LOW POWER SYNCHRONOUS DESIGN OF HARDWARE ARCHITECTURE FOR IEEE 754 SINGLE PRECISION FLOATING POINT FAST FOURIER TRANSFORM

By

SUNKARI SAI KIRAN

Bachelor of Technology in Electronics & Communication Engineering

Jawaharlal Nehru Technological University

Anantapur, Andhra, India.

April, 2013.

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 2015

LOW POWER SYNCHRONOUS DESIGN OF

HARDWARE ARCHITECTURE FOR IEEE 754 SINGLE

PRECISION FLOATING POINT FFT

Thesis Approved:

Dr. James E. Stine

Thesis Adviser

Dr. Subhash C Kak

Dr. Weihua Sheng

ACKNOWLEDGEMENTS

I render my sincere thanks to Dr. James Stine for introducing me to the concepts of CMOS VLSI design. I owe to his patience for handling me in all my tough situations and motivating me to self –learn the interpersonal and conceptual skills. I thank Dr. Subhash Kak and Dr. Weihua Sheng for their support and advice in the final stages of thesis presentation and report review work. I am very thankful to OSU, especially the staff of CASTs for providing me employment and supporting my living at OSU. Finally, I thank my family, friends and well-wishers for giving me hope to live a better life and encouraging me financially and emotionally to pursue Masters at OSU.

Name: Sai Kiran Sunkari

Institution: Oklahoma State University

Date of Degree: December, 2015

Location: Stillwater, Oklahoma

Title of Study: LOW POWER SYNCHRONOUS DESIGN OF HARDWARE ARCHITECTURE FOR IEEE 754 SINGLE PRECISION FLOATING POINT FAST FOURIER TRANSFORM

Pages in Study: 52

Major Field: Electrical and Computer Engineering

Abstract: Signal Processing, communication systems, Digital information systems and many other fields of DSP have the wide need for Fast Fourier Transformation computations. Hardware architecture for computing IEEE 754 single precision floating point FFT is proposed here and the work is focused on power optimization of the design. Cooley-Tukey's (DIF) Decimation in Frequency domain butterfly algorithm is used for the design implementation. Proposed design is a synchronous architecture and proved to be an efficient compared to the earlier parallel architectures. The clock latency and hardware over head of the design is productive and cost effective compared to the designs known earlier. The design is implemented in RTL Verilog and logically verified using Altera-Model Sim. Synthesis of the design is carried out in gscl-45 nm library, 1.1 v process using Synopsys design vision and prime time tools. The power reports showed that the proposed design consumes 90% less power with 50% reduced clock latency compared to earlier designs. Frequency of the design is compromised to an extent but can be improved using the suggested novel sub-designs of floating point add/sub and multiply blocks. Techniques for further power optimization are also given for future implementations.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.0) Introduction to Digital Signal Processing:

Digital Signal Processing (DSP) is one of the widely researched engineering concepts with applications in diverse fields like air traffic control, military radar systems, radio, mobile and satellite communications, weather forecasting. DSP provides an efficient and better way of communication for transferring the digital signals from source to destination by processing and analyzing the signals. This is because DSP is application – specific towards its intended application. A general DSP scheme in communications as shown in Figure 1.

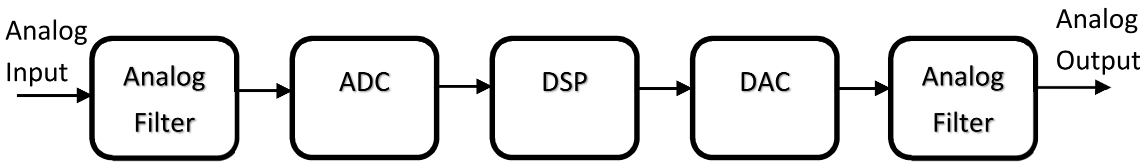

Figure 1: DSP Scheme in Communications (adapted from [1])

In Figure 1 in a general DSP scheme, the input is usually an analog signal generated naturally and the analog filter allows required frequency bandwidths to the next stage. An Analog to Digital Converter (ADC) converts the analog signals to digital signals and the DSP block analyzes and processes these digital signals for efficient and error free transmission. At the receiver end, the processed digital signal is converted back to an analog signal by a Digital to Analog converter (DAC) and filtered by an analog filter. The present thesis work is focused on hardware design for one of the core functions of a DSP block.

The primary function of DSP block is to perform mathematical and numerical manipulations on digital signals. For example, let an analog signal with maximum frequency f having some noise that is sampled at the Nyquist rate (T_s = 1/2f) and quantized to give a digital signal. This digital signal may undergo any mathematical modifications, like frequency transformations, differentiation, integration, noise filtering for the desired output. Generally, functions on a digital signal can be implemented using a software approach with the help of a general computer or by special DSP hardware. Though a general purpose computer is a straight forward and convenient way for illustrating DSP theory and applications, high speed real time signal processing require special purpose DSP hardware for faster calculations and accuracy. Between these two extremes sits the programmable microprocessors possibly attached to general purpose host computers [1].

1.1) Need for Frequency Transformation of a Digital Signal:

Mathematical analysis of a signal can be done in either time domain or frequency domain. The time domain functions of a signal are not feasible and flexible for implementation on hardware, so the signals are transformed into the frequency domain to provide a better way for real time hardware implementations. Frequency domain analysis is a better approach due to the following three main reasons. a) The time domain analysis allows alterations to only the amplitude and phase of input signal, whereas, the frequency domain analysis allows modifications to the underlying frequencies of input signal b) The frequency ranges of a signal when obtained, helps in enhancing and suppressing, band stopping, band filtering of desired frequencies from the original signal. c) If the input signal is defined by its frequency spectrum and a system by its frequency response, then the output is a simple multiplication of these terms compared to difficult convolution operation in time domain.

1.2) Frequency Spectrum of Digital Sequences:

1.2.1) Periodic Digital Sequences: If a digital signal is strictly periodic, then the frequency spectrum of the signal has discrete spectral lines, called the line spectrum and are harmonically related. This series of spectral lines is called the Discrete Fourier Series. The Discrete Fourier Series analysis equation is given in Equation 1.1 with 'a_k' denoting the Kth frequency spectral component and K varying from 0 to N-1 [1] .

$$a_k = \frac{1}{N}\sum_{n=0}^{N-1} x[n] * e^{-\frac{j2\pi kn}{N}} \; ; \; 0 \le k \le \mathrm{N}-1 \tag{1.1}$$

1.2.2) Aperiodic Digital Sequences: The Discrete Fourier series equation can be modified to calculate the frequency spectrum of aperiodic signals. By observing Equation (1.1), it can be deduced that increasing period N makes the frequency spectral lines (a_k values) more close to each other. Therefore, as N (period) tends to move towards infinity the input signal can be considered almost aperiodic and the frequency spectral lines bunch together forming a continuous spectrum. The product $N * a_k$ in Equation (1.1) is finite though N tends to move towards infinity. Rewriting the Discrete Fourier Series Equation (1.1) by replacing factor $2\pi k/N$ with Ω gives Equation (1.2) [1] .

$$X(\Omega) = n * a_k = \sum_{n=0}^{N-1} x[n] * e^{-j\Omega n} \tag{1.2}$$

For aperiodic signals, in a general sense, the summation is calculated over all the positive and negative values of n, so changing the limits in Equation (1.2) produces Equation (1.3) [1]:

$$X(\Omega) = \sum_{n=-\infty}^{\infty} x[n] * e^{-j\Omega n} \qquad (1.3)$$

The modified Discrete Fourier Series equation for aperiodic signals in Equation (1.3) is called the Fourier transform (X(Ω)) and is a continuous frequency spectrum of aperiodic digital sequence ($x[n]$).

1.2.3) Aperiodic Sequences with Finite Length (Discrete Fourier Transform (DFT)): In the previous section, the frequency spectrum of aperiodic signals over entire sequence with *n* ranging from -∞ to +∞ was calculated. But in the real world, aperiodic sequences with finite number of non-zero sample values are more common. Therefore, the frequency spectrum of these finite length aperiodic sequences is calculated by assuming that the finite non-zero sample values are periodic over an entire length. For example, if the aperiodic sequence is finite with N samples, then the sequence is treated as periodic with period N over the entire length. This simplifies the calculation of frequency spectrum to Discrete Fourier series (periodic sequences) discussed earlier. Multiplying and replacing $a_k * N$ in Equation (1.1) with X (k), and rewriting exponential factor as ${W_N}^{kn}$, the Discrete Fourier Transform equation is obtained as [1].

$$X(k) = \sum_{n=0}^{N-1} x[n]\, {W_N}^{kn}, \qquad 0 \le k \le \mathrm{N} - 1 \qquad (1.4)$$

The above equation gives the frequency spectrum of aperiodic sequence with finite N samples. On the other hand, the IDFT (Inverse Discrete Fourier Transform) regenerates the original sequence $x[n]$ from X (k) called the synthesis equation and is given in Equation (1.5) [1].

$$x[n] = \frac{1}{N} \sum_{n=0}^{N-1} X[K] * {W_N}^{-kn} \qquad (1.5)$$

1.3) Fast Fourier Transform:

Expanding equation (1.4) for Discrete Fourier Transform the following frequency spectral components at the output is obtained.

$X(0) = x[0]W_8{}^0 + x[0]W_8{}^0 + x[0]W_8{}^0 + x[0]W_8{}^0 + x[0]W_8{}^0 + x[0]W_8{}^0 + x[0]W_8{}^0 + x[0]W_8{}^0$
$X(1) = x[0]W_8{}^0 + x[0]W_8{}^1 + x[0]W_8{}^2 + x[0]W_8{}^3 + x[0]W_8{}^4 + x[0]W_8{}^5 + x[0]W_8{}^6 + x[0]W_8{}^7$
$X(2) = x[0]W_8{}^0 + x[0]W_8{}^2 + x[0]W_8{}^4 + x[0]W_8{}^6 + x[0]W_8{}^8 + x[0]W_8{}^{10} + x[0]W_8{}^{12} + x[0]W_8{}^{14}$
$X(3) = x[0]W_8{}^0 + x[0]W_8{}^3 + x[0]W_8{}^6 + x[0]W_8{}^9 + x[0]W_8{}^{12} + x[0]W_8{}^{15} + x[0]W_8{}^{18} + x[0]W_8{}^{21}$
$X(4) = x[0]W_8{}^0 + x[0]W_8{}^4 + x[0]W_8{}^8 + x[0]W_8{}^{12} + x[0]W_8{}^{16} + x[0]W_8{}^{20} + x[0]W_8{}^{24} + x[0]W_8{}^{28}$
$X(5) = x[0]W_8{}^0 + x[0]W_8{}^5 + x[0]W_8{}^{10} + x[0]W_8{}^{15} + x[0]W_8{}^{20} + x[0]W_8{}^{25} + x[0]W_8{}^{30} + x[0]W_8{}^{35}$
$X(6) = x[0]W_8{}^0 + x[0]W_8{}^6 + x[0]W_8{}^{12} + x[0]W_8{}^{18} + x[0]W_8{}^{24} + x[0]W_8{}^{30} + x[0]W_8{}^{36} + x[0]W_8{}^{42}$
$X(7) = x[0]W_8{}^0 + x[0]W_8{}^7 + x[0]W_8{}^{14} + x[0]W_8{}^{21} + x[0]W_8{}^{28} + x[0]W_8{}^{35} + x[0]W_8{}^{42} + x[0]W_8{}^{49}$

Table 1: Frequency Components of 8-point DFT [adapted from [2]]

From the equations in Table 1, an observation can be made that the number of multiplications required for an *8* point DFT are a total of *64.* In general, for an N point DFT, the total complex multiplications required are $2*N^2$. If the sequence is either even or odd, the computational complexity is still N^2 multiplications. However, for a large sample number $N = 1000$, total of 10^6 multiplications are required in calculating DFT which is the most time consuming effort in hardware. To reduce this time of computation and increase the calculation speed, an underlying property of redundancy in the DFT is used.

It may be observed that, while calculating the DFT, the same values of $x[n]\ W_N{}^{kn}$ are calculated many times as the computation proceeds. This is because the factor $W_N{}^{kn}$ is a periodic function with limited number of distinct values. Therefore, using the symmetry and redundancy within the DFT, efficient algorithm is derived to reduce the computational complexity of DFT. These highly efficient algorithms for speeding up the calculation of DFT are called the Fast Fourier

Transforms [1]. Many FFT algorithms with different features are known in calculating DFT of which the butterfly algorithm is the most commonly-used FFT algorithm and is widely used in almost all of the hardware implementations for calculating DFTs.

