LUT optimization for Fast Fourier Transform

A Project Report

submitted by

MOHANKUMAR R

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

MAY 2019

THESIS CERTIFICATE

This is to certify that the thesis titled **LUT optimization for Fast Fourier Transform**, submitted by **MOHANKUMAR R**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology**, is a bonafide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr.K.SRIDHARAN
Research Guide
Professor
Dept. of Electrical Engineering
IIT-Madras, 600 036

Place: Chennai

Date: 5th May 2019

ACKNOWLEDGEMENTS

I would like to express my profound gratitude to my project guide, Dr.K.SRIDHARAN , for giving me the opportunity to work under him, on this project. His vast knowledge, outlook towards research, patience and willingness to help, was instrumental in helping me complete my project.

I would also like to thank my friends, my parents and the faculty at IIT Madras for being a great source of motivation and encouragement.

ABSTRACT

KEYWORDS: LUT optimization, Fast Fourier Transform, Field Programmable Gate Array

Fast Fourier Transform (FFT) remains of a great importance due to its substantial role in the field of signal processing and imagery. Multiple designs of 8 point FFT is proposed using various algorithms . The material resources of the FPGA are limited, particularly the integrated DSP blocks, hence different approaches are used using the Verilog description with the aim to reduce the resource usage.

The experimental validation was done using ISIM simulation tool, where the numerical synthesis and the post and route described in Verilog, was realized using ISE Design Suite 14.7. The FFT modules of all the implementations were tested using a python script against various corner input test cases.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

FPGA	Field-Programmable Gate Array
HDL	Hardware Description Language
ASIC	Application-Specific Integrated Circuit
FFT	Fast Fourier Transform
FHT	Fast Hartley Transform
DFT	Discrete Fourier Transform
OMS	Odd Multiple Storage
APC	Anti-Symmetric Coding
CORDIC	COordinate Rotation DIgital Computer
VHDL	VHSIC Hardware Description Language
LUT	Look-Up Table

CHAPTER 1

Introduction

1.1 What is FPGA?

A field-programmable gate array (FPGA) is an integrated circuit designed to be configured by a customer or a designer after manufacturing âĂŞ hence the term "field-programmable". The FPGA configuration is generally specified using a hardware description language (HDL), similar to that used for an Application-Specific Integrated Circuit (ASIC). Circuit diagrams were previously used to specify the configuration, but this is increasingly rare due to the advent of electronic design automation tools.

FPGA contains an array of programmable logic blocks, and a hierarchy of "reconfigurable interconnects" that allow the blocks to be "wired together", like many logic gates that can be inter-wired in different configurations. Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory. Many FPGAs can be reprogrammed to implement different logic functions, allowing flexible re-configurable computing as performed in computer software.

1.2 What is FFT?

A fast Fourier transform (FFT) is an algorithm that computes the discrete Fourier transform (DFT) of a sequence. Fourier analysis converts a signal from its original domain (often time or space) to a representation in the frequency domain and vice versa. The DFT is obtained by decomposing a sequence of values into components of different frequencies. This operation is useful in many fields, but computing it directly from the definition is often too slow to be practical. An FFT rapidly computes such transformations by factorizing the DFT matrix into a product of sparse (mostly zero) factors.

As a result, it manages to reduce the complexity of computing the DFT from $O(n^2)$, which arises if one simply applies the definition of DFT, to $O(n \log n)$, where n is the data size. The difference in speed can be enormous, especially for long data sets where N may be in the thousands or millions. In the presence of round-off error, many FFT algorithms are much more accurate than evaluating the DFT definition directly. There are many different FFT algorithms based on a wide range of published theories, from simple complex-number arithmetic to group theory and number theory.

1.3 What is a LUT?

A LUT is a Look-Up Table. Modern FPGA's are built out of large arrays of these lookup tables. Using a lookup table, we can build any logic we want ,as long as we don't exceed the number of elements in the lookup table. As an example, the 7-series Xilinx FPGAs are composed of "configurable logic blocks", each of which contain two "slices", of which each of those "slices" contain four 6-input LUTs. Each of these LUT's can handle either one six input lookup, or two five input lookups-as long as the two share the same inputs. The Altera Cyclone IV, on the other hand, has only 4-input LUTs.The point being that every FPGA implements our logic via a combination of LUTs. Chips differ by the capability of their LUTs, as well as by the number of LUTs on board. In general, the more LUTs we have, the more logic your chip can do, but also the more our FPGA chip is going to cost. It's all a trade off. To get the most logic for a given price, the FPGA design engineer needs to be able to code efficiently, and pack their code into the fewest LUTs possible.

1.4 Problem Statement

FPGAs are an efficient hardware target when only small series are needed, or for rapid prototyping. The FPGAs are complex enough to implement more than glue logic, including complex designs up to several thousands gates. As the logic capacity of FPGAs increases, LUT optimization is becoming more important. FPGA's resources are limited , hence optimal utilization of resources is important. FFT units can be optimized in various ways by compromising on factors like speed , input type , etc.. .

CHAPTER 2

Literature Review

2.1 LUT Optimization for Memory-Based Computation

Semiconductor memory has become cheaper, faster, and more power-efficient. The transistor packing density of memory components is not only higher but also increasing much faster than those of logic components. Apart from that, memory-based computing structures are more regular than the multiply - accumulate structures and offer many other advantages, e.g., greater potential for high-throughput and low-latency implementation and less dynamic power consumption. Memory-based computing is well suited for many digital signal processing (DSP) algorithms, which involve multiplication with a fixed set of coefficients.

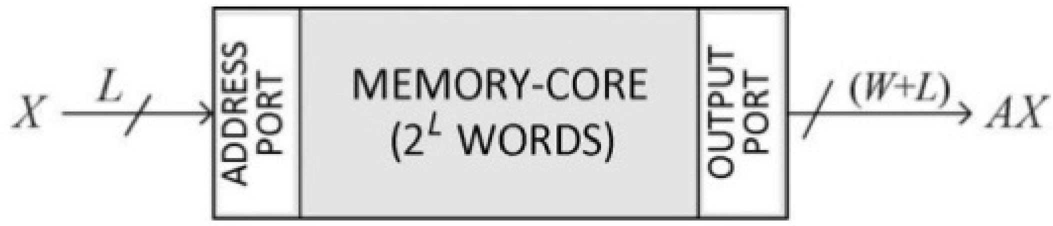


Figure 2.1: Conventional LUT based multiplier

Fig. 2.1 shows a conventional memory based multiplier where every possible combination of input word X of length L is multiplied with a constant A and stored in memory.The total memory-core size is 2^L (all combination of X).Using APC and OMS technique, and by taking advantage of symmetry, the size of memory core can be reduced, thereby reducing the number of LUTs used.

FFT implementation requires multiplication to be performed in various parts of butterfly structure. Most multiplications in the FFT algorithm involves one of the operand being constant (which is sin or cos of some angle). Due to the fact that one of the operand is constant, memory based multiplier can be used to achieve higher speeds.

This paper discusses anti-symmetric product coding (APC) and odd-multiple-storage (OMS) techniques for lookup-table (LUT) design for memory-based multipliers to be used in digital signal processing applications.These techniques are used in the reduction of the LUT size by a factor of two.In this a different form of APC and a modified OMS scheme is shown, in order to combine them for efficient memory-based multiplication. The proposed combined approach provides a reduction in LUT size to one-fourth of the conventional LUT. It has also suggested a simple technique for selective sign reversal to be used in the proposed design. It is shown that the proposed LUT design for small input sizes can be used for efficient implementation of high-precision multiplication by input operand decomposition.

2.2 CORDIC algorithm based on FPGA

CORDIC algorithm provides an efficient way of rotating the vectors in a plane by simple shift add operation to estimate the basic elementary functions like trigonometric operations, multiplication, division and some other operations like logarithmic functions, square roots and exponential functions.

The CORDIC algorithm is used to evaluate real time calculation of the exponential and logarithmic functions using the iterative rotation of the input vector. This rotation of a given vector (x_i, y_i) is realized by means of a sequence of rotations with fixed angles which results in overall rotation through a given angle or result in a final angular argument of zero.

CORDIC algorithm can be used efficiently for realizing a multiplication between a scalar and vector, by changing the initial magnitude of the rotation vector , thereby reducing total number of multiplications from two to one. Fig. 2.2 shows the rotation of the input vector, and by multiplying the scalar operand with the initial magnitude of vector operand we obtain M_in. Now M_in is rotated in small steps for the total angle θ and thereby obtaining the resulting vector.

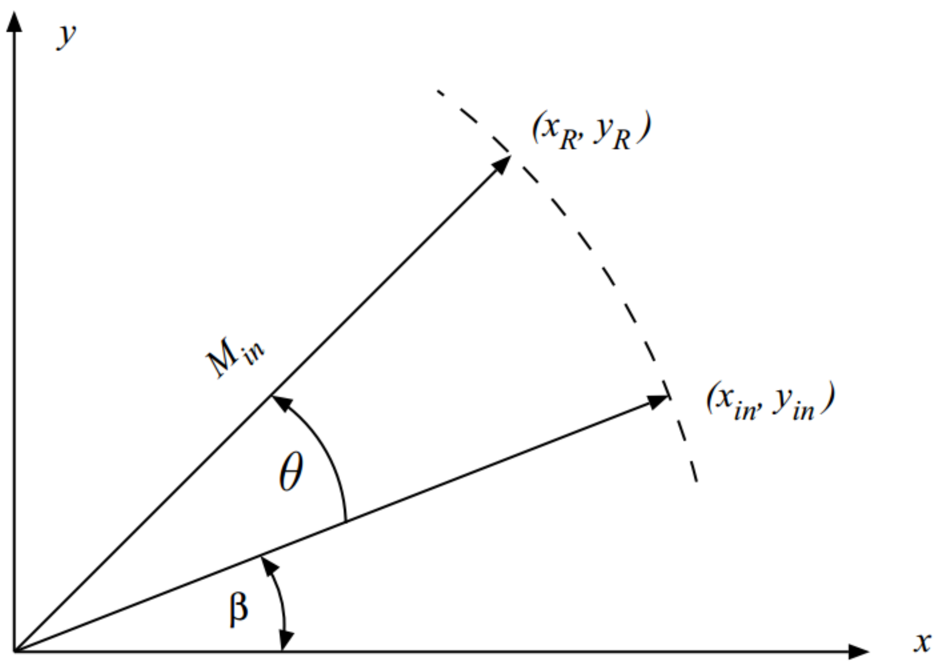


Figure 2.2: CORDIC algorithm showing rotation through an angle θ.

