2004

Implementation of Chirp-z Discrete Fourier Transform on Virtex II FPGA

Hariharan Natarajan
IMPLEMENTATION OF CHIRP-Z DISCRETE FOURIER TRANSFORM ON VIRTEX II FPGA

By

HARIHARAN NATARAJAN

A Thesis submitted to the Department of Electrical and Computer Engineering in partial fulfillment of the requirements for the degree of Master of Science

Degree Awarded: Spring Semester, 2004
The members of the Committee approve the Thesis of Hariharan Natarajan defended on 9th April 2004.

Dr. Uwe Meyer-Baese  
Professor Directing Thesis

Dr. Anke Meyer-Baese  
Committee Member

Dr. Simon Foo  
Committee Member

Approved:

Dr. Reginald Perry, Chair, Dept. of Electrical and Computer Engineering

Dr. Ching-Jen Chen, Dean, College of Engineering

The Office of Graduate Studies has verified and approved the above named committee members.
Dedicated to my parents S. Natarajan, Anandhi Natarajan
And my brother Venkataraman Natarajan
I would like to express my gratitude to all those who helped to complete this thesis. I want to thank Dr. Anke Meyer-Baese for her constant support and Dr. Simon Foo for all the help he has offered during the course of my studies.

I am deeply indebted to my advisor Dr. Uwe Meyer-Baese whose help; suggestions and encouragement helped me during the time of research for and writing of this thesis. I want to thank all the faculty members and the staff of Electrical and Computer engineering department for their support during my entire graduate studies. I also want to thank my friends at FSU for their immense support and encouragement.
TABLE OF CONTENTS

List of Acronyms ........................................................................................................ vii
List of Figures ........................................................................................................... ix
List of Tables ............................................................................................................. x
Abstract ................................................................................................................... xi

1. INTRODUCTION .................................................................................................. 1

2. PROGRAMMABLE DEVICE ARCHITECTURES ............................................. 2
   2.1 Programmable Logic Array ................................................................. 2
   2.2 Programmable Array Logic (PAL) Device ........................................... 3
   2.3 Complex Programmable Logic Device (CPLD) ..................................... 4
   2.4 Field Programmable Gate Array (FPGA) ............................................ 6
   2.5 Xilinx Virtex II Family ......................................................................... 9

3. DISCRETE FOURIER TRANSFORM ............................................................... 13
   3.1 Introduction ......................................................................................... 13
   3.2 Fourier Transform .............................................................................. 13
   3.3 Discrete Fourier Transform ............................................................... 13
   3.4 Properties of Discrete Fourier Transform ......................................... 14
   3.5 Fast Fourier Transform ...................................................................... 14
   3.6 The Bluestein Chirp-z Transform ..................................................... 15

4. IMPLEMENTATION OF CHIRP-Z DISCRETE FOURIER TRANSFORM ....... 21
   4.1 Implementation of ROM to store the twiddle factor............................ 21
   4.2 Complex number multiplier ............................................................... 21
   4.3 FIR Filter ............................................................................................ 22

5. COMPUTER ARITHMATIC ALGORITHMS FOR FILTER DESIGN .......... 24
   5.1 Canonical Signed Digit System ........................................................... 24
   5.2 Multiplier Adder Graph (MAG) ........................................................... 26
   5.3 Reduced Adder Graph (RAG) .............................................................. 28
LIST OF ACRONYMS

A/D: Analog to Digital
ASIC: Application Specific Integrated Circuit
CLB: Configurable Logic Block
CMOS: Complimentary Metal Oxide Semi-conductor
CPLD: Complex Programmable Logic Device
CSA: Carry Save Adder
CSD: Canonical Signed Digits
CZT: Chirp-z Transform
DCM: Digital clock Manager
DFT: Discrete Fourier Transform
DSP: Digital Signal Processing
EPROM: Electrically Programmable Read Only Memory
EEPROM: Electrically Erasable Programmable Read Only Memory
FIR: Finite Impulse Response
FFT: Fast Fourier Transform
FPGA: Field Programmable Gate Arrays
FPLD: Field programmable Logic Devices.
IIR: Infinite Impulse Response
ISE: Integrated Synthesis Environment
LAB: Logic Array Blocks
LC: Logic Cell
LE: Logic Element
LSB: Least Significant Bit
LTI: Linear Time Invariant
LUT: Look-Up Table
MAC: Multiply Accumulate
MAG: Multiplier Adder Graph
MAX: Multiple Array Matrix
MSB: Most Significant Bit
PAL: Programmable Array Logic
PIA: Programmable Interconnect Array
PLA: Programmable Logic Array
PLL: Phase Locked Loop
RAG: Reduced Order Graph
RAM: Random Access Memory
ROM: Read Only Memory
SD: Signed Digit
SOC: System On Chip
SRAM: Synchronous Random Access Memory
VHSIC: Very High Speed Integrated Circuit
VHDL: VHSIC Hardware Descriptive Language
VLSI: Very Large Scale Integrated Circuit.
LIST OF FIGURES

Figure 2.1.1 Programmable Logic Array.................................................. 2
Figure 2.2.1 Programmable Array Logic (PAL) Device............................. 3
Figure 2.3.1 CPLD Architecture............................................................... 4
Figure 2.3.2 Simple Programmable Logic Device Architecture..................... 5
Figure 2.4.1 Medium Density FPGA Architecture..................................... 6
Figure 2.4.2 Logic Block Architecture...................................................... 7
Figure 2.4.3 Altera Flex 10 K Device Architecture.................................... 8
Figure 2.4.4 Xilinx Spartan II Device Architecture.................................. 8
Figure 2.5.1 Xilinx Virtex II FPGA Architecture..................................... 9
Figure 3.6.1 The Bluestein Chirp z DFT.................................................... 18
Figure 4.3.1 Direct Form FIR Filter......................................................... 23
Figure 4.3.2 Transposed Form FIR filter.................................................. 23
Figure 5.2.1 Comparison of CSD coding and MAG coding.......................... 27
Figure 5.2.2 Possible Cost one to four graphs for CSD and RAG................... 27
Figure 5.3.1 RAG implementation example.............................................. 29
Figure 5.3.2 FIR Filter with multiplier block......................................... 29
Figure 6.1.1 DFT length v/s SNR plot...................................................... 32
LIST OF TABLES