1.4) Butterfly Algorithm for Fast Fourier Transform [2]:

A N point DFT can be broken down into N/2 DFTs. This is typically called decomposition [1]. Conventional decomposition assumes that the number of samples N is not a prime and in more general N is restricted to the lengths of $N=2^i$ (called radix 2) where i is a positive integer. Procedures for $N=4^i$ (called radix-4) can also be implemented, but are less prevalent than radix 2 DFTs. Breaking down the input sequences $x[n]$ into shorter, interleaved, subsequences by conventional decomposition is referred as the Decimation in Time domain FFT. Furthermore, Decomposition of input sequences is achieved by using the properties of symmetry and periodicity of ${W_N}^{kn}(twiddle\ factors)$ values. The present focus for this thesis is N=8 point FFT hardware. A set of twiddle factors for N=8 is given in Table 2, to show the property of symmetry and periodicity.

$$Symmetery\ {W_N}^{k+N/2} = -{W_N}^k;\ periodicity\ {W_N}^{k+N} = {W_N}^k;$$

${W_8}^{48} = {W_8}^{40} = {W_8}^{32} = {W_8}^{24} = {W_8}^{16} = {W_8}^{8} = {W_8}^{0} = 1$
${W_8}^{49} = {W_8}^{41} = {W_8}^{33} = {W_8}^{25} = {W_8}^{17} = {W_8}^{9} = {W_8}^{1}$
${W_8}^{42} = {W_8}^{34} = {W_8}^{26} = {W_8}^{18} = {W_8}^{10} = {W_8}^{2}$
${W_8}^{43} = {W_8}^{35} = {W_8}^{27} = {W_8}^{19} = {W_8}^{11} = {W_8}^{3}$
${W_8}^{44} = {W_8}^{36} = {W_8}^{28} = {W_8}^{20} = {W_8}^{12} = {W_8}^{4} = -{W_8}^{0} = -1$
${W_8}^{45} = {W_8}^{37} = {W_8}^{29} = {W_8}^{21} = {W_8}^{13} = {W_8}^{5} = -{W_8}^{1}$
${W_8}^{46} = {W_8}^{38} = {W_8}^{30} = {W_8}^{22} = {W_8}^{14} = {W_8}^{6} = -{W_8}^{2}$
${W_8}^{47} = {W_8}^{39} = {W_8}^{31} = {W_8}^{23} = {W_8}^{15} = {W_8}^{7} = -{W_8}^{3}$

Table 2: Twiddle Factors of N =8

Using the above properties and twiddle factor values, a *N=8* point DFT can be split into two *4*-point DFTs. Decimating further, each *4*-point DFT can be split into two *2*-Point DFTs. For example, if $x[n]$ is defined for n = {0,1,2,3,4,5,6,7}, DFT of $x[n]$ can be achieved in terms of four *2*-point DFTs by splitting 'n' as {0,4},{2,6},{1,5},{3,7}. Two point DFTs for each pair are computed and these four *2* point DFTs are combined into two four point DFTs. The two 2-point DFTs are again combined to give the final output X(k). Therefore, an *8* point DFT can be achieved by calculating four *2*-point DFTs termed as radix-*2* decimation. This decimation in Time domain is classified as a Cooley-Tuckey algorithm for a FFT [30] and is generally represented as butterfly structure in flow graph. A simplified *2*-point butterfly and complete butterfly structure of *8* point Decimation in Time domain FFT are given in the Figure 2 and Figure 3, respectively.

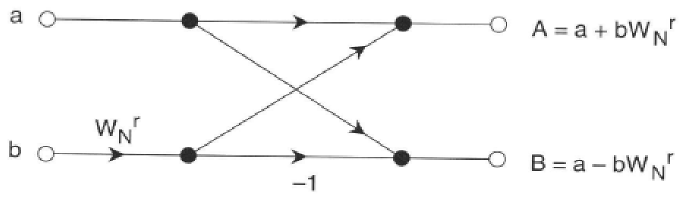

Figure 2: Simplified 2-Point Decimation in Time FFT (adapted from [2])

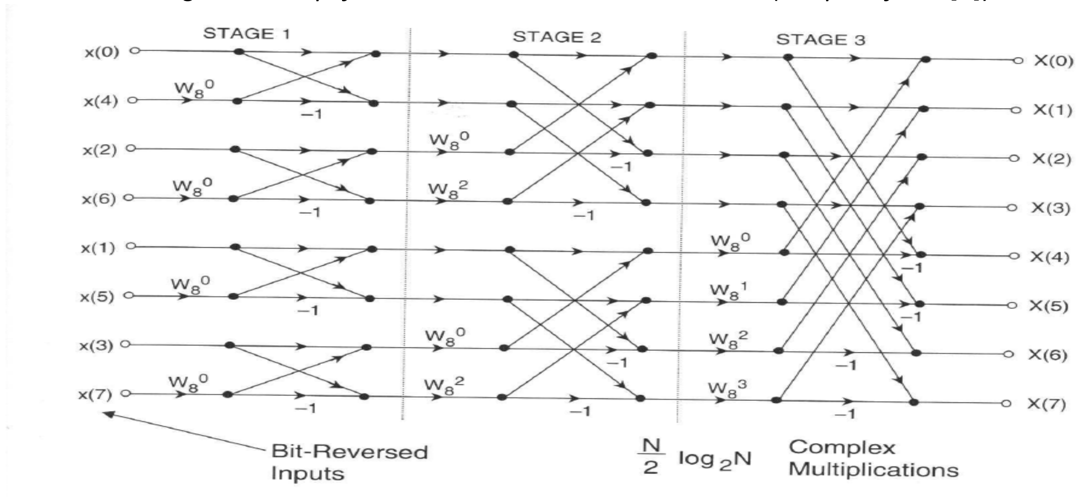

Figure 3: Butterfly Structure of 8-point Decimation in Time Domain FFT (adapted from [2], [30])

It is important to note that the inputs of decimation in time domain FFT algorithm do not follow a natural order whereas, the output frequency spectra follow a natural order. Therefore, bit reversal ordering must be applied to the input samples before proceeding to the actual butterfly operation. Therefore, each input is assigned with a binary equivalent of decimation index 'n' and the binary bits are placed in reverse order and converted back to the decimal number, as shown in Table 3. This passes the correct order of inputs to get the outputs in natural order.

Decimal	0	1	2	3	4	5	6	7
Binary Equivalent	000	001	010	011	100	101	110	111
Bit reversal	000	100	010	110	001	101	011	111
Decimal Equivalent	0	4	2	6	1	5	3	7

Table 3: Bit Reversal of FFT

1.5) Decimation in Frequency, FFT algorithm:

As discussed earlier, the decimation in time domain FFT divides the input data into subsequences grouping for the even and odd parts. The decimation of a sequence in frequency domain of the FFT is the counterpart of decimation in time where the output is decimated instead of input. In short, the decimation in frequency can be achieved from decimation in time domain by simply reversing the signal flow direction and interchanging inputs and outputs. In this scenario, the bit reversal is applied to the frequency components at outputs as decimation starts from frequency components X (k). Figure 4 gives the butterfly structure of decimation in frequency domain.

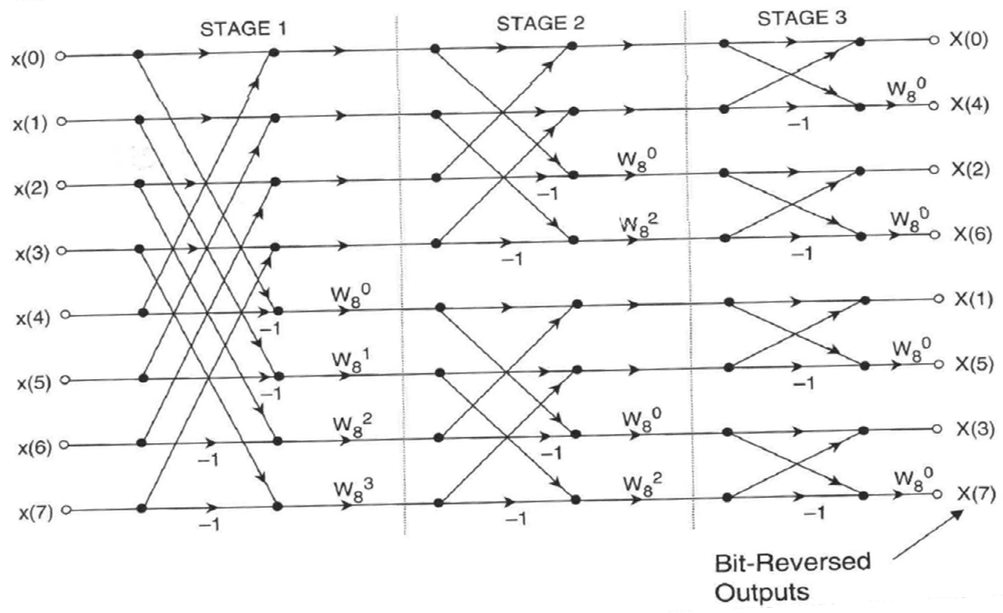

Figure 4: Butterfly Structure of 8-point DIF- FFT (adapted from [2], [30])

Throughout the implementation of the hardware architecture for the FFT, the decimation in frequency butterfly FFT (Figure 4) is followed in this report and the reordering of outputs to get the natural order of the sequence is assumed to be utilized.

1.6) Importance of FFT:

During a frequency transformation, the FFT is much more important compared to DFT, because of its speed advantage. As stated earlier, the number of complex multiplications required in calculating DFT is of the order N^2. By breaking down N-point DFT into *2*-point transforms, there are $\log_2 N$ stages of computation (Figure 3). For a N sampled sequence, there are $\log_2 N$ stages and each stage have a nominal N/2 number of complex multiplications. Thus, the total computation speed is of the order N/2 $\log_2 N$ for an FFT. The speed efficiency of FFT over DFT tabulated in Table 4.

$$Efficiency = \frac{Order\ of\ complex\ multiplications\ in\ DFT}{Order\ of\ complex\ multiplications\ in\ FFT} = \frac{N^2}{\frac{N}{2}log_2N} = \frac{2N^2}{Nlog_2N} \qquad (1.6)$$

N	DFT Multiplications	FFT Multiplications	FFT Efficiency
8	64	12	16:3
256	65,536	1024	64:1
512	2,62,144	2,304	114:1
1024	1,048,576	5,120	205:1
2048	4,194,304	11,264	372:1
4096	16,777,216	24,576	683:1

Table 4: Speed Efficiency of FFT over DFT [adapted from [2]]

Therefore, to reduce the computational time and increase the overall speed, the FFT is a more feasible and convenient approach for implementing the frequency transformation function within software or hardware implementations.

1.6.1) Necessity for Special FFT Hardware: DSP algorithms like the FFT can be implemented either using software on general-purpose computers or by designing a separate hardware processor for a DSP. When the FFT is implemented using software, the run time is significantly higher as well as an increase in power consumption. In the worst case, if the number of samples in an input sequence increase, a normal microprocessor may fail due to the tremendous power consumption. Also, as the frequency components of a digital signal are extensively useful, such as in earth quake sensing, magnetic resonance imaging (MRI), weather forecasting, mobile communications (multiplexing and de-multiplexing), there is large need for separate hardware for DSP functions like the FFT. Besides this, the DSP function involves lot of feedback and loop calculations when compared to traditional arithmetic functions. This requires separate memory elements to store the values for handling loops and feedback operations. A general microprocessor or microcontroller fails to provide efficient feedback and loop operations, thereby, making processors important for application –specific applications. The FFTW [4] is a widely accepted free software that generates source codes and C-subroutines for computing FFT used in many applications. The performance measures of FFTW on various commercial Intel microprocessors is published in [5].

1.6.2) Example of FFT using MATLAB Software

A simple example of *8*-point FFT using software approach (MATLAB [31]) is given below.

```
x = [1+1i 3+3i 4+4i 5+5i 8+8i 9+9i 10+10i 32+32i];

X= fft(x);
```

The FFT of the above given inputs as obtained from the MATLAB is

```
X= 72.0000+72.0000i, -21.4853+37.1838i, -30.0000+20.0000i, -39.1838-4.5147i, -26.0000-26.0000i, -4.5147-39.1838i, 20.0000-30.0000i, 37.1838-21.4853i.
```

1.7) Thesis Organization:

This thesis work targets low power hardware design for computing the *8*-point floating point FFT and the next chapters discuss the hardware realization of the Cooley-Tukey butterfly algorithm using a synchronous pipelined architecture approach.

Chapter *2* presents the real time example of Texas instrument's hardware FFT accelerator attached to C55x DSP. In addition, background for the IEEE *754* Single Precision format and background of CMOS VLSI, use of CMOS in digital design, power consumption and reduction factors in CMOS, parallel & pipeline architectures of FFT are introduced.

Chapter *3* details the implementation of floating point FFT architecture, design of the butterfly module, control logic for butterfly operation, architecture for floating point addition/ subtraction and floating point multiplication.

Chapter *4* gives the reports of logical verification and power estimation of the design. VHDL description of the design, simulation results, output waveform and comparison of the results to MATLAB results, power & performance comparison to earlier designs are also detailed in chapter *4*. Chapter *5* concludes the thesis with suggestions for future modifications to the present design in achieving higher speeds and more power optimized results.

CHAPTER II

BACKGROUND

2.0) FFT Core as Accelerator in TI-C55x DSPs [6]:

Texas instruments (TI) is one of the leading producers of low-power Digital Signal Processors(DSPs). They are currently producing cores of lowest active power solution that are less than 0.20mw/MHZ and delivering a performance that exceeds 12GFLOPs/W [6]. TI produces DSP cores for the C55xx, C674x, C66x Digital Signal Processors series with power efficiency ranging from 1mw/MHZ to 10mw/MHZ and speeds ranging from 60MHz to 1.2GHz [6]. The C55x DSP processors supporting an FFT that is tightly attached to its core is briefly presented here.