2.3 Cooley-Tukey FFT Algorithms

The direct way of computing the DFT problem of size N takes $O(N^2)$ operations, where each operation consists of multiplication and addition of complex values. While using Cooley-Tukey FFT Algorithms, the computation time can be reduced to $O(Nlog(N))$. A special case of such algorithm when N is a power of 2 is used to calculate the DFT , which is otherwise known as FFT.

$$X(k) = \sum_{x=0}^{N-1} x(n) W_N^{kx}, 0 \leq k \leq N-1$$

$$W_N^{kx} = e^{-j2\pi/N}$$

The above expression is for computing FFT and W_N are the twiddle factors.This expression can be best represented as a butterfly structure shown in Fig. 2.3

Fig. 2.3 shows the butterfly structure implementation for computing FFT using Cooley-Tukey Algorithm. The inputs $x[i]$ to the first stage of butterfly are complex numbers and the FFT is computed over 3 stages for 8 point FFT. The total number of

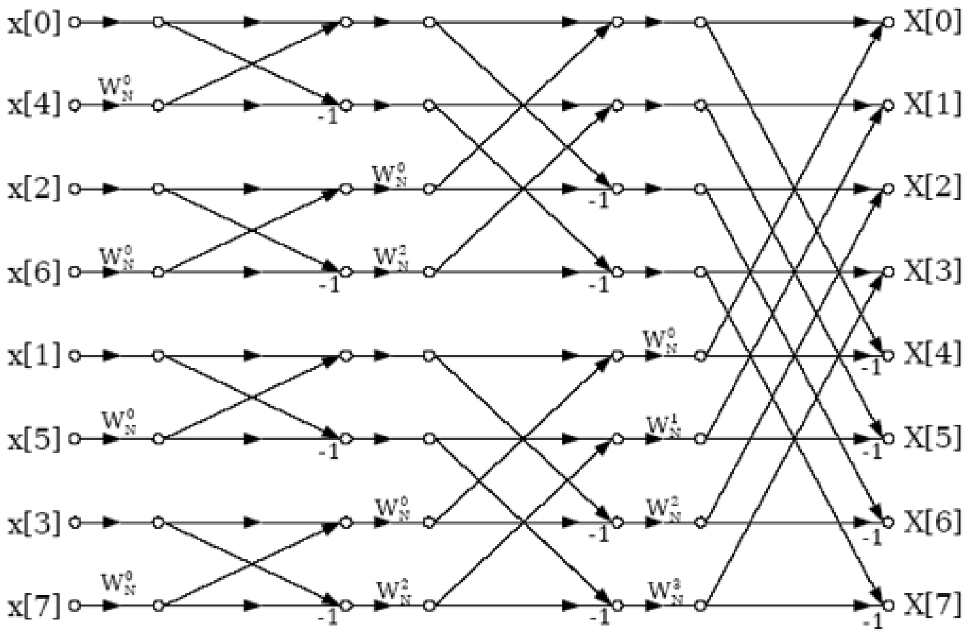


Figure 2.3: Butterfly structure for Cooley-Tukey Algorithm

stages is related to the number of inputs N as show below.

$$Total\ Stages : log_2 N$$

The output $X[i]$ is also a complex number. Twiddle Factors are multiplied in various stages represented by W_N^i

2.4 Doing Hartley Smartly

FFT can also be computed by first computing another transform first, called as Hartley transform , given by the following expression.

$$H(f) = \frac{1}{N} \sum_{i=0}^{N-1} X(t) \left[cos(\frac{2\pi ft}{N}) + sin(\frac{2\pi ft}{N}) \right]$$

$$F_{real}(f) = H(f) + H(N - f)$$

$$F_{imag}(f) = H(f) - H(N - f)$$

Here $F_{real}(f)$ and $F_{imag}(f)$ are the real and imaginary parts of FFT. Classically, FFT performance has been evaluated by counting the number of multiplications, additions, and subtractions that are involved. In these terms, the FHT does very well. If we disregard the simplicity of the first two rounds, we have each round requiring N multiplications and 2N additions or subtractions. The number of rounds is log2N. By comparison, the FFT requires 2N multiplications and 7N/2 additions or subtractions in each round. In both cases these numbers assume that no redundant operations are performed.

Table 2.1: Comparison of total operations performed by computing FFT directly and by computing FHT first and then FFT.

	FFT	FHT
Multiplication	N	2N
Addition or Subtraction	2N	7N/2
Rounds	$log_2 N$	$log_2 N$

The biggest disadvantage of this algorithm is that, the input for the FFT needs to be real valued samples only. In most of the real world DSP applications like image processing, audio processing, and so on, the input is real valued samples, and this algorithm can be used to improve efficiency and speed.

CHAPTER 3

Direct Cooley-Tukey algorithm using DSP based multiplier

3.1 Algorithm

Cooley and Tukey showed that it's possible to divide the DFT computation into two smaller parts. From the definition of the DFT we have

$$X_k = \sum_{n=0}^{N-1} x_n . e^{-i2\pi kn/N} \tag{3.1}$$

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} . e^{-i2\pi k(2m)/N} + \sum_{m=0}^{N/2-1} x_{2m+1} . e^{-i2\pi k(2m+1)/N} \tag{3.2}$$

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} . e^{-i2\pi km/(N/2)} + e^{-i2\pi k/N} \sum_{m=0}^{N/2-1} x_{2m+1} . e^{-i2\pi km/(N/2)} \tag{3.3}$$

We've split the single Discrete Fourier transform into two terms which themselves look very similar to smaller Discrete Fourier Transforms, one on the odd-numbered values, and one on the even-numbered values. Each term consists of $(N^2/2)$ computations, for a total of N^2.

The trick comes in making use of symmetries in each of these terms. Because the range of k is $0 \leqslant k < N$, while the range of n is $0 \leqslant n < N/2$, we see from the symmetry properties above that we need only perform half the computations for each sub-problem. Our $O[N^2]$ computation has become $O[M^2]$, with M half the size of N.

As long as our smaller Fourier transforms have an even-valued M, we can reapply this divide-and-conquer approach, halving the computational cost each time, until our arrays are small enough that the strategy is no longer beneficial. In the asymptotic limit, this recursive approach scales as $O[NlogN]$. Since recursively the scales reduce to half everytime , FFT can be implemented only if N is in the form of 2^k.

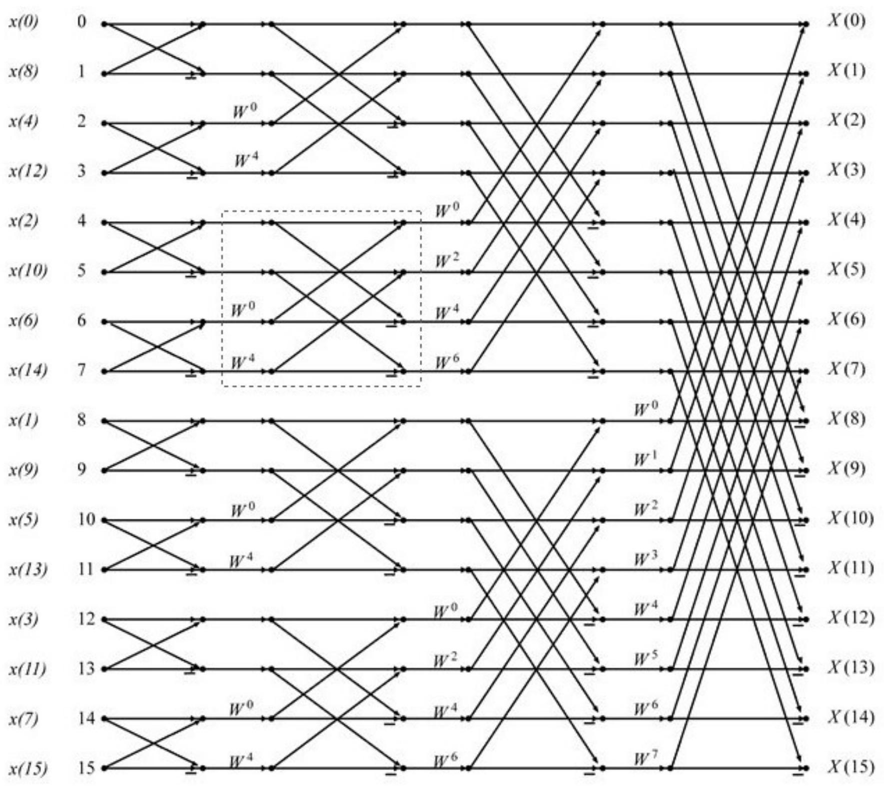


Figure 3.1: Butterfly structure used for calculating 16 length FFT

After we reduce the FFT of larger sizes recursively, we reach the smallest unit called the butterfly unit which is the main building block for FFT. Fig. 3.1 shows the implementation of a 16 length input FFT. The smallest structure with 2 input and 2 output unit is called the butterfly unit and the entire computation can be build using these buttefly structures.

3.2 Calculation

Cooley-Tukey algorithm re-expresses the discrete Fourier transform (DFT) of an arbitrary composite size N = $N_1 N_2$ in terms of N_1 smaller DFTs of sizes N_2, recursively, to reduce the computation time to **O**(N log N) for highly composite N.

This is a direct implementation of Cooley-Tukey Algorithm , and for a 8 point FFT , it involves 2 multiplications for output calculation. The equations involving the calculation of FFT in shown below.

$t_1 = D(0) + D(4); m_3 = D(0) - D(4);$

$t_2 = D(6) + D(2); m_6 = j * (D(6) - D(2));$

$t_3 = D(1) + D(5); t_4 = D(1) - D(5);$

$t_5 = D(3) + D(7); t_6 = D(3) - D(7);$

$t_8 = t_5 + t_3; m_5 = j * (t_5 - t_3);$

$t_7 = t_1 + t_2; m_2 = t_1 - t_2;$

$m_0 = t_7 + t_8; m_1 = t_7 - t_8;$

$m_4 = sin(\pi/4) * (t_4 - t_6); m_7 = -j * sin(\pi/4) * (t_4 + t_6);$

$s_1 = m_3 + m_4; s2 = m_3 - m_4;$

$s_3 = m_6 + m_7; s4 = m_6 - m_7;$

$DO(0) = m_0; DO(4) = m_1;$

$DO(1) = s_1 + s_3; DO(7) = s_1 - s_3;$

$DO(2) = m_2 + m_5; DO(6) = m_2 - m_5;$

$DO(5) = s_2 + s_4; DO(3) = s_2 - s_4;$

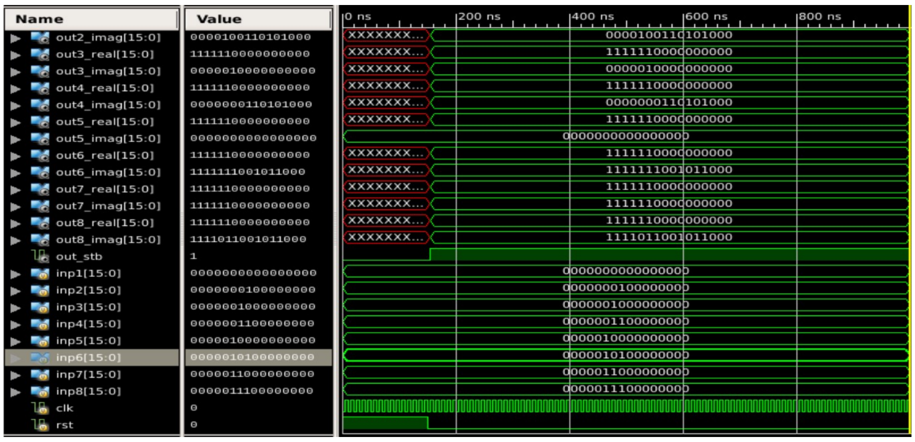
where D and DO are input and output arrays of the complex data t_1,...,t_8, m_1,.., m_7, s_1,..,s_4 are the intermediate complex results. As we see the algorithm contains only 2 multiplications to the non-trivial coefficient sin(π/4) = 0.7071, and 22 real additions and subtractions. The multiplication to a coefficient j means the negation the imaginary part and swapping real and imaginary parts.