Table 2.5.1 Virtex II Family Devices................................................................. 10
Table 2.5.2 Comparison of High-density FPGA families........................................ 12
Table 3.6.1 Table of DFT length and filter coefficients........................................ 19
Table 3.6.2 DFT length and non-trivial coefficients............................................. 20
Table 5.1.1 Cost incurred using CSD algorithm for 8 bit numbers......................... 25
Table 5.2.1 Coefficients improved due to MAG. ................................................. 26
Table 5.3.1 Example for implementation of RAG................................................. 28
Table 6.1 DFT Synthesis Results for optimized speed......................................... 30
Table 6.2 DFT Synthesis Results for optimized area............................................ 30
Table 6.3 SNR for various DFT lengths.............................................................. 32
ABSTRACT

The application of Fourier results in conversion in representation of a signal in time domain to frequency domain. Hence, it forms an important tool for frequency analysis. With advent of digital computers, we can perform frequency analysis faster and more efficiently. Thus discrete Fourier transform is important for frequency analysis of signal in discrete form.

Discrete Fourier Transform (DFT) and Fast Fourier Transform (FFT) algorithms have been invented in several variations. This thesis focuses on implementation of the Bluestein Chirp-z transform algorithm. This method uses chirp signals, which are complex exponential signals, which increase linearly with time. Hence, the name chirp-z Transform. The transform is implemented on Xilinx Inc.’s Virtex II FPGA. Virtex II family has two of the world’s largest programmable device with gate count up to 8 million. It’s features like embedded multiplier and memory make it ideal for digital signal processing applications. The implementation of chirp-z transform would involve designing a ROM to store the twiddle factors; a complex number multiplier and FIR filter for convolution. Again, we look at various algorithms for calculation of filter coefficient for minimum cost of adder and multiplier.

DFT are implemented for length 4 point, 16 point, 32 point and 64 point. We analyze each of the above-mentioned implementations and especially the space occupied and the speed of the device.
CHAPTER 1

INTRODUCTION

Programmable Devices are integrated circuits which can be programmed “in house” or on the field. The design and implementation of an application on these devices can be achieved with the help of software tools. Hardware descriptive languages such as Verilog and VHDL are widely used for this purpose. The codes written in these languages can also be synthesized using a third party Electronic Design and Automation tool (EDA) tool or the software tool provided by the vendor. With the help of these tools it is also possible to optimize the design for speed or space.

In this thesis we focus on implementation of chirp-\(z\) transform (CZT) on a third generation programmable device (FPGA) from the Xilinx Inc. called the Virtex II. Chapter 2 discusses the evolution of programmable device architectures and in particularly the FPGAs and then to features of the device used. Then we shift our focus on the application, the Discrete Fourier Transform (DFT) in chapter 3. Here, we discuss some of their properties and concentrate on the chirp-\(z\) algorithm, the basic theory and some of it’s advantages and properties. The basic building- blocks of the CZT system is discussed in chapter 4. Chapter 5 deals with various arithmetic algorithms used for the FIR filter design and how we conclude on the most efficient algorithm. Lastly, we look at the results of DFTs performed for various lengths- 4 points, 16 points, 32 points and 64 points. The codes can be found in the appendix section.
CHAPTER 2

PROGRAMMABLE DEVICE ARCHITECTURES

The programmable devices have gradually grown to immense prominence in the field of digital signal processing. The gates have grown from a group of AND/OR gates accomplishing simple “sum of products” operations to millions of gates on a single substrate with additional devices like memory elements, multiplier and also in built processors (Virtex II Pro from Xilinx Inc.). In this chapter we shall discuss the evolution of programmable devices architecture. We shall discuss various FPGA technologies and in particular Virtex II family from Xilinx Inc. which we use to implement Discrete Fourier Transform.

2.1 Programmable Logic Array

Programmable logic devices are one of the first programmable devices developed. They consist of a layer each of AND gates and OR gates. They accomplish the Sum of Product operation. Hence, the number of inputs cannot exceed the number of AND gates.

Figure 2.1.1 Programmable Logic Array
2.2 Programmable Array Logic

Programmable Array Logic devices are similar to PLA. They were introduced by Monolithic Memories (now a part of Advanced Micro Devices). These devices also have only AND and OR gates. However, unlike PLA, only AND gates are programmable and every OR gate is connected to a bunch of AND gates. Thus, the maximum number of minterms allowed for an OR gate is equal to the number of inputs to the OR gate. The logic function of higher minterms can be implemented by routing the output of one OR gate to input of another minterm.

Figure 2.2.1 Programmable Array Logic (PAL) Device
2.3 Complex Programmable Logic Device (CPLD)

A complex programmable logic device, as the name suggests, is more complex than the previously discussed architectures. Firstly, they are large granularity devices and consist of a group of arrays of logic elements or logic cell, which are connected through wide busses (called Programmable Interconnect Array (PIA) by Altera) with short delays. The logic cells typically have 8 to 10 inputs, 3 to 4 outputs, and support around 20 product terms. The data path is not unidirectional from input to output of the IC; instead outputs of all the arrays are fed back to the PIA. The output of the LC that is required to be fed as input into another cell is first routed back to the common interconnect lines and then connected to designation logic. By combining the bus and the fixed LC timing, it is possible to provide predictable and short pin-to-pin delays in CPLDs.

Figure 2.3.1 CPLD Architecture
Altera Corporation released the Multiple Array Matrix (MAX) devices as a part of the CPLD family. These devices comprised of MAX 5000, MAX 3000A, MAX 7000 and MAX 9000. While MAX 5000 uses EPROM technology, other devices use EEPROM technology. Xilinx Inc. released the XPLA2, ‘Cool Runner XPLA3’ and XC 9500 as a part of its CPLD family which use Flash memory technology. Though both Flash and EEPROM are electrically erasable technology the difference is that in EEPROM the chip erase bit by bit where as in Flash, large blocks are erased at a time.
2.4 Field Programmable Gate Array (FPGA)

Traditional gate arrays contain a number of building blocks or primitive cells etched on a single silicon substrate. The connections between cells are permanent and made later. These are non-reprogrammable high-density devices containing about 5 millions gates. The FPGAs have similar structure to gate arrays however they have programmable elements. The programmable cell is called Logic Element (LE) in case of Altera device and Configurable Logic Block (CLB) in Xilinx devices. FPGA use the Complementary Metal Oxide Semiconductor SRAM technology and are thus reset at power off.