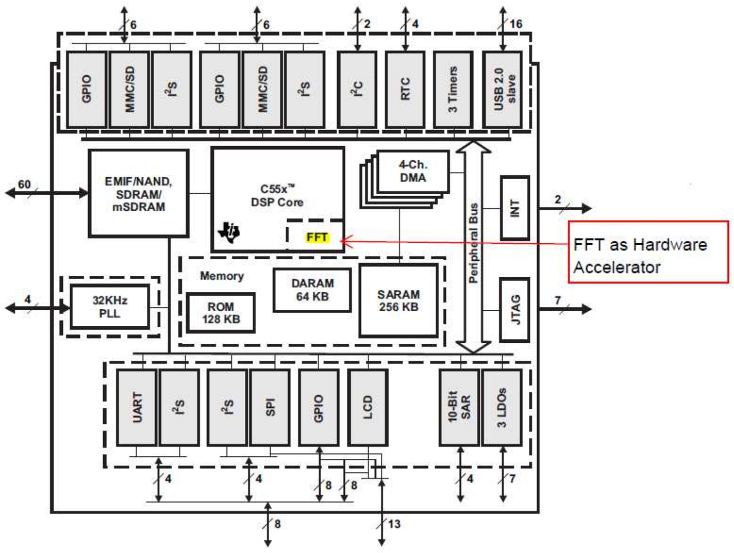

Figure 5: Block Diagram of TI C55x DSP with HWAFFT (adapted from [6])

Figure 5 gives the block diagram of TI-C55x DSP with a FFT hardware accelerator tightly coupled to the DSP core. The FFT hardware accelerator (HWAFFT) that comes with C55x can perform complex valued DIT -FFT (Decimation in Time domain FFT) for 8 to 1024 fixed points. It also performs the inverse FFT and bit reversal order operations of FFT. The HWAFFT is physically present outside the DSP core, but is tightly coupled to the core and have access to full memory bandwidth, core's internal registers, accumulators and address generation units.

The core of HWAFFT consists of a single Radix-2 DIT butterfly implemented in hardware. It supports two stages, a single stage mode and double-stage mode where one FFT stage is performed in each pass of single stage mode and two FFT stages are performed in each pass of double stage mode. The logic of HWAFFT is pipelined to deliver maximum throughput. The HWAFFT core's first pipeline stage computes complex multiplication with twiddle factors and second pipeline stage performs complex addition and subtraction. The twiddle factors used in complex multiplication are stored in a look up table (LUT) in the HWAFFT core.

The 512 complex twiddle factors (16bit each) are made available in LUT for computing up to 1024 point FFTs. The latency of HWAFFT core is 5 cycles in single stage mode and 9 cycles in double stage mode. The communication between the CPU and HWAFFT is achieved through the software. The CPU instruction set architecture (ISA) includes a class of co-processor instructions allowing CPU to initialize, pass data to and execute butterfly computations of the HWAFFT. The HWAFFT shown in Figure 5 supports the input and output vectors of complex numbers with real and imaginary parts represented in two's complement, 16 bit fixed-point floating numbers. Table 5 compares the FFT performance of the HWAFFT vs FFT using the CPU (with CPU operated at V_{core} = 1.05volts and PLL = 60MHz) [6].

Complex FFT	FFT with HWA	CPU(Scale)		HWA versus CPU		
	FFT + BR Cycles	Energy/FFT (nJ/FFT)	FFT + BR Cycles	Energy/FFT (nJ/FFT)	x Times Faster (Scale)	x Times Energy Efficient (Scale)
8 pt	92 + 38 = 130	23.6	196 + 95 = 291	95.1	2.2	4
16 pt	115 + 55 = 170	32.1	344 + 117 = 461	157.1	2.7	4.9
32 pt	234 + 87 = 321	69.5	609 + 139 = 748	269.9	2.3	3.9
64 pt	285 + 151 = 436	98.5	1194 + 211 = 1,405	531.7	3.2	5.4
128 pt	633 + 27 9 = 912	219.2	2499 + 299 =2,798	1,090.4	3.1	5
256 pt	1133 + 535 = 1668	407.2	5404 + 543 = 5,947	2,354.2	3.6	5.8
512 pt	2693 + 1047 =3740	939.7	11829 + 907 = 12,736	5,097.5	3.4	5.4
1024 pt	5244 + 2071 = 7315	1,836.2	25934 + 1783 = 27,717	11,097.9	3.8	6

BR = Bit Reverse

Table 5: FFT Performance on HWAFFT vs CPU (adapted from [6])

2.1) IEEE 754 -1985 Single Precision Floating Point Number:

The IEEE 754 Single precision floating point format is a computer number format as mentioned in IEEE 754-1985 standard [7], this standard is commonly used representation for numbers, on present day computers. The IEEE 754 single precision floating point number occupies 4 bytes = 32bits in a computer memory. The precision of floating point numbers is about 6 to 9 digits of decimal representation. The 32 bit single precision floating point representation is shown in Equation 2.1

1 bit	8 bits	23 bits
Sign	Exponent	Mantissa

$$Number = (-1)^{sign}\ 2^{exponent-127}\,(1.\text{Mantissa}) \qquad (2.1)$$

The 32 bit number representation is usually normalized (1.Mantissa) which says that all the numbers lie between [1,2) in the IEEE 754 format. Since, all binary numbers except zero start with a leading '1'. This gives an extra bit of precision when normalized.

2.1.1) Advantage of Floating Point DSPs over Fixed Point DSPs: All the present day integrated circuits are developed using efficient electronic design automation (EDA) software tools which follow floating point number systems. Developing an IC for fixed point FFT core causes more effort on EDA software tools as it requires the need for the conversion of fixed point numbers into

floating point representation when realizing the hardware. Therefore floating point architectures are widely preferred to reduce the development time and effort of the IC. Also, though the fixed point DSP processors are cheaper compared to floating point processors, the floating point DSPs have an advantage of better precision, dynamic range and development time compared to fixed point DSPs.

In floating point representation, the gap between a floating point number and its adjacent number is usually one ten-millionth of the value of the number [8]. Therefore, when a signal value is represented using floating point, there is an added noise to the signal during rounding off of the number. This added noise results in signal to noise ratio of approximately 30 million to one [8].

In fixed point number representation, the gap between the number and its adjacent number is usually one ten-thousandth of the value of the number which is very large compared to floating point numbers. This large gap results in more noise added to the fixed point number when rounding off and approximately leaves the signal to noise ratio to be, ten thousand to one. If the DSP has to perform 500 iterations over the signal, it results an added noise on to fixed point number representation during round off in each iteration, leaving the signal to noise ratio to be about twenty to one. This is really poor performance that is undesired and can be overcome using floating point representation. Moreover, most of the floating point DSPs can also execute fixed point operations though the execution efficiency differs for both operations.

2.2) CMOS LOGIC:

Many IC technologies are available in the market and the choice of particular fabrication technology depends on the design goal of end product. Early bipolar transistors (RTL-Resistor Transistor Logic) are certainly the right choice for faster IC designs compared to MOSFET (Metal Oxide Semiconductor Field Effect Transistor) families. But, the power consumption and heat dissipation of the RTL family limits the number of transistors to be integrated in a single chip, found in [9].

The introduction of MOSFET led to CMOS logic which solved the problem of transistor limiting and power consumption in an IC. CMOS logic with its complementary nature and being operated with smaller voltages is the best available technology for low-power designs [9]. Also, Speed is compromised to an extent in CMOS compared to RTL, but this can be overruled with the increased count of transistors and smaller less power consumption foot prints in CMOS. Complementary N-MOS pull down and P-MOS pull up networks are used in building a CMOS logic function as shown in Figure 6 a). The examples of inverter and OR gates implemented in CMOS logic are given in Figure 6 b) and c).

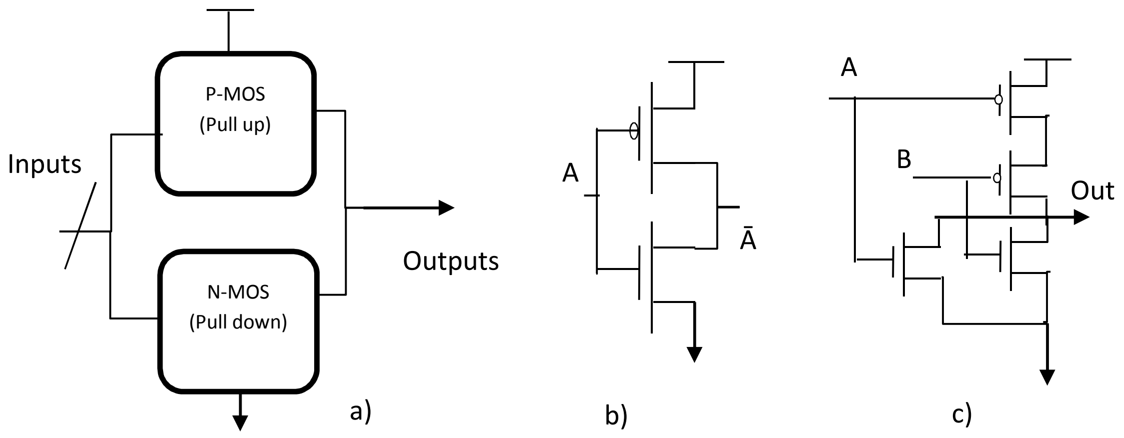

Figure 6: a) CMOS Logic network b) CMOS- Inverter c) CMOS-OR gate

<u>*2.2.1)*</u> <u>Voltage and Current (I-V) Characteristics of MOSFET [10]</u>: A p-type or n-type MOSFET transistor has three regions of operation which are cut-off region, linear region and saturation region. The voltage-current equation between source and drain for the three regions are given below for most transistors that operate under strong inversion [10].

$$I_{ds} = 0 \; ; for \; V_{gs} = 0 \; (Cut \, off \, region) \tag{2.2}$$

$$I_{ds} = \beta \left(V_{gs} - V_t - \frac{V_{ds}}{2} \right) \ ; for\ V_{ds} < V_{sat}\ (linear\ region) \qquad (2.3)$$

$$I_{ds} = \frac{\beta\,(V_{dd} - V_t)^2}{2};\ for\ V_{ds} > V_{sat}\ (saturation\ region) \qquad (2.4)$$

where V_{gs} *is the voltage between gate and source.*

V_{ds} *is the voltage between drain and source*

V_{dd} *is the supply voltage.*

The gain or β value is dependent on geometry and the technology node utilized in the manufacturing of MOSFET. The β value is given in Equation (2.5)

$$\beta = \frac{\mu\, C_{OX}\, W}{L} \qquad (2.5)$$

where μ *is the mobility of the charge carriers in the channel*

C_{ox} *is capacitance per unit area of gate oxide.*

The gate capacitance (parallel plate capacitance between gate and channel in MOSFET) is given by the equation (2.6)

$$C_g = C_{OX} W L \qquad (2.6)$$

Where W *is the width of the gate*

L *is the length of the channel*

Most of the technology nodes choose the minimum possible length L to reduce the delay and power consumption.

The gate capacitance is rewritten in terms of technology parameters with constant $C_{OX}L$ denoted $C_{permicron}$

$$C_g = C_{permicron} W \qquad (2.7)$$

$$C_{OX} L = C_{permicron} \qquad (2.8).$$

2.3) Power Consumption Factors in CMOS:

Power in any electrical circuit is calculated using the product of current in the circuit I to the voltage V, i.e. Power = I * V. For CMOS transistor the I-V characteristics describe earlier results in static and dynamic power consumptions discussed in subsequent sections.

2.3.1) Static Power Consumption [11]:

For CMOS inverters (Figure 6 b), during steady state, any one of the transistors either PMOS or NMOS are always OFF and the other is ON because of complement nature. This always breaks the direct contact from VDD to GND during the steady state of the inverter. So, in any steady logic state, there is no dc path between VDD and GND and this leaves steady state quiescent current to be zero. But, due to the reverse bias currents in the transistor that is OFF, there are some leakage currents possible resulting in power consumption. This is called static power consumption and is given as

$$static\ power = I_{sub} + I_{gate} + I_{junc} + I_{cont} \qquad (2.9)$$

Where I sub = subthreshold leakage current

I gate = gate leakage current

I junc = Junction leakge current

I cont = Contention current

2.3.2) Dynamic Power Consumption:

The power consumed when a gate switches from logic level high to low or vice versa is said to be dynamic power consumption. Dynamic power consumption is mainly due to two factors: switching of load capacitances and short circuit current during switching.

a) Switching Power: During switching activity, the power consumed with the output capacitor charging and discharging is estimated by [11].

$$P_{switching} = C\, {V_{dd}}^2\ f_{sw} \qquad (2.10)$$

f_{sw} is switching frequency and is given in equation (2.11)

$$f_{sw} = \alpha f \qquad (2.11)$$

Here α is called activity factor and f is the systems clock frequency.

b) <u>Internal Short Circuit Current:</u> During the switching of transistors, both the NMOS and PMOS may be ON for a temporary time. This leads to a moment of circuit current between VDD and GND, called the Crow-bar current [12]. Also, the internal capacitance in P-MOS and N-MOS get charged and discharged leading to some power loss. These factors result in short circuit current during switching.