The implementation of this algorithm is given in Appendix A.1 written in Verilog and tested with Xilinx ISE design suite 14.7. inp1,...inp8 are the 16 bit signed floating point inputs of Q format and out1-real , out1-imag, ...,out8-real,out8-imag are the real and imaginary outputs in 16 bit Q format of the FFT8 module.clk , rst are the clock , reset inputs respectively , and output-stb is the output strobe , which is enabled once the output is calculated.

The implementation was cross verified with the python script shown in Appendix A.5 , which tests the FFT block against various test vectors and verifying the result with the default FFT module in numerical python library.

3.3 Simulation

Fig. 3.2 shows the simulation of this implementation in ISIM.



```
Name                Value
out2_imag[15:0]     0000100110101000
out3_real[15:0]     1111110000000000
out3_imag[15:0]     0000010000000000
out4_real[15:0]     1111110000000000
out4_imag[15:0]     0000000110101000
out5_real[15:0]     1111110000000000
out5_imag[15:0]     0000000000000000
out6_real[15:0]     1111110000000000
out6_imag[15:0]     1111111001011000
out7_real[15:0]     1111110000000000
out7_imag[15:0]     1111110000000000
out8_real[15:0]     1111110000000000
out8_imag[15:0]     1111011001011000
out_stb             1
inp1[15:0]          0000000000000000
inp2[15:0]          0000000100000000
inp3[15:0]          0000001000000000
inp4[15:0]          0000001100000000
inp5[15:0]          0000010000000000
inp6[15:0]          0000010100000000
inp7[15:0]          0000011000000000
inp8[15:0]          0000011100000000
clk                 0
rst                 0
```

Figure 3.2: ISIM simulation for Direct Cooley-Tukey algorithm using DSP based multiplier

3.4 Result

Table 3.1: resource utilization for direct implementation of Cooley-Tukey algorithm with DSP481As

Logic Utilization	Quantity
Number of Slice Registers	47
Number of Slice LUTs	337
Number of fully used LUT-FF pairs	47
Number of BUFG/BUFGCTRLs	1
Number of DSP48A1s	2

Table. 3.1 shows the resource utilization summary of this current implementation. In total there are 2 multiplications that is performed in parallel in to improve speed of calculation. Since 2 multiplications run in parallel , 2 hardware based multiplier are used as seen in the Table. 3.1.Hardware Multiplier are scarce resources and Spartan-3e contains only 8 DSP48A1s in total and more power consuming compared to LUT based multipliers.This implementation gives the maximum speed compared to others shown here , but compromising on more resource usage.

CHAPTER 4

Modified Cooley-Tukey algorithm reusing single DSP based multiplier

4.1 Algorithm

From the chapter 3 , it is seen that 2 multiplications are necessary for calculating 8 point FFT. In this algorithm those 2 multiplications are performed serially one by one , hence reusing a single multiplier. Fig. 4.1 shows the basic block diagram , representing multiplexers combined with hardware multipliers.both the multiplication operands are given as inputs to multiplexer , where Operand-1A,Operand-2A and Operand-1B,Operand-2B are the input operands and a select is used to select which multiplication to be performed. This way the number of hardware multipliers used can be reduced.

This block diagram is used for 8 point FFTs since only 2 multiplications are involved , this can be extended to any other input size by using large multiplexers of different size.

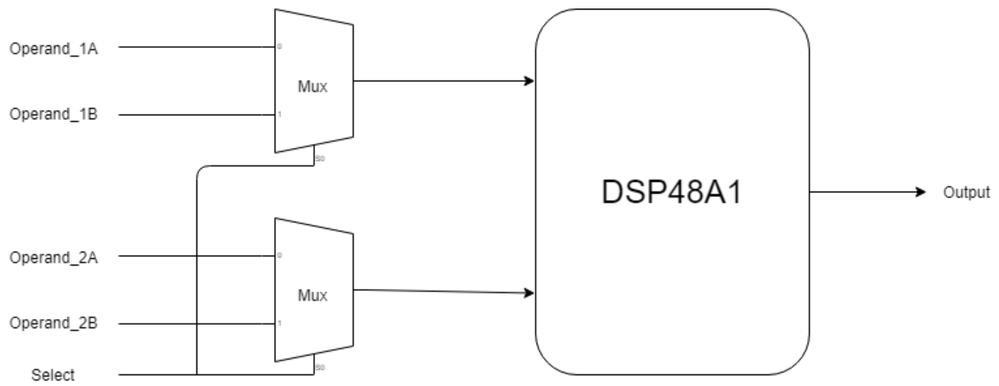


Figure 4.1: Block diagram representing DSP48A1s coupled with multiplexer

4.2 Calculation

Cooley-Tukey algorithm for 8 point FFT involves 2 multiplication , hence using 2 multiplicative blocks as seen in the first implementation.In this modified algorithm , the multiplications are pipe lined to run one by one , by reusing the same multiplier. The calculations of various intermediate complex numbers is divided into groups and executed group by group.The equation involving different groups are mentioned below.

$Group-1$

$t_1 = D(0) + D(4);$

$m_3 = D(0) - D(4);$

$t_2 = D(6) + D(2);$

$m_6 = j * (D(6) - D(2));$

$t_3 = D(1) + D(5);$

$t_4 = D(1) - D(5);$

$t_5 = D(3) + D(7);$

$t_6 = D(3) - D(7);$

$t_8 = t_5 + t_3;$

$m_5 = j * (t_5 - t_3);$

$t_7 = t_1 + t_2;$

$m_2 = t_1 - t_2;$

$m_0 = t_7 + t_8;$

$m_1 = t_7 - t_8;$

$m_4 = sin(\pi/4) * (t_4 - t_6);$

$Group-2$

$m_7 = -j * sin(\pi/4) * (t_4 + t_6);$

$s_1 = m_3 + m_4;$

$s2 = m_3 - m_4;$

$s_3 = m_6 + m_7;$

$s4 = m_6 - m_7;$

$DO(0) = m_0;$

$DO(4) = m_1;$

$DO(1) = s_1 + s_3;$

$DO(7) = s_1 - s_3;$

$DO(2) = m_2 + m_5;$

$DO(6) = m_2 - m_5;$

$DO(5) = s_2 + s_4;$

$DO(3) = s_2 - s_4;$

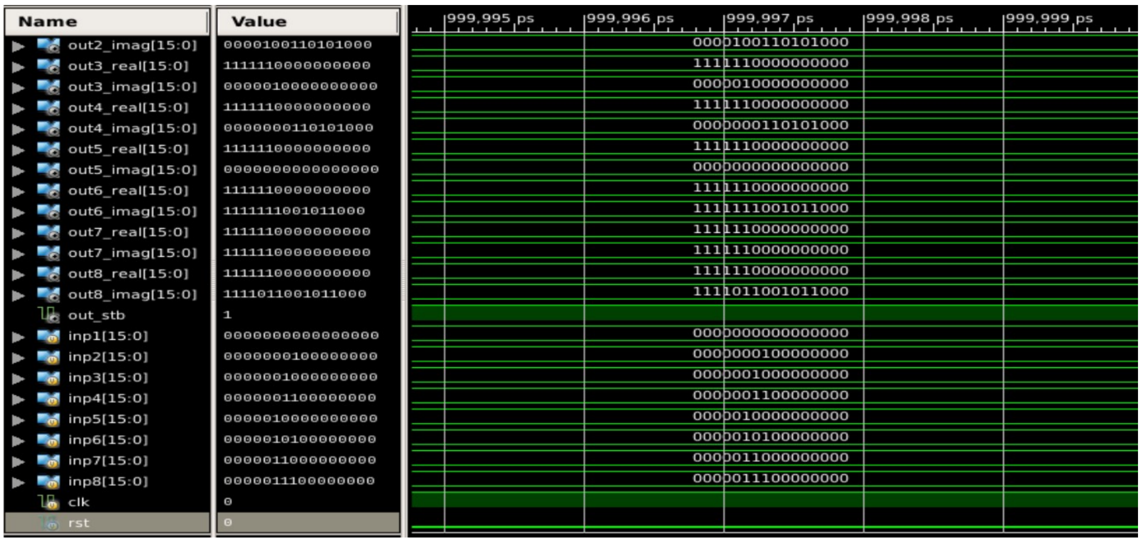
where D and DO are input and output arrays of the complex data t_1,..,t_8, m_1,.., m_7, s_1,..,s_4 are the intermediate complex results. As we see the algorithm contains only 2 multiplications to the non-trivial coefficient sin(π/4) = 0.7071, and 22 real additions and subtractions. The multiplication to a coefficient j means the negation the imaginary part and swapping real and imaginary parts.

The implementation of this algorithm is given in Appendix A.2 written in Verilog and tested with Xilinx ISE design suite 14.7. inp1,...inp8 are the 16 bit signed floating point inputs of Q format and out1-real , out1-imag, ...,out8-real,out8-imag are the real and imaginary outputs in 16 bit Q format of the FFT8 module.clk , rst are the clock , reset inputs respectively , and output-stb is the output strobe , which is enabled once the output is calculated.

The implementation was cross verified with the python script shown in Appendix A.5 , which tests the FFT block against various test vectors and verifying the result with the default FFT module in numerical python library.