Figure 2.4.1 Medium Density FPGA Architecture
The competing second-generation mid-density programmable logic devices are FLEX devices from Altera and Spartan devices (XC 3000, XC 4000 and XC 5200) family from Xilinx Inc. These devices not only have higher gate count but also have embedded Phase Locked Loops (PLL), and hence clocks are synchronized.

The next generations of Xilinx FPGAs are high-density devices that have upto 8 million gates. Some of these low power consuming devices also have specialized block like multipliers (Virtex II), microprocessor (Virtex II Pro), etc.
Figure 2.4.3 Altera Flex 10 K Device Architecture

Figure 2.4.4 Xilinx Spartan II Device Architecture
2.5 Xilinx Virtex II Family

The Virtex-II family is one of Xilinx’s premier FPGA offering. The Virtex-II family features two of the world’s largest programmable logic devices (the six million gate 2V6000 and the eight million gate 2V8000). Virtex-II FPGAs are loaded with features that advanced designs require such as XCITE technology (Digitally Controlled Impedance), advanced Digital Clock Managers (DCMs) for clock synchronization, embedded memory and embedded hardware multipliers and hence suitable for System on Chip (SOC) designs. The Virtex II family consists of 11 devices covering a wide range of density. The details of each device are given in the Table 2.5.1.

Figure 2.5.1 Xilinx Virtex II FPGA Architecture
<table>
<thead>
<tr>
<th>Virtex II Device</th>
<th>Max. System Gates</th>
<th>Logic Cells</th>
<th>Embedded BRAM Memory</th>
<th>Max. Distributed Memory</th>
<th>18x18 Multiplier Block</th>
<th>DCMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>XC2V8000</td>
<td>8M</td>
<td>104,832</td>
<td>3.024 Mbits</td>
<td>1.456 Mbits</td>
<td>168</td>
<td>12</td>
</tr>
<tr>
<td>XC2V6000</td>
<td>6M</td>
<td>76,032</td>
<td>2.592 Mbits</td>
<td>1.056 Mbits</td>
<td>144</td>
<td>12</td>
</tr>
<tr>
<td>XC2V4000</td>
<td>4M</td>
<td>51,840</td>
<td>2.160 Mbits</td>
<td>720 Kbits</td>
<td>120</td>
<td>12</td>
</tr>
<tr>
<td>XC2V3000</td>
<td>3M</td>
<td>32,256</td>
<td>1.728 Mbits</td>
<td>448 Kbits</td>
<td>96</td>
<td>12</td>
</tr>
<tr>
<td>XC2V2000</td>
<td>2M</td>
<td>24,192</td>
<td>1.008 Mbits</td>
<td>336 Kbits</td>
<td>56</td>
<td>8</td>
</tr>
<tr>
<td>XC2V1500</td>
<td>1.5M</td>
<td>17,280</td>
<td>864 Kbits</td>
<td>240 Kbits</td>
<td>48</td>
<td>8</td>
</tr>
<tr>
<td>XC2V1000</td>
<td>1M</td>
<td>11,520</td>
<td>720 Kbits</td>
<td>160 Kbits</td>
<td>40</td>
<td>8</td>
</tr>
<tr>
<td>XC2V500</td>
<td>500K</td>
<td>6,912</td>
<td>576 Kbits</td>
<td>96 Kbits</td>
<td>32</td>
<td>8</td>
</tr>
<tr>
<td>XC2V250</td>
<td>250K</td>
<td>3,456</td>
<td>432 Kbits</td>
<td>48 Kbits</td>
<td>24</td>
<td>8</td>
</tr>
<tr>
<td>XC2V80</td>
<td>80K</td>
<td>1,152</td>
<td>144 Kbits</td>
<td>16 Kbits</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>XC2V40</td>
<td>40K</td>
<td>596</td>
<td>72 Kbits</td>
<td>8 Kbits</td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>
The Virtex-II Family offers the following resources:

- **Superior on-chip RAM** - Up to 3Mbits of block SelectRAM, plus up to 1Mb of distributed SelectRAM.

- **Fast efficient hardware multipliers** - Virtex-II devices offer the industries first hard-embedded multiplier built directly into the FPGA fabric.

- **Industry leading Configurable Logic Blocks** – Each logic block contains four logic slices. Each slice contains two registers and two 4-input functions generators that can be used as 16 bits of distributed RAM, a 4 –input LUT, or a 16-bit variable tap shift register.

- **Flexible Digital Clock Management** – Virtex-II FPGAs can have up to 12 DCM blocks which provide self-calibrating, fully digital solutions for clock distribution, delay compensation, clock multiplication and division, and clock phase shifting.

- **Advanced Select-I/O Ultra** – Xilinx Virtex-II devices offer support for the widest area of industry I/O standards in an FPGA, including support for the most advanced single-ended and differential I/O standards available.

