TUKEY FFT algorithm:

The most popular Cooley-Tukey FFTs are those where the transform length is a power of a basis r, i.e., N = r^S. These algorithms are referred to as radix-r algorithms. The most commonly used are those of basis r = 2 (radix2) and r = 4 (radix-4). Those algorithms and others such as radix-2^i and Split-Radix have been developed based on the basic Cooley-Tukey algorithm to further reduce the computational complexity.

For r = 2 and S stages, for instance, the following index mapping of Cooley-Tukey algorithm gives:



The Cooley-Tukey algorithm is based on a divide and conquer approach in the frequency domain and is therefore referred to as Decimation-In-Frequency (DIF) FFT. The DFT formula is split into two summations.

$$x(k) = \sum_{n=0}^{N-1} x(n)\, W_N^{nk} + \sum_{n=0}^{N-1} x(n)\, W_N^{nk}$$

$$= \sum_{n=0}^{N/2-1} x(2n)\, W_N^{2nk} + \sum_{n=0}^{N/2-1} x(2n+1)\, W_N^{(2n+1)k}$$

$x_e(n) = x(2n) \rightarrow$ even samples

$x_o(n) = x(2n+1) \rightarrow$ odd samples

$$X(k) = \sum_{n=0}^{N/2-1} x_e(n)\, W_N^{2nk} + W_N^k \sum_{n=0}^{N/2-1} x_o(n)\, W_N^{(2n+1)k}$$

$$W_N^2 = \left[e^{-j\left(\frac{2\pi}{N}\right)}\right]^2 = W_{N/2}$$

$$X(k) = \sum_{n=0}^{N/2-1} x_e(n)\, W_{N/2}^{kn} + W_N^k \sum_{n=0}^{N/2-1} x_o(n)\, W_{N/2}^{k}$$

The computational procedure can be repeated through decimation of the N/2-point DFTs X (2k) and DFTs X(2k+1). The entire algorithm involves log2N stages. Where each stage involves N/2 operations units (Butterflies). The computation of the N point DFT via the decimation-in-frequency FFT as in the decimation-in-time algorithm requires (N/2)·log2N complex multiplications and N.log2N complex addition.

2.3 Implementation of FFT:

In general, the other fast algorithms like radix-4, radix-8, radix-2 2 and split-radix (based on the same approach) recursively divide the FFT computation into odd- and even-half parts and then obtain as many common twiddle factors as possible. The number of needed real additions and multiplications is generally used to compare efficiency of different FFT algorithms.

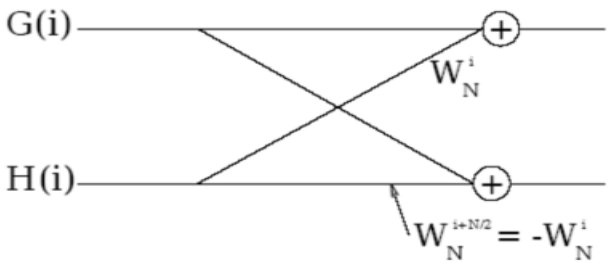


Fig 2.1 : Basic butterfly computation in the DIT FFT algorithm

Since the FFT is a divide and conquer algorithm, the various steps can be implemented in multiple passes in a shader by rendering the result of each pass to a texture. These steps are called **butterflies**, because of the shape of the data-flow diagram. The FFT algorithm recursively breaks down a DFT of composite size N = n1n2 into n1 smaller transforms of size n2. These smaller DFTs are then combined with size-n1 butterflies, which themselves are DFTs of size n1 (performed n2 times on corresponding outputs of the sub-transforms) pre-multiplied by roots of unity (known as twiddle factors).

The application employs four 2D textures - two for the ping-pong operations, one for the source data (for each pass) and the one for holding the indices and weights for performing the butterfly steps. The textures used for ping-ponging are marked as either a RenderTarget texture or a source depending on whether it is used as destination or source in the current pass.This enables the shader to use the output of the previous pass as input in the current pass. Following are the steps in implementing the FFT algorithm:

- compute indices and weights for performing the butterfly operations
- compute log2(Width) horizontal butterflies
- compute log2(Height) vertical butterflies

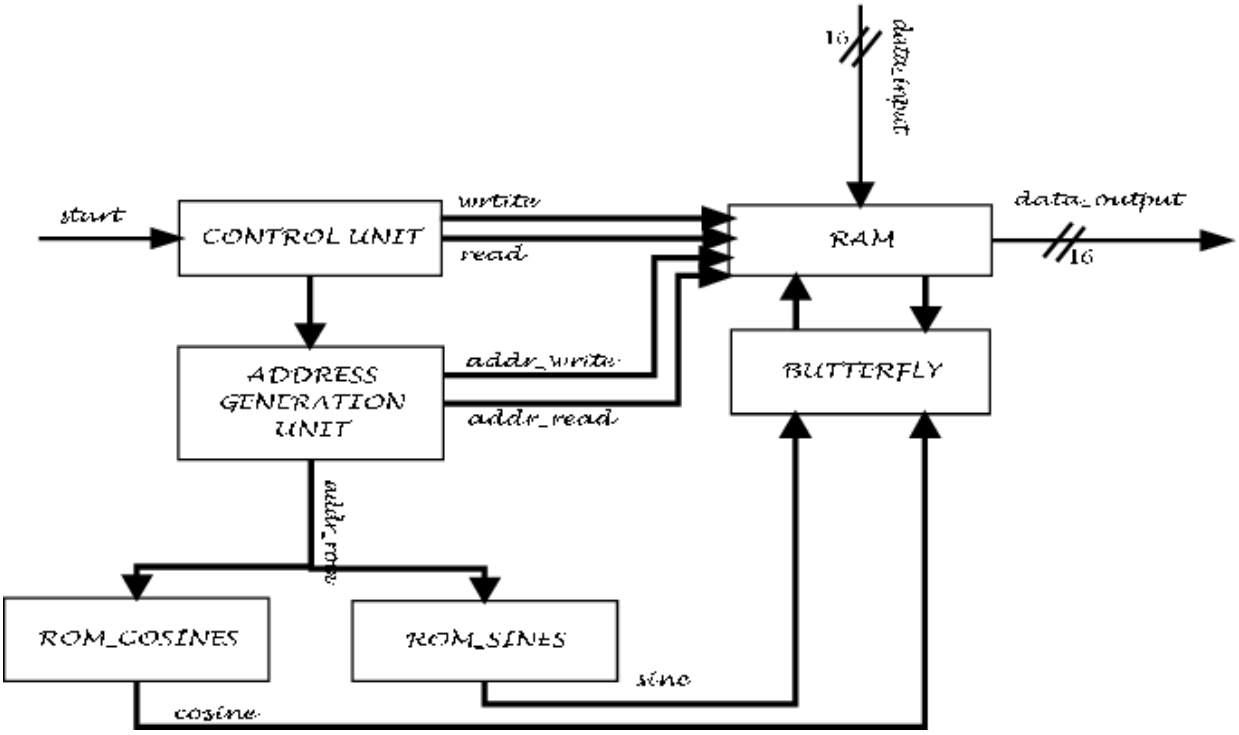


Fig 2.2 : Block diagram of FFT

If the height and width of the image are equal, then only one texture can be used for the butterfly values, for both the horizontal and vertical passes. Also note that in the vertical butterfly pass the input is the result of the horizontal butterfly pass. In each butterfly pass the current pixel is combined with another pixel using a complex multiply and add and written to the current location. In other words if the current pixel is a then,

$$a = a + wb$$

Where w is a complex number representing the weight and b is some other pixel. Each pixel of the butterfly texture contains the locations of a, b and the value of w and passed to the shader by the application. Note that for simplicity, only gray scale images are considered in the current implementation. However, extending the algorithm to multiple color channels is straight forward and will only require more textures for additional channels.

The shader looks-up the butterfly texture for indices of the two pixels to be combined and the weight and computes the result by performing the complex math a + wb or a - wb. To make matters simple, the sign is encoded as part of the weight and hence only the addition is performed in the shader program all the times. The result is returned with the real value in the first three components and the imaginary value in the fourth component.

To transform the image from frequency domain to the spatial domain the exact same operations are performed but on the frequencies. Since the frequencies are not in the range to be displayed on the screen they are scaled by a factor of 1/(Width*Height).

2.4 32-point FFT:

The synthesis of 32-points FFT is divided into eight stages, first stage is used to supply inputs where second stage consists of a finite state machine (FSM), and next five stages are used for calculation of radix-2 DIT FFT through butterfly operations and last stage is used to store the outputs of 32-point FFT i.e., 32-point sequences which are now converted into frequency domain.

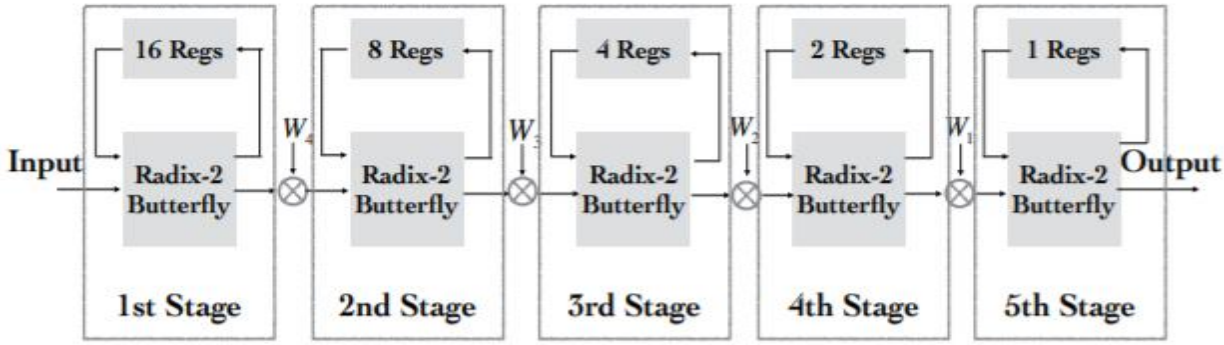


Fig 2.3: The five stages of the 32-point DIF FFT architecture

2.5 Block diagram of 32 point FFT:

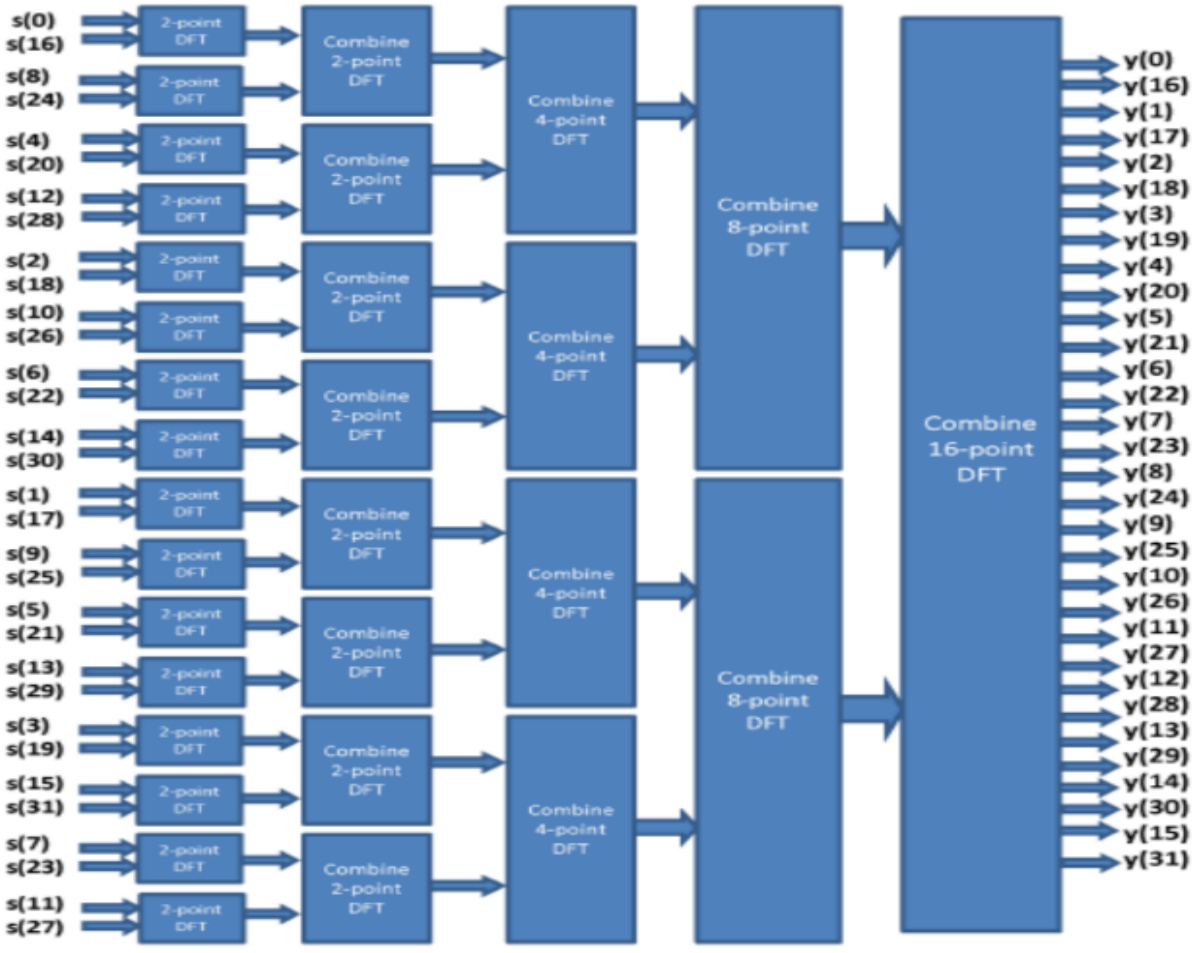


Fig 2.4 : Block diagram of 32 point FFT

FFT is an efficient implementation of DFT, and is used, apart from other fields, in digital image processing. Fast Fourier Transform is applied to convert an image from the spatial domain to the frequency domain. Applying filters to images in frequency domain is computationally faster than to do the same in the image domain. FFT turns the complicated convolution operations into simple multiplications.