4.3 Simulation

Fig. 4.2 shows the simulation of this implementation in ISIM



```
Name               Value               |999,995 ps  |999,996 ps  |999,997 ps  |999,998 ps  |999,999 ps
out2_imag[15:0]    0000100110101000
out3_real[15:0]    1111110000000000
out3_imag[15:0]    0000010000000000
out4_real[15:0]    1111110000000000
out4_imag[15:0]    0000000110101000
out5_real[15:0]    1111110000000000
out5_imag[15:0]    0000000000000000
out6_real[15:0]    1111110000000000
out6_imag[15:0]    1111111001011000
out7_real[15:0]    1111110000000000
out7_imag[15:0]    1111110000000000
out8_real[15:0]    1111110000000000
out8_imag[15:0]    1111011001011000
out_stb            1
inp1[15:0]         0000000000000000
inp2[15:0]         0000000100000000
inp3[15:0]         0000001000000000
inp4[15:0]         0000001100000000
inp5[15:0]         0000010000000000
inp6[15:0]         0000010100000000
inp7[15:0]         0000011000000000
inp8[15:0]         0000011100000000
clk                0
rst                0
```

Figure 4.2: ISIM simulation for Modified Cooley-Tukey algorithm reusing single DSP based multiplier

4.4 Result

Table 4.1: resource utilization for Cooley-Tukey algorithm by reusing same DSP481A multiplier.

Logic Utilization	Quantity
Number of Slice Registers	317
Number of Slice LUTs	374
Number of fully used LUT-FF pairs	138
Number of BUFG/BUFGCTRLs	1
Number of DSP48A1s	1

Table. 4.1 shows the resource utilization summary of this current implementation. In total there are 2 multiplications that is performed in one after another . Since 2 multiplications run one by one , 1 hardware based multiplier are used as seen in the Table. 4.1.Hardware Multiplier are scarce resources and Spartan-3e contains only 8 DSP48A1s in total and more power consuming compared to LUT based multipliers.This Algorithm takes twice the time taken by Algorithm 3, but uses less power since single multiplier is used. Also we save 12.5 percent of the resource utilization of hardware multiplier compared to Algorithm 3, and this Algorithm can be used in places where the DSP multiplier are very less. Also since the DSP48A1s are connected with multiplexer to run computation in stages , more number of LUTs are used compared to that of Algorithm 3.

CHAPTER 5

Modified Cooley-Tukey algorithm using LUT based multiplier

5.1 Algorithm

Chapter 4 shows the Cooley-Tukey algorithm implemented to perform multiplication sequentially using hardware based multipliers.This chapter discusses the same algorithm being implemented using LUT based multiplier. More precisely a shift and add based multiplier is used here.

Shift-and-add multiplication is similar to the multiplication performed by paper and pencil. This method adds the multiplicand X to itself Y times, where Y denotes the multiplier. To multiply two numbers by paper and pencil, the algorithm is to take the digits of the multiplier one at a time from right to left, multiplying the multiplicand by a single digit of the multiplier and placing the intermediate product in the appropriate positions to the left of the earlier results.

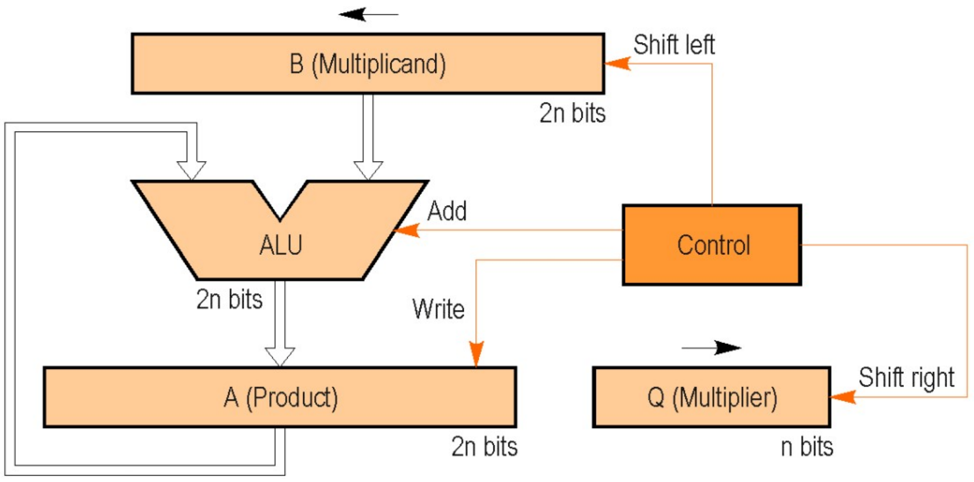


Figure 5.1: Shift-Add multiplier circuit

A version of the multiplier circuit, which implements the shift-and-add multiplication method for two n-bit numbers, is shown in Fig. 5.1. The $2n$ bit product register

(A) is initialized to 0. Since the basic algorithm shifts the multiplicand register (B) left one position each step to align the multiplicand with the sum being accumulated in the product register, we use a $2n$-bit multiplicand register with the multiplicand placed in the right half of the register and with 0 in the left half.

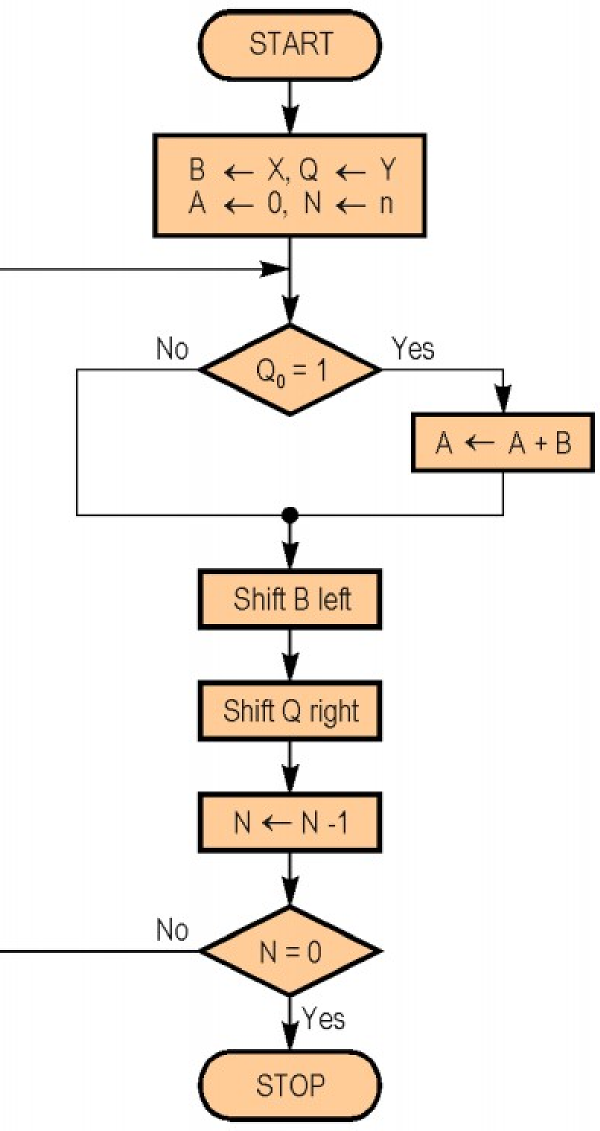


Figure 5.2: Shift-Add multiplier Algorithm

Fig. 5.2. shows the basic steps needed for the multiplication. The algorithm starts by loading the multiplicand into the B register, loading the multiplier into the Q register, and initializing the A register to 0. The counter N is initialized to n. The least significant bit of the multiplier register (Q_0) determines whether the multiplicand is added to the product register. The left shift of the multiplicand has the effect of shifting the intermediate products to the left, just as when multiplying by paper and pencil. The right shift of the multiplier prepares the next bit of the multiplier to examine in the following iteration.

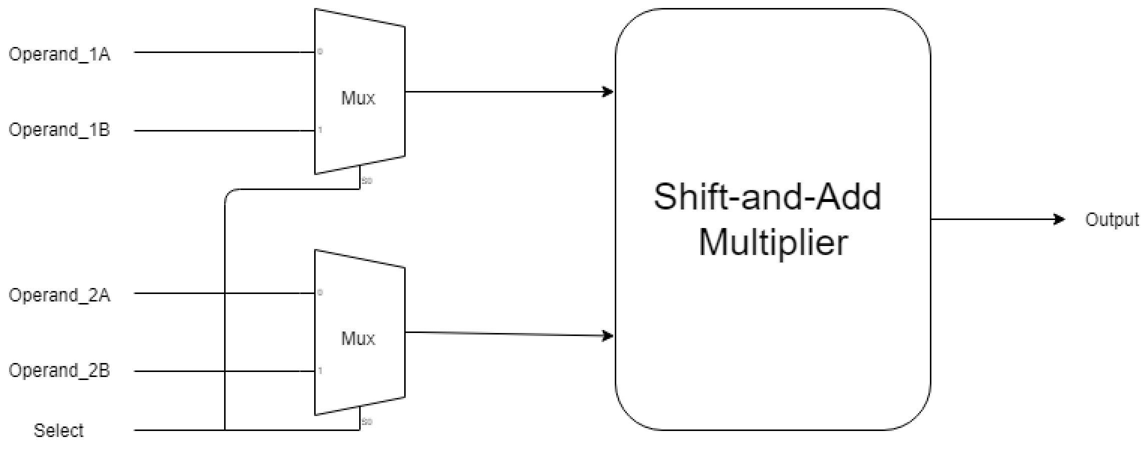


Figure 5.3: Shift-Add multiplier coupled with multiplexer

Now the shift-add multiplier is coupled with multiplexer as shown in Fig. 5.3, which is very similar to chapter 4. Based on the input sizes of the FFT block, Multiplexer of various sizes can be used to run multiplications serially.

5.2 Calculation

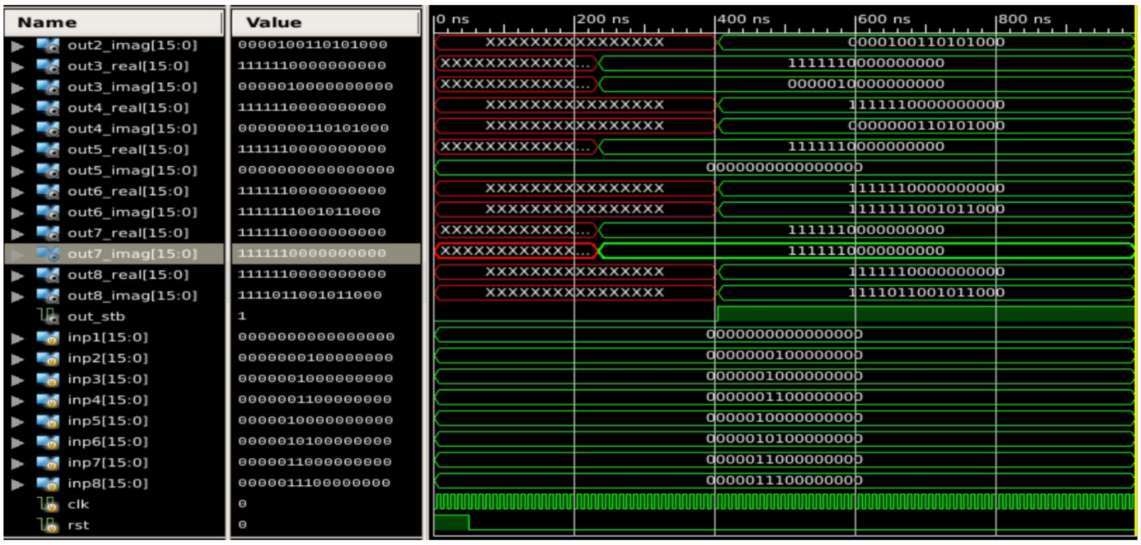
This is implemented by using a Shift-Add based multiplier using purely LUTs in FPGA.This algorithm requires lot of area in an FPGA since multiplier block is complex and it consumes most of the LUTs, but highly power efficient and slower in speed compared to that of DSP based multipliers. The calculation are same as shown in section 4.2.