The CLB logic in Virtex-II FPGA devices allows users to implement combinatorial or synchronous logic designs using the same block. The fast routing within a CLB allows users to build incredibly fast and efficient arithmetic logic out of the available slices. The 4-input LUTs can be programmed as 16-bits of distributed SelectRAM (perfect for storing coefficients for DSP algorithms), 16-bit shift registers, or normal 4-input logic. This functionality is only available in Xilinx FPGAs. The comparison between Xilinx Virtex II with Altera’s APEX 10KE is shown in Table 2.5.2.
Table 2.5.2 Comparison of High-density FPGA families

<table>
<thead>
<tr>
<th></th>
<th><strong>Altera</strong></th>
<th><strong>Xilinx</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Device families</strong></td>
<td>APEX 10KE</td>
<td>Virtex II</td>
</tr>
<tr>
<td><strong>Architecture</strong></td>
<td>Uses both CPLD and gate array techniques</td>
<td>FPGA</td>
</tr>
<tr>
<td><strong>Process Technology</strong></td>
<td>0.22 micron</td>
<td>0.15/0.12 micron</td>
</tr>
<tr>
<td><strong>Usable typical gates</strong></td>
<td>5.25 Million</td>
<td>8 Million</td>
</tr>
<tr>
<td><strong>Salient feature</strong></td>
<td>CAM</td>
<td>Dedicated Multiplier</td>
</tr>
<tr>
<td><strong>Memory Bits</strong></td>
<td>1.15 Mb</td>
<td>3 Mb of select RAM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.5 Mb of CLB</td>
</tr>
<tr>
<td><strong>I/O Pins</strong></td>
<td>1060</td>
<td>1108</td>
</tr>
<tr>
<td><strong>Software Support</strong></td>
<td>Quartus II</td>
<td>Xilinx Integrated Synthesis Environment (ISE™)</td>
</tr>
</tbody>
</table>
CHAPTER 3
DISCRETE FOURIER TRANSFORM

3.1 Introduction

The Fourier Transform provides the means of transforming a signal defined in the time domain into one defined in the frequency domain. When a function is evaluated by numerical procedures, it is always necessary to sample it in some fashion. This means that in order to fully evaluate a Fourier transform with digital operations, it is necessary that the time and frequency functions be sampled in some form or another. Thus the digital or Discrete Fourier Transform (DFT) is of primary interest.

3.2 Fourier Transform

The Fourier transform is used to transform a continuous time signal into the frequency domain. It describes the continuous spectrum of a nonperiodic time signal. The Fourier transform $X(f)$ of a continuous time function $x(t)$ can be expressed as

$$X(f) = \int_{-\infty}^{\infty} x(t) e^{-j2\pi ft} \, dt$$

The inverse transform is

$$x(t) = \int_{-\infty}^{\infty} X(f) e^{j2\pi ft} \, df$$

3.3 The Discrete Fourier Transform

This is used in the case where both the time and the frequency variables are discrete (which they are if digital computers are being used to perform the analysis). Let $x(n)$ represent the discrete time signal, and let $X(k)$ represent the discrete frequency transform function. The Discrete Fourier Transform (DFT) is given by

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-j2\pi kn/N} \quad \text{where} \quad n = 0,1,2,\ldots,N-1$$
It’s inverse is defined as

\[
x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k)e^{2\pi i kn/N}
\]

where \(k=0,1,2,3\ldots N-1\)

### 3.4 Properties of Discrete Fourier Transform

1. **Periodicity:** If \(x(n)\) and \(x(k)\) are an \(N\)-point DFT pair then

\[x(n+N) = x(n) \text{ for all } n\]

\[X(k+N) = X(k) \text{ for all } k\]

2. **Superposition:** If the DFT of \(x_1(n)\) and \(x_2(n)\) are \(X_1(k)\) and \(X_2(k)\) respectively, then for any real valued or complex valued constant \(a_1\) and \(a_2\),

\[a_1x_1(n) + a_2x_2(n) \xrightarrow{\text{DFT}} a_1X_1(k) + a_2X_2(k)\]

3. **Multiplication:** If the DFT of \(x_1(n)\) and \(x_2(n)\) are \(X_1(k)\) and \(X_2(k)\) respectively, then

\[x_1(n)x_2(n) \xrightarrow{\text{DFT}} (X_1(k) \otimes X_2(k))/N\]

4. **Convolution:** If the DFT of \(x_1(n)\) and \(x_2(n)\) are \(X_1(k)\) and \(X_2(k)\) respectively, then

\[(x_1(n) \otimes x_2(n)) \xrightarrow{\text{DFT}} X_1(k)X_2(k)\]

### 3.5 The Fast Fourier Transform

The **Fast Fourier Transform** (FFT) is simply a class of special algorithms, which implement the discrete Fourier transform with considerable savings in computational time. It must be pointed out that the FFT is not a different transform from the DFT, but rather just a means of computing the DFT with a considerable reduction in the number of calculations required. While it is possible to develop FFT algorithms that work with any
number of points, maximum efficiency of computation is obtained by constraining the number of time points to be an integer power of two, e.g. 1024 or 2048.

### 3.6 The Bluestein Chirp-z Transform

The DFT of an \( N \)-point data sequence can be viewed as the \( z \)-transform of \( x(n) \) evaluated at \( N \) equally spaced points on a unit circle.

Suppose we wish to compute the values of the \( z \)-transform of \( x(n) \) at a set of points \( \{z_k\} \). Then,

\[
X(z_k) = \sum_{n=0}^{N-1} x(n) z_k^{-n} \quad k=0,1,2\ldots N-1
\]

(1)

If the contour is a circle of radius \( r \) and the \( z_k \) are \( N \) equally spaced points, then

\[
Z_k = r e^{j2\pi n/N} \quad k=0,1,2\ldots N-1
\]

\[
X(z_k) = \sum_{n=0}^{N-1} [x(n) r^{-n}] e^{-j2\pi n/N} \quad k=0,1,2\ldots N-1
\]

(2)

More generally, suppose that the points \( z_k \) in the \( z \)-plane fall on an arc which begins at some point

\[
Z_0 = r_0 e^{j\theta_0}
\]

and spirals either in toward the origin or out away from the origin such that the points \( \{z_k\} \) are defined as

\[
Z_k = r_0 e^{j\Phi_0} (R_0 e^{j\theta_0})^k \quad k=0,1,2,3\ldots L-1
\]

(3)