So, Dynamic Power = Switching Power + Short Circuit power.

Generally Short circuit power is neglected as it is mostly less than 10% of total dynamic power.

2.4) Power Reduction Techniques [12] [13]:

2.4.1) Clock Gating: To avoid switching activity (dynamic power consumption), registers unused are turn off by stopping the clock [12]. This saves the clock activity, eliminating switching activity in the unused block. General clock gating technique is given in the figure. Clock gating technique is one of the best way to reduce dynamic power.

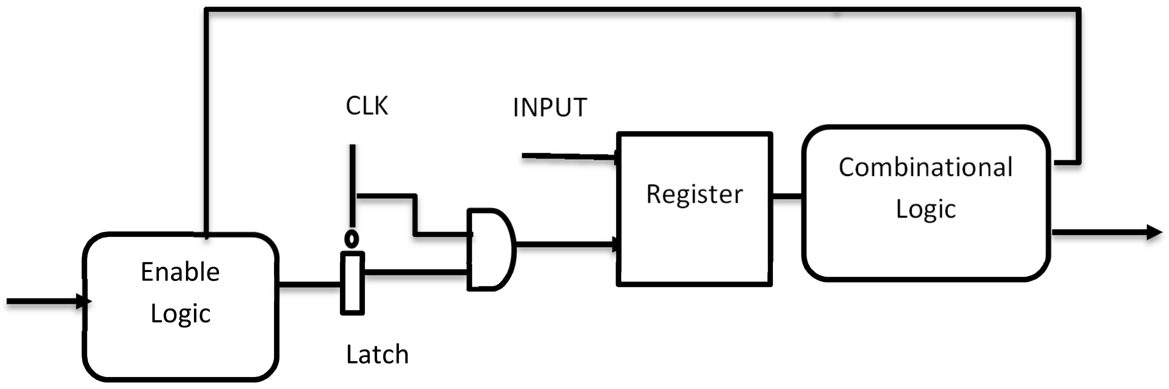

Figure 7: Clock Gating Technique

2.4.2) Power Gating [13], [14]: Leakage power is dissipated by short circuit current during switch activity and also in steady state mode. The leakage power is caused mostly by I_{gate} and I_{subt} (gate and sub threshold) leakage currents. This can be reduced by the idea of power gating. In power gating technique, sleep transistors are used as switches for block of cells instead of individual gates to turn of power to the block of cells. The sleep transistors are treated as either virtual VDD [12] or virtual GND [13]. The outputs of block of cells are also sent to sleep mode to prevent invalid logic levels passing on to next stage.

2.4.3) Disadvantages: Switching of sleep transistors during normal operation costs dynamic power and voltage drop across the sleep transistor degrades performance of the design. The sleep transistor power gating technique is given in the Figure 8.

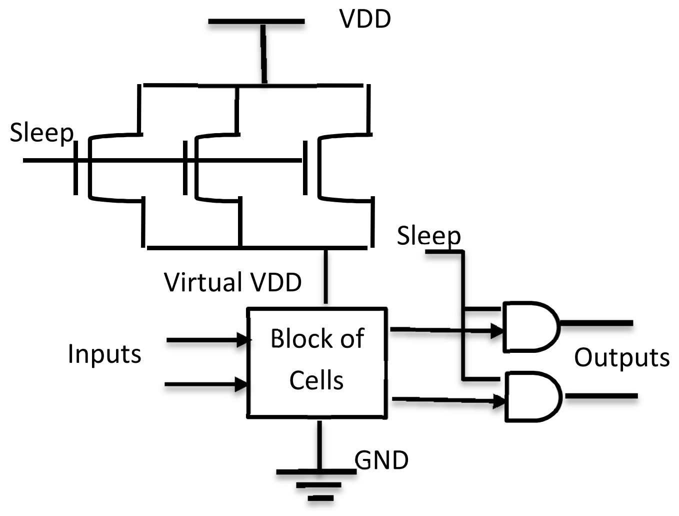

Figure 8: Power Gating Technique

2.5) Pipelined vs Parallel Arithmetic Architecture:

Parallel processing architectures consist of multiple processing units interconnected. The parallel processing architectures have the advantage of speed, but they require more hardware and results in more area and power consumption. Parallel architectures are best used in getting high throughput though the implementation results in high hardware cost. The parallel architecture approaches for

FFT implementation are given in [5] and the parallel FFT architecture implemented using only combinational circuits is given in [6]. During a pipelined arithmetic architecture, the arithmetic operations are executed by an overlap operation in stages, where each stage executes an operation that is required immediately for the next stage. Each stage is connected in a straight order to carry out the instruction flow. High throughput with low latency and reduced hardware cost are the main factors for the implementation of pipelined architecture. The pipelined stages with combinational logic between registers gives the high throughput when clock period 'T' used for registers is as small as possible. Pipeline efficiency is limited by register delay and the uneven time delays may be possible between pipeline stages. Though FFT implementation using parallel architecture results in faster computations, power consumption of large number of butterfly cores adds up resulting in poor power efficiency [6]. On the other hand, pipelined architectures uses less hardware giving better power efficiency.

CHAPTER III

IMPLEMENTATION

3.1) Design of Pipeline Architecture for FFT:

The GALS (Globally Asynchronous and Locally Synchronous) three stage pipelined hardware architecture for the implementation of FFT is given in [18]. This pipeline structure constitutes of three mux stages with twiddle factor multiplications in between two consecutive stages shown in Figure 9.

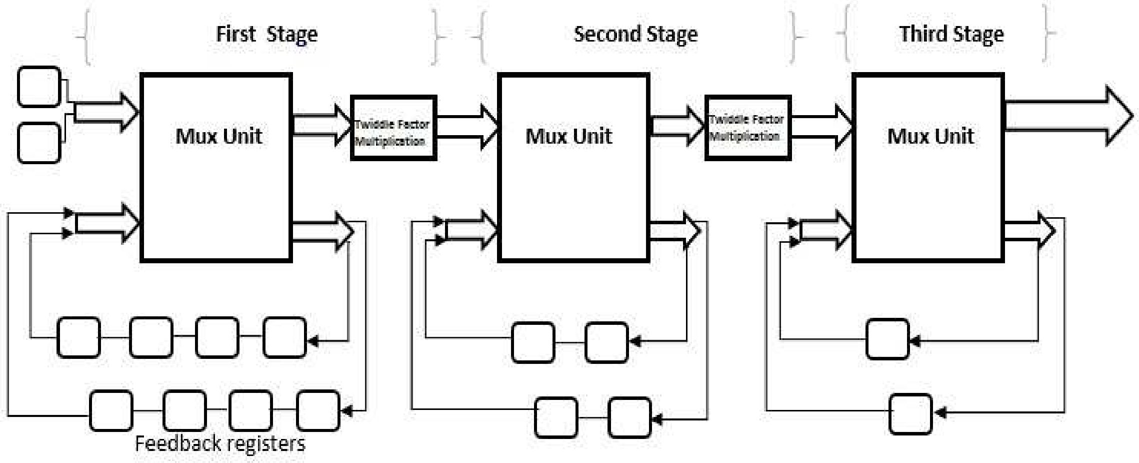

Figure 9: Three stage Pipelined Architecture Design of FFT [adapted from [18]]

Our work is focused on hardware realization and power estimation of the above pipelined architecture using synchronous clocks and detailing methods for low power approaches. Five pipelined stages that include three mux stages and two twiddle factor multiplication stages are used for the complete 8-point DIF-FFT computation. The block diagram for the five stage implementation of 8-point FFT is shown in Figure 10. In Figure 10, the MUX stage performs the mux operations in choosing right operands for complex floating point addition/subtraction and MULT stage performs the complex multiplication between the twiddle factors (loaded from LUTs or external registers) and operands.

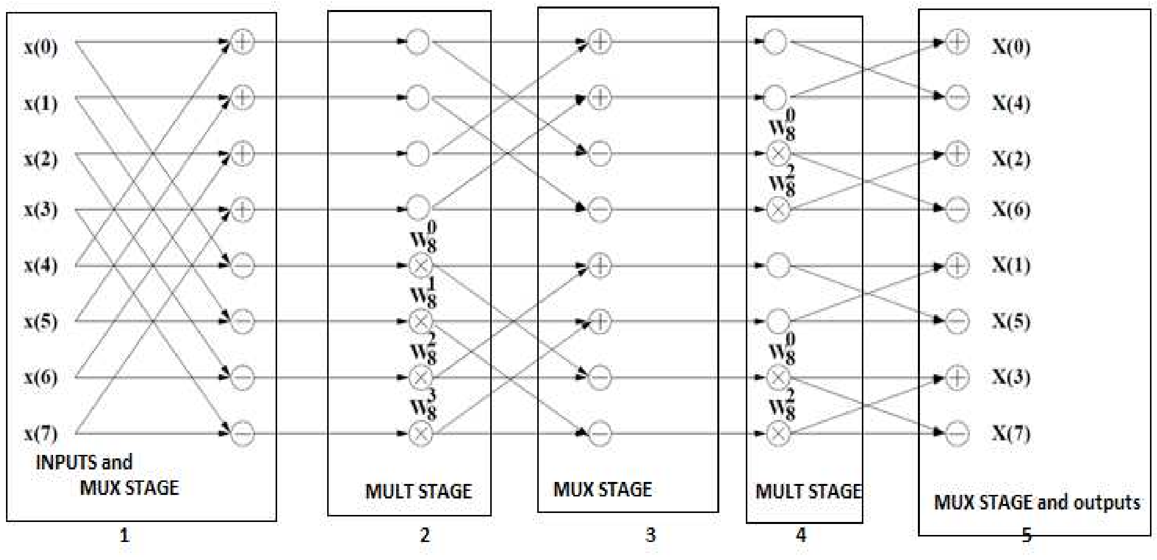

Figure 10: Block Diagram of 5-stage Pipelined Design in Computing FFT

<u>*3.2)* Design of Butterfly Mux (MUX stage)</u>:

The proposed design for MUX Unit in Figure 9 takes eight IEEE single precision floating point inputs fed serially one by one into the unit at each clock cycle. The first butterfly mux block is shown in the Figure 8 and consists of a floating point add/sub unit which computes the complex addition and subtraction on the correct operands. The design of floating point add/sub module is discussed in the subsequent sections.

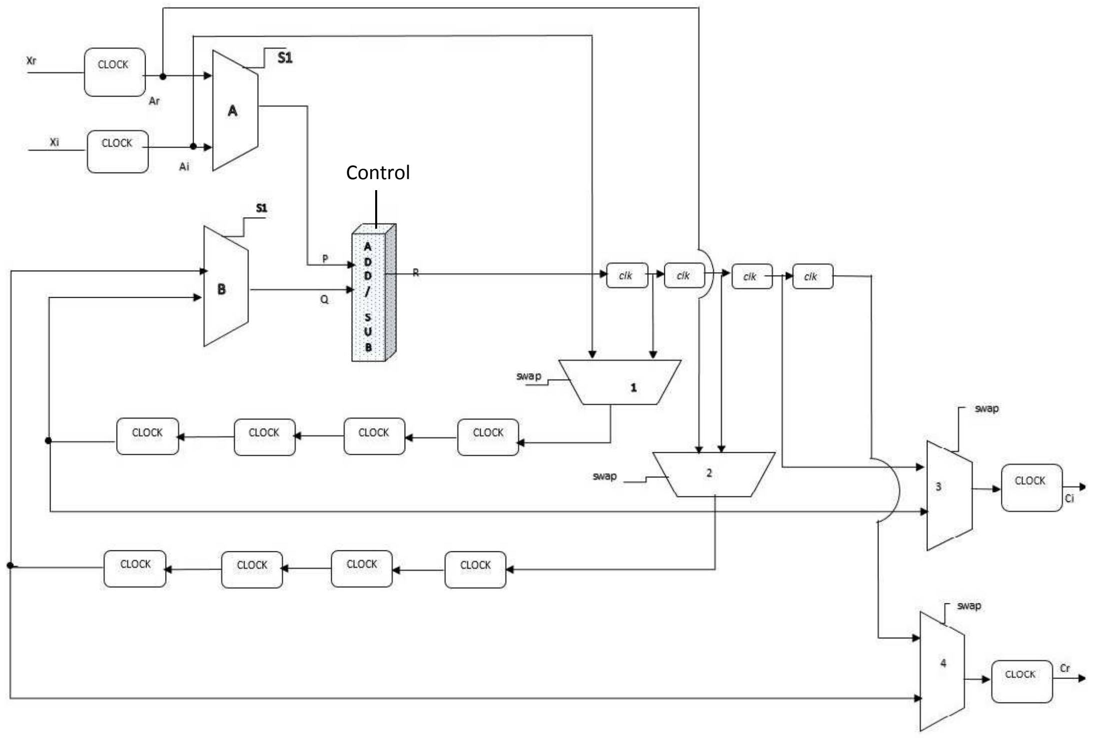

Figure 11: First Mux Stage

There are four feedback registers that help in storing the input operands until the matching operands are reached at input of floating point ADD/SUB block. For example, the input *x0* have to be stored until *x3* is reached at input side to perform floating point add/sub (*x0 + x3, x0 - x3),* and the feedback registers helps in this delaying of inputs. The feedback registers are operated with a clock mentioned "CLOCK" and supposed to be operated with time period represented as T. The internal registers operates with a scaled clock time period T/4 and are represented as CLK. The timing of clock cycles for both internal and external registers are to be properly synchronized to avoid errors in calculation of complex addition and complex subtraction on the correct operands. The clock synchronizing of internal and external registers plays an important role in giving the correct output without errors. Figure 9 gives the clock scaling for the mux stage and also the synchronization of the selection line S1 of first two multiplexers A & B and control signal of floating pointADD/SUB unit.