2.6 Mathematical calculations of 32 point FFT:

Number of stages:

$\mathbf{M = \log_2(N)} = \log_2(32) = (\log_{10}(1/32))/(\log_{10}(2)) = 5$

Number of butterflies per stage:

$\mathbf{N/2} = 32/2 = 16$

Number of input/output for each butterfly in stage "m" is seperated by:

$\mathbf{2^{m-1}}$ samples.

stage 1 : 1 sample

stage 2 : 2 samples

stage 3 : 4 samples

stage 4 : 8 samples

stage 5 : 16 samples

Number of complex multiplications is given by:

$\mathbf{(N/2)\log_2(N)} = 16*5 = 80$

Number of complex addition is given by:

$\mathbf{N\log_2(N)} = 32*5 = 160$

Number of butterflies in each stage:

$\mathbf{2^{M-m}}$

stage 1 : $2^{5-1} = 16$

stage 2 : $2^{5-2} = 8$

stage 3 : $2^{5-3} = 4$

stage 4 : $2^{5-4} = 2$

stage 5 : $2^{5-5} = 1$

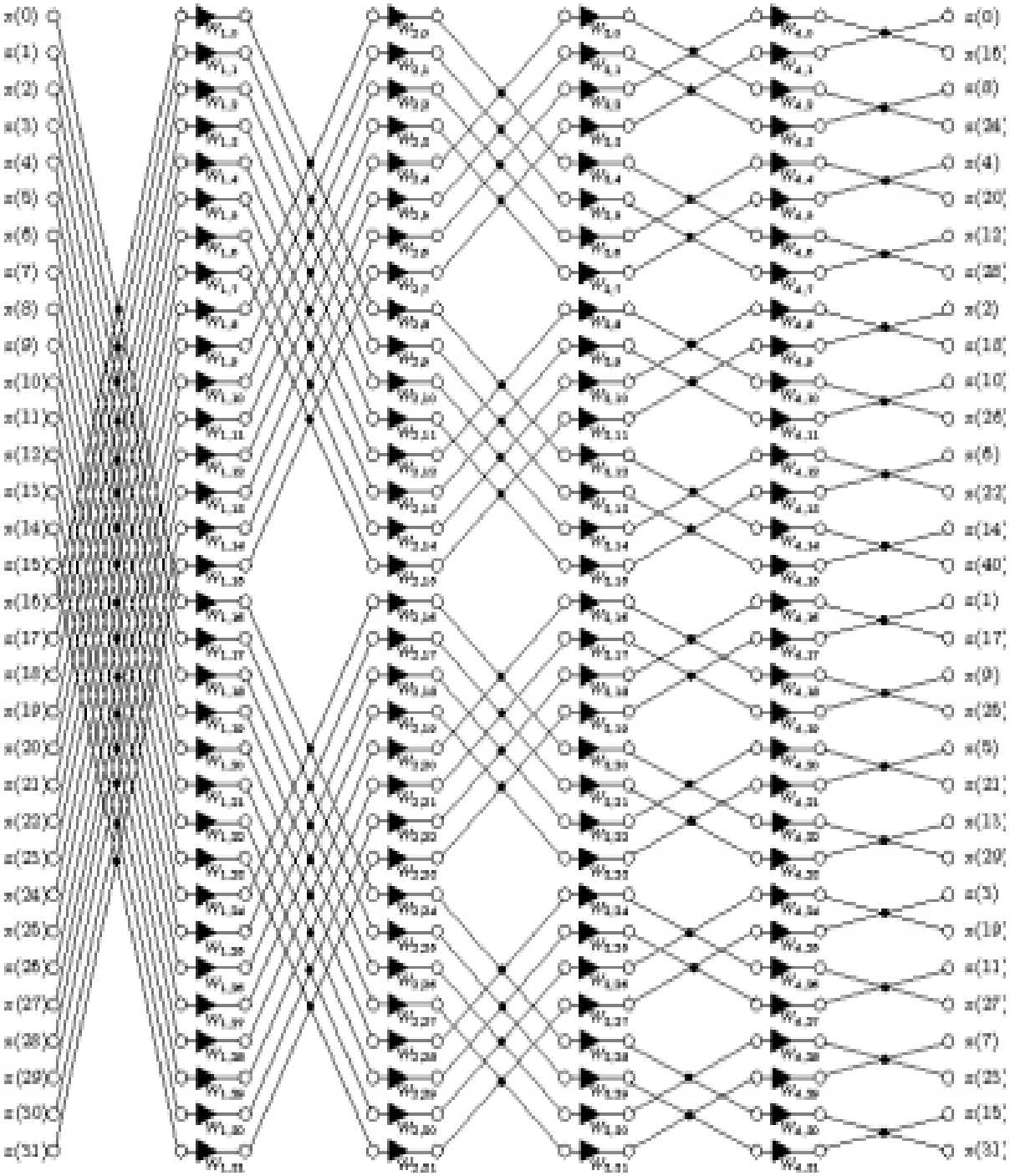


Fig 2.5: Signal flow graph of a 32-point DIT-FFT with Radix-2

2.7 Advantages of FFT over DFT:

FFT helps in converting the time domain in frequency domain which makes the calculations easier as we always deal with various frequency bands in communication system another very big advantage is that it can convert the discrete data into a **continuous data type** available at various frequencies.

FFT is based on divide and conquer algorithm where you divide the signal into two smaller signals, compute the DFT of the two smaller signals and join them to get the DFT of the larger signal. The order of complexity of DFT is O(n^2) while that of FFT is O(n. logn) hence, FFT is faster than DFT.

2.8 Disadvantages of FFT:

A **disadvantage** associated with the **FFT** is the restricted range of waveform data that can be transformed and the need to apply a window weighting function to the waveform to compensate for spectral leakage.

In the case of FFT, if the real frequency doesn't match FFT frequency grid exactly, power spectrum is spreading mostly among neighboring frequency bins (among all bins in general, but most of it is focused in a few adjacent ones).

2.9 Applications of FFT:

- fast large-integer and polynomial multiplication,
- efficient matrix–vector multiplication for Toeplitz, circulant and other structured matrices,
- filtering algorithms
- fast algorithms for discrete cosine or sine transforms (e.g. fast DCT used for JPEG and MPEG/MP3 encoding and decoding),
- fast Chebyshev approximation,
- solving difference equations,
- computation of isotopic distributions.

- modulation and demodulation of complex data symbols using orthogonal frequency division multiplexing (OFDM) for 5G, LTE, Wi-Fi, DSL, and other modern communication systems.

2.10 Reliability:

Fast Fourier transform (FFT)-based computations can be far more accurate than the slow transforms suggest. Discrete Fourier transforms computed through the FFT are far more accurate than slow transforms, and convolutions computed via FFT are far more accurate than the direct results. However, these results depend critically on the accuracy of the FFT software employed, which should generally be considered suspect. Popular recursions for fast computation of the sine/cosine table (or twiddle factors) are inaccurate due to inherent instability. Some analyses of these recursions that have appeared heretofore in print, suggesting stability, are incorrect. Even in higher dimensions, the FFT is remarkably stable.

The FFT is just a fast algorithm for implementing the discrete Fourier transform (DFT), nothing more. Instead, there is an inherent tradeoff in time and frequency resolution due to the Heisenberg uncertainty principle. While its statement is explicitly focused at quantum mechanics, the same underlying principle remains true: the more precisely you know the frequency of a signal, the less able you are to localize it in time.

With that said, there are another class of techniques known as bilinear time-frequency distributions that are appropriate for some applications. One example is the Wigner-Ville distribution. In short, these techniques can provide simultaneously high resolution in time and frequency. The cost, however, is the presence of spurious features in their resulting outputs.

CHAPTER – 3

RADIX 2 BUTTERFLY DIAGRAM

3.1 Butterfly Diagram:

In the context of fast Fourier transform algorithms, a butterfly is a portion of the computation that combines the results of smaller discrete Fourier transforms (DFTs) into a larger DFT, or vice versa (breaking a larger DFT up into sub transforms). The name "butterfly" comes from the shape of the data-flow diagram in the radix-2 case, as described below. The earliest occurrence in print of the term is thought to be in a 1969 MIT technical report. The same structure can also be found in the Viterbi algorithm, used for finding the most likely sequence of hidden states.

Most commonly, the term "butterfly" appears in the context of the Cooley–Tukey FFT algorithm, which recursively breaks down a DFT of composite size $n = rm$ into r smaller transforms of size m where r is the "radix" of the transform. These smaller DFTs are then combined via size-r butterflies, which themselves are DFTs of size r (performed m times on corresponding outputs of the sub-transforms) pre-multiplied by roots of unity (known as twiddle factors). This is the "decimation in time" case; one can also perform the steps in reverse, known as "decimation in frequency", where the butterflies come first and are post-multiplied by twiddle factors.

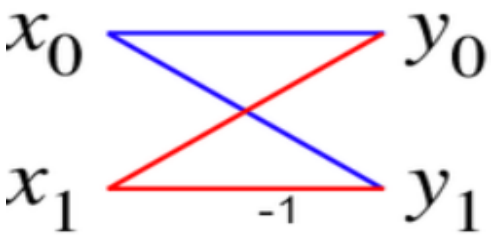


Fig 3.1 : Butterfly diagram

The above diagram is about signal flow graph connecting the inputs x (left) to the outputs y that depend on them (right) for a "butterfly" step of a radix-2 Cooley–Turkey FFT. This diagram resembles a butterfly, hence it is also called as hourglass diagram.

3.2 Radix 2 Butterfly:

In the case of the radix-2 Cooley–Tukey algorithm, the butterfly is simply a DFT of size-2 that takes two inputs (x_0, x_1) (corresponding outputs of the two sub-transforms) and gives two outputs (y_0, y_1) by the formula (not including twiddle factors):

$$y_0 = x_0 + x_1$$
$$y_1 = x_0 - x_1.$$

If one draws the data-flow diagram for this pair of operations, the (x_0, x_1) to (y_0, y_1) lines cross and resemble the wings of a butterfly.

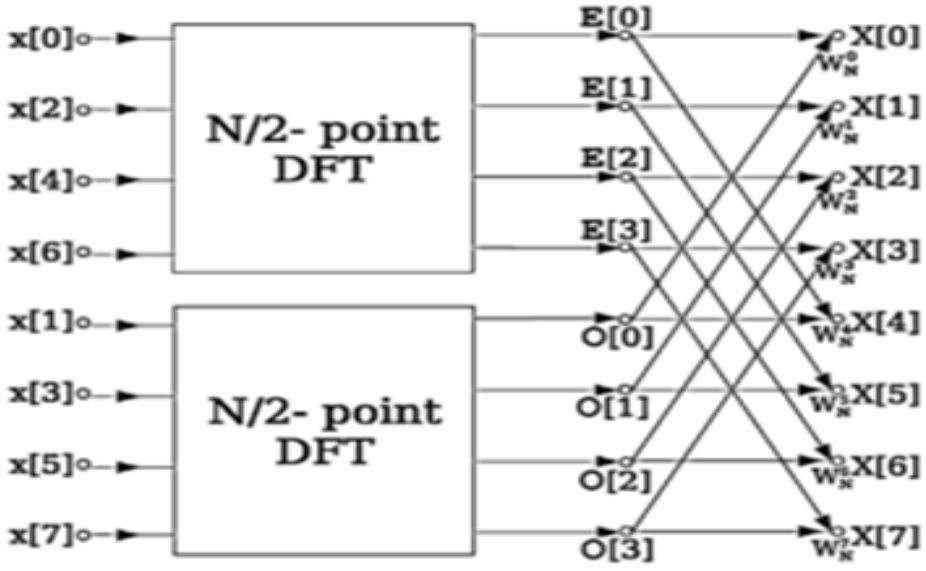


Fig 3.2 : DIT Radix 2 FFT

The above diagram is about decimation-in-time radix-2 FFT breaks a length-N DFT into two length-N/2 DFTs followed by a combining stage consisting of many butterfly operations. More specifically, a radix-2 decimation-in-time FFT algorithm on $n = 2^p$ inputs with respect to a primitive n-th root of unity relies on O ($n \log_2 n$) butterflies of the form:

$$y_0 = x_0 + x_1 \omega_n^k$$
$$y_1 = x_0 - x_1 \omega_n^k,$$

where k is an integer depending on the part of the transform being computed. Whereas the corresponding inverse transform can mathematically be performed by

replacing ω with ω^{-1} (and possibly multiplying by an overall scale factor, depending on the normalization convention), one may also directly invert the butterflies:

$$x_0 = \frac{1}{2}(y_0 + y_1)$$
$$x_1 = \frac{\omega_n^{-k}}{2}(y_0 - y_1),$$

corresponding to a decimation-in-frequency FFT algorithm.

3.3 Radix 2:

The Radix indicates the size of FFT decomposition. In this paper Radix is 2 which is single-Radix FFT. For single Radix FFTs, transform size must be choose according to the power of Radix. Here we use 32and 64 sizes, which is 25 and 26. The Radix-2 Decimation-in-Time FFT (DIT-FFT) is applied to the two Points N/2 DFT's. To find the number of butterfly stages required to compute N length sequence can be M=log2N, and N/2 butterfly operations are computed in each stage. In this paper, there are 5 butterfly stages and 16 butterfly operations are computed to produce 32 Point FFT. Similarly, 6 butterfly stages and 32 butterfly operations are computed to produce 64 Point FFT. Fig 4 and Fig 5 shows the butterfly stages whereas, Fig 4 and Fig 5 shows the butterfly diagram of each and every stage. In DIT-FFT the given input sequence is in shuffled order and the output sequence is in natural order. By using Bit-Reversal input sequence gets shuffled.

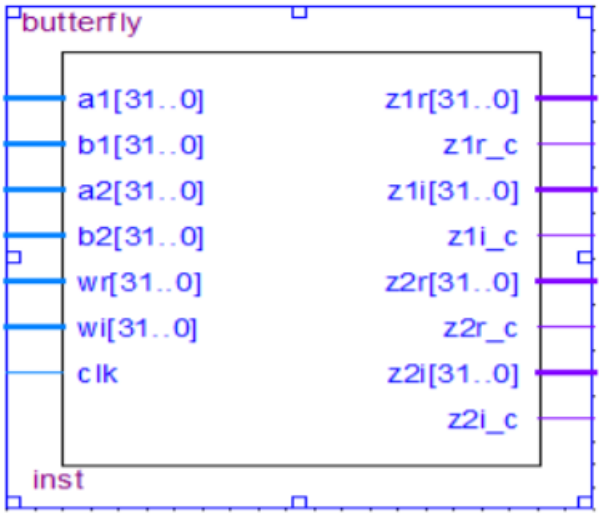


Fig 3.3 : 32 point FFT

3.4 Other uses of butterfly diagram:

The butterfly can also be used to improve the randomness of large arrays of partially random numbers, by bringing every 32 or 64 bit word into causal contact with every other word through a desired hashing algorithm, so that a change in any one bit has the possibility of changing all the bits in the large array.

CHAPTER - 4

INTRODUCTION TO VERILOG

4.1 Introduction:

Verilog, standardized as IEEE 1364, is a hardware description language (HDL) used to model electronic systems. It is most commonly used in design and verification of digital circuits at the regular -transfer level of abstraction. It is also used in verification of analog circuits and mixed signal circuits HDLs allows the design to be simulated earlier in the design circuits in order to correct errors or experiments with different architectures.

Designs described in HDL are technology independent, easy to design and debug, and are usually more readable than schematics, particularly for large circuits. Verilog can be used to describe designs at four levels if abstractions:

- Algorithmic level (much like as code if, case and loop statements).
- Register transfer level (RTL uses registers connected by Boolean equations)
- Gate level (interconnected AND, NOR etc.).
- Switch level (the switches are MOS transistors inside gates).

A Verilog design consists of a hierarchy of modules. Modules encapsulate design hierarchy, and communicate with other modules through a set of declared input, output and bidirectional ports. Internally, a module can contain any combination of the following: net/variable declarations (wire, reg, integer, etc.), concurrent and sequential statement blocks, and instances of other modules (sub-hierarchies).

4.2 Features of Verilog HDL:

Verilog HDL offers many useful features for hardware design.

- Verilog (verify logic) HDL is general purpose hardware description language that is easy to learn and easy to use. It is similar in syntax to C programming language.
- Verilog HDL allows different levels of abstraction to be mixed in the same model. Thus a designer can define a hardware model in terms of switches, or behavior code.
- Most popular logic synthesis tools support Verilog HDL. This makes it the language of choice for designers.
- All fabrication vendors provide Verilog HDL libraries for post logic synthesis simulation

- Thus designing a chip in Verilog HDL allows the widest choice of vendors.
- The Programming language interface (PLI) is a powerful feature that allows the user to custom C code to interact with the internal data structures of Verilog.

4.3 Module Declaration:

A module is the principal design entity in Verilog. The first line of a module declaration specifies the name and port list (arguments). The next few lines specifies the I/O type (**input**, **output** or **inout**) and width of each port. The default port with is 1 bit. Then the port variables must be declared **wire**, **reg**. The default is **wire**.

Typically inputs are **wire** since their data is latched outside the module. Outputs are type **reg** if their signals were stored inside always or **initial** block

Syntax

module model_ name(port_list);

input[msb:lsb] input_port_list;

output[msb:lsb] output_port_list;

inout[msb:lsb] inout_port_list;

…….statements……

endmodule

Verilog has four levels of modelling:

1. The switch level Modeling.
2. Gate- level Modeling.
3. The Data-Flow level.
4. The behavioural or procedural level.

1.Switch level Modeling

A circuit is defined by explicity showing how to constrct it using transistors like pmos and nmos, predefined modules.