The implementation of this algorithm is given in Appendix A.3 written in Verilog and tested with Xilinx ISE design suite 14.7. inp1,...inp8 are the 16 bit signed floating point inputs of Q format and out1-real , out1-imag, ...,out8-real,out8-imag are the real and imaginary outputs in 16 bit Q format of the FFT8 module.clk , rst are the clock , reset inputs respectively , and output-stb is the output strobe , which is enabled once the output is calculated.

The implementation was cross verified with the python script shown in Appendix A.5 , which tests the FFT block against various test vectors and verifying the result with the default FFT module in numerical python library.

5.3 Simulation

Fig. 5.4 shows the simulation of this implementation in ISIM



Name	Value
out2_imag[15:0]	0000100110101000
out3_real[15:0]	1111110000000000
out3_imag[15:0]	0000010000000000
out4_real[15:0]	1111110000000000
out4_imag[15:0]	0000000110101000
out5_real[15:0]	1111110000000000
out5_imag[15:0]	0000000000000000
out6_real[15:0]	1111110000000000
out6_imag[15:0]	1111111001011000
out7_real[15:0]	1111110000000000
out7_imag[15:0]	1111110000000000
out8_real[15:0]	1111110000000000
out8_imag[15:0]	1111011001011000
out_stb	1
inp1[15:0]	0000000000000000
inp2[15:0]	0000000100000000
inp3[15:0]	0000001000000000
inp4[15:0]	0000001100000000
inp5[15:0]	0000010000000000
inp6[15:0]	0000010100000000
inp7[15:0]	0000011000000000
inp8[15:0]	0000011100000000
clk	0
rst	0

Figure 5.4: ISIM simulation for Modified Cooley-Tukey algorithm using LUT based multiplier

5.4 Result

Table 5.1: resource utilization for Cooley-Tukey algorithm by using LUT based multiplier.

Logic Utilization	Quantity
Number of Slice Registers	357
Number of Slice LUTs	456
Number of fully used LUT-FF pairs	188
Number of BUFG/BUFGCTRLs	1
Number of DSP48A1s	0

Table. 5.1 shows the resource utilization summary of this current implementation.This Implementation is purely using LUTs and hence number of LUTs used is comparatively higher is number than that of Chapter 3 and Chapter 4. Also as seen , the number of hardware multipliers used are zero, therefore saving it for other blocks in the DSP applications like filters, etc...This implementation is slower compared to the Chapter 3 and Chapter4 , since LUT based multiplier use shift-add algorithm which is slower and

each multiplication takes more number of clock cycles to compute which is also equal to the length of the operand. Hence this algorithm is slower than Chapter 4 in terms of two times the length of input since 2 multiplications are performed serially.

CHAPTER 6

Modified Cooley-Tukey algorithm using CORDIC based multiplier

6.1 Algorithm

Chapter 5 shows the Cooley-Tukey algorithm implemented to perform multiplication sequentially using LUT based multipliers.This chapter discusses the same algorithm being implemented using CORDIC based multiplier. The CORDIC multiplier is implemented by using LUTs and one DSP48A1.

The CORDIC algorithm is a clever method for accurately computing trigonometric functions using only additions, bitshifts and a small lookup table. It's well known that rotating the vector (1,0) anticlockwise about the origin by an angle θ gives the vector $(cos\theta, sin\theta)$. We will use this as the basis of our algorithm. When we replace the initial vector (1,0) with (a,0) , we will get $(acos\theta, asin\theta)$.

Every iteration calculates a rotation, which is performed by multiplying the vector v_{i-1} with the rotation matrix R_i:

$$v_i = v_{i-1} * R_i \tag{6.1}$$

where Rotation matrix R_i is given by

$$R_i = \begin{bmatrix} cos(\gamma_i) & -sin(\gamma_i) \\ sin(\gamma_i) & cos(\gamma_i) \end{bmatrix} \tag{6.2}$$

by solving above equations we get ,

$$v_i = \frac{1}{\sqrt{1 + tan^2(\gamma_i)}} * \begin{bmatrix} 1 & tan(\gamma_i) \\ tan(\gamma_i) & 1 \end{bmatrix} * \begin{bmatrix} x_{i-1} \\ y_{i-1} \end{bmatrix} \tag{6.3}$$

where x_{i-1} and y_{i-1} are the components of v_{i-1}. Restricting the angles γ_i so that $tan(\gamma_i)$

takes on the values $\pm 2^{-i}$, the multiplication with the tangent can be replaced by a division by a power of two, which is efficiently done in digital computer hardware using a bit shift. The expression then becomes:

$$v_i = K_i * \begin{bmatrix} 1 & -\sigma_i 2^{-i} \\ \sigma_i 2^{-i} & 1 \end{bmatrix} * \begin{bmatrix} x_{i-1} \\ y_{i-1} \end{bmatrix} \tag{6.4}$$

where

$$K_i = \frac{1}{\sqrt{1+2^{-2i}}} \tag{6.5}$$

and σ_i can have the values of âĹŠ1 or 1, and is used to determine the direction of the rotation; if the angle γ_i is positive then σ_i is +1, otherwise it is âĹŠ1.

K_i can be ignored in the iterative process and then applied afterward with a scaling factor:

$$K(n) = \prod_{i=0}^{n-1} K_i = \prod_{i=0}^{n-1} \frac{1}{\sqrt{1+2^{-2i}}} \tag{6.6}$$

$$K = \lim_{n\to\infty} K(n) \approx 0.6072529350088812561694 \tag{6.7}$$

This initial constant K is now multiplied with a scalar a to give $acos(\theta)$ and $asin(\theta)$ and hence this can be used as a multiplier.Now as seen in Chapter 5, its coupled with multiplexer to be reused for other computations.You can see that , the approximation is more true as limit N tends to infinity , meaning the value of cosθ and sinθ are more accurate when the step angle for rotation tends to zero.

6.2 Calculation

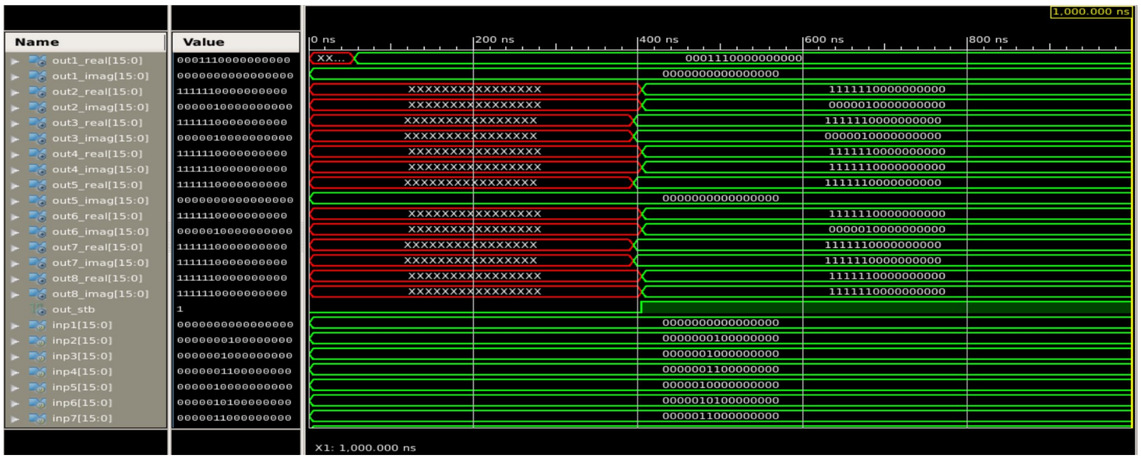
This Algorithm implements the same algorithm as shown in Chapter 5 , but instead of LUT based shift-add multiplier , CORDIC multiplier is used. Since the Multiplications in Cooley-Tukey Method are multiplications with one operand scalar and another being sin or cos of some angle θ , CORDIC based multiplier can be used.

In CORDIC method , the initial magnitude is multiplied with the scalar operand and rotated in small steps covering angle θ , thereby giving the result.The obtained result is used in Cooley-Tukey algorithm for computing FFT.

The implementation of this algorithm is given in Appendix A.4 written in Verilog and tested with Xilinx ISE design suite 14.7. inp1,...inp8 are the 16 bit signed floating point inputs of Q format and out1-real , out1-imag, ...,out8-real,out8-imag are the real and imaginary outputs in 16 bit Q format of the FFT8 module.clk , rst are the clock , reset inputs respectively , and output-stb is the output strobe , which is enabled once the output is calculated.

6.3 Simulation

Fig. 6.1 shows the simulation of this implementation in ISIM



Name	Value
out1_real[15:0]	0001110000000000
out1_imag[15:0]	0000000000000000
out2_real[15:0]	1111110000000000
out2_imag[15:0]	0000010000000000
out3_real[15:0]	1111110000000000
out3_imag[15:0]	0000010000000000
out4_real[15:0]	1111110000000000
out4_imag[15:0]	1111110000000000
out5_real[15:0]	1111110000000000
out5_imag[15:0]	0000000000000000
out6_real[15:0]	1111110000000000
out6_imag[15:0]	0000010000000000
out7_real[15:0]	1111110000000000
out7_imag[15:0]	1111110000000000
out8_real[15:0]	1111110000000000
out8_imag[15:0]	1111110000000000
out_stb	1
inp1[15:0]	0000000000000000
inp2[15:0]	0000000100000000
inp3[15:0]	0000001000000000
inp4[15:0]	0000001100000000
inp5[15:0]	0000010000000000
inp6[15:0]	0000010100000000
inp7[15:0]	0000011000000000

Figure 6.1: ISIM simulation for Modified Cooley-Tukey algorithm using CORDIC based multiplier

6.4 Result

Table 6.1: resource utilization for Cooley-Tukey algorithm by using CORDIC based multiplier.