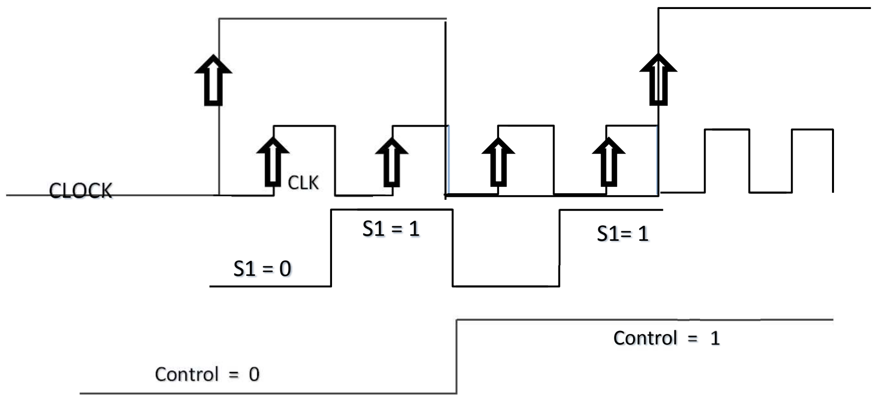

Figure 12: Synchronization of Signals in Mux block

3.2.1) Control Logic for Butterfly Mux: The control logic for the operation of the above MUX stage is given in the Table 6. The control signal *'SWAP'* is the selection line on multiplexers in the MUX stage that swaps the input operands to either the feedback registers or computing the floating point arithmetic results to the next stage. The selection lines 'S1' on the multiplexers at the input side selects either real or imaginary values of input. The 'Swap1', 'Swap2' & 'Swap3' signals are the control signal of mux units in the first, second and third stages, respectively.

negedge (CLOCK/4)	Select(S1)
0	0
0.25	1
0.5	0
0.75	1

negedge(CLOCK/2)	Add/Sub control
0.5	0
1	1

Table 6: Synchronous clocks with varied time periods

negedge (CLOCK)	Swap1	Swap2	Swap3
0	0	X	X
1	0	X	X
2	0	X	X
3	0	X	X
4	0	X	X
5	1	X	X
6	1	X	X
7	1	0	X
8	1	0	X
9	X	1	X
10	X	1	X
11	X	0	0
12	X	0	1
13	X	1	0
14	X	1	1
15	X	X	0
16	X	X	1
17	X	X	0
18	X	X	1

Table 7: Control Logic for Butterfly Mux (adapted from [19])

3.3) Design of Twiddle Factor Multiplication Block:

The twiddle factor multiplication stage multiplies the suitable twiddle factor constant values as shown in Figure 10, with the outputs of the first stage. This stage makes use of an IEEE 754 floating point multiplier and add/sub unit which are discussed later in this chapter. The registers denoted as *'reg'* works with a clock period *T/2.* Therefore, three clocks with time periods *T, T/2, T/4* are used to operate the entire top level design. If (*ar*+ *bi)* is the complex number input, with (*cr*+*di)* being a twiddle factor constant, this stage performs the complex multiplication between the complex input and twiddle factor constant giving the output *(ac-bd)* as real part and *(ad+bc)* as the imaginary part.

It can be observed that this complex number multiplication requires total six floating point operations to be performed (four multiplications {*ac,bd,ad,bc*}, one addition {*ad +bc}* and one subtraction *{ad-bc}*). The two multiplexers at the input side in Figure 10 operate with same selection lines as multiplexers A & B from Figure 8.

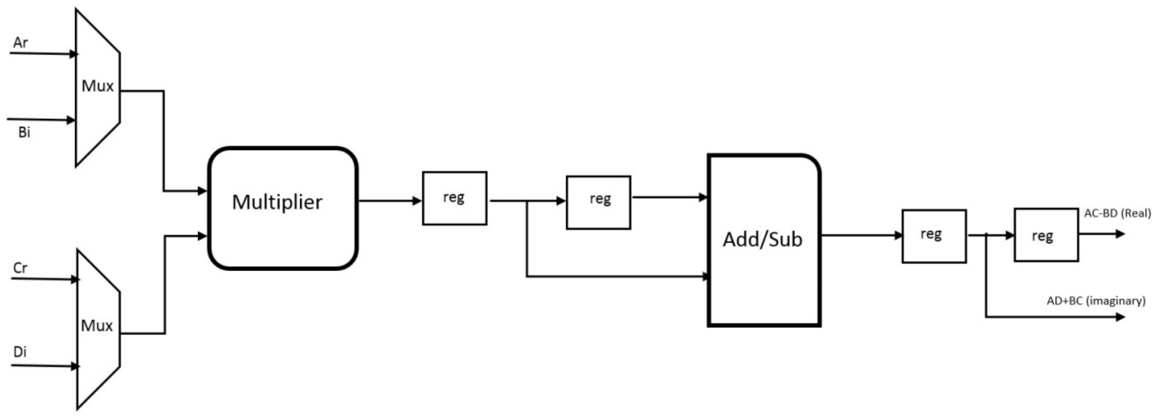

Figure 13: Twiddle Factor Multiplication Architecture

3.4) Floating Point Add/Sub Design:

A traditional floating point add/sub architectures used for Digital signal Processing is utilized for this thesis [9]. The floating point add/sub block is designed using RTL Verilog for the present design and the logic is given in the next sections.

3.4.1) Sign Logic: Using two 32 bit floating point numbers, the most significant bit (32nd) bit is the sign bit of the operands. Based on the sign bit of two operands and the control signal to add/subtract, we can use '*XOR*' gate to perform an actual add/sub operation [20].

Operand A-Sign	Operand B-Sign	Control C	Condition <u>Result</u> <u>XOR of A,B,C</u>	Sign of Output
+	+	Add +	Addition +	+
+	-	Add +	Subtraction -	Greater sign
-	+	Add +	Subtraction -	Greater sign
-	-	Add +	Addition +	-
+	+	Sub -	Subtraction -	Greater sign
+	-	Sub -	Addition +	+
-	+	Sub -	Addition+	-
-	-	Sub -	Subtraction-	Greater sign

Table 8: Logic for Actual Floating Point Add/Sub and Sign Output

<u>3.4.2) Equaling Exponents and Mantissa Shifting:</u> The next step is to obtain the greater of the two operands for an actual floating point subtraction operation. This is done by comparing the exponents and mantissas of each operand and unpack the sign of greater operand. The sign of the greater operand helps in deciding the output sign when actual floating point subtraction is performed. If the exponents are not equal, the difference between the exponents is calculated and the smaller mantissa is shifted towards right to the count equal to difference of exponents. Generally, barrel shift registers are used to shift the mantissa in floating point arithmetic. Barrel shift registers can shift the required number of digits in one clock cycle, however, they pose the problem of hardware and power consumption as described in [18] [20]. Therefore, for a low power implementation of a floating point add/sub module, single shift registers can be used [20]. For the present design implementation, the shift is described in RTL Verilog and the synthesis tool performs the necessary gate level implementation of shifting.

3.4.3) Addition and Normalization on Mantissas: An addition/subtraction operation is done on the mantissas once the exponents are made equal. Normalization of addition result is simple compared to subtraction and always falls into two scenarios, that is, either a carryout generated by an addition or not. If the carry out is generated, carry is discarded and the remaining bits becomes resultant mantissa. If the carry is not generated the bits after the most significant 1 becomes output mantissa. This reduces the burden of normalization in addition as the search for most significant '1'is easy and is available readily in first two MSB places.

3.4.4) Subtraction and Normalization on Mantissas: For subtraction, the resultant mantissas may have the significant 1 in any one of the 27 bit positions (normalizing 1 bit, 23 bits mantissa, extra 3bits). The extra 3 bits here, a guard bit, round bit and sticky bit needed for rounding and precision [20]. Finding the leading 1 for normalization of subtraction result, is a complicated function and a leading zero detector (LZD) is generally used to detect this leading '1' bit in the result. More efficient algorithms available for LZD designs [21] can be employed while designing floating point add/sub units for optimum performance.

The resultant mantissa must be normalized by left shift until the leading '1' is made the most significant bit. The count of left shifting is subtracted from the output exponents to get the final result of exponent. This gives the final normalized mantissa and final result of exponent in subtraction. The sign bit of output is calculated from the condition of addition/subtraction and the greater operand sign while comparing the exponents and mantissas. The advanced approaches for designing low power floating point add/subtract units which makes FFT computation more power efficient are found in [22][23].

3.5) Floating Point Multiplication [20]: Architecture of floating point multiplication unit is also a key factor in deciding performance of the entire FFT. The floating point multiplication for the thesis is implemented in a traditional way described as follows:

3.5.1) Output Sign: The sign bit output when two floating point numbers are multiplied is the XOR of sign bits of the input operands.

3.5.2) Output Exponent: When two floating point numbers are multiplied, the resultant exponent is the sum of exponents of input operands. As the exponent in floating point representation is biased to decimal 128 (binary 8'b01111111), the addition of two floating point exponents adds the biasing value twice and care must be taken to remove the double biasing addition. Therefore, the exponent of the operands when added, the output result is subtracted with 128 (8'b01111111) to cancel the double addition of biasing value 128.

3.5.3) Output Mantissa: The mantissas of input operands are multiplied using 24*24 bit multiplier. The 24 bits of each operand include the normalizing bit 1' and the actual 23 mantissa bits of input operand. When two 24 bit operands are multiplied, the resultant product is a 48 bit binary output which determines the final mantissa and exponent. If the 47th bit in the product is '1' the final output mantissa is [46:24], else it is [45:23], the remaining lower order 23 bits [22:0] are neglected to preserve the floating point format though the product loose it's precision. If the 47th bit is 1, the exponent is increased by 1, otherwise the exponent remains the same as it is calculated in the previous step. If the input is given as all 0's, the design uses the proper exponent value to preserve the value of exponent biasing.

CHAPTER IV

RESULTS

4.1) RTL Description of the Design: All the stages of the hardware architecture presented in Chapter 3 are implemented in the Verilog hardware description language following the IEEE Verilog standard 1364-2005 [35]. The test bench for the design written in Verilog includes all the control logic for SWAP control, Select S1, S2, S3 signals of mux blocks in three stages and clock timing of the circuit. The switching activity information required for the power calculation is captured using a *.vcd* file while testing the design using a test bench.

4.2) Waveform & Simulation Results: Mentor Graphics Model Sim software [32] is used for the simulation and logical verification of the design. The output simulation waveform for a set of sample test vectors (show in hexadecimal format) is given in the Figure 14. The inputs and corresponding outputs obtained are shown in Table 8 and are compared with MATLAB outputs.