Example

```
module inverter (out,in);
output out;
input in;
Supply0gnd;
Supply1vdd;
nmosx1(out, in, gnd);
pmosx2(out, in, vdd);
endmodule
```

2.Gate level modeling

A circuit is defined by explicitly showing how to correct it using logic gates. Predefined modules, and the connections between them.In this first we think of our circuit as a box or module which is encapsulated from its outer environment, in such a way that its only communication with the outer environment, is through input and output ports. We then set out to describe structure within the module by explicitly describing its gates and sub modules, and how they connect with one another as well as to the module ports.

In other words, structural modelling is used to draw aschematic diagram for the circuit.

As an example, consider the full-adder below.

Example

```
module fulladder (a, b, sum, Cout);
input a, b;
output sum, Cout;
xor x1(a, b, y);
xor x2(a, b, y);
endmodule
```

3.Data-flow modelling

Dataflow modeling uses Boolean expressions and operators. In this we use assign statement.

Example

```
module fulladder (a, b, sum, Cout);
```

input a, b;

output sum, Cout;

assign sum=a^b;

assign Cout=a^b;

endmodule

4.Behavioural modeling

It is higher level of modeling where behaviour of logic is modelled. Verilog behavioural Code is inside procedure blocks, but there is an exception:some behavioural code also exist outside procedure blocks.

There are two types of procedural blocks in Verilog

Initial: initial blocks execute only once at time zero (start execution at time zero)

Always: always blocks loop to execute over and over again; in other words, as other words as the name suggests, it executes always.

An always statement executes repeatedly, it starts and its execution at other 0 ns

Syntax:

always@ (sensitivity list)

Begin

Procedural statements

end

Example

module fulladder (a, b, clk, sum);

input a, b, clk;

output sum;

always@ (posedgeclk)

begin

sum= a+b;

endmodule

CHAPTER-5

INTRODUCTION TO SOFTWARE TOOLS

5.1 SOFTWARE TOOLS USED

5.1.1 MATLAB

Introduction:

MATLAB (Matrix Laboratory) is a programming platform developed by MathWorks, which uses it's proprietary MATLAB programming language. The MATLAB programming language is a matrix-based language which allows matrix manipulations, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs written in other languages, including C, C++, C#, Java, Fortran and Python. It is used in a wide range of application domains from Embedded Systems to AI, mainly to analyze data, develop algorithms, and create models and applications.

Usage of MATLAB software in this project:

DFT plays important role in DSP and used in wide variety of applications like correlation and spectral analysis. Understanding quantization errors in its computation is an important aspect of its design either for s/w or h/w implementation.
A complex DFT of N points has N complex (4N real) mutliplications. In the fixed point implementation of the DFT each mutliplication introduces a quantization error (all the errors are mutually uncorrelated and uncorrelated with input sequence as well).

FFT provides an efficient way to compute DFT. Even though FFT has significantly less number of multiplications the quantization errors do not decreate accordingly. Each butterfly operation (ignoring some multiplications are trivial +/-1) involves one complex (four real) multiplication(s). The quantization errors introduced in each butterfly propagate through N/2 stages.

Analysis of quantization related noise effects in a FFT is a challenge. Since FFT algorithm consists of sequence of stages, it is possible to have different scaling strategies in each stage. The attached model shows how fixed point numeric type and fimath can be changed at each stage of the FFT (implemented as a sub function in the embedded MATLAB script).

To achieve desired numerical behavior of the algorithm for the specific application requirements, each multiplication and sum in the attached model can be tweaked independently. The complex magnitude result of FFT output is plotted and compared to its corresponding behavioral block.

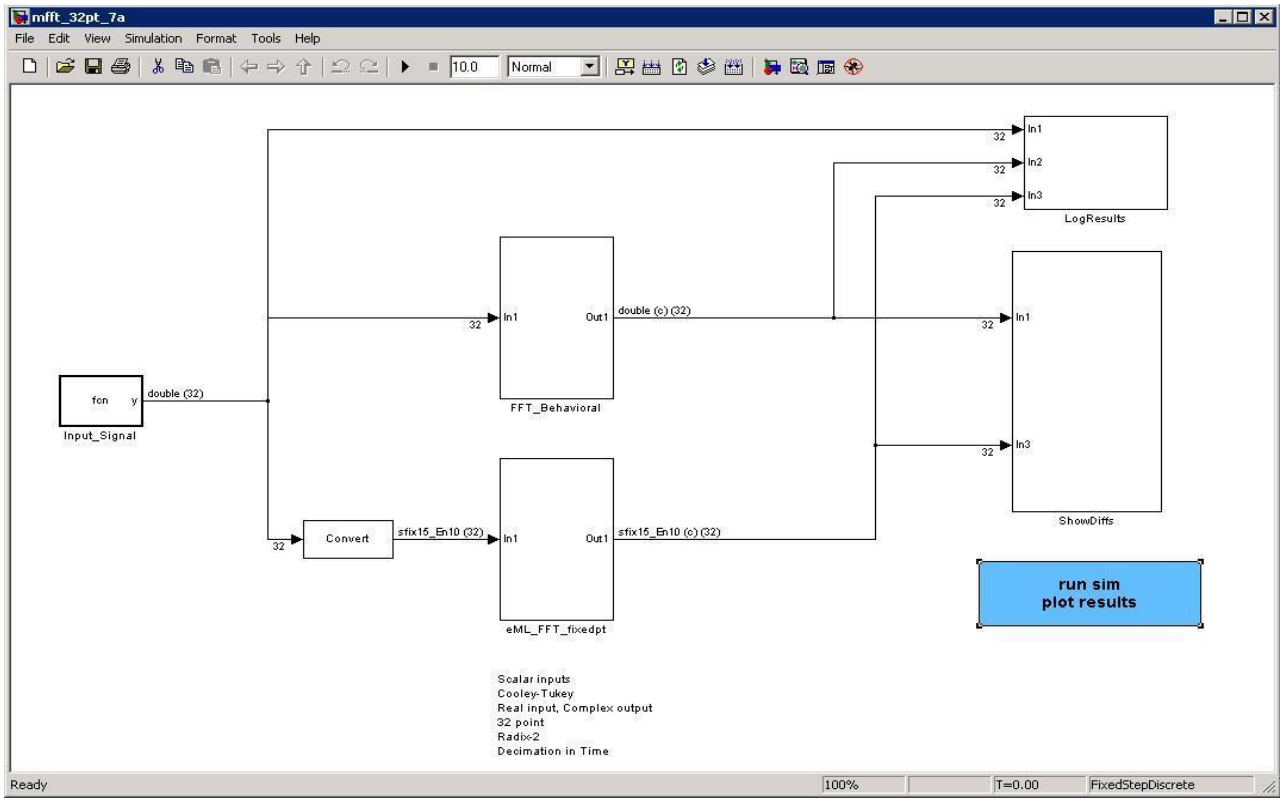


Fig 5.1 : 32 point FFT in MATLAB

5.1.2 MODELSIM

MODELSIM is a multi-language HDL simulation environment by Mentor Graphics, for simulation of hardware description languages such as VHDL, Verilog and SystemC, and includes a built-in C debugger.ModelSim can be used independently, or in conjunction with Intel Quartus Prime, Xilinx ISE or Xilinx Vivado. Simulation is performed using the graphical user interface (GUI), or automatically using scripts.

Language support

ModelSim uses a unified kernel for simulation of all supported languages, and the method of debugging embedded C code is the same as VHDL or Verilog. ModelSim and QuestaSim products enable simulation, verification and debugging for the following languages:

VHDL
Verilog
Verilog 2020
SystemVerilog
PSL

Usage of ModelSim in this project:

We used ModelSim to implement the logic in verilog it is used to check the simulation results and graphs minimize the errors in the verilog code.

Verilog:

Verilog is a Hardware Description Language (HDL). It is a language used for describing a digital system such as a network switch, a microprocessor, a memory, or a flip-flop. We can describe any digital hardware by using HDL at any level. Designs described in HDL are independent of technology, very easy for designing and debugging, and are normally more useful than schematics, particularly for large circuits.

Verilog is a HARDWARE DESCRIPTION LANGUAGE (HDL), which is used to describe a digital system such as a network switch or a microprocessor or a memory a flip-flop. Verilog was developed to simplify the process and make the HDL more robust and

flexible. Today, Verilog is the most popular HDL used and practiced throughout the semiconductor industry.

HDL was developed to enhance the design process by allowing engineers to describe the desired hardware's functionality and let automation tools convert that behavior into actual hardware elements like combinational gates and sequential logic.

Verilog is like any other hardware description language. It permits the designers to design the designs in either Bottom-up or Top-down methodology.

- **Bottom-Up Design:** The traditional method of electronic design is bottom-up. Each design is performed at the gate-level using the standards gates. This design gives a way to design new structural, hierarchical design methods.
- **Top-Down Design:** It allows early testing, easy change of different technologies, and structured system design and offers many other benefits.

Verilog creates a level of abstraction that helps hide away the details of its implementation and technology.

For example, a D flip-flop design would require the knowledge of how the transistors need to be arranged to achieve a positive-edge triggered FF and what the rise, fall, and CLK-Q times required to latch the value onto a flop among much other technology-oriented details.

5.1.3 XILINX VIVADO

Vivado enables developers to synthesize their designs, perform timing analysis examine RTL diagrams, simulate a design's reaction to different stimuli, and configure the target device with the programmer. Vivado is a design environment for FPGA products from Xilinx, and is tightly-coupled to the architecture of such chips, and cannot be used with FPGA products from other vendors.

Language support

The Vivado High-Level Synthesis compiler enables C, C++ and SystemC programs to be directly targeted into Xilinx devices without the need to manually create RTL. Vivado HLS is widely reviewed to increase developer productivity, and is confirmed to support C++ classes, templates, functions and operator overloading.

Xilinx vivado enables simulation, verification and synthesis for the following languages

VHDL Verilog

System Verilog

5.2 XILINX VIVADO DESIGN SUITE (2020.2 version)

Xilinx is a powerful software tool that is used to design, synthesize, simulate, test and verify digital circuit designs. The designer can describe the digital design by either using the schematic entry tool or a hardware description language. In this software we will create Verilog design input files – the hardware description of the logic circuit, compile Verilog source files, create a test bench and simulate the design to make sure of the correct operation of the design (functional simulation). The purpose of this is to give new users an exposure to the basic and necessary steps to implement and examine your own designs using Vivado environment. In this, we will design one simple module (OR gate); however, in the future, you will be designing such modules and completing the overall circuit design from these existing files. A Verilog input file in the Xilinx environment consists of: Entity Declarations: module name and interface specifications (I/O) – list of input and output ports; their mode, which is direction of data flow; and data type. Architecture: defines a component's logic operation.

There are different styles for the architecture body: (i)Behavioural – set of sequential assignment statements (ii) Data Flow – set of concurrent assignments o Structural – set of interconnected components A combination of these could be used, but in this tutorial we will use Dataflow. In its simplest form, the architectural body will take the following format,

regardless of the style: architecture architecture_name of entity_name is begin … -- statement end architecture_name;

ISE (Integrated Software Environment) is a software tool produced by Xilinx for synthesis and analysis of HDL designs, enabling the developer to synthesize (“compile”) their designs, perform timing analysis, examine RTL diagrams, simulate a design’s reaction to different simuli, and configure the target device with the programmer.

Xilinx is an American technology company, primarily a supplier of programmable logic devices. It is known for inventing FPGA. The Xilinx Vivado is primarily used for circuit synthesis and design, while the Modelsim logic simulator is used for system-level testing.

5.3 Project Navigator:

In this section, we introduce the reader to the main components of an “Project Navigator” window, which allows us to manage our design files and move our design process from creation to synthesis and to simulation phase.

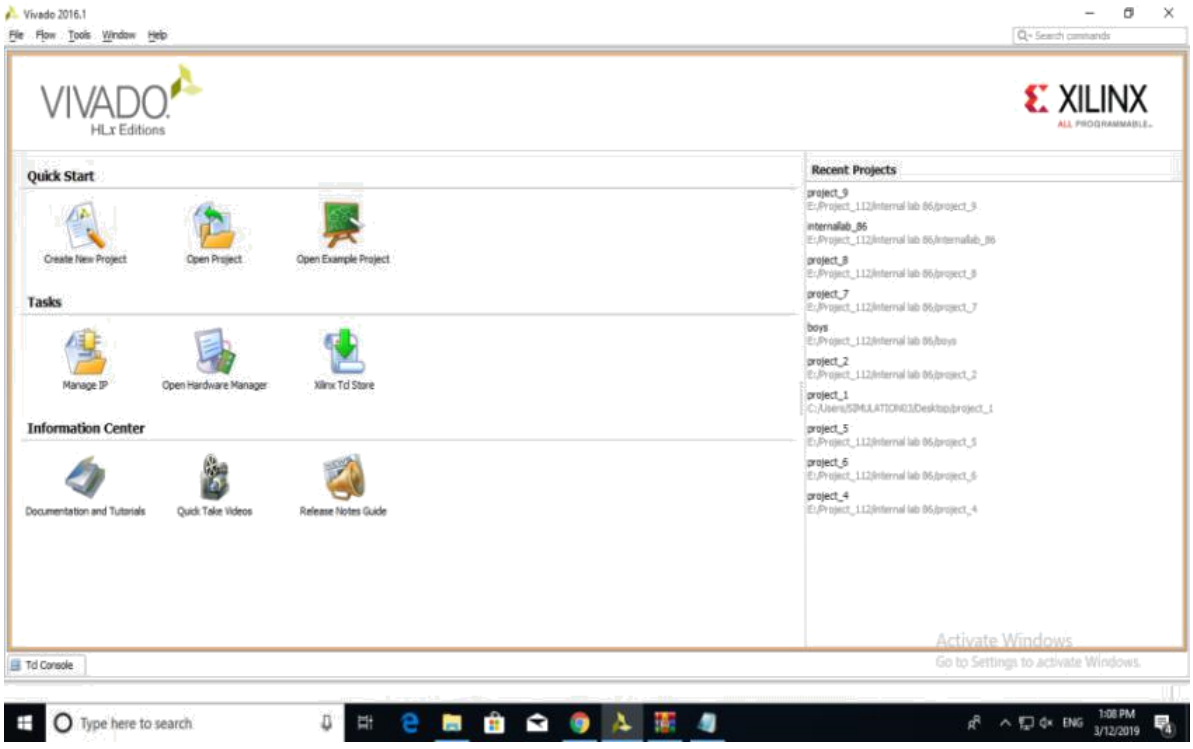


Fig 5.2 : Xilinx Vivado Project navigator window

By opening the Xilinx Vivado Design suite, we will come to see the 3 main points. They are

1.Quick start

2.Tasks

3.Information Center

In the Quick start block, We have create a new project, open project and open example project.

In the Tasks, We have Manage IP, open hardware manager, xilinx Td store.

In the Information center, we have documentation and tutorials,quick take videos and release notes guide.

This section describes the four basic steps to working with a project.

Step 1—— Creating a New Project

This creates .xpr file and a working library.

Step 2—— Adding Items to the project

Projects can reference or include source files, folders for organizations, simulations, and any other files you want to associate with the project. You can copy files into the project directory or simply create mappings to files in other locations.

Step 3—— Compiling the Files

This checks syntax and semantics and creates the pseudo machine code that Viavdo uses for simulation.

Step 4—— Simulating a Design

This specifies the design unit you want to simulate and opens a structure tab in the workspace pane.