Therefore, if \( R_0 < 1 \), the points fall on a contour that spirals towards the origin and if \( R_0 > 1 \), the contour spirals away from the origin. If \( R_0 = 1 \), the contour is a circular arc of radius \( r_0 \). If \( r_0 = 1 \) and \( R_0 = 1 \), the contour is an arc of the unit circle. The latter contour would allow us to compute the frequency content of the sequence \( x(n) \) at a dense set of \( L \) frequencies in the range covered by the arc without having to compute a large DFT, that is, a DFT of the sequence \( x(n) \) padded with many zeros to obtain the desired resolution in frequency. Finally, if \( r_0 = R_0 = 1 \), \( \theta_0 = 0 \), \( \Phi_0 = 2\pi/N \), and \( L=N \), the contour is the entire unit circle and the frequencies are those of DFT.
When points \{z_k\} from eq. (3) are substituted into the expression for z-transform, we obtain

\[ X(z_k) = \sum_{n=0}^{N-1} x(n)z_k^{-n} \]

\[ = \sum_{n=0}^{N-1} x(n)(r_0 e^{j\theta_0})^{-n} V^{-nk} \text{ where, } V = R_0 e^{j\theta_0} \tag{4} \]

We can express eq. (4) in the form of a convolution, by noting that

\[ nk = \frac{1}{2} \left[ n^2 + k^2 - (k-n)^2 \right] \tag{5} \]

Substitution of (5) into (4) gives

\[ X(z_k) = V^{-k^2/2} \sum_{n=0}^{N-1} \left[ x(n)(r_0 e^{j\theta_0})^{-n} V^{-n^2/2} \right] V^{(k-n)^2/2} \tag{6} \]

Let us define a new sequence \( g(n) \) as

\[ g(n) = x(n)(r_0 e^{j\theta_0})^{-n} V^{-n^2/2} \tag{7} \]

Then eq. (6) can be expressed

\[ X(z_k) = V^{-k^2/2} \sum_{n=0}^{N-1} g(n)V^{(k-n)^2/2} \tag{8} \]

The summation can interpreted as the convolution of the sequence \( g(n) \) with the impulse response \( h(n) \) of a filter, where

\[ h(n) = V^{-n^2/2} \]

Hence, eq. (8) can be expressed as

\[ X(z_k) = V^{-k^2/2} y(k) \]

\[ = \frac{y(k)}{h(k)} \quad k = 0,1 \ldots N-1 \tag{9} \]

where \( y(k) \) is the output of the filter

\[ y(k) = \sum_{n=0}^{N-1} g(n)h(k-n) \quad k=0,1 \ldots N-1 \tag{10} \]

Both \( g(n) \) and \( h(n) \) are complex valued sequences.
The sequence \( h(n) \) with \( R_0 = I \) has the form of a complex exponential with argument \( \phi_0 = n^2 \phi_0/2 = (n\phi/2)n \). The quantity \( (n\phi/2) \) represents the frequency of the complex exponential signal, which increases linearly with time. Such signals are used in radar systems and are called *chirp signals*. Hence the z-transform is called *chirp-z transform*.

Consider, the circular convolution of the \( N \)-point sequence \( g(n) \) with an \( M \)-point section \( h(n) \), where \( M > N \). In such a case, we know that the first \( N-1 \) points contain aliasing and that the remaining \( M-N+1 \) points are identical to the result obtained from linear convolution of \( h(n) \) and \( g(n) \). Hence, for

\[
M = L+N-1
\]

It would yield \( N-1 \) corrupt points and \( L \) valid points.

The chirp-z transform is implemented in hardware to compute the DFT of signals. For the computation of the DFT, we select \( \theta_0=0, \phi_0=2\pi/N \). In this case,

\[
V^{-n^2/2} = e^{-j\pi^2/N}
\]

\[
= \cos \frac{\pi n^2}{N} - j \sin \frac{\pi n^2}{N}
\]

The chirp filter with impulse response

\[
h(n) = V^{n^2/2}
\]

\[
= \cos \frac{\pi n^2}{N} + j \sin \frac{\pi n^2}{N}
\]

\[
= h_r(n) + j h_i(n) \quad \text{for } 0 < n < 2N-1
\]

has been implemented as a pair of FIR filters (of length \( 2N-1 \)) with coefficient \( h_r(n) \) and \( h_i(n) \), respectively. The cosine and sine sequences are stored in a read-only memory (ROM).

The algorithm can be summarized as:

1. \( N \) multiplication of \( x[n] \) with \( W_n^{n^2/2} \).
2. Linear convolution \( x[n]W_N^{n^2/2} \ast W_N^{n^2/2} \)

3. \( N \) multiplications with \( W_N^{n^2/2} \)

Where \( W_N = e^{j2\pi/N} \)

For a complete transform, we therefore need a length \( N \) convolution and \( 2N \) complex multiplications. The advantage, compared with the Rader algorithm is that there is no restriction to primes in the transform length \( N \). CZT can be defined for every length.

\[
\begin{align*}
  &\text{Premultiplication with chirp signal} \\
  \exp(-j\pi n^2/2N) \\
  \text{Linear convolution} \\
  \exp(-j\pi k^2/2N) \\
  &\text{Postmultiplication with chirp signal}
\end{align*}
\]

Figure 3.6.1 The Bluestein chirp-z DFT

Hence, for a complete transform, we need a length \( N \) convolution and \( 2N \) multiplications. The major advantage compared with the Rader algorithm, is that there is no restriction to primes in the transform length \( N \) and can be defined for every length.

An interesting aspect noticed (by Narasimha) in CZT algorithm is that many coefficients of the FIR filter were either trivial or identical. This can be attributed to the periodicity of Sine and Cosine functions. For example 8-point DFT has a filter length of 16 but only has four coefficients to be computed. They are 1, j, and \( \pm e^{22.5^\circ} \), i.e. only two non-trivial coefficients. The number of coefficients (\( C_N \)) for other lengths is described in Table 3.6.1.
Table 3.6.1 Table of DFT length and complex $C_N$ filter coefficients

<table>
<thead>
<tr>
<th>DFT Length</th>
<th>$C_N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>16</td>
<td>7</td>
</tr>
<tr>
<td>24</td>
<td>8</td>
</tr>
<tr>
<td>40</td>
<td>12</td>
</tr>
<tr>
<td>48</td>
<td>14</td>
</tr>
<tr>
<td>72</td>
<td>16</td>
</tr>
<tr>
<td>80</td>
<td>21</td>
</tr>
<tr>
<td>120</td>
<td>24</td>
</tr>
<tr>
<td>144</td>
<td>28</td>
</tr>
<tr>
<td>168</td>
<td>32</td>
</tr>
<tr>
<td>180</td>
<td>36</td>
</tr>
<tr>
<td>240</td>
<td>42</td>
</tr>
<tr>
<td>360</td>
<td>48</td>
</tr>
<tr>
<td>504</td>
<td>64</td>
</tr>
</tbody>
</table>