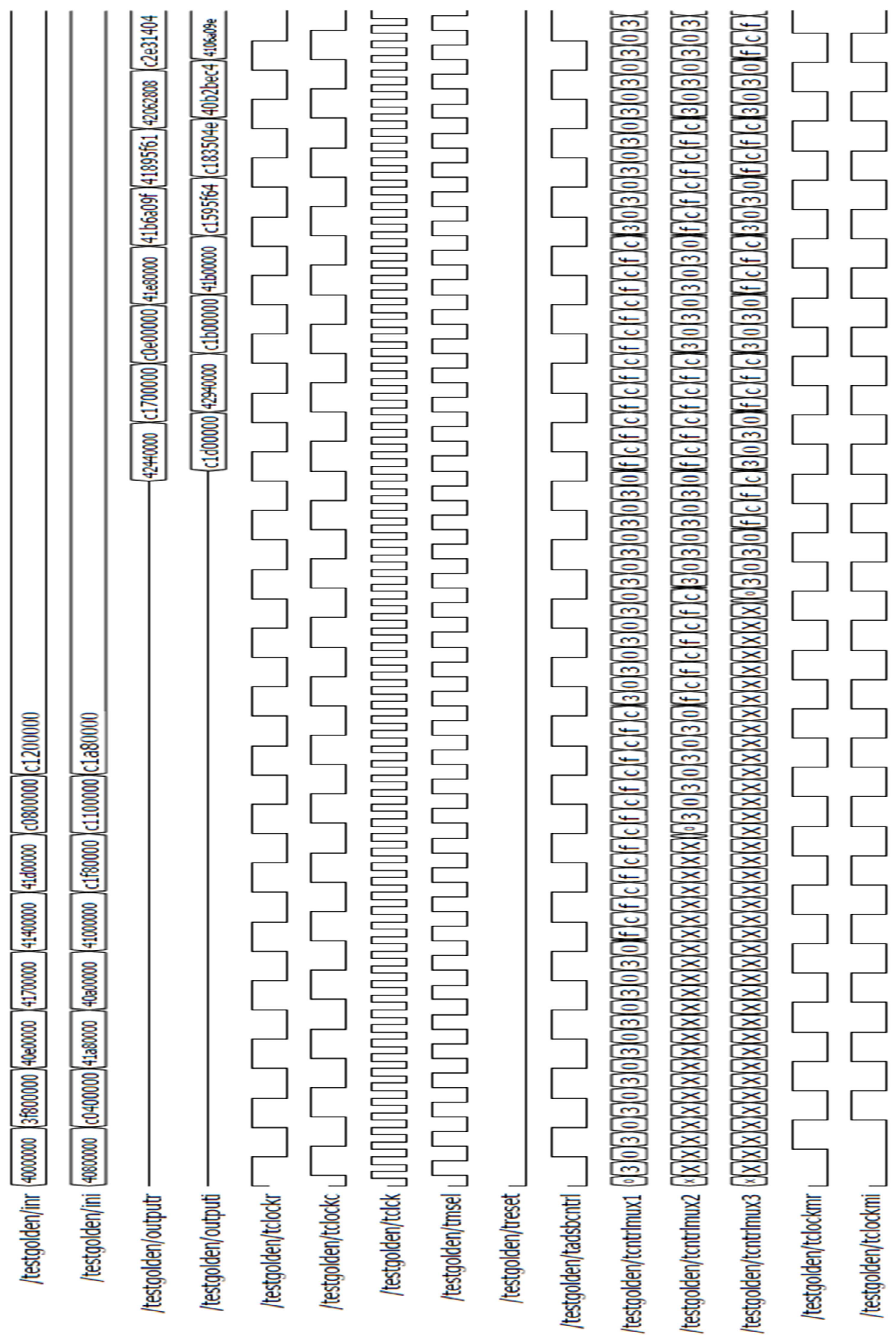

Figure 14: Simulation Output Waveform of the Design

Test Inputs	Design Simulation outputs	MATLAB outputs
32'h40000000 + 32'h40800000 i (2+4i)	32'h42440000 + 32'hC1D00000 i (49-26i)	49.00 – 26.00i
32'h3f800000 + 32'hc0400000 i (1-3i)	32'hC1700000 + 32'h42940000 i (-15+74i)	-15.00+74.00i
32'h40e00000 + 32'h41A80000 i (7+21i)	32'hC0E00000 + 32'hC1B00000 i (-7-22i)	-7.00 - 22.00i
32'h41700000 + 32'h40A00000 i (15+5i)	32'h41E80000 + 32'h41B00000 i (29+22i)	29.00+22.00i
32'h41400000 + 32'h41000000 i (12+8i)	32'h41B6A09F+ 32'hC1595f64 i (22.828428 -13.585789i)	22.83 -13.59i
32'h41D00000 + 32'hC1F80000 i (26-31i)	32'h41895f61 + 32'hC183504E i (17.171572-16.414211i)	17.17-16.41i
32'hC0800000 + 32'hC1100000 i (-4.0 -9.0i)	32'h42062808 + 32'h40B2BEC4 i (33.539093+5.585787i)	33.54+5.59i
32'hC1200000 + 32'hC1A80000 i (-10.0 -21.0i)	32'hC2E31404 + 32'h4106A09E i (-113.53909+8.414213i)	-113.54+8.41i

Table 8: Design Outputs Compared with MATLAB Results

4.3) Synthesis® with Design Compiler™:

The Synopsys® Design Compiler ™ [34] is used to synthesis and optimize the RTL high level description of the design. The RTL is top-down compiled using the Design Compiler. Pre-defined Standard library Cells 'gscl45nm' [27] is used to compile the design. The gate level net list generated is tested using the test bench. The frequency of the design is limited between 20 to 30 MHz and is primarily dependent on the frequency of functional blocks that are floating point add/sub and multiplication units in the design. Power reports of the design are generated by monitoring the switching activity of the design using the VCD file. The design is expected to run with greater performance and better power efficiency when a low power floating point adder and multiplier designs are utilized [25] [26].

4.4) Power Estimation with Synopsys® Primetime™ [33]*:*

Synopsys Prime time tool is used to estimate the static and dynamic power consumption of the design. The switching activity of the design is obtained as follows

4.4.1) VCD (Value Change Dump) File for Switching Activity:

The change in the values of the signals in the design can be captured using the *$dumpvar* Verilog Task function in Test bench. While running the test simulation of the design, the VCD file captures the time and value of the transition that occurs in signals in the design. The VCD file has the information for the toggling activity of all signals and also the time when the toggling occurs. This file is used in Synopsys® Prime Time™ to read the toggling activity of net lists into the tool for calculating power consumed during dynamic switching of the cells in the design.

Also, the switching activity of the design is monitored by giving 500 test vectors to the test bench. The 500 test vectors are randomly generated using a C programming and the code used for generating random test vectors is given in appendix.

4.4.2) SAIF (Switching Activity Interchange Format):

The Switching activity file gives almost the same information as VCD file except that a SAIF file does not record the dynamic timing when the toggling activity occur. SAIF files captures only the switching activity and duration of switching, thus calculating only average power. Synopsys Prime time tool handles both VCD and SAIF files for power optimization and timing analysis.

4.5) Library Specifications and Operating Conditions:

Synopsys Design compiler [33] is utilized to generate a gate level net list of the design. The technology library used for logical mapping of the design is gscl45nm and the attributes of the library are given below.

4.5.1) Library Unit Attributes:

Time unit: “1nano second”

Voltage unit: “1Volt”

Current Unit: “1 micro Ampere”

Pulling resistance unit: “1kohm”

Dynamic Power unit: “1 milli watt”

Leakage Power unit: “1nano watt”

Capacitive load unit: “1 pico farad”

4.5.2) Operating Conditions:

Voltage = 1.1 Volts

Temperature = 27

4.6) Power Report:

Cell Internal Power	= 6.4694 mW	(54%)
Net Switching Power	= 5.4361 mW	(46%)
Total Dynamic Power	= 11.9055 mW	(100%)
Cell Leakage Power	= 102.9736 uW	

4.7) Power Comparison: Power results for the given pipelined architecture are compared with a parallel architecture in [16] and are given Table 10. It can be observed from Table 10 that the power efficiency is 10 times better compared to the parallel and combinational architecture in [16]. More importantly, optimized and low power results can be achieved by using power efficient floating point add/sub [23] [24] and multiplication blocks [25] [26] within the pipelined design.

	Simple FFT [16]	Low power FFT [16] (mw)	Pipelined FFT (mw)
Cell internal Power	428.5302 mw	68.4336 mw	6.4694 mw
Net Switching power	309.0902 mw	44.6828 mw	5.4361 mw
Total Dynamic Power	737.62 mw	113.1164mw	11.905 mw

Table 10: Power Results Compared with Earlier Designs

4.8) Performance Comparison: The latency of the presented pipelined design for an N point is estimated by generating a formula for total number of clock cycles required in complete computation of N point FFT . The equation formulated is given in (4.1)

$$Total\ clock\ cycles\ for\ N\ point\ FFT = 2 * (N + log_2 N - 1) \tag{4.1}$$

Input points	8 Point	16 Point	32 Point	64 Point	128 Point	256 Point	512 Point	1024 Point
Stage Number	Clock Cycles required at each decimation stage							
Stage 10								1
Stage 9							1	2
Stage 8						1	2	4
Stage 7					1	2	4	8
Stage 6				1	2	4	8	16
Stage 5			1	2	4	8	16	32
Stage 4		1	2	4	8	16	32	64
Stage 3	1	2	4	8	16	32	64	128
Stage 2	2	4	8	16	32	64	128	256
Stage 1	4	8	16	32	64	128	256	512
Stage 0	8	16	32	64	128	256	512	1024
Twiddle stages	2	3	4	5	6	7	8	9
Mux stages	3	4	5	6	7	8	9	10
Total Clock Cycles required	20	38	72	138	268	526	1040	2066

Table 11: Clock Latency for Higher Point FFTs

From Table 11, the total clock latency of a 1024 point FFT is 2,066 clock cycles which is efficient when compared to the designs given in [28] with latency 5,220 clock cycles. Though the frequency of operation in [28] is greater (100 MHz) compared to the presented design, the total performance of the design in [28] lags because of the greater computation cycles utilized. The performance comparison for both designs is given in Table 11 expecting the pipelined design to run at the same frequency as in [28]. The formula of processing time for computation given in Equation (4.2).

$$Performance = \frac{Latency\ in\ clock\ cycles}{Frequency\ of\ operation} \tag{4.2}$$

	Traditional FFT processors	FFT processors in [28]	Pipelined FFT processor
Frequency	≥ 100 MHz	≥ 150 MHz	≥ 150 MHz (if expected)
Number of cycles	10240	5220	2066
Processing time	102.4 µs	34. 8 µs	13.77 µs

Table 12: Performance Comparison to Previous Design.

Better processing times can be achieved for the current design if the frequency of the design is improved by improving the speeds of floating point arithmetic units.

CHAPTER V

CONCLUSION

This thesis presents the hardware realization of synchronous design for floating point FFT computation that consumes less power than the parallel architecture in [16]. The clock latency of the proposed design is very efficient computing the complete 1024 point FFT in 2066 clock cycles compared to that in [28]. Future work can be address the power consumption of the overall architecture by designing a control unit which monitors and stops the underlying unnecessary computations when the design is not in use. Clock gating techniques and power gating techniques given in [13] [14] can also be implemented to reduce the switching activity of the design as this significantly reduces the dynamic power consumption. Using power efficient functional blocks such as low power dual path floating point fused addition/subtraction in [22] [23] [24] and low power floating multiplication units in [25] [26], the design can be further improved for better performance and power efficiency.

REFERENCES

[1] P.Lynn and W.Fuerst, *Introductory digital signal processing with computer applications*. Chichester: Wiley, 1989.

[2] W.Kester, *Mixed-signal and DSP design techniques*. Amsterdam: Newnes, 2003.

[3] W.Smith and J.Smith, *Handbook of real-time fast Fourier transforms*. New York: IEEE Press, 1995.

[4] Fftw.org, 'FFTW Home Page', 2015. [Online]. Available: http://www.fftw.org/. [Accessed: 09- Nov- 2015].

[5] D.Orozco, L.Xue, M.Bolat, X.Li and G.Gao, 'Experience of Optimizing FFT on Intel Architectures', *2007 IEEE International Parallel and Distributed Processing Symposium*, 2007.

[6] Texas Instruments, 'FFT Implementation on the TMS320VC5505, TMS320C5505, and TMS320C5515 DSPs', Texas Instruments, 2013.

[7] '754-2008', *754-2008 - IEEE Standard for Floating-Point Arithmetic*, no. 978-0-7381-6981-1, pp. 1 - 82, 2008.

[8] S.Smith, *The scientist and engineer's guide to digital signal processing*. [San Diego, Calif.]: California Technical Pub., 2002.

[9] Wikipedia, 'Logic family', 2015. [Online]. Available: https://en.wikipedia.org/wiki/Logic_family.

[10] N.Weste, D.Harris and N.Weste, *CMOS VLSI design*. Boston: Pearson/Addison-Wesley, 2005.

[11] V.D.Agrawal and R.Srivaths, 'Low-Power Design and Test, Dynamic and Static Power in CMOS', Hyderabad, July 30-31 2007, 2007.

[12] J.Stine, 'VLSI Digital System Design', Oklahoma State University, 2013.

[13] E.Macii, L.Bolzani, A.Calimera, A.Macii and M.Poncino, 'Integrating Clock Gating and Power Gating for Combined Dynamic and Leakage Power Optimization in Digital CMOS Circuits', *2008 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools*, 2008.

[14] Changbo Long and Lei He, 'Distributed sleep transistor network for power reduction', *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 9, pp. 937-946, 2004.

[15] Lehigh University Lehigh Preserve, 'Parallel FFT algorighms and architectures by Xiaofeng Pen', Lehigh University, 2015.

[16] U.Ghate, A.Gurjar and V.Ghate, 'Power optimization of single precision floating point FFT design using fully combinational circuits', *2013 15th International Conference on Advanced Computing Technologies (ICACT)*, pp. 1-5, 2013.

[17] C.Ramamoorthy and H.Li, 'Pipeline Architecture', *CSUR*, vol. 9, no. 1, pp. 61-102, 1977.

[18] K.Dabbagh-Sadeghipour and M.Eshghi, 'A self-timed, pipelined floating point FFT processor architecture', *SCS 2003. International Symposium on Signals, Circuits and Systems. Proceedings (Cat. No.03EX720)*, 2003.

[19] J.Stine, 'IEEE 754 Single precision FFT', Oklahoma State University, 2013.

[20] B.Parhami, *Computer arithmetic*. New York: Oxford University Press, 2000.

[21] G.Dimitrakopoulos, K.Galanopoulos, C.Mavrokefalidis and D.Nikolos, 'Low-Power Leading-Zero Counting and Anticipation Logic for High-Speed Floating Point Units', *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 7, pp. 837-850, 2008.

[22] J.Min, J.Sohn and E.Swartzlander, 'A low-power dual-path floating-point fused add-subtract unit', *2012 Conference Record of the Forty Sixth Asilomar Conference on Signals, Systems and Computers (ASILOMAR)*, 2012.