You specify will be used to create a working library subdirectory within the Project

In order to start Vivado double click the desktop icon: Or click**:**

Creating a New Project

After launching Vivado, from the startup page click the "Create New Project" icon. Alternatively, you can select **File -> New Project**

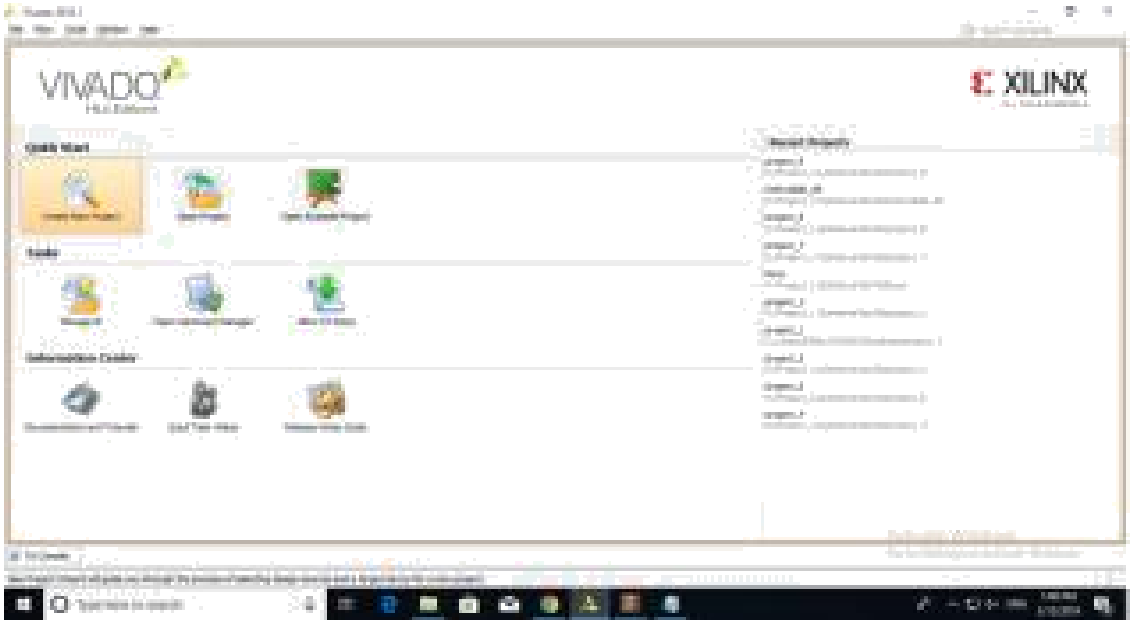


Fig 5.3 : Creating new project window

The New Project wizard will launch, click the “Next >” button to proceed

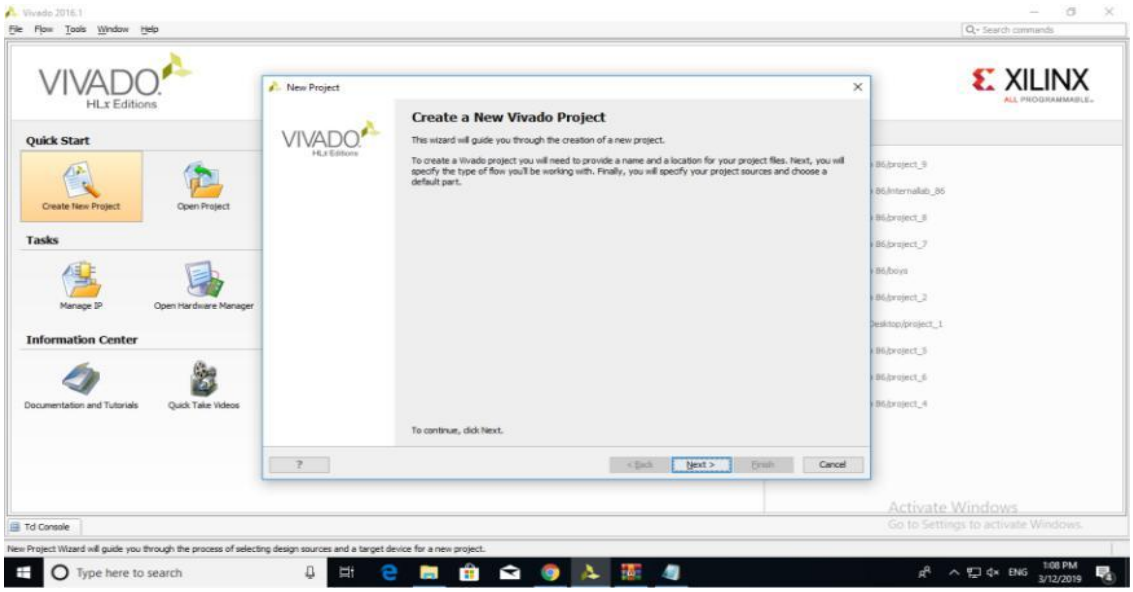


Fig 5.4 : Guiding wizard for the project

Enter a project name and select a project location. **Make certain there are NO SPACES in either!** It’s not a bad idea to only use letters, numbers, and underscores as well. If necessary simply create a new directory for your Xilinx Vivado projects in your root drive (e.g.C:\Vivado). You will likely always want to select the “Create project sub-directory” check-box as well. This keeps things neatly organized with a directory for each project and

helps avoid problems. Click the “Next >” button to proceed.

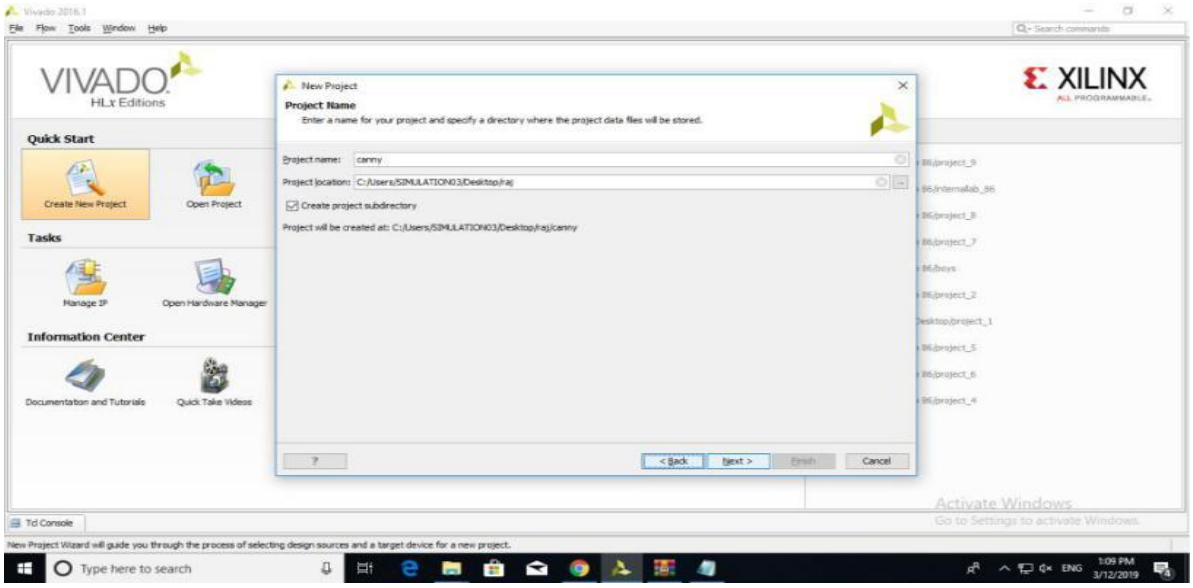


Fig 5.5 : Creating a project name

Select the “RTL Project” radial and select the “Do not specify sources at this time” check-box. If you don’t select the check-box the wizard will take you through some additional steps to optionally add pre existing items such as VHDL or Verilog source files, Vivado IP blocks, and .XDC constraint files for device pin and timing configuration. For this first project you will add the necessary items later. Click the “Next >” button to proceed.

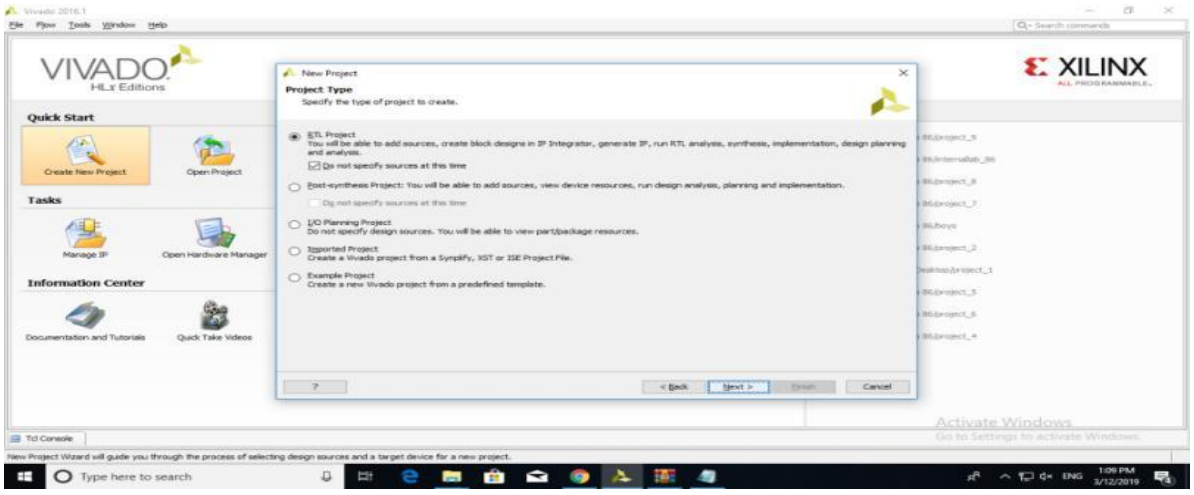


Fig 5.6 : Specifying the RTL project

You need to filter down to and select the specific part number for your project. You can physically read the markings on your chip or refer to your board’s documentation to find

its part number. In the case of the Basys 3 it's the Artix-7 chip that's on the board, and the filters shown will help you get to the correct device that's highlighted. Once you select the correct device click the "Next >" button to proceed.

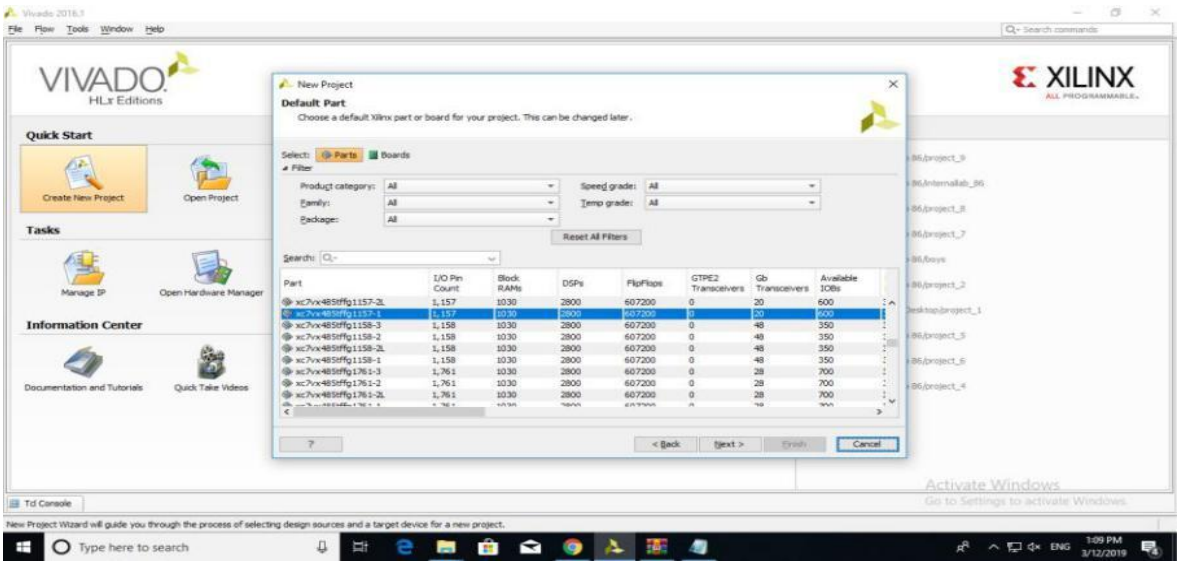


Fig 5.7 : Choosing a board for project

Click the "Finish" button and Vivado will proceed to create your project as specified.

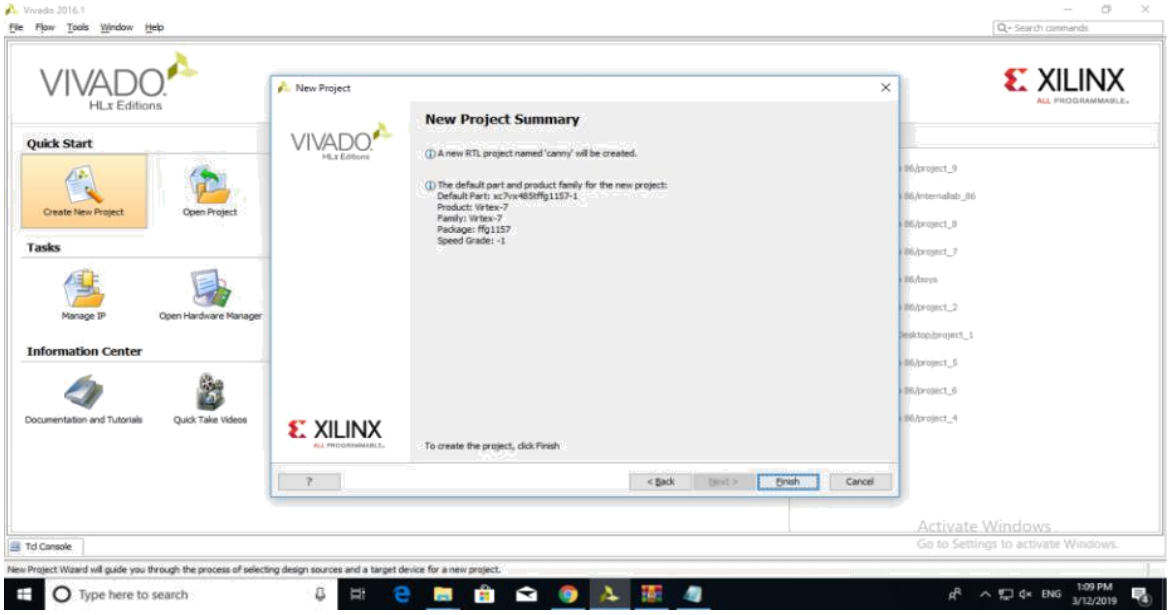


Fig 5.8 : Project summary

5.4 Steps for design entry:

Working through the Basic Project Flow:

The Vivado project window contains a lot of information, and the information displayed can change depending on what part of the design you currently have open as you work through the steps of your project. Keep this in mind as you work through this guide, because if you don't see a specific sub-window or sub-window tab it's possible you aren't in the correct part of the design.

The "Flow Navigator" on the left side of the screen has all the major project phases organized from top to bottom in their natural chronological order. You begin in the "Project Manager" portion of the flow and the header at the top of the screen next to the Flow Navigator reflects this. This header and the corresponding highlighted section in the Flow Navigator will tell you which phase of the design you have open.