The above table however is not an indicator of the implementation effort, because some coefficients may be trivial (i.e. $\pm 1$ or $\pm j$) or may be symmetrical. Table 3.6.2 shows the number of non-trivial coefficients for various lengths of DFT.
Table 3.6.2 DFT length and non-trivial coefficients

<table>
<thead>
<tr>
<th>DFT length</th>
<th>Non-trivial coefficients</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>32</td>
<td>6</td>
</tr>
<tr>
<td>40</td>
<td>8</td>
</tr>
<tr>
<td>50</td>
<td>10</td>
</tr>
<tr>
<td>80</td>
<td>11</td>
</tr>
<tr>
<td>96</td>
<td>14</td>
</tr>
<tr>
<td>160</td>
<td>20</td>
</tr>
<tr>
<td>192</td>
<td>25</td>
</tr>
</tbody>
</table>

Hence, length 16 and 32 are maximum length DFTs with only 3 and 6 multipliers respectively.
CHAPTER 4

IMPLEMENTATION OF CHIRP-Z TRANSFORM

Implementation of the Chirp-z algorithm consists of three modules:

I. A ROM to store the coefficients
II. Complex number multiplier
III. FIR filter to perform convolution

4.1 Implementation to store ROM

The ROM with twiddle factors can be easily obtained using a case statement table. But we need to keep in mind that the numbers are fractions. The most significant bit represents the sign (‘0’ for positive and ‘1’ for negative). For example:

“00110000” = 0 + 0 + (1/2)^2 + (1/2)^3 + 0 + 0 + 0 + 0 = 0.375

“11110000” = (-1) + (1/2)^1 + (1/2)^2 + (1/2)^3 = -0.125

4.2 Complex Number Multiplier

The multiplication operation performs the operation:

\[ R + jI = (A + jB) \times (C + jS) \]

Where,

A, B: Real and imaginary parts of input.

C, S: Real and imaginary parts of coefficient.

\[ R = (AC - BS) \quad I = (AS + BC) \]

This requires 4 multipliers. However, the same result can be computed as
\[ R = (C-S) \cdot Y + Z \quad \quad I = (C+S) \cdot X - Z \]

Where \( Z = C \cdot (X-Y) \). Hence, we can reduce the number of multipliers from 4 to 3. The inputs are padded for a possible growth for sign extension. Similarly final results are scaled to eight bits.

### 4.3 FIR Filter

An FIR with constant coefficient is a linear time invariant (LTI) filter digital filter. The output of a FIR of the order or length \( L \), to an input time-series \( x[n] \), is given by a finite version of convolution sum given by:

\[
y = x[n] \ast f[n] = \sum_{k=0}^{L-1} f[k]x[n-k]
\]

where \( f[0] \neq 0 \) through \( f[L-1] \neq 0 \) are filter’s \( L \) coefficients.

In z-domain the above equation is expressed as

\[
Y(z) = F(z) \cdot X(z)
\]

where \( F(z) \) is the FIR’s transfer function defined in the z-domain by

\[
F(z) = \sum_{k=0}^{L-1} f[k]z^{-k}
\]

The \( L \)th order FIR filter consists of a group of tapped delay lines, adders and multipliers. One of the operands of the multipliers is called the “tap weight” or the filter coefficients. The roots of the polynomial \( F(z) \) define the zeros of the filter. Due to the presence of only zeros, the FIR filters are also referred to as all zero filters.
A variation of the direct FIR filter structure is the transposed structure. A direct form FIR filter can be converted into transposed form by simply:

- Exchanging the input and the output.
- Inverting the direction of signal flow
- Substituting an adder by a fork, and vice versa.

The advantage of the transposed structure is that we don’t need an extra shift register and there is no need for an extra pipeline stage for the adder (tree) of the products to achieve high throughput.
CHAPTER 5

COMPUTER ARITHMATIC ALGORITHMS FOR FILTER DESIGN

In this chapter we shall look into the arithmetic algorithms used compute the coefficient of the FIR filter. The main criterion for the choice of algorithm is the cost factor i.e. number of adders. Let’s look three algorithm/ number representation systems and compare their performances.

5.1 Canonical Signed Digit System (CSD)

The signed digit (SD) system unlike the regular binary number system has is represented in three values 0, 1, -1 (-1 is represented as 1). They are successful in implementing carry free adders or multipliers with less complexity. Since the complexity of the multiplier is typically estimated through the number of nonzero elements, which can be reduced by using SD numbers. The SD representation, unlike a 2C code, is non-unique. We call a canonic digit system or CSD, the system that has least number of non-zero elements. The algorithm of the classical CSD is as follows:

**Algorithm for Classical CSD coding**

Starting from the LSB substitute all 1 sequences equal or larger than two, with 10…. 01.

However, the use of above algorithm does not always result in optimal CSD coding with reference to hardware complexity. In some of the cases additions are also substituted by subtractions, when such substitutions are not required. For e.g.: 110₂ is coded as 10ī. When this is used in a constant multiplication, the subtraction will need a full adder instead of half adder for LSB. Hence, the following CSD coding will produce a CSD with a minimum of none-zero terms.
The Table 5.1.1 below shows the cost factor (number of adders) for all coefficients from 0 to 256 (all 8 bits data). For e.g.:

\[ 45_{10} = 00101101_2 = 00110\overline{1}01 \text{ (in CSD)} = 32 + 16 + (-4) + 1 \]

Cost of 45\textsubscript{10} is 4; the same result can be inferred from the Table 5.1.1.