[23] R.Pillai, D.Al-Khalili and A.Al-Khalili, 'A low power approach to floating point adder design', *Proceedings International Conference on Computer Design VLSI in Computers and*

Processors, 1997.

[24] S.Kukati, D.Sujana, S.Udaykumar, P.Jayakrishnan and R.Dhanabal, 'Design and implementation of low power floating point arithmetic unit', *2013 International Conference on Green Computing, Communication and Conservation of Energy (ICGCE)*, 2013.

[25] H.Zhang, W.Zhang and J.Lach, 'A low-power accuracy-configurable floating point multiplier', *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, 2014.

[26] N.Babu and R.Sarma, 'A novel low power and high speed Multiply-accumulate (MAC) unit design for floating-point numbers', *2015 International Conference on Smart Technologies and Management for Computing, Communication, Controls, Energy and Materials (ICSTM)*, 2015.

[27] Eda.ncsu.edu, 'FreePDK - NCSU EDA Wiki', 2015. [Online]. Available: http://www.eda.ncsu.edu/wiki/FreePDK.

[28] S.Mou and X.Yang, 'Design of a High-speed FPGA-based 32-bit Floating-point FFT Processor', *Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007)*, 2007.

[29] K.Lasith and A.Thomas, 'Efficient implementation of single precision floating point processor in FPGA', *2014 Annual International Conference on Emerging Research Areas: Magnetics, Machines and Drives (AICERA/iCMMD)*, 2014.

[30] J.Cooley and J.Tukey, 'An Algorithm for the Machine Calculation of Complex Fourier Series', *Mathematics of Computation*, vol. 19, no. 90, p. 297, 1965.

[31] Mathworks.com, 'MATLAB - The Language of Technical Computing', 2015. [Online]. Available: http://www.mathworks.com/products/matlab/.

[32] Altera.com, 'ModelSim-Altera Software', 2015. [Online]. Available: https://www.altera.com/products/design-software/model---simulation/modelsim-altera-software.tablet.html.

[33] Synopsys, 'Synopsys PrimeTime', 2015. [Online]. Available: http://www.synopsys.com/Tools/Implementation/SignOff/Pages/PrimeTime.aspx.

[34] Synopsys, 'Design Compiler - Synopsys', 2015. [Online]. Available:

http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler/Pages/default.aspx.

[35] IEEE Member Digital Library', *IEEE Transactions on Power Electronics*, vol. 19, no. 5, pp. 1364-1364, 2004.

APPENDICES

A-1) <u>Verilog design of floating point multiplication:</u>

```
module spmult (out,a,b);

output [31:0]out;

wire [31:0]spproduct;  // output declaration of 32 bit number

input [31:0]a,b; // input declarations of two 32 bit numbers

// internal nets

wire [47:0]prtemp; //temporary net used for storing 48bits in the product before truncation

wire [7:0]pexptemp; //temporary exponent used before checking the most significand bit in the
product

//RTL Verilog code using ASSIGN statememts only

assign spproduct[31] = a[31] ^ b[31];  // sign bits of inputs are operated seperately to get the sign
bit of product

assign pexptemp = a[30:23] + b[30:23] - 8'b01111111;  // extra bias (127) that is added twice is
removed once to get the actual bias

assign prtemp[47:0] = {1'b1,a[22:0]} * {1'b1,b[22:0]};  // 1 is appended to the 23 bit mantissas to
multiply the significands perfectly and get the 48bit result

assign spproduct[22:0] = prtemp[47]==1'b1 ? prtemp[46:24] : prtemp[45:23];//if 47th position(i.e
48th bit) is 1,mantissa is from 46 to 24 positions else 45 to 23

assign spproduct[30:23] = prtemp[47]==1'b1 ? pexptemp+1'b1 : pexptemp; //if 48th bit in the
product is 1,47th bit is shifted to after the floating point, hence exp increased by 1y

assign
out[31:0]=(a==32'h00000000)?32'h01800000:(b==32'h00000000)?32'h01800000:spproduct[31:0
];

endmodule
```

A-2) Verilog design of floating point ADD/SUB :

```
module subadd(out,a,b,cntrl); //toplvl module
output [31:0]out; //final ouptut
input [31:0]a,b; //input IEEE 754 numbers
input cntrl; //cntrl 0_adds 1_subtracts
wire cndtn,sign; //actual add or sub
wire [31:0] grt,smll;
wire [7:0] preexp,nadexp,nsbexp;
wire [27:0]add,sub,nrmsm;
wire [22:0]nrmam;
wire [23:0]newm;
expshift t(grt[31:0],smll[31:0],preexp[7:0],a[31:0],b[31:0],sign); //exponent comparisionn
shift s(newm[23:0],grt[30:23],smll[30:23],smll[22:0]);//shifting exponents
addition d(add[27:0],{1'b1,grt[22:0]},newm[23:0]);//addition without normalizing
subtraction q(sub[27:0],grt[22:0],newm[23:0]); //subtraction without normalizing
normaladd an(nrmam,nadexp,add[27:0],preexp[7:0]); //normalizing addition output
normalsubs sb(nrmsm,nsbexp,sub[27:0],preexp[7:0]);//normalizing subtraction output
assign cndtn=cntrl^a[31]^b[31];//xor operation tells whether actually to add or subtract
assign out[22:0]=cndtn?nrmsm[27:5]:nrmam[22:0];//assigning mantissa according to condtn
assign out[30:23]=cndtn?nsbexp[7:0]:nadexp[7:0];//assigning exponents according to condtn
//logic used to get the sign bit when subtraction is done
assign out[31]=(cntrl==1'b0)?grt[31]:(a[31]==1'b0 && b[31]==1'b1)?1'b0:(a[31]==1'b1 &&
b[31]==1'b0)?1'b1:(a[31]&&b[31]&&sign)?1'b0:(a[31]==1'b0 && b[31]==1'b0 &&
sign==1'b0)?1'b0:1'b1;//sign is always greater number sign
endmodule

//comparing the exponents for further shifting
module expshift(grt,smll,reexp,a,b,sign);
input [31:0]a,b;
output[31:0]grt,smll;
output [7:0]reexp;
output sign;
assign smll[31:0]=(a[30:0]<b[30:0])?a:b;
assign grt[31:0]=(a[30:0]>b[30:0])?a:b;
assign reexp[7:0]=grt[30:23];
assign sign=(grt[31:0]==a[31:0])?1'b0:1'b1;
endmodule

//shfting exponents
module shift(newm,grtexp,smllexp,smm);//equaling the exponents
input [22:0]smm;
input [7:0]grtexp,smllexp;
output [23:0]newm;
```

```
wire  [31:0]count;
assign count=grtexp-smllexp; //count is used to shift the mantissa and make expopents equal
assign newm[23:0]={1'b1,smm[22:0]}>>count;//shifting the mantissas to equalize exponents
endmodule

//module for adding mantissa
module addition(outadd,a,b);
output [27:0]outadd;
input [23:0]a,b;
assign outadd[27:0]={a[23:0],3'h0}+{b[23:0],3'h0};
endmodule

//subtraction operation
module subtraction(outsub,a,b);
output[27:0]outsub;//output
input [22:0]a;//input mantissas (equal exponents)
input [23:0]b;
wire [27:0]reneg;//28th bit to check whether the result is negative
wire [27:0]tempaa;
assign reneg[27:0]={1'b0,1'b1,a[22:0],3'b0}-{1'b0,b[23:0],3'b0};
assign tempaa=0-reneg[27:0];//twos complement of result
assign outsub=(reneg[27]==1'b1)?tempaa:reneg[27:0];//twoscomplement is output if result is -
endmodule

//normalizing the sum mantissa
module normaladd (newam,newaexp,oldman,oldexp);
output [22:0]newam;
output [7:0]newaexp;
input [27:0]oldman;
input [7:0]oldexp;
assign newam[22:0]=(oldman[27]==1'b1)?oldman[26:4]:oldman[25:3];
assign newaexp=(oldman[27]==1'b1)?oldexp+1:oldexp;//increasing the exponent
endmodule

//normalizing the subtraction mantissas
module normalsubs(newman,newexp,inman,oldexp);
output [7:0]newexp;
output [27:0]newman;
input [27:0]inman;
input [7:0]oldexp;
assign
newman[27:0]=(inman[27]==1'b1)?inman[27:0]<<32'd1:(inman[26]==1'b1)?inman[27:0]<<32'd
2:(inman[25]==1'b1)?inman[27:0]<<32'd3:(inman[24]==1'b1)?inman[27:0]<<32'd4:(inman[23]=
=1'b1)?inman[27:0]<<32'd5:(inman[22]==1'b1)?inman[27:0]<<32'd6:(inman[21]==1'b1)?inman[
27:0]<<32'd7:(inman[20]==1'b1)?inman[27:0]<<32'd8:(inman[19]==1'b1)?inman[27:0]<<32'd9:(
inman[18]==1'b1)?inman[27:0]<<32'd10:(inman[17]==1'b1)?inman[27:0]<<32'd11:(inman[16]=
=1'b1)?inman[27:0]<<32'd12:(inman[15]==1'b1)?inman[27:0]<<32'd13:(inman[14]==1'b1)?inma
n[27:0]<<32'd14:(inman[13]==1'b1)?inman[27:0]<<32'd15:(inman[12]==1'b1)?inman[27:0]<<3
2'd16:(inman[11]==1'b1)?inman[27:0]<<32'd17:(inman[10]==1'b1)?inman[27:0]<<32'd18:(inma
n[9]==1'b1)?inman[27:0]<<32'd19:(inman[8]==1'b1)?inman[27:0]<<32'd20:(inman[7]==1'b1)?in
man[27:0]<<32'd21:(inman[6]==1'b1)?inman[27:0]<<32'd22:(inman[5]==1'b1)?inman[27:0]<<3
2'd23:(inman[4]==1'b1)?inman[27:0]<<32'd24:(inman[3]==1'b1)?inman[27:0]<<32'd25:(inman[
2]==1'b1)?inman[27:0]<<32'd26:(inman[1]==1'b1)?inman[27:0]<<32'd27:32'd0;
assign
newexp[7:0]=(inman[27]==1'b1)?oldexp+32'd1:(inman[26]==1'b1)?oldexp:(inman[25]==1'b1)?o
ldexp-32'd1:(inman[24]==1'b1)?oldexp-32'd2:(inman[23]==1'b1)?oldexp-
32'd3:(inman[22]==1'b1)?oldexp-32'd4:(inman[21]==1'b1)?oldexp-
32'd5:(inman[20]==1'b1)?oldexp-32'd6:(inman[19]==1'b1)?oldexp-
32'd7:(inman[18]==1'b1)?oldexp-32'd8:(inman[17]==1'b1)?oldexp-
32'd9:(inman[16]==1'b1)?oldexp-32'd10:(inman[15]==1'b1)?oldexp-
32'd11:(inman[14]==1'b1)?oldexp-32'd12:(inman[13]==1'b1)?oldexp-
32'd13:(inman[12]==1'b1)?oldexp-32'd14:(inman[11]==1'b1)?oldexp-
32'd15:(inman[10]==1'b1)?oldexp-32'd16:(inman[9]==1'b1)?oldexp-
32'd17:(inman[8]==1'b1)?oldexp-32'd18:(inman[7]==1'b1)?oldexp-
32'd19:(inman[6]==1'b1)?oldexp-32'd20:(inman[5]==1'b1)?oldexp-
32'd21:(inman[4]==1'b1)?oldexp-32'd22:(inman[3]==1'b1)?oldexp-
32'd23:(inman[2]==1'b1)?oldexp-32'd24:(inman[1]==1'b1)?oldexp-
32'd25:(inman[0]==1'b1)?oldexp-32'd26:8'b00000000;
endmodule
```

A-3) Verilog design of first MUX Stage

```
//module of first mux unit first stage
module firstmux(ar,ai,cr,ci,select,swap1,reset,cntrl,clock,clck);
output [31:0]cr,ci; //outputs of first mux
input [31:0]ar,ai; //inputs to first mux
input select,swap1; //selection lines that control simple 2x1 muxes used in MUX unit
input reset; //reset pin of all the flipflops used
input clock,clck; //clockr operates on real values, clockc on imaginary and clck on internal regs
input cntrl; //cntrl is when to add or when to subtract
wire [31:0]ar1,ac1,br,bi,p,q,r,dr,di,q1,q2,q3,q4,d1r,d2r,d3r,d1c,d2c,d3c,cr1,ci1; //intermediate
outputs joined as wire between blocks

//simple muxes used (4 totally)
assign dr=(swap1==1'b1)?q2:ar1;
assign di=(swap1==1'b1)?q1:ac1;
assign cr1=(swap1==1'b1)?q4:br;
assign ci1=(swap1==1'b1)?q3:bi;
assign p=(select==1'b1)?ac1:ar1;
assign q=(select==1'b1)?bi:br;
subadd t1(r,q,p,cntrl);
//input flipflops
D_flpflop arin(ar,ar1,clock,reset);
D_flpflop acin(ai,ac1,clock,reset);

//internal Feed back registers
//feedbackr sum_diff(r,q4,reset,clck,q1,q2,q3);
D_flpflop am(r,q1,clck,reset);
D_flpflop an(q1,q2,clck,reset);
D_flpflop ap(q2,q3,clck,reset);
D_flpflop aq(q3,q4,clck,reset);

//External feed back registers real
//feedbackr extr(dr,br,reset,clock,d1r,d2r,d3r);
D_flpflop er1(dr,d1r,clock,reset);
D_flpflop er2(d1r,d2r,clock,reset);
D_flpflop er3(d2r,d3r,clock,reset);
D_flpflop er4(d3r,br,clock,reset);

//External feedback registers - imaginary
//feedbackr exti(di,bi,reset,clock,d1c,d2c,d3c);
D_flpflop ei1(di,d1c,clock,reset);
D_flpflop ei2(d1c,d2c,clock,reset);
D_flpflop ei3(d2c,d3c,clock,reset);
D_flpflop ei4(d3c,bi,clock,reset);

//output fliflops
D_flpflop cre(cr1,cr,clock,reset);
D_flpflop cim(ci1,ci,clock,reset);
endmodule
```