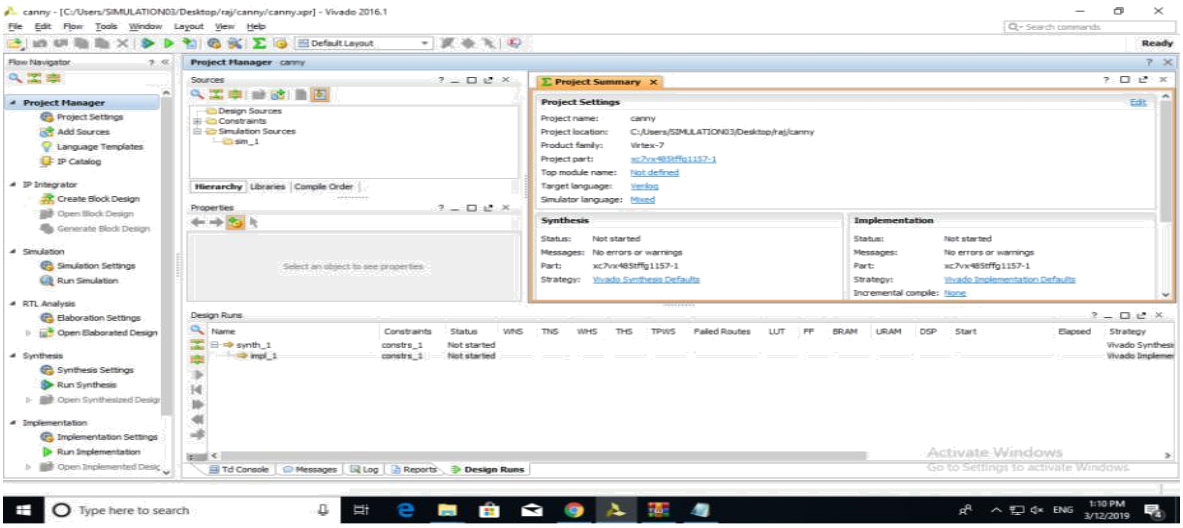


Fig 5.9 : Main window for the project

Project Manager:

1.Project Settings:

Begin by clicking on "Project Settings" under the Project Manager phase of the Flow Navigator

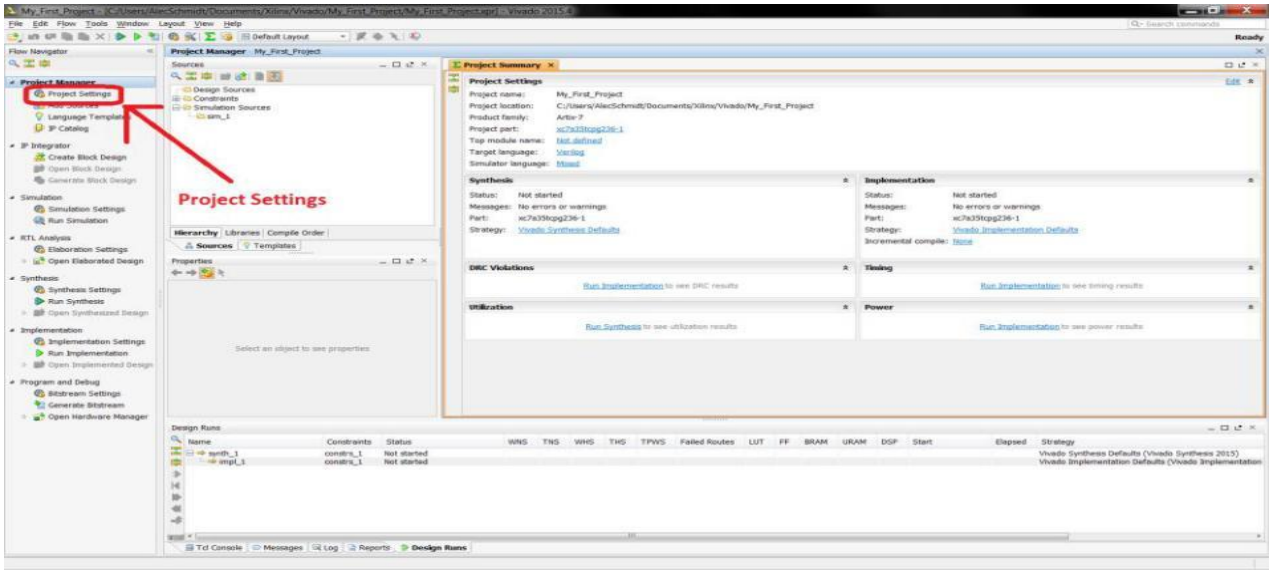


Fig 5.10 : Project settings window

There are a lot of settings available here for all phases of the project flow, but for now just select “System verilog” from the drop-down for the “Target language” in the “General” project settings and click the “OK” button.

2.Add Sources:

Now click on “Add Sources” under the Project Manager phase of the Flow Navigator

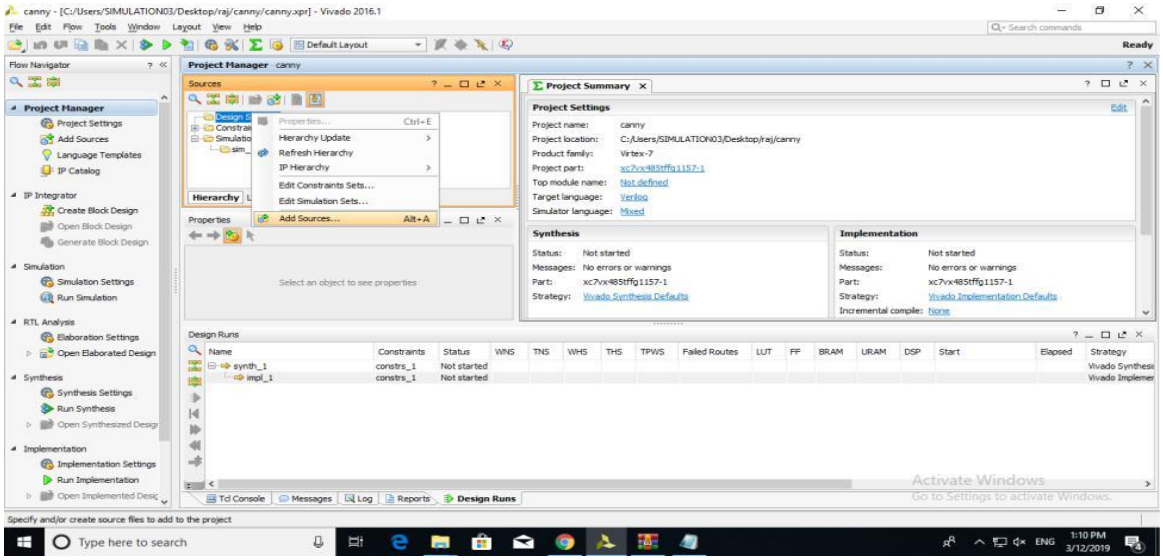


Fig 5.11 : Adding the source files

Select the “Add or create design sources” radial and then click the “Next >” button.

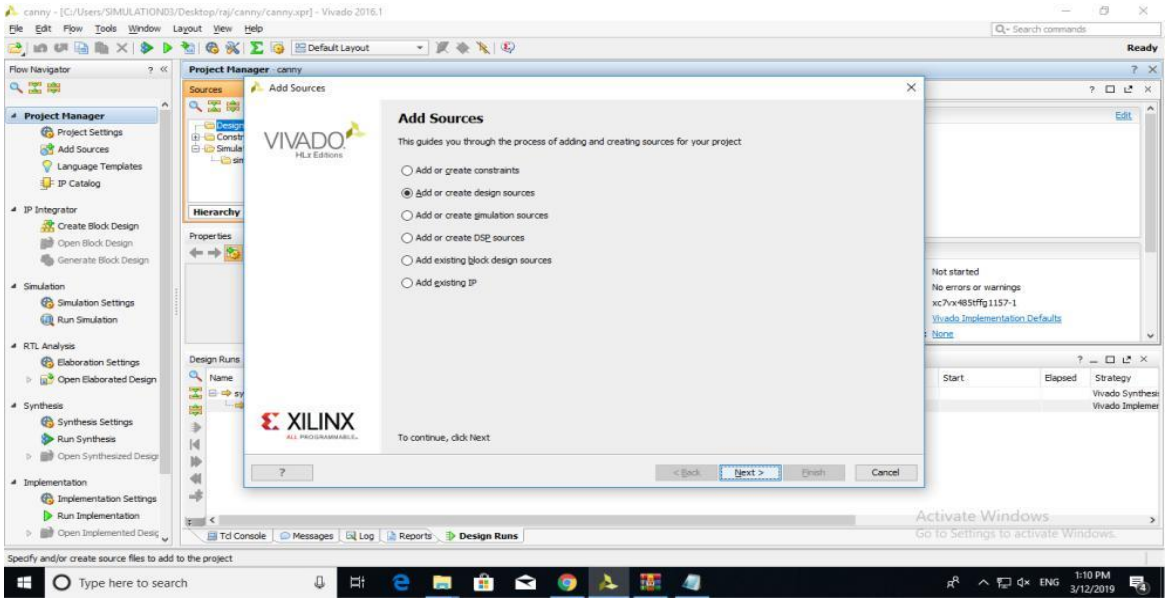


Fig 5.12 : wizard that shows to the design source

Click the “Create File” button or click the green “+” symbol in the upper left corner and select the “Create File…” option.

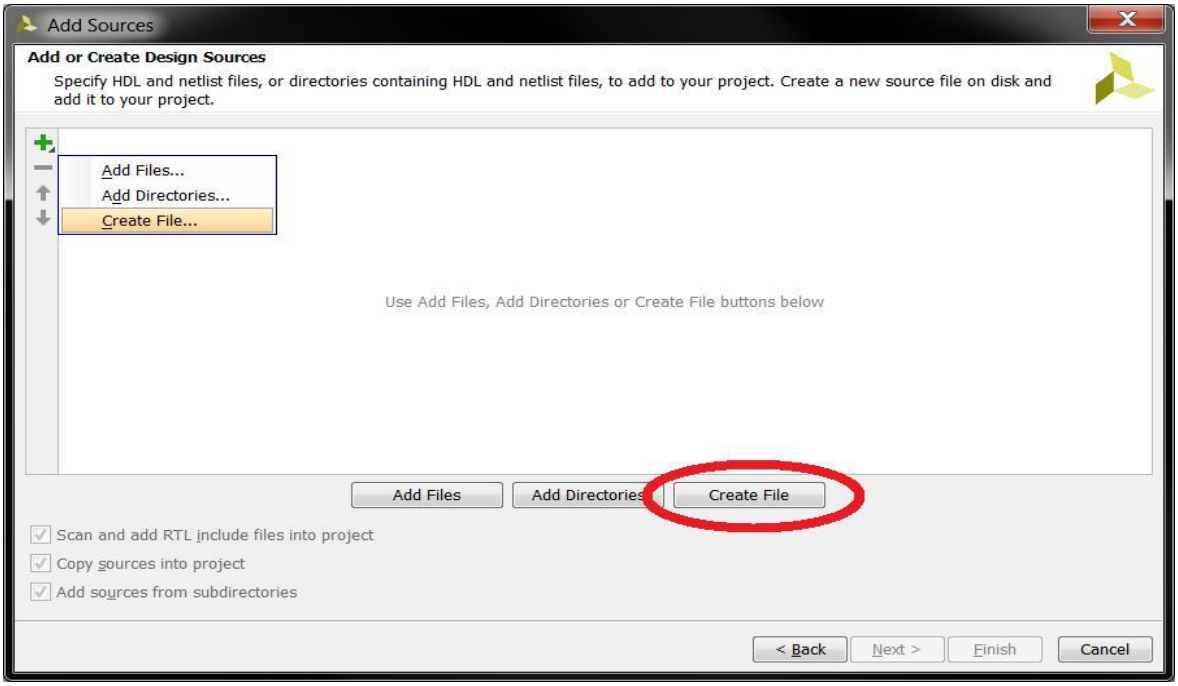


Fig 5.13 : Creating a new file name for new design source

Make sure the options shown are selected in the “Create Source File” popup, and for the sake of following along enter “convolution(Gaussian filter)” for the “File name”. Click the “OK” button when finished.

You can normally enter anything you like for the “File name” as long as it’s valid, but always make certain there are NO SPACES!

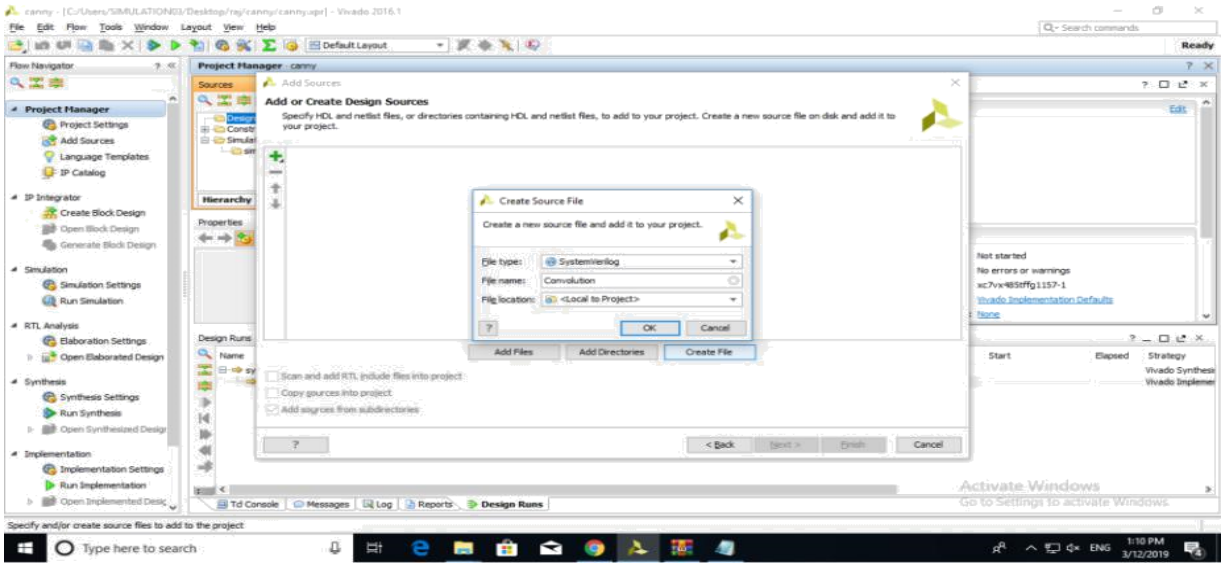


Fig 5.14 : Selecting the type of file and location

Click the “Finish” button and Vivado will then bring up the “Define Module” window.

3.Define Module:

You can use the “Define Module” window to automatically write some of the VHDL code for you. Additional “I/O Port Definitions” can be added by either clicking the green “+” symbol in the upper left or by simply clicking on the next empty line. The “Entity name” and “Architecture name” will be the corresponding Verilog HDL identifiers used in the code, as will whatever is typed in for each “Port Name”. Any valid verilog HDL identifier can be used for any of these, but for the sake of following along enter the information as shown. Make sure the proper “Direction” is set for each. Click the “OK” button when finished.

Note that if you would rather write your own code from scratch you can simply click the “Cancel” button and Vivado will create a completely blank System verilog VHDL source

file inside your project. If you click the “OK” button without defining any “I/O Port Definitions” Vivado will still write the basic Verilog HDL code structure but the port definition will be empty and commented out for you to uncomment and fill later.

Also note that the port names here match the silkscreen reference designators of the switches and LEDs on the Basys 3 board that will be utilized for the example. This is for the convenience of those following along with the Basys 3, but should not be inferred as a requirement by beginners; each name is simply an arbitrary identifier.