Table 5.1.1 Cost incurred using CSD algorithm for 8 bit numbers

<table>
<thead>
<tr>
<th>Cost</th>
<th>Coefficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4, 8, 16, 32, 64, 128, 256</td>
</tr>
<tr>
<td>1</td>
<td>3, 5, 6, 7, 9, 10, 12, 14, 15, 17, 18, 20, 24, 28, 30, 31, 33, 34, 36, 40, 48, 56, 60, 62, 63, 65, 66, 68, 72, 80, 96, 102, 120, 124, 126, 127, 129, 130, 132, 136, 144, 160, 192, 224, 240, 248, 252, 254, 255</td>
</tr>
<tr>
<td>4</td>
<td>171, 173, 179, 181, 203, 205, 211, 213</td>
</tr>
</tbody>
</table>
5.2 Multiplier Adder Graph (MAG)

We know that the number of non-zero elements determines the cost of multiplication. Sometimes a more efficient way of reducing the cost is by factorizing the number and realize CSD code for each factors. For e.g.:

\[45_{10} = 00101101_2 = 00110\bar{1}01_{1} \text{ (in CSD)} = 32+16+(-4)+1\]

It is evident from above that 45\( _{10} \) requires three adders in CSD coding. However, using MAG 45\( _{10} \) can be factorized as

\[45_{10} = 9 * 5 = (8+1)*(4+1)\]

In all 31 coefficients (for 8 bit data) can be improved using MAG algorithm. They are listed in Table 6.2.1. with their factors.

Table 5.2.1 Coefficients improved due to MAG

<table>
<thead>
<tr>
<th>Cost</th>
<th>Coefficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>45 = 5 x 9, 51 = 3 x 17, 75 = 5 x 15, 85 = 5 x 17, 90 = 2 x 9 x 5, 93 = 3 x 31, 99 = 3 x 33, 102 = 2 x 3 x 17, 105 = 7 x 15, 150 = 2 x 5 x 15, 153 = 9 x 17, 155 = 5 x 31, 165 = 5 x 33, 170 = 2 x 5 x 17, 180 = 4 x 5 x 9, 186 = 2 x 3 x 31, 189 = 7 x 9, 195 = 3 x 65, 198 = 2 x 3 x 33, 204 = 4 x 3 x 17, 210 = 2 x 7 x 15, 217 = 7 x 31, 231 = 7 x 33</td>
</tr>
<tr>
<td>3</td>
<td>171 = 3 x 57, 173 = 8 + 165, 179 = 51 + 128, 181 = 1 + 180, 211 = 1 + 210, 213 = 3 x 71, 205 = 5 x 41, 203 = 7 x 29</td>
</tr>
</tbody>
</table>

Figure 5.2.1 shows a comparison in implementation of CSD and MAG algorithm for 93. While in CSD, the cost factor is 3; using MAG algorithm the cost is reduced to 2. Figure 5.2.2 shows cost 1 to 4 graphs of all coefficients from 0 to 256 (all 8 bit data) where each path represents gain of the order \(2^k\) and each nodal point is an adder. For example 93 in CSD is represented by graph 1 from cost 3 graph (or simply can be put as graph 3.1) and it’s RAG equivalent is represented by graph 2 from cost 2 (graph 2.2) column.
Figure 5.2.1 Comparison of CSD coding and MAG coding

Figure 5.2.2 Possible Cost one to four graphs for CSD and RAG
5.3 Reduced Adder Graph (RAG)

One of the advantages of the transposed filter as we have seen in earlier chapters is repetitions of coefficients due to the periodicity of sine and cosine function and symmetry due presence of complex conjugates. The reduced adder graph (RAG) exploits this property. In this method the coefficients of filters are calculated by using previously calculated coefficient values. For instance if the 9 and 11 are coefficients of the filter, by using RAG algorithm the total cost is 3 (Cost 1 for 9 + cost 2 for 11). But in RAG algorithm the total cost is cut down to 2 i.e. cost 1 for 9 and 11= 9+2. Thus the RAG algorithm can be summarized as

Reduced Adder Graph Algorithm
- Remove the sign of the coefficient because the sign can be realized by a subtraction in the filter’s tapped delay.
- Remove all the coefficients and factors, which are a power of two, since they can be implemented by hard-wired data shift.
- Realize all cost “1” coefficients
- Use cost “1” coefficients in building the multiplier of higher cost.

The first three steps in the above algorithm are straightforward but the last step tends to get complex for higher values. Table 5.3.1 shows RAG algorithm implementation for coefficients 9, -44,208 and 346 of filter F6.

Table 5.3.1 Example for implementation of RAG.

<table>
<thead>
<tr>
<th>Step</th>
<th>To be realized</th>
<th>Already realized</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0)</td>
<td>346,208,-44,9</td>
<td>-</td>
<td>Initialization</td>
</tr>
<tr>
<td>1)</td>
<td>346,208,44,9</td>
<td>-</td>
<td>No negative coefficients</td>
</tr>
<tr>
<td>2)</td>
<td>346,208,44,9</td>
<td>-</td>
<td>Remove $2^k$ coefficients</td>
</tr>
<tr>
<td>3)</td>
<td>173,13,11,9</td>
<td>-</td>
<td>Remove $2^k$ factors</td>
</tr>
<tr>
<td>4)</td>
<td>173,13,11</td>
<td>9</td>
<td>Realize cost 1 coefficients</td>
</tr>
<tr>
<td>5)</td>
<td>173,13,11</td>
<td>9</td>
<td>Other coefficients are prime</td>
</tr>
<tr>
<td>6)</td>
<td>173,13</td>
<td>9,11</td>
<td>For 11=9+2</td>
</tr>
<tr>
<td>7)</td>
<td>173</td>
<td>9,11,13</td>
<td>For 13=9+4</td>
</tr>
<tr>
<td>8)</td>
<td>-</td>
<td>9,11,13,173</td>
<td>For 173=(4+1)9+128</td>
</tr>
</tbody>
</table>
The realization of the multiplier block is shown in Figure 5.3.1. The multiplier block requires 5 adders to implement all the coefficients. The final filter with multiplier is shown in Figure 5.3.2.