Logic Utilization	Quantity
Number of Slice Registers	458
Number of Slice LUTs	942
Number of fully used LUT-FF pairs	263
Number of BUFG/BUFGCTRLs	1
Number of DSP48A1s	1

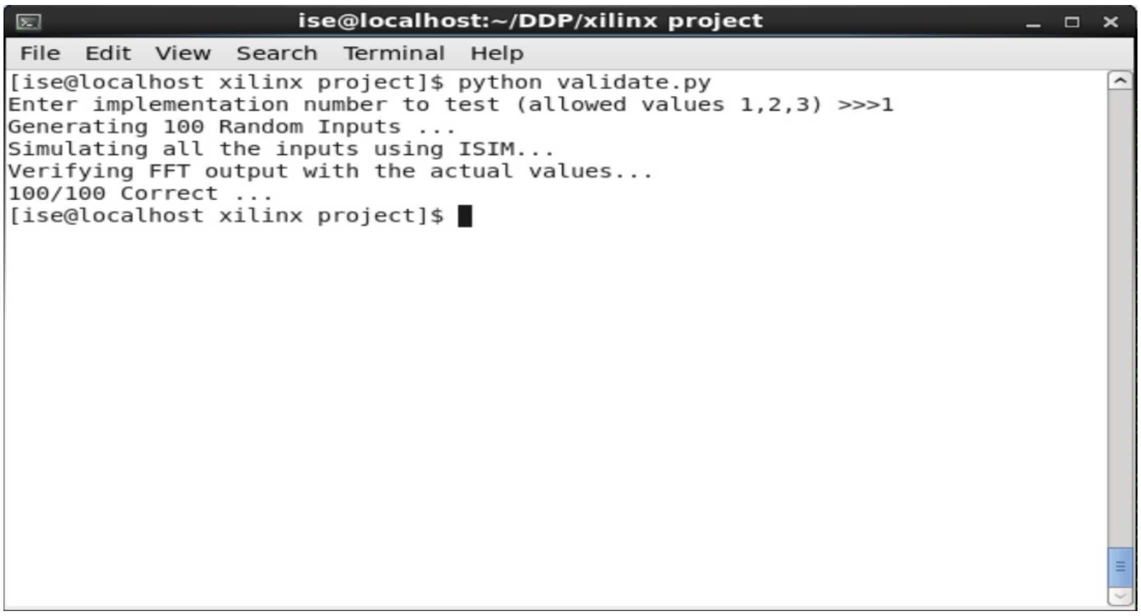
Table. 6.1 shows the resource utilization summary of this current implementation. This Algorithm is implemented with mixture of LUTs and DSP48A1s. The hardware multiplier is used to multiply the initial vector in the CORDIC rotation with the scalar operand. The number of LUTs are high because , the CORDIC multiplier is complex in terms of implementing the rotation. This Algorithm is more useful if accuracy is the primary concern. Its easier to increase accuracy to large extent by reducing the step size in CORDIC rotation.

CHAPTER 7

Python Validation Script

A python Validation script is written to validate the Verilog FFT Blocks. The python script below uses Numpy library to generate 100 random floating point test inputs and using ISIM command line options, it simulates all the inputs and validates them with the standard FFT function present in the numpy library of python.

The script is written in Python 2.x version and requires ISIM to be pre installed in the system for working.Fig. 7.1 shows the sample working of the script.The complete script can be found in Appendix A.5.


ise@localhost:~/DDP/xilinx project
File Edit View Search Terminal Help

```
[ise@localhost xilinx project]$ python validate.py
Enter implementation number to test (allowed values 1,2,3) >>>1
Generating 100 Random Inputs ...
Simulating all the inputs using ISIM...
Verifying FFT output with the actual values...
100/100 Correct ...
[ise@localhost xilinx project]$ 
```

Figure 7.1: Sample working of Python validation script

CHAPTER 8

Conclusion

All the four Algorithms shown have different resource utilization and Table. 8.1 compares the various resources used by different algorithm. FPGAs available today vary in terms of available resources like Slice Registers, LUTs, LUT-FF pairs, Hardware Multiplier, etc...

Table 8.1: Comparison of resource utilization for different implementations shown

Logic Utilization	Algo-1	Algo-2	Algo-3	Algo-4
Number of Slice Registers	47	317	357	458
Number of Slice LUTs	337	374	456	942
Number of fully used LUT-FF pairs	47	138	188	263
Number of BUFG/BUFGCTRLs	1	1	1	1
Number of DSP48A1s	2	1	0	1

Based on the resource utilization shown in Table. 8.1 and FPGA used , appropriate algorithm can be used to optimally fit and use less area for FFT blocks , and use most of the resources for other DSP blocks like Filters, etc.. .Out of all the resources , the most scarce resource is DSP48A1s which are hardware multipliers and its required for most of the DSP applications. Algo-3 can be used for applications where hardware multiplier are less and required for other blocks.When speed is the primary concern Algo-1 can be used since its faster and running all multiplications in parallel with multiple hardware multipliers. Hardware Multipliers use more power in general compared to LUT based multiplier.Hence Algo-3 can be used for applications where the power is primary concern and required for long endurance. Algo-4 implements CODRIC based multiplier, hence accuracy of the FFT can be increased by reducing the step angle for rotation.Hence it can be used for application requiring high precision by changing the step angle as per the requirement.

APPENDIX A

Appendix

A.1 Code for direct Cooley-Tukey algorithm implementation

```
`timescale 1ns / 1ps

module fft8(
        input signed [15:0] inp1,
        input signed [15:0] inp2,
        input signed [15:0] inp3,
        input signed [15:0] inp4,
        input signed [15:0] inp5,
        input signed [15:0] inp6,
        input signed [15:0] inp7,
        input signed [15:0] inp8,
        input clk,
        input rst,
        output signed [15:0] out1_real,
        output signed [15:0] out1_imag,
        output signed [15:0] out2_real,
        output signed [15:0] out2_imag,
        output signed [15:0] out3_real,
        output signed [15:0] out3_imag,
        output signed [15:0] out4_real,
        output signed [15:0] out4_imag,
        output signed [15:0] out5_real,
        output signed [15:0] out5_imag,
        output signed [15:0] out6_real,
        output signed [15:0] out6_imag,
        output signed [15:0] out7_real,
        output signed [15:0] out7_imag,
        output signed [15:0] out8_real,
        output signed [15:0] out8_imag,
        output out_stb
);

  localparam signed sin_45 = 16'b00000000_10110101;
  localparam signed sin_315 = 16'b11111111_01001011;
  reg signed [31:0] t1_46,t2_46;
  reg signed [15:0] t1,t2,t3,t4,t5,t6,t7,t8,m0,m1,m2,m3,m4;
  reg signed [15:0] m7_imag,s1,s2,s3_imag,s4_imag;
  reg signed [15:0] m5_imag,m6_imag
  reg output_stb;

  initial
    begin
      output_stb = 1'b0;
    end

  always @( posedge clk )
    begin
      if (rst == 1'b1)
        begin
          output_stb = 1'b0;
        end
      else
        begin
          t1 = inp1 + inp5;
          t2 = inp7 + inp3;
          t3 = inp2 + inp6;
          t5 = inp4 + inp8;
          m3 = inp1 - inp5;
          m6_imag = inp7 - inp3;
          t4 = inp2 - inp6;
```

```
            t6 = inp4 - inp8;
            t8 = t5 + t3;
            t7 = t1 + t2;
            m0 = t7 + t8;
            t1_46 = sin_45 * ( t4 - t6);
            m4 = t1_46 [23:8];
            m5_imag = t5 - t3;
            m2 = t1 - t2;
            m1 = t7 - t8;
            t2_46 = sin_315 * ( t4 + t6);
            m7_imag = t2_46 [23:8];
            s1 = m3 + m4;
            s2 = m3 - m4;
            s3_imag = m6_imag + m7_imag;
            s4_imag = m6_imag - m7_imag;
            output_stb = 1'b1;
          end
        end
    assign out1_real = m0;
    assign out1_imag = 16'b0000000000000000;
    assign out2_real = s1;
    assign out2_imag = s3_imag;
    assign out3_real = m2;
    assign out3_imag = m5_imag;
    assign out4_real = s2;
    assign out4_imag = ~s4_imag + 1'b1;
    assign out5_real = m1;
    assign out5_imag = 16'b0000000000000000;
    assign out6_real = s2;
    assign out6_imag = s4_imag;
    assign out7_real = m2;
    assign out7_imag = ~m5_imag + 1'b1;
    assign out8_real = s1;
    assign out8_imag = ~s3_imag + 1'b1;
    assign out_stb = output_stb;
```

```
endmodule
```

A.2 Code for modified Cooley-Tukey algorithm with one multiplier reused

```
`timescale 1ns / 1ps

module multiplier(
    input signed [15:0] inp1,
    input signed [15:0] inp2,
    output signed[15:0] out
    );
  reg signed [31:0] inp12;
  always @ ( * )
    begin
      inp12 = inp1*inp2;
    end
  assign out = inp12 [23:8];
endmodule


module fft8(
  input signed [15:0] inp1,
  input signed [15:0] inp2,
  input signed [15:0] inp3,
  input signed [15:0] inp4,
  input signed [15:0] inp5,
  input signed [15:0] inp6,
  input signed [15:0] inp7,
  input signed [15:0] inp8,
  input clk,
  input rst,
  output signed [15:0] out1_real,
  output signed [15:0] out1_imag,
  output signed [15:0] out2_real,
  output signed [15:0] out2_imag,
  output signed [15:0] out3_real,
  output signed [15:0] out3_imag,
  output signed [15:0] out4_real,
  output signed [15:0] out4_imag,
  output signed [15:0] out5_real,
  output signed [15:0] out5_imag,
  output signed [15:0] out6_real,
  output signed [15:0] out6_imag,
  output signed [15:0] out7_real,
  output signed [15:0] out7_imag,
  output signed [15:0] out8_real,
  output signed [15:0] out8_imag,
  output out_stb
);

  localparam signed sin_45 = 16'b00000000_10110101;
  localparam signed sin_315 = 16'b11111111_01001011;
  localparam signed sf = 2.0**-8.0;

  reg signed [31:0] t1_46,t2_46;
  reg signed [15:0] t1,t2,t3,t4,t5,t6,t7,t8,m0,m1,m2,m3,m4;
  reg signed [15:0] m7_imag,s1,s2,s3_imag,s4_imag;
  reg signed [15:0] m5_imag,m6_imag;
  reg [15:0] mult_inp1,mult_inp2;
  wire [15:0] mult_out;
  reg [1:0] stage;
  reg output_stb;

  multiplier mult (
    .inp1(mult_inp1),
    .inp2(mult_inp2),
    .out(mult_out)
  );
```

```

initial
  begin
    stage = 2'b00;
    output_stb = 1'b0;
  end

always @( posedge clk)
  begin
    if (rst == 1'b1)
      begin
        output_stb = 1'b0;
      end
    if (stage == 2'b00 && rst == 1'b0)
      begin
        $display("stage-1");
        t1 = inp1 + inp5;
        t2 = inp7 + inp3;
        t3 = inp2 + inp6;
        t5 = inp4 + inp8;
        m3 = inp1 - inp5;
        m6_imag = inp7 - inp3;
        t4 = inp2 - inp6;
        t6 = inp4 - inp8;
        t8 = t5 + t3;
        t7 = t1 + t2;
        m0 = t7 + t8;
        mult_inp1 = t4 - t6;
        mult_inp2 = sin_45;
        stage = 2'b01;
      end
    else if (stage == 2'b01)
      begin
        $display("stage-2");
        m4 = mult_out;
        m5_imag = t5 - t3;
        m2 = t1 - t2;
        m1 = t7 - t8;
        mult_inp1 = t4 + t6;
        mult_inp2 = sin_315;
        stage = 2'b10;
      end
    else if (stage == 2'b10)
      begin
        m7_imag = mult_out;
        s1 = m3 + m4;
        s2 = m3 - m4;
        s3_imag = m6_imag + m7_imag;
        s4_imag = m6_imag - m7_imag;
        stage = 2'b00;
        output_stb = 1'b1;
      end
  end
assign out_stb = output_stb;
assign out1_real = m0;
assign out1_imag = 16'b0000000000000000;
assign out2_real = s1;
assign out2_imag = s3_imag;
assign out3_real = m2;
assign out3_imag = m5_imag;
assign out4_real = s2;
assign out4_imag = ~s4_imag + 1'b1;
assign out5_real = m1;
assign out5_imag = 16'b0000000000000000;
assign out6_real = s2;
assign out6_imag = s4_imag;
assign out7_real = m2;
assign out7_imag = ~m5_imag + 1'b1;
assign out8_real = s1;
assign out8_imag = ~s3_imag + 1'b1;
```

```
endmodule
```