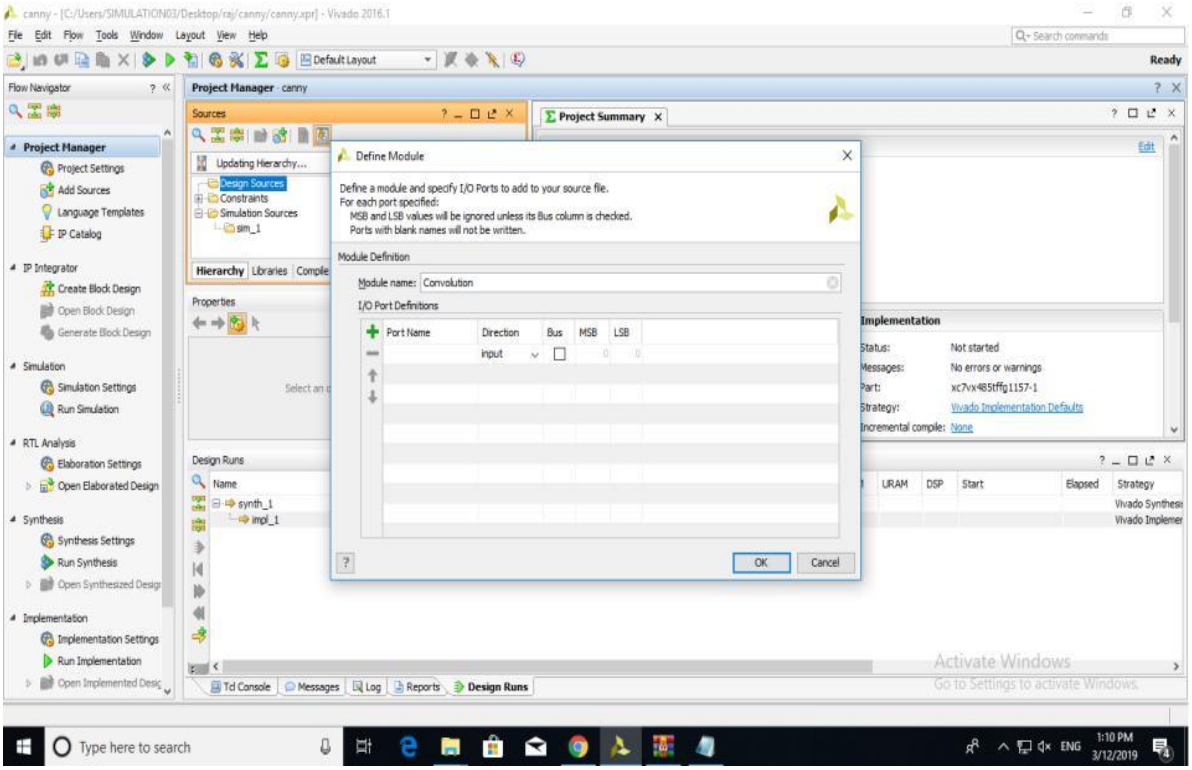


Fig 5.15 : Module defining with ports

The System Verilog HDL source file generated will be added to your project in the “Design Sources” folder as shown. Double click it and it will open up in a new tab for you to view/edit. All the code here was generated by the previous “Define Module” window, and for this example you only need to manually enter the three highlighted lines between the “begin” and “end” keywords. If we want to create a simulation source we have to select a new simulation source by right clicking the add source block in the panel.

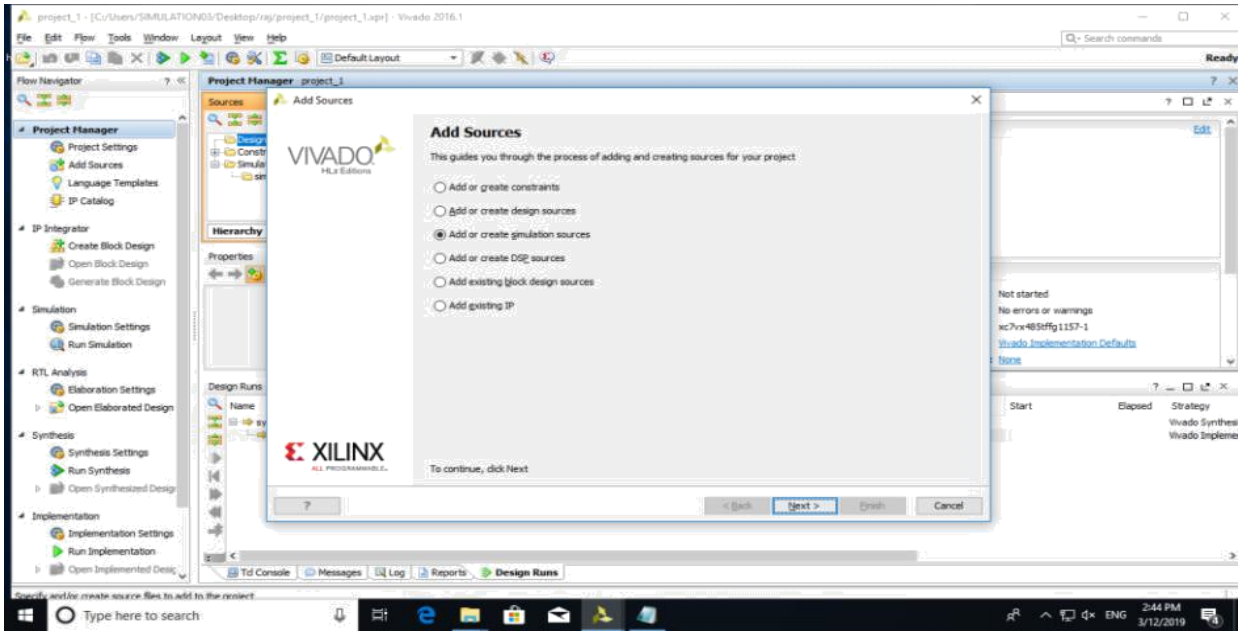


Fig 5.16 : Creating the simulation sources

CHAPTER 6

SIMULATION RESULTS AND RESULTS

Simulation of the FFT and results obtained from its detailed synthesis report shows that the proposed design of 32-point DIT FFT with radix-2 is very efficient in terms of area as well as it is also efficient in terms of speed.

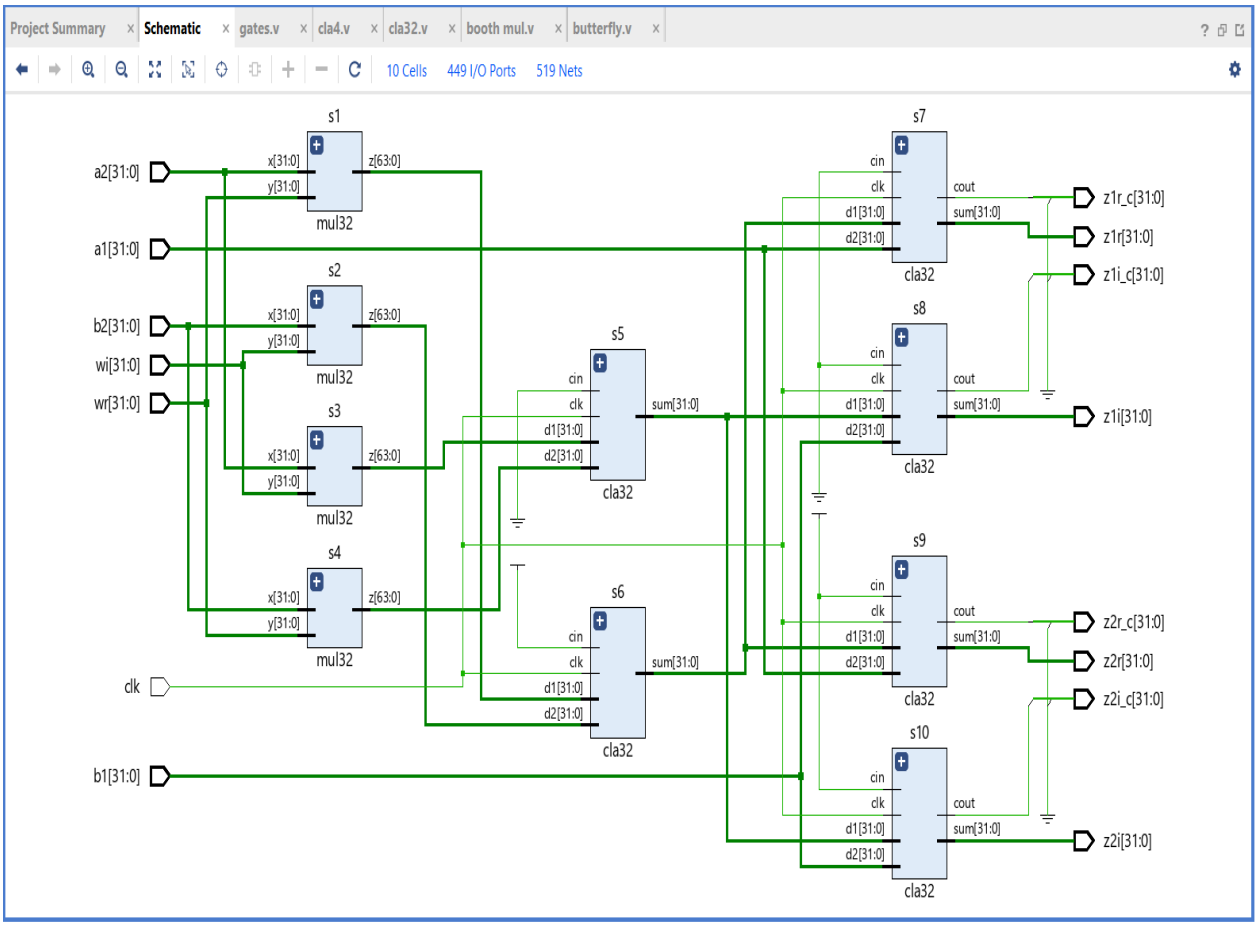
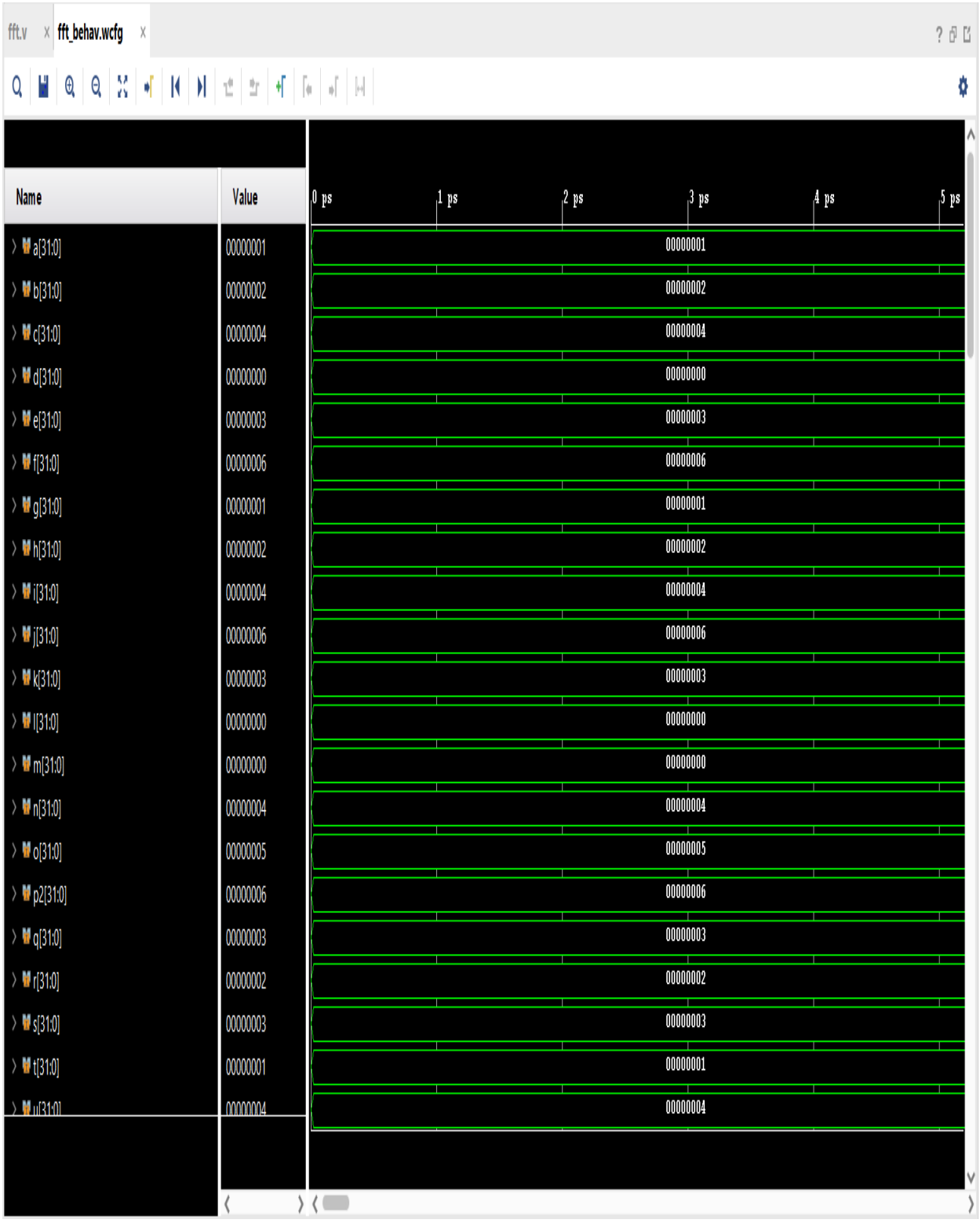
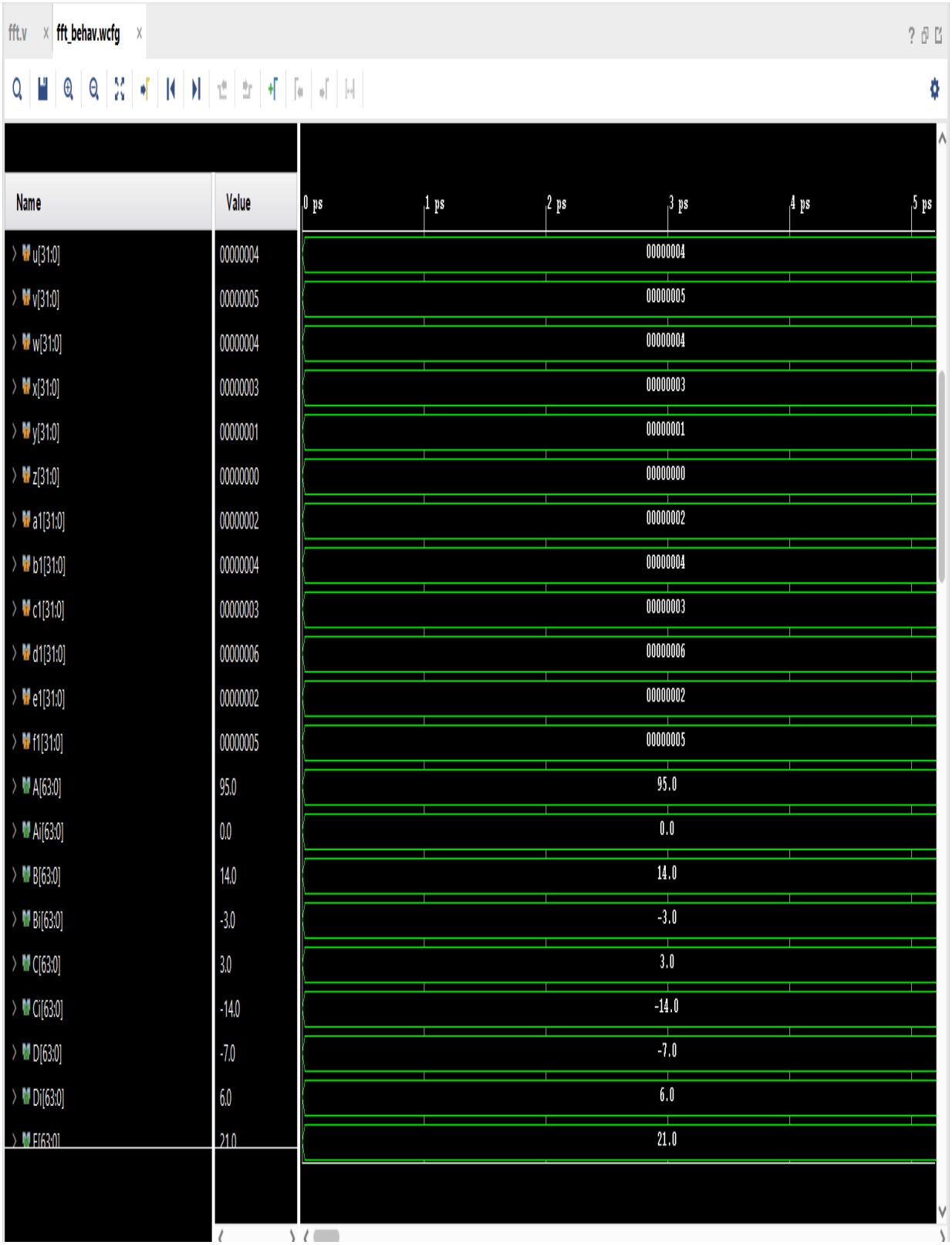