A-4) Verilog design of Second MUX Stage:

```
//module of second mux unit second stage
module secondmux(ar,ai,cr,ci,select,swap2,reset,cntrl,clock,clck);
output [31:0]cr,ci; //outputs of first mux
input [31:0]ar,ai; //inputs to first mux
input select,swap2; //selection lines that control simple 2x1 muxes used in MUX unit
input reset; //reset pin of all the flipflops used
input clock,clck; //clockr operates on real values, clockc on imaginary and clck on internal regs
input cntrl; //cntrl is when to add or when to subtract
wire [31:0]ar1,ac1,br,bi,p,q,r,dr,di,q1,q2,q3,q4,d1r,d2r,d3r,d1c,d2c,d3c,cr1,ci1; //intermediate
outputs joined as wire between blocks
//registers before mux starting that takes in input ai and ar
D_flpflop arin(ar,ar1,clock,reset);
D_flpflop acin(ai,ac1,clock,reset);
//simple muxes used (4 totally)
assign p=(select==1'b1)?ac1:ar1;
assign q=(select==1'b1)?bi:br;

//internal serial registers working on clck (high frequency) to store add/sub real/cmp values
subadd t1(r,q,p,cntrl);
//feedbackr sum_diff(r,q4,reset,clck,q1,q2,q3);
D_flpflop am(r,q1,clck,reset);
D_flpflop an(q1,q2,clck,reset);
D_flpflop ap(q2,q3,clck,reset);
D_flpflop aq(q3,q4,clck,reset);
//one adder/subtractor used
assign dr=(swap2==1'b1)?q2:ar1;
assign di=(swap2==1'b1)?q1:ac1;
assign cr1=(swap2==1'b1)?q4:br;
assign ci1=(swap2==1'b1)?q3:bi;
//muxes at the inputs using selection
//smplemux ia(ar1,ac1,p,select);
//smplemux ib(br,bi,q,select);
//feed back flipflops (2 in number)
D_flpflop er1(dr,d1r,clock,reset);
D_flpflop er2(d1r,br,clock,reset);
//feedback registers imaginary
D_flpflop ei1(di,d1c,clock,reset);
D_flpflop ei2(d1c,bi,clock,reset);
//for complex values

//registers used to store the output values after first stage
D_flpflop cre(cr1,cr,clock,reset);
D_flpflop cim(ci1,ci,clock,reset);
endmodule
```

A-5) Verilog design of third MUX Stage:

```
//module of third mux unit third stage
module thirdmux(ar,ai,cr,ci,select,swap3,reset,cntrl,clock,clck);
```

```
output [31:0]cr,ci; //outputs of third mux
input [31:0]ar,ai; //inputs to third mux
input select,swap3; //selection lines that control simple 2x1 muxes used in MUX unit
input reset; //reset pin of all the flipflops used
input clock,clck; //clockr operates on real values, clockc on imaginary and clck on internal regs
input cntrl; //cntrl is when to add or when to subtract
wire [31:0]ar1,ac1,br,bi,p,q,r,dr,di,q1,q2,q3,q4,d1r,d2r,d3r,d1c,d2c,d3c,cr1,ci1; //intermediate
outputs joined as wire between blocks

//registers before mux starting that takes in input ai and ar
D_flpflop arin(ar,ar1,clock,reset);
D_flpflop acin(ai,ac1,clock,reset);
//simple muxes used
assign dr=(swap3==1'b1)?q2:ar1;
assign di=(swap3==1'b1)?q1:ac1;
assign cr1=(swap3==1'b1)?q4:br;
assign ci1=(swap3==1'b1)?q3:bi;
assign p=(select==1'b1)?ac1:ar1;
assign q=(select==1'b1)?bi:br;
//feedbackr sum_diff(r,q4,reset,clck,q1,q2,q3);
//internal serial registers working on clck (high frequency) to store add/sub real/cmp values
D_flpflop am(r,q1,clck,reset);
D_flpflop an(q1,q2,clck,reset);
D_flpflop ap(q2,q3,clck,reset);
D_flpflop aq(q3,q4,clck,reset);
//one adder/subtractor used
subadd t1(r,q,p,cntrl);
//registers used to store the output values after third stage
D_flpflop cre(cr1,cr,clock,reset);
D_flpflop cim(ci1,ci,clock,reset);
//External feed back registers
D_flpflop fdb3a(dr,br,clock,reset);
D_flpflop fdb3b(di,bi,clock,reset);//for complex values
endmodule
```

A-6) Verilog design of 1st Twiddle Factor Multiplication stage

```
module mult1(ar,ai,outmr,outmi,clock,clckm,clck,mcntrl,sltm,sltn,selectm,reset,prod,m1,m2,sb1);
input [31:0]ar,ai;
input  [2:0]selectm;
input clock,clck,clckm,mcntrl,reset;
output [31:0]outmr,outmi;
input  sltm,sltn;
```

```
inout [31:0]prod,m1,m2,sb1;
wire [31:0] c,d,in1,in2;
reg [31:0] w80r=32'h3f800000,w80c=32'h00000000,
w81r=32'h3f800000,w81c=32'h00000000,
 w82r=32'h3f800000, w82c=32'h00000000,
 w83r=32'h3f800000, w83c=32'h00000000,
 w84r=32'h3f800000, w84c=32'h00000000,
 w85r=32'h3f3504f3, w85c=32'hbf3504f3,
w86r=32'h00000000, w86c=32'hbf800000,
 w87r=32'hbf3504f3, w87c=32'hbf3504f3;
assign
c[31:0]=(selectm==3'b000)?w80r:(selectm==3'b001)?w81r:(selectm==3'b010)?w82r:(selectm==
3'b011)?w83r:(selectm==3'b100)?w84r:(selectm==3'b101)?w85r:(selectm==3'b110)?w86r:(selec
tm==3'b111)?w87r:32'h00000000;
 assign
d[31:0]=(selectm==3'b000)?w80c:(selectm==3'b001)?w81c:(selectm==3'b010)?w82c:(selectm==
3'b011)?w83c:(selectm==3'b100)?w84c:(selectm==3'b101)?w85c:(selectm==3'b110)?w86c:(sele
ctm==3'b111)?w87c:32'h00000000;
 assign in1[31:0]=(sltm==1'b0)?ar[31:0]:ai[31:0];
 assign in2[31:0]=(sltn==1'b0)?c[31:0]:d[31:0];
//combinational multiplication block used
spmult sr(prod,in1,in2);
//intermediate products stored
D_flpflop min1(prod,m1,clck,reset);
D_flpflop min2(m1,m2,clck,reset);
//combinational add/sub unit
subadd s1(sb1,m2,m1,mcntrl);
//Dflipflops that store output values
D_flpflop dl(sb1,outmi,clckm,reset);
D_flpflop d2(outmi,outmr,clckm,reset);
Endmodule
```

A-7) Verilog design of 2nd Twiddle Factor Multiplication stage

```
module mult2(ar,ai,outmr,outmi,clock,clckm,clck,mcntrl,sltm,sltn,selectm2,reset);
input [31:0]ar,ai;
input  [2:0]selectm2;
input clock,clckm,clck,mcntrl,reset;
output [31:0]outmr,outmi;
input  sltm,sltn;
wire [31:0]c,d,in1,in2;
reg [31:0] w80r=32'h3f800000,w80c=32'h00000000, w81r=32'h3f800000, w81c=32'h00000000,
 w82r=32'h3f800000, w82c=32'h00000000,
 w83r=32'h00000000, w83c=32'hbf800000,
 w84r=32'h3f800000, w84c=32'h00000000,
 w85r=32'h3f800000, w85c=32'h00000000,
 w86r=32'h3f800000, w86c=32'h00000000,
 w87r=32'h00000000, w87c=32'hbf800000;
wire [31:0]prod,m1,m2,sb1,outar,outai;
spmult sr2(prod,in1,in2);
 assign
c[31:0]=(selectm2==3'b000)?w80r:(selectm2==3'b001)?w81r:(selectm2==3'b010)?w82r:(selectm
2==3'b011)?w83r:(selectm2==3'b100)?w84r:(selectm2==3'b101)?w85r:(selectm2==3'b110)?w86
r:(selectm2==3'b111)?w87r:32'h00000000;
 assign
d[31:0]=(selectm2==3'b000)?w80c:(selectm2==3'b001)?w81c:(selectm2==3'b010)?w82c:(select
m2==3'b011)?w83c:(selectm2==3'b100)?w84c:(selectm2==3'b101)?w85c:(selectm2==3'b110)?
w86c:(selectm2==3'b111)?w87c:32'h00000000;
 assign in1[31:0]=(sltm==1'b0)?ar[31:0]:ai[31:0];
 assign in2[31:0]=(sltn==1'b0)?c[31:0]:d[31:0];
D_flpflop min1(prod,m1,clck,reset);
D_flpflop min2(m1,m2,clck,reset);
subadd s2s(sb1,m2,m1,mcntrl);
D_flpflop dl2(sb1,outmi,clckm,reset);
D_flpflop d22(outmi,outmr,clckm,reset);
endmodule
```

A-8) Verilog design of complete FFT (Top Level Design):

```
//toplevel that has all the 5 pipelined stages
//Top level Module that combines all the blocks
module
toplvl(ar,ai,outr,outi,clock,clckm,clck,select,reset,cntrl,swap1,swap2,swap3,mcntrl,selectm,select
m2,sltm,sltn,or1,oi1,orm1,oim1,or2,oi2,orm2,oim2,prod,m1,m2,sb1);
input [31:0]ar,ai; //input ports
output [31:0]outr,outi; //outports
input clock,clck,clckm;
input select,reset,cntrl,mcntrl,sltm,sltn; //control lines of all the stages
input [2:0]selectm,selectm2; //twidle factor selection line for multiplication
input swap1,swap2,swap3; //selection lines of each mux 1,2,3 (logic table desired)
inout [31:0]or1,oi1,orm1,oim1,or2,oi2,orm2,oim2,prod,m1,m2,sb1; //intermediate outputs at each
stage joined by wire
//instantiation of stages
firstmux mx1(ar,ai,or1,oi1,select,swap1,reset,cntrl,clock,clck);
//first twidle factor multiplication unit
mult1 ma1(or1,oi1,orm1,oim1,clock,clckm,clck,mcntrl,sltm,sltn,selectm,reset,prod,m1,m2,sb1);
//second mux unit
secondmux mux2(orm1,oim1,or2,oi2,select,swap2,reset,cntrl,clock,clck);
```

```
//second stage twidle factor multiplication
mult2 ma2(or2,oi2,orm2,oim2,clock,clckm,clck,mcntrl,sltm,sltn,selectm2,reset);
//final mux unit
thirdmux mux3(orm2,oim2,outr,outi,select,swap3,reset,cntrl,clock,clck);
endmodule
```

A-9) <u>C-Program for Generating 500 Random Vectors as Inputs to Design.</u>

```
//C-program  for generating 500 random inputs to the design

#include <stdio.h>
#include <stdlib.h>

int main()
{
  int i, n;
  time_t t;

  n = 500;

  /* Intializes random number generator */
  srand((unsigned) time(&t));

  /* Print all the random numbers in the given range in hexadecimal format */
  for( i = 0 ; i < n ; i++ ) {
    printf("#8 ar= %08x;\n", rand() % 99999999999);
    printf("   ai=%08x;\n", rand() % 8888888888);
  }

 return(0);
}
```

VITA

Sai Kiran, Sunkari

Candidate for the Degree of

Master of Science

Thesis: LOW POWER SYNCHRONOUS DESIGN OF HARDWARE ARCHITECTURE FOR IEEE 754 SINGLE PRECISION FLOATING POINT FAST FOURIER TRANSFORM

Major Field: Electrical & Computer Engineering

Biographical:

Education: Received Bachelor of Technology Degree in Electronics & Communication Engineering from Jawaharlal Technological University-Anantapur, Andhra in April 2013. Completed the requirements for the Master of Science in Electrical Engineering at Oklahoma State University, Stillwater, Oklahoma in December, 2015.

Experience: Employed by Oklahoma State University, Department of Electrical and Computer Engineering as Lab Teaching Assistant in spring 2015.