A.3 Code for modified Cooley-Tukey algorithm with LUT based multiplier

```
`timescale 1ns / 1ps

module multiplier(
    input signed [15:0] inp1,
    input signed [15:0] inp2,
    input rst,
    input clk,
    output signed [15:0] out,
    output out_stb
);
  localparam sf = 2.0**-8.0;
  reg [29:0] inp12;
  reg [14:0] input_1;
  reg [14:0] input_2;
  reg output_stb;
  reg out_sign;
  integer counter;
  assign out_stb = output_stb;
  initial
    begin
      output_stb = 1'b0;
      inp12 = 32'b0;
      counter = 0;
    end

always @ ( posedge clk )
begin
if (rst == 1'b1)
  begin
  output_stb = 1’b0;
  inp12 = 30’b0;
  counter = 0;
  end
else if (output_stb == 1’b0 && counter < 15)
  begin
  if (counter == 0)
  begin
  out_sign = (inp1[15] && ~inp2[15]) + (inp2[15] && ~inp1[15]);
  input_1 = inp1[15]==1’b0 ? inp1[14:0] : ~inp1[14:0]+1’b1;
  input_2 = inp2[15]==1’b0 ? inp2[14:0] : ~inp2[14:0]+1’b1;
  end
  if(input_1[counter]==1’b1)
  begin
  inp12[29:14] = inp12[29:14] + input_2[14:0];
  end
  inp12 = inp12 >> 1;
  counter = counter + 1;
  end
else if (counter >= 15)
  begin
    output_stb = 1’b1;
  end
end
assign out = {out_sign,inp12[22:7]};
endmodule



module fft8(
  input signed [15:0] inp1,
  input signed [15:0] inp2,
  input signed [15:0] inp3,
  input signed [15:0] inp4,
  input signed [15:0] inp5,
  input signed [15:0] inp6,
  input signed [15:0] inp7,
  input signed [15:0] inp8,
  input clk,
  input rst,
  output signed [15:0] out1_real,
  output signed [15:0] out1_imag,
  output signed [15:0] out2_real,
  output signed [15:0] out2_imag,
  output signed [15:0] out3_real,
  output signed [15:0] out3_imag,
  output signed [15:0] out4_real,
  output signed [15:0] out4_imag,
  output signed [15:0] out5_real,
  output signed [15:0] out5_imag,
  output signed [15:0] out6_real,
  output signed [15:0] out6_imag,
  output signed [15:0] out7_real,
  output signed [15:0] out7_imag,
  output signed [15:0] out8_real,
  output signed [15:0] out8_imag,
  output out_stb
);

  localparam signed sin_45 = 16'b00000000_10110101;
  localparam signed sin_315 = 16'b11111111_01001011;
  localparam signed sf = 2.0**-8.0;

  reg signed [31:0] t1_46,t2_46;
  reg signed [15:0] t1,t2,t3,t4,t5,t6,t7,t8,m0,m1,m2,m3,m4;
  reg signed [15:0] m5_imag,m6_imag,m7_imag,s1,s2;
  reg [15:0] mult_inp1,mult_inp2,s3_imag,s4_imag;
  wire [15:0] mult_out;
  reg [1:0] stage;
  reg output_stb,mult_rst;
```

```
wire mult_stb;

multiplier mult (
  .inp1(mult_inp1),
  .inp2(mult_inp2),
  .rst(mult_rst),
  .out(mult_out),
  .clk(clk),
  .out_stb(mult_stb)
);

initial
  begin
    stage = 2'b00;
    output_stb = 1'b0;
    mult_rst = 1'b1;
  end

always @( posedge clk)
  begin
    if (rst == 1'b1)
      begin
        output_stb = 1'b0;
        mult_rst = 1'b1;
      end
    if (stage == 2'b00 && rst == 1'b0 && output_stb == 1'b0)
      begin
        t1 = inp1 + inp5;
        t2 = inp7 + inp3;
        t3 = inp2 + inp6;
        t5 = inp4 + inp8;
        m3 = inp1 - inp5;
        m6_imag = inp7 - inp3;
        t4 = inp2 - inp6;
        t6 = inp4 - inp8;
```

```
          t8 = t5 + t3;
          t7 = t1 + t2;
          m0 = t7 + t8;
          mult_inp1 = t4 - t6;
          mult_inp2 = sin_45;
          stage = 2'b01;
          mult_rst = 1'b0;
        end
      else if (stage == 2'b01 && mult_stb == 1'b1)
        begin
          m4 = mult_out;
          mult_rst = 1'b1;
          stage = 2'b10;
        end
      else if (stage == 2'b10)
        begin
          m5_imag = t5 - t3;
          m2 = t1 - t2;
          m1 = t7 - t8;
          mult_inp1 = t4 + t6;
          mult_inp2 = sin_315;
          mult_rst = 1'b0;
          stage = 2'b11;
        end
      else if (stage == 2'b11 && mult_stb == 1'b1)
        begin
          m7_imag = mult_out;
          s1 = m3 + m4;
          s2 = m3 - m4;
          s3_imag = m6_imag + m7_imag;
          s4_imag = m6_imag - m7_imag;
          mult_rst = 1'b1;
          stage = 2'b00;
          output_stb = 1'b1;
        end
```

```
    end
  assign out_stb = output_stb;
  assign out1_real = m0;
  assign out1_imag = 16'b0000000000000000;
  assign out2_real = s1;
  assign out2_imag = s3_imag;
  assign out3_real = m2;
  assign out3_imag = m5_imag;
  assign out4_real = s2;
  assign out4_imag = ~s4_imag + 1'b1;
  assign out5_real = m1;
  assign out5_imag = 16'b0000000000000000;
  assign out6_real = s2;
  assign out6_imag = s4_imag;
  assign out7_real = m2;
  assign out7_imag = ~m5_imag + 1'b1;
  assign out8_real = s1;
  assign out8_imag = ~s3_imag + 1'b1;
endmodule
```

A.4 Code for modified Cooley-Tukey algorithm with CORDIC based multiplier

```
`timescale 1ns / 1ps

`define K 32'h26dd3b6a
`define BETA_0  32'h3243f6a9
`define BETA_1  32'h1dac6705
`define BETA_2  32'h0fadbafd
`define BETA_3  32'h07f56ea7
`define BETA_4  32'h03feab77
`define BETA_5  32'h01ffd55c
`define BETA_6  32'h00fffaab
`define BETA_7  32'h007fff55
```

```verilog
`define BETA_8  32'h003fffeb
`define BETA_9  32'h001ffffd
`define BETA_10 32'h00100000
`define BETA_11 32'h00080000
`define BETA_12 32'h00040000
`define BETA_13 32'h00020000
`define BETA_14 32'h00010000
`define BETA_15 32'h00008000
`define BETA_16 32'h00004000
`define BETA_17 32'h00002000
`define BETA_18 32'h00001000
`define BETA_19 32'h00000800
`define BETA_20 32'h00000400
`define BETA_21 32'h00000200
`define BETA_22 32'h00000100
`define BETA_23 32'h00000080
`define BETA_24 32'h00000040
`define BETA_25 32'h00000020
`define BETA_26 32'h00000010
`define BETA_27 32'h00000008
`define BETA_28 32'h00000004
`define BETA_29 32'h00000002
`define BETA_30 32'h00000001
`define BETA_31 32'h00000000

module multiplier(
  clock,
  reset,
  start,
  angle_in,
  sin_out,
  initial_value,
  out_stb
);