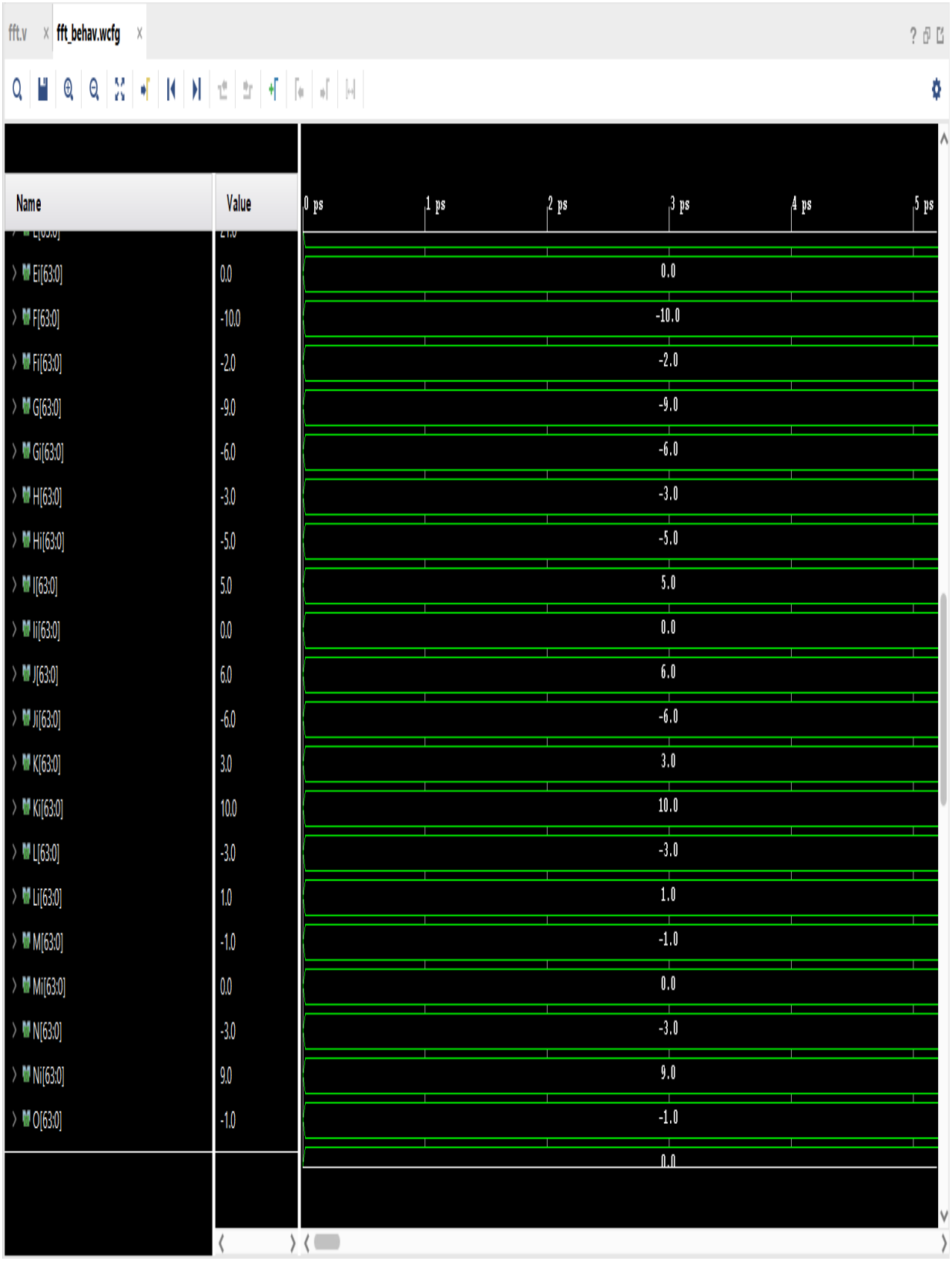
Fig 6.1 : Internal Architecture of The Butterfly Component

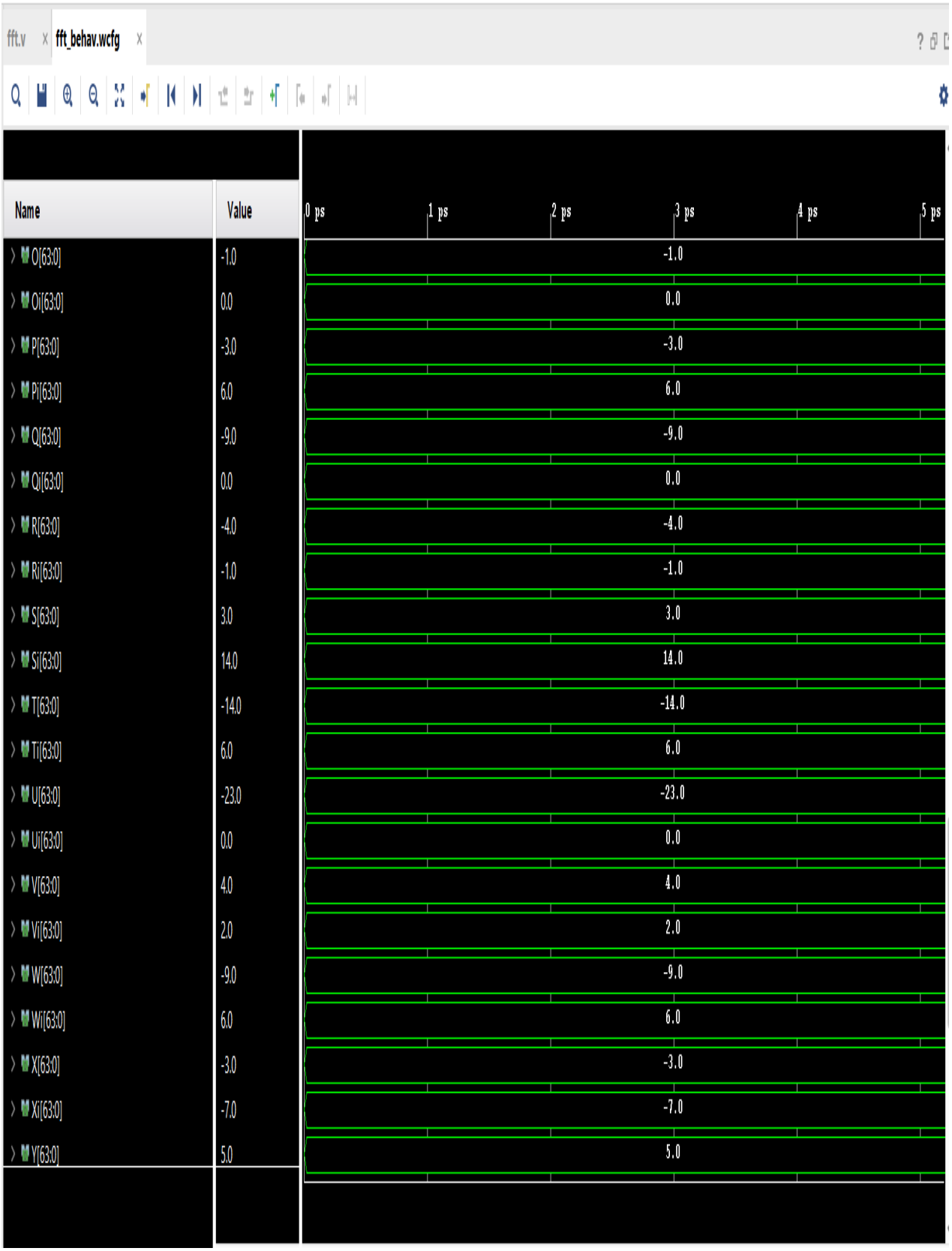
The simulation of this project is done with the help of Xilinx Vivado 14.7 in which 32-point input sequence is provided with the help of an input block and twiddle factors are also provided by calculating and converting them into their 16-bit binary equivalents. The following figures shows the waveforms obtained after simulation.

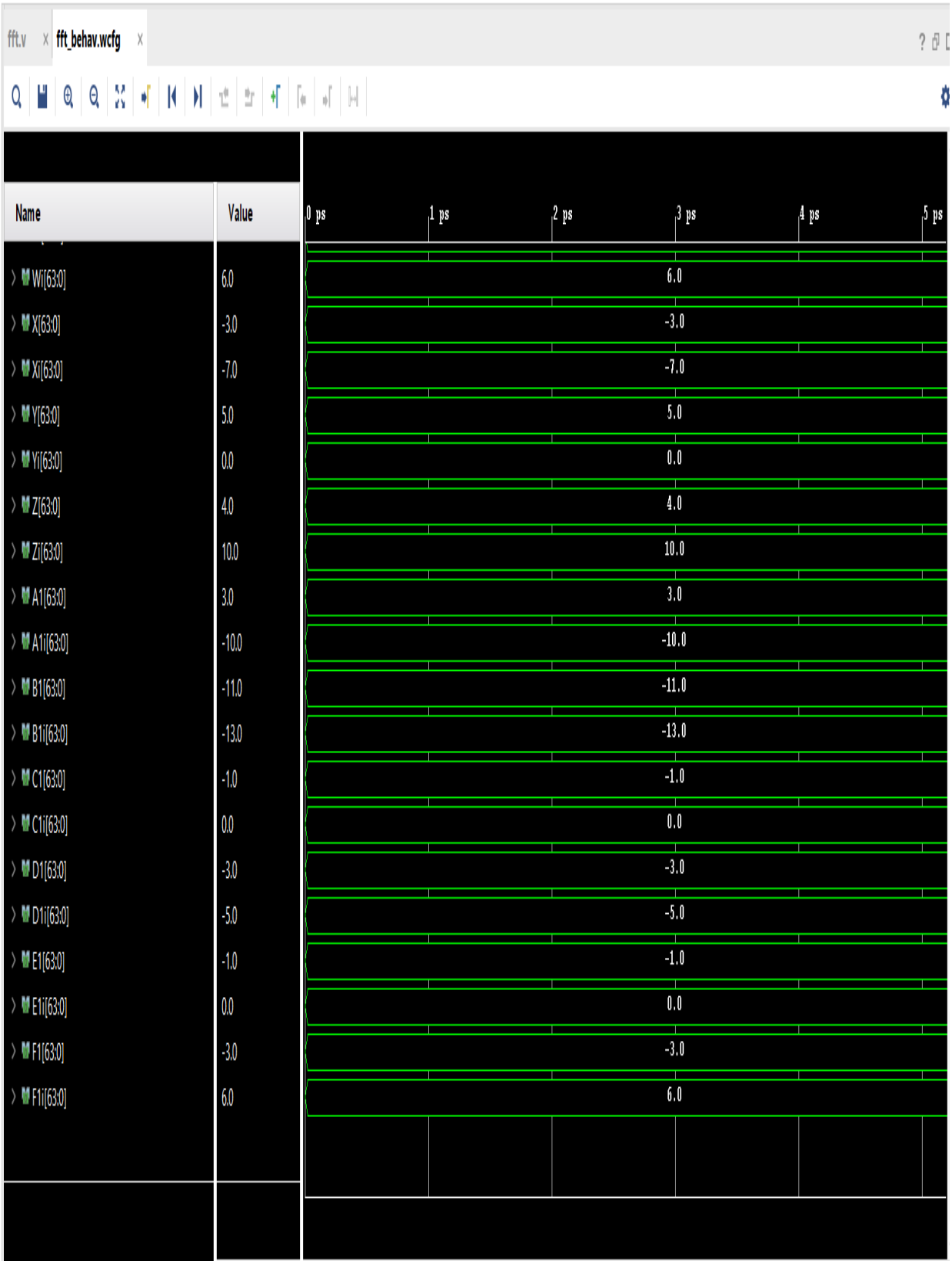
6.1 Simulation results of 32 point FFT:




fft.v
fft_behav.wcfg
Name
Value
0 ps
1 ps
2 ps
3 ps
4 ps
5 ps
u[31:0] 00000004
v[31:0] 00000005
w[31:0] 00000004
x[31:0] 00000003
y[31:0] 00000001
z[31:0] 00000000
a1[31:0] 00000002
b1[31:0] 00000004
c1[31:0] 00000003
d1[31:0] 00000006
e1[31:0] 00000002
f1[31:0] 00000005
A[63:0] 95.0
Ai[63:0] 0.0
B[63:0] 14.0
Bi[63:0] -3.0
C[63:0] 3.0
Ci[63:0] -14.0
D[63:0] -7.0
Di[63:0] 6.0
F[63:0] 21.0


fft.v
fft_behav.wcfg
Name
Value
0 ps
1 ps
2 ps
3 ps
4 ps
5 ps
Ei[63:0]
0.0
F[63:0]
-10.0
Fi[63:0]
-2.0
G[63:0]
-9.0
Gi[63:0]
-6.0
H[63:0]
-3.0
Hi[63:0]
-5.0
I[63:0]
5.0
Ii[63:0]
0.0
J[63:0]
6.0
Ji[63:0]
-6.0
K[63:0]
3.0
Ki[63:0]
10.0
L[63:0]
-3.0
Li[63:0]
1.0
M[63:0]
-1.0
Mi[63:0]
0.0
N[63:0]
-3.0
Ni[63:0]
9.0
O[63:0]
-1.0


fft.v
fft_behav.wcfg
Name
Value
0 ps
1 ps
2 ps
3 ps
4 ps
5 ps
O[63:0] -1.0
Oi[63:0] 0.0
P[63:0] -3.0
Pi[63:0] 6.0
Q[63:0] -9.0
Qi[63:0] 0.0
R[63:0] -4.0
Ri[63:0] -1.0
S[63:0] 3.0
Si[63:0] 14.0
T[63:0] -14.0
Ti[63:0] 6.0
U[63:0] -23.0
Ui[63:0] 0.0
V[63:0] 4.0
Vi[63:0] 2.0
W[63:0] -9.0
Wi[63:0] 6.0
X[63:0] -3.0
Xi[63:0] -7.0
Y[63:0] 5.0


fft.v
fft_behav.wcfg
Name
Value
0 ps
1 ps
2 ps
3 ps
4 ps
5 ps
Wi[63:0] 6.0
X[63:0] -3.0
Xi[63:0] -7.0
Y[63:0] 5.0
Yi[63:0] 0.0
Z[63:0] 4.0
Zi[63:0] 10.0
A1[63:0] 3.0
A1i[63:0] -10.0
B1[63:0] -11.0
B1i[63:0] -13.0
C1[63:0] -1.0
C1i[63:0] 0.0
D1[63:0] -3.0
D1i[63:0] -5.0
E1[63:0] -1.0
E1i[63:0] 0.0
F1[63:0] -3.0
F1i[63:0] 6.0