```

```
input clock;
input reset;
input [31:0] angle_in;
input start;
input signed [15:0] initial_value;
output out_stb;
reg out_stb_reg;
output signed [15:0] sin_out;
assign out_stb = out_stb_reg;
wire [15:0] sin_out = sin_final[23:8];

reg signed [31:0] cos_final;
reg signed [31:0] sin_final;

reg signed [31:0] cos;
reg signed [31:0] sin;
reg [31:0] angle;
reg [4:0] count;
reg state;

reg [31:0] cos_next;
reg [31:0] sin_next;
reg [31:0] angle_next;
reg [4:0] count_next;
reg state_next;

always @(posedge clock or posedge reset) begin
  if (reset)
    begin
      cos <= 0;
      sin <= 0;
      angle <= 0;
      count <= 0;
      state <= 0;
    end
  else
    begin
      cos <= cos_next;
      sin <= sin_next;
      angle <= angle_next;
      count <= count_next;
      state <= state_next;
    end
end

always @* begin
  cos_next = cos;
  sin_next = sin;
  angle_next = angle;
  count_next = count;
  state_next = state;
  if (state) begin
    cos_next = cos + (direction_negative ? sin_shr : -sin_shr);
    sin_next = sin + (direction_negative ? -cos_shr : cos_shr);
    angle_next = angle + (direction_negative ? beta : -beta);
    count_next = count + 1;
    if (count == 31) begin
      state_next = 0;
    out_stb_reg = 1'b1;
    sin_final = {sin[31],6'b000000,sin[31:22]} * initial_value;
    end
  end
  else begin
    if (start) begin
      cos_next = `K;
      sin_next = 0;
      angle_next = angle_in;
      count_next = 0;
      state_next = 1;
    end
  end
end

wire [31:0] cos_signbits = {32{cos[31]}};
wire [31:0] sin_signbits = {32{sin[31]}};
wire [31:0] cos_shr = {cos_signbits, cos} >> count;
wire [31:0] sin_shr = {sin_signbits, sin} >> count;
wire direction_negative = angle[31];
wire [31:0] beta_lut [0:31];
assign beta_lut[0] = `BETA_0;
assign beta_lut[1] = `BETA_1;
assign beta_lut[2] = `BETA_2;
assign beta_lut[3] = `BETA_3;
assign beta_lut[4] = `BETA_4;
assign beta_lut[5] = `BETA_5;
assign beta_lut[6] = `BETA_6;
assign beta_lut[7] = `BETA_7;
assign beta_lut[8] = `BETA_8;
assign beta_lut[9] = `BETA_9;
assign beta_lut[10] = `BETA_10;
assign beta_lut[11] = `BETA_11;
assign beta_lut[12] = `BETA_12;
assign beta_lut[13] = `BETA_13;
assign beta_lut[14] = `BETA_14;
assign beta_lut[15] = `BETA_15;
assign beta_lut[16] = `BETA_16;
assign beta_lut[17] = `BETA_17;
assign beta_lut[18] = `BETA_18;
assign beta_lut[19] = `BETA_19;
assign beta_lut[20] = `BETA_20;
assign beta_lut[21] = `BETA_21;
assign beta_lut[22] = `BETA_22;
assign beta_lut[23] = `BETA_23;
assign beta_lut[24] = `BETA_24;
assign beta_lut[25] = `BETA_25;
assign beta_lut[26] = `BETA_26;
assign beta_lut[27] = `BETA_27;
assign beta_lut[28] = `BETA_28;
assign beta_lut[29] = `BETA_29;
assign beta_lut[30] = `BETA_30;
assign beta_lut[31] = `BETA_31;
wire [31:0] beta = beta_lut[count];
endmodule

module fft8(
  input signed [15:0] inp1,
  input signed [15:0] inp2,
  input signed [15:0] inp3,
  input signed [15:0] inp4,
  input signed [15:0] inp5,
  input signed [15:0] inp6,
  input signed [15:0] inp7,
  input signed [15:0] inp8,
  input clk,
  input rst,
  output signed [15:0] out1_real,
  output signed [15:0] out1_imag,
  output signed [15:0] out2_real,
  output signed [15:0] out2_imag,
  output signed [15:0] out3_real,
  output signed [15:0] out3_imag,
  output signed [15:0] out4_real,
  output signed [15:0] out4_imag,
  output signed [15:0] out5_real,
  output signed [15:0] out5_imag,
  output signed [15:0] out6_real,
  output signed [15:0] out6_imag,
  output signed [15:0] out7_real,
  output signed [15:0] out7_imag,
  output signed [15:0] out8_real,
  output signed [15:0] out8_imag,
  output out_stb
);

  localparam signed sin_45 = 16’b00000000_10110101;
  localparam signed sin_315 = 16’b11111111_01001011;
  localparam signed sf = 2.0**-8.0;
  reg signed [31:0] t1_46,t2_46,mult_inp1;
  reg signed [15:0] t1,t2,t3,t4,t5,t6,t7,t8,m0,m1,m2,m3,m4;
  reg signed [15:0] m5_imag,m6_imag,m7_imag,s1,s2,s3_imag,s4_imag;
  reg [15:0] mult_inp2;
  wire [15:0] mult_out;
  reg [1:0] stage;
  reg output_stb,mult_rst,mult_start;
  wire mult_stb;

  multiplier mult (
    .angle_in(mult_inp1),
    .initial_value(mult_inp2),
    .reset(mult_rst),
    .sin_out(mult_out),
    .clock(clk),
    .out_stb(mult_stb),
    .start(mult_start)
  );

  initial
    begin
      stage = 2’b00;
      output_stb = 1’b0;
      mult_rst = 1’b1;
    end

  always @( posedge clk)
    begin
```

```
      if (rst == 1'b1)
        begin
          output_stb = 1'b0;
          mult_rst = 1'b1;
        end
      if (stage == 2'b00 && rst == 1'b0 && output_stb == 1'b0)
        begin
          t1 = inp1 + inp5;
          t2 = inp7 + inp3;
          t3 = inp2 + inp6;
          t5 = inp4 + inp8;
          m3 = inp1 - inp5;
          m6_imag = inp7 - inp3;
          t4 = inp2 - inp6;
          t6 = inp4 - inp8;
          t8 = t5 + t3;
          t7 = t1 + t2;
          m0 = t7 + t8;
          mult_inp1 = t4 - t6;
          mult_inp2 = 32'h3243f6a9;
          stage = 2'b01;
          mult_rst = 1'b0;
          mult_start = 1'b1;
        end
      else if (stage == 2'b01 && mult_stb == 1'b1)
        begin
          m4 = mult_out;
          mult_rst = 1'b1;
          stage = 2'b10;
        end
      else if (stage == 2'b10)
        begin
          m5_imag = t5 - t3;
          m2 = t1 - t2;
          m1 = t7 - t8;
          mult_inp1 = -t4 - t6;
          mult_inp2 = 32'h3243f6a9;
          mult_rst = 1'b0;
          stage = 2'b11;
        end
      else if (stage == 2'b11 && mult_stb == 1'b1)
        begin
          m7_imag = mult_out;
          s1 = m3 + m4;
          s2 = m3 - m4;
          s3_imag = m6_imag + m7_imag;
          s4_imag = m6_imag - m7_imag;
          mult_rst = 1'b1;
          stage = 2'b00;
          output_stb = 1'b1;
        end
    end
  assign out_stb = output_stb;
  assign out1_real = m0;
  assign out1_imag = 16'b0000000000000000;
  assign out2_real = s1;
  assign out2_imag = s3_imag;
  assign out3_real = m2;
  assign out3_imag = m5_imag;
  assign out4_real = s2;
  assign out4_imag = ~s4_imag + 1'b1;
  assign out5_real = m1;
  assign out5_imag = 16'b0000000000000000;
  assign out6_real = s2;
  assign out6_imag = s4_imag;
  assign out7_real = m2;
  assign out7_imag = ~m5_imag + 1'b1;
  assign out8_real = s1;
  assign out8_imag = ~s3_imag + 1'b1;
endmodule
```

A.5 Python Code for Verifying fft8 blocks

```python

import numpy as np
from bitstring import Bits
import os
implement=raw_input("design to test(allowed values 1,2,3,4)")
working_directory=r"/FFT8_implementation_"+str(implement)

print("Generating 100 Random Inputs ... ")
cmds = ""
for i in range(100):
    input_array=np.random.uniform(low=-8.0, high=8.0, size=(8,))
    binary_array=[]
    for inp in input_array:
        binary_array.append(Bits(int=int(inp*2**8), length=16).bin)
   cmd="""
     isim force add {/fft8_tb/inp1} %s -radix bin
     isim force add {/fft8_tb/inp2} %s -radix bin
     isim force add {/fft8_tb/inp3} %s -radix bin
     isim force add {/fft8_tb/inp4} %s -radix bin
     isim force add {/fft8_tb/inp5} %s -radix bin
     isim force add {/fft8_tb/inp6} %s -radix bin
     isim force add {/fft8_tb/inp7} %s -radix bin
     isim force add {/fft8_tb/inp8} %s -radix bin
     isim force add {/fft8_tb/clk} 1 -radix bin -value 0 -radix
     bin -time 2500 ps -repeat 5 ns -cancel 1 us
     isim force add {/fft8_tb/rst} 1 -radix bin -cancel 20 ns
     isim force add {/fft8_tb/rst} 0 -radix bin -time 20 ps
    -cancel 1 us
     run
     dump
    """%tuple(binary_array)
   cmds=cmds+cmd

os.chdir(working_directory)
f=open("inp.test","w")
f.write(cmds)
f.close()
```

```
print("Simulating all the inputs using ISIM...")
cmd='"'+working_directory+r'/fft8_tb_isim_beh.exe" < inp.test > out.test'
os.system(cmd)
f=open("out.test","r")
values=[]
inp={}
out={}
for line in f.readlines():
 print line
 if "Signal:" in line:
  out[line.strip().split("{")[1].split("}")[0].split("[")[0].strip()]=line.strip().split(":")[-1].strip()
 if "Variable:" in line:
  inp[line.strip().split("{")[1].split("}")[0].split("[")[0].strip()]=line.strip().split(":")[-1].strip()
 if "{rst}" in line.strip():
  values.append([inp,out])
  inp={}
  out={}

f.close()

print("Verifying FFT output with the actual values...")
correct=0
count=0
for val in values:
  count=count+1
  inp = np.asarray([Bits(bin=val[0]['inp'+str(i)]).int/(2.0**8) for i in range(1,9)])
  out1 = np.array([Bits(bin=val[1]['out'+str(i)+"_real"]).int/(2.0**8) for i in range(1,9)])
  out2 = np.array([Bits(bin=val[1]['out'+str(i)+"_imag"]).int/(2.0**8) for i in range(1,9)])
  out=out1 + 1j*out2
  if np.allclose(out,np.fft.fft(inp),1e-2):
   correct=correct+1

print(str(correct)+r"/"+str(count)+" Correct ...")
```

REFERENCES

Li, Junwei Fang, Jiandong Li, Bajin Zhao, Yudong. (2016). *Study of CORDIC algorithm based on FPGA*. 4338-4343. 10.1109/CCDC.2016.7531747.

Robert Scott (2000). *Doing Hartley Smartly*.Embedded systems programming.

Amente Bekele(2016). *Cooley-Tukey FFT Algorithms*.COMP 5703: ADVANCED ALGORITHMS, FALL 2016

Meher, P.K.. (2010). *LUT Optimization for Memory-Based Computation*. IEEE Trans. on Circuits and Systems. 57-II. 285-289. 10.1109/TCSII.2010.2043467.

Lin, Sheng Liu, Ning Nazemi, Mahdi Li, Hongjia Ding, Caiwen Wang, Yetang Pedram, Massoud. (2017). *FFT-Based Deep Learning Deployment in Embedded Systems*.

Memon, Tayab Pathan, Aneela. (2018). *An approach to LUT based multiplier for short word length DSP systems*. 276-280. 10.1109/ICSIGSYS.2018.8372772.

Beaudoin, Normand Beauchemin, Steven. (2002). *An accurate discrete Fourier transform for image processing*. Proceedings - International Conference on Pattern Recognition. 3. 935 - 939 vol.3. 10.1109/ICPR.2002.1048189.

Roxburgh, Alastair. (2013). *On Computing the Discrete Fourier Transform*.