6.2 Synthesis Report

Device Utilization Summary (estimated values)				
Logic Utilization	**Used**	**Available**	**Utilization**	
Number of slice LUT's	926852	1303680	71%	Resources left
Number of fully used LUT-FF pair's	0	2607360	0%	Not used
Number of bonded IOBs	1638	2016	81%	Resource left
Number of DSP	7016	9024	77%	

Table 6.1 : Summary of Virtex Ultrascale + HBM features used in the 32 point FFT

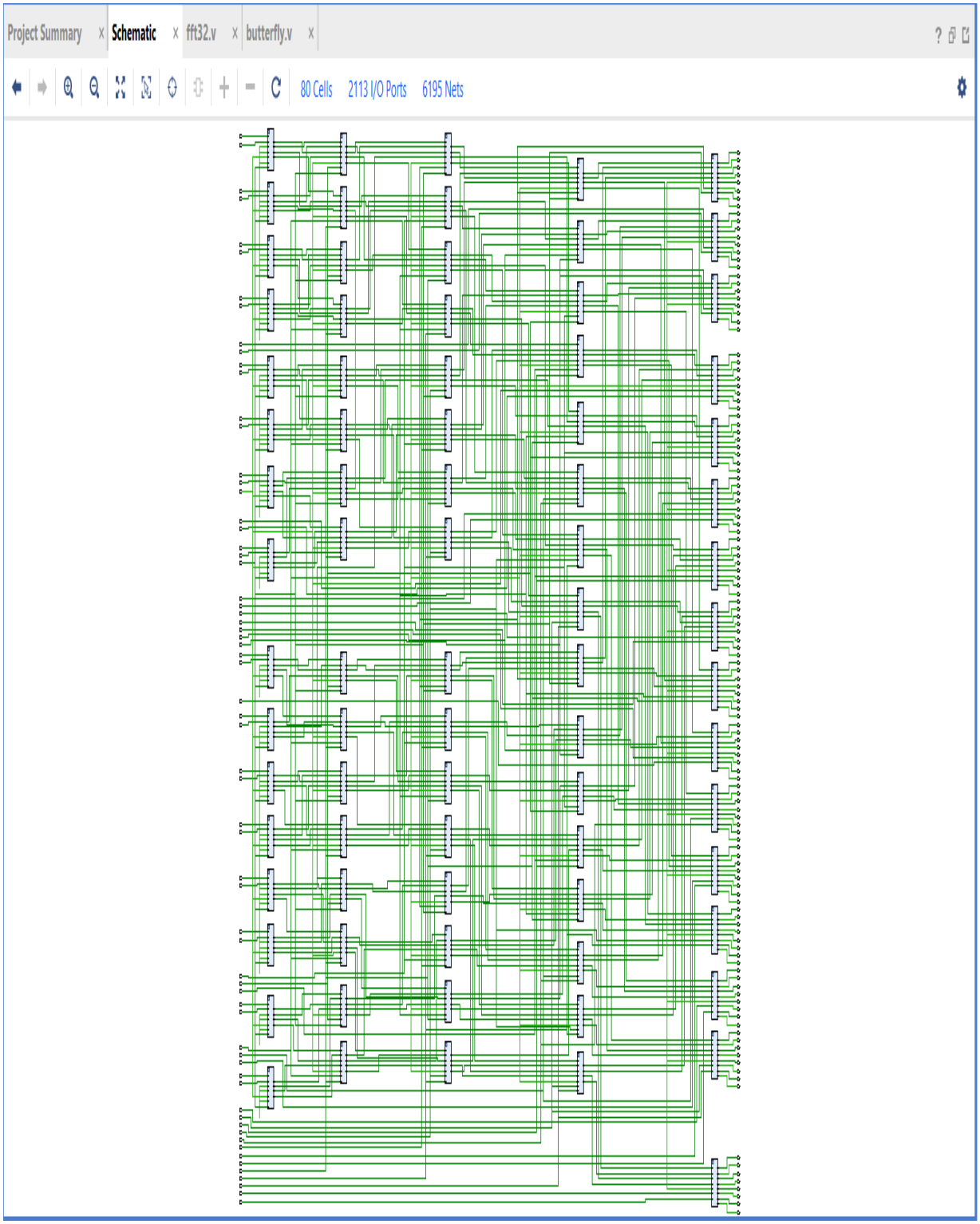


Fig 6.2 : RTL analysis of 32 point FFT

6.3 MATLAB Results:

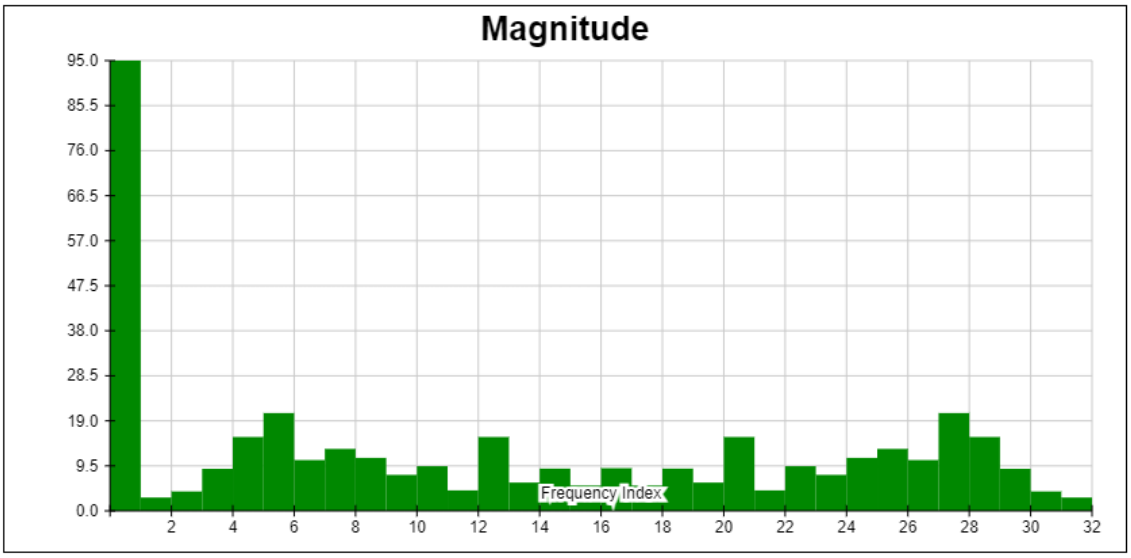


Fig 6.3 : Magnitude of 32 point FFT

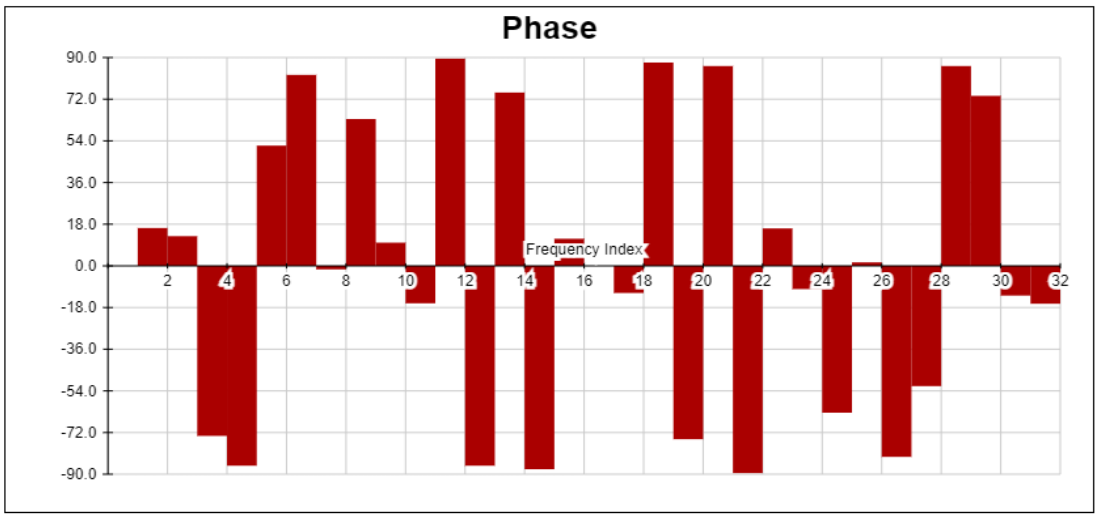


Fig 6.4 : Phase of 32 point FFT

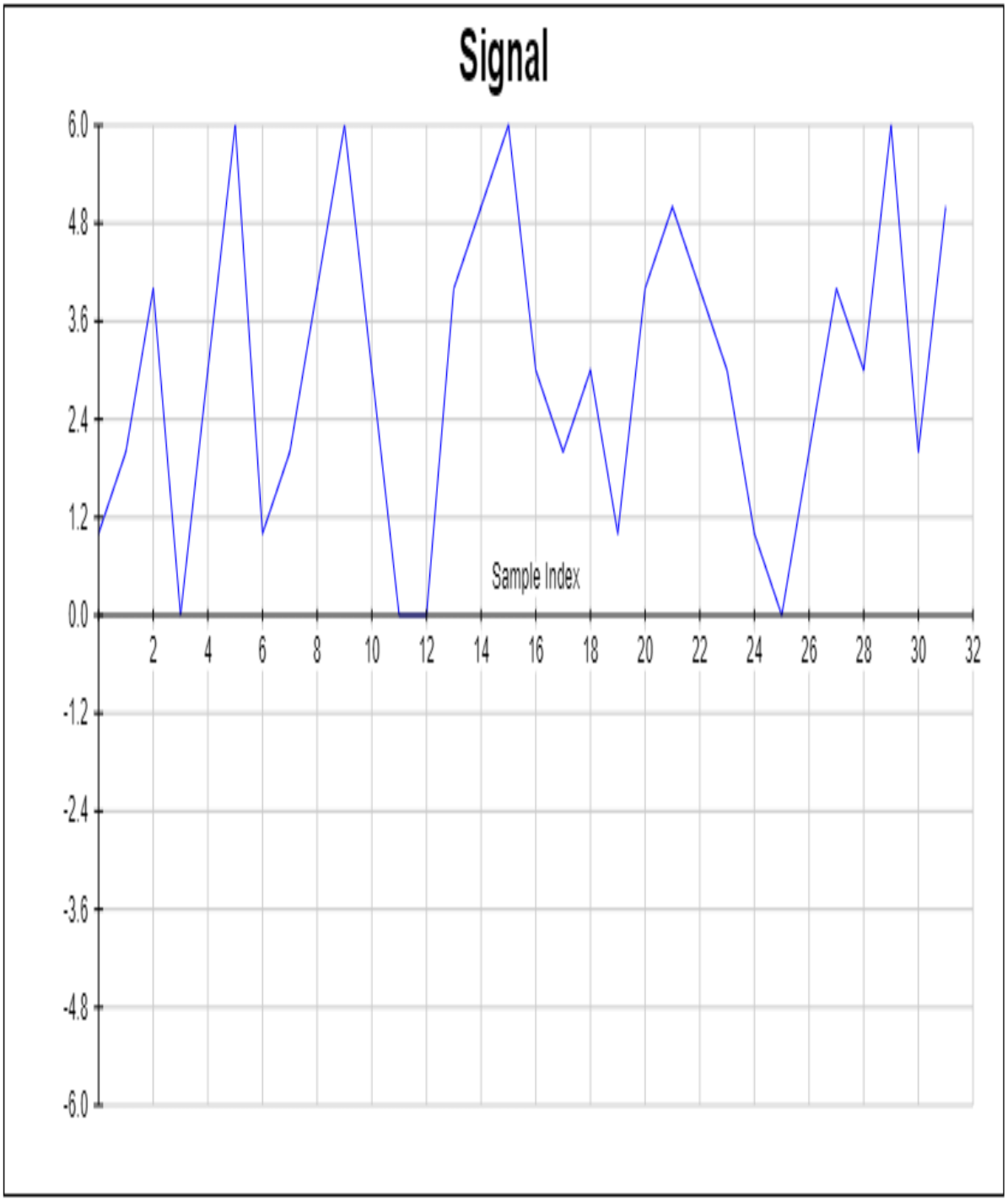


Fig 6.5 : Signal view in matlab

Frequency Index	Real (Cosine)	Imaginary (Sine)	Magnitude	Phase
0	95	0	95	0
1	-2.701	-0.792	2.815	16.349
2	3.993	0.909	4.095	12.827
3	-2.526	8.497	8.864	-73.446
4	-1	15.556	15.588	-86.322
5	-12.726	-16.256	20.644	51.945
6	1.398	10.626	10.717	82.506
7	13.065	-0.344	13.069	-1.51
8	-5	-10	11.18	63.435
9	-7.479	-1.322	7.595	10.021
10	-9.055	2.626	9.428	-16.171
11	0.037	4.323	4.323	89.512
12	-1	15.556	15.588	-86.322
13	1.557	5.762	5.969	74.874
14	-0.336	8.909	8.916	-87.838
15	-5.227	-1.083	5.338	11.709
16	-9	0	9	0
17	-5.227	1.083	5.338	-11.709
18	-0.336	-8.909	8.916	87.838
19	1.557	-5.762	5.969	-74.874
20	-1	-15.556	15.588	86.322
21	0.037	-4.323	4.323	-89.512
22	-9.055	-2.626	9.428	16.171
23	-7.479	1.322	7.595	-10.021
24	-5	10	11.18	-63.435
25	13.065	0.344	13.069	1.51
26	1.398	-10.626	10.717	-82.506
27	-12.726	16.256	20.644	-51.945
28	-1	-15.556	15.588	86.322
29	-2.526	-8.497	8.864	73.446
30	3.993	-0.909	4.095	-12.827
31	-2.701	0.792	2.815	-16.349

Table 6.2: Simulation Results of 32 point FFT in matlab

CONCLUSION

FFT has the benefit over DFT because it has less computations. Implemented butterflies algorithm on VIVADO reduces calculations and makes system more efficient and fast. The FFT algorithm is used information extracting for ECG, EEG signals. FFT helps in converting the time domain in frequency domain which makes the calculations easier as we always deal with various frequency bands in communication systems. This proposed design is helpful in solving complex computations faster and there is a large scope in increasing the processing points speed for bigger signal processing systems. It is also powerful tool for new technology IoMT.

FUTURE WORK

The programming code which we implemented can made more robust and can be optimized to more number of inputs with better precision. By this, the quality of tracing back a signal can be improved and it becomes more flexible and easier instead using heavier operations. There are also some advanced FFT algorithms i.e., Good-Thomas algorithm, Rader algorithm…etc. helps in improving conversion rate.

REFERENCES

[1] The Scientist and Engineer's Guide to Digital Signal Processing, Steven W. Smith, Second Edition.

[2] SnehaN.kherde, MeghanaHasamnis, "Efficient Design and Implementation of FFT".

[3] Ahmed Saeed, M. Elbably, G. Abdelfadeel, and M. I. Eladawy, "Efficient FPGA implementation of FFT/IFFT Processor".

[4] Alan V. Oppenheim, Ronald W. Schafer with John R. Buck, Discrete Time Signal Processing.

[5] AsmitaHaveliya, "Design and Simulation of 32-Point FFT Using Radix-2 Algorithm for FPGA Implementation", IEEE 2012.

[6] Hardware Description Language.URL:http://en.wikipedia.org/wiki/Hardware_description_ Language.

[7] James W. Cooley and John W. Tukey, an Algorithm for the Machine Calculation of Complex Fourier series.

[8] B. Parhami, Computer Arithmetic, Algorithms and Hardware Designs, 1999.

[9]Vivado tutorial https://reference.digilentinc.com/vivado/gettingstarted/start.

[10]Verilog tutorial https://www.chipverify.com/verilog/verilog-tutorial.