![Figure 5.3.1 RAG realization of filter F6.](image)

![Figure 5.3.2 FIR Filter with multiplier block](image)
CHAPTER 6
CONCLUSION

The chirp-z algorithm for DFT (Discrete Fourier Transform) was successfully implemented on a Virtex II device. The results for 4 point, 16 point DFT, 32 Point DFT and 64 Point DFT are shown in table 6.1. The code is optimized for speed and area. All the designs were run on 2v1000fg256-6 devices from Virtex II family. The results of synthesis for optimized speed and area are shown in Table 6.1 and Table 6.2 respectively.

Table 6.1 DFT Synthesis Results for speed optimization

<table>
<thead>
<tr>
<th>DFT Length</th>
<th>4 Point</th>
<th>16 Point</th>
<th>32 Point</th>
<th>64 Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equivalent Gate count</td>
<td>34,713</td>
<td>52,559</td>
<td>90,495</td>
<td>154,486</td>
</tr>
<tr>
<td>Number of 4 input LUT</td>
<td>705/10240(6%)</td>
<td>2581/10240(25%)</td>
<td>5068/10240(49%)</td>
<td>8809/10240(86%)</td>
</tr>
<tr>
<td>Number of bonded IOBs</td>
<td>101/172(58%)</td>
<td>121/172(70%)</td>
<td>127/172(73%)</td>
<td>122/328(71%)</td>
</tr>
<tr>
<td>Number of MULT18x18s</td>
<td>6/40(15%)</td>
<td>3/40(7%)</td>
<td>3/40(7%)</td>
<td>3/40(7%)</td>
</tr>
<tr>
<td>Timing</td>
<td>24.728 ns</td>
<td>24.100 ns</td>
<td>27.977 ns</td>
<td>20.279 ns</td>
</tr>
<tr>
<td>Peak memory Usage</td>
<td>80 MB</td>
<td>100 MB</td>
<td>125 MB</td>
<td>163 MB</td>
</tr>
</tbody>
</table>
Table 6.2 DFT Synthesis Results for Area optimization

<table>
<thead>
<tr>
<th>DFT Length</th>
<th>4 Point</th>
<th>16 Point</th>
<th>32 Point</th>
<th>64 Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equivalent Gate count</td>
<td>34,973</td>
<td>52,509</td>
<td>90,502</td>
<td>154,435</td>
</tr>
<tr>
<td>Number of 4 input LUT</td>
<td>738/10240(6%)</td>
<td>2588/10240(25%)</td>
<td>50778/10240(49%)</td>
<td>8812/10240(86%)</td>
</tr>
<tr>
<td>Number of bonded IOBs</td>
<td>101/172(58%)</td>
<td>121/172(70%)</td>
<td>127/172(73%)</td>
<td>123/328(71%)</td>
</tr>
<tr>
<td>Number of MULT18x18s</td>
<td>6/40(15%)</td>
<td>3/40(7%)</td>
<td>3/40(7%)</td>
<td>3/40(7%)</td>
</tr>
<tr>
<td>Timing</td>
<td>25.738 ns</td>
<td>24.314 ns</td>
<td>25.637 ns</td>
<td>22.734 ns</td>
</tr>
<tr>
<td>Peak memory Usage</td>
<td>79 MB</td>
<td>98 MB</td>
<td>123 MB</td>
<td>162 MB</td>
</tr>
</tbody>
</table>

From the above results it is evident that there is no significant change in timing or gate count between area optimized synthesis and the speed optimized synthesis. Hence, we can say that the algorithm strikes a balance between speed and area. The following results are for input data width 8 bits and output data width 8 bits. The final results of simulation have some quantization error, which is negligible. However, it must be kept in mind that the bit width is decided upon the permissible signal noise ratio (SNR) as

\[
\text{SNR} = 10 \log_{10}(2^{2b-v-1})
\]

Where,

- \(b\): bit width
- \(v\): \(\log_2(\text{Length of DFT})\)
Table 6.3 SNR for various DFT lengths.

<table>
<thead>
<tr>
<th>DFT Length</th>
<th>4</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>SNR</td>
<td>39.13 dB</td>
<td>33.11 dB</td>
<td>30.10 dB</td>
<td>27.09 dB</td>
</tr>
</tbody>
</table>

Table 6.3 shows the SNR for various lengths of DFT that have been synthesized. It is evident from the table and from the Figure 6.1 that for a constant bit length, as the length of DFT increases there is increase in quantization error and hence the signal to noise ratio decreases indicating the increasing presence of noise.

As future work, the synthesized DFT blocks can be used to perform Fast Fourier Transforms or FFT. For e.g. to implement FFT of length 64, we can use DFT blocks of length 16 and length 4 (64=4x16). However, we need an additional module for indexing the input and output.
APPENDIX A

VHDL CODES FOR 4 POINT DFT

I.

--Project Name: thesis.npl
--File Name: czt8h.vhd
--Description: Implementation of 4 point DFT
--Author: Dr. Uwe Meyer-Baese and Hariharan Natarajan
--Date: 03/30/04

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

ENTITY czt4b8h IS                      ------> Interface
  GENERIC (Wfir: INTEGER := 19;    -- Filter accu bit width
            W2  : INTEGER := 16;    -- Multiplier bit width
            W1  : INTEGER := 2;    -- log2(DFT length)
            W   : INTEGER := 8);    -- Data/Coeff bit width
  PORT ( clk            : IN  STD_LOGIC;
         x_real, x_imag : IN  STD_LOGIC_VECTOR(8-1 DOWNTO 0);
         t1_real_out,t1_imag_out, t1_xr_out,t1_xi_out : OUT STD_LOGIC_VECTOR(8-1 DOWNTO 0);
         r0_out, i0_out, t2_real_out, t2_imag_out : OUT STD_LOGIC_VECTOR(8-1 DOWNTO 0);
        );
ARCHITECTURE flex OF czt4b8h IS

subtype WORD8 is integer range -2**7 TO 2**7 -1;
subtype WORD11 is integer range -(2**10) TO 2**10-1;
subtype WORD19 is integer range -(2**19) TO 2**19-1;

type ARRAY_WORD is array (0 to 6) of WORD19;

COMPONENT diag4b8
  PORT ( table_in : IN STD_LOGIC_VECTOR(2-1 DOWNTO 0);
         re_out    : OUT STD_LOGIC_VECTOR(8-1 DOWNTO 0);
         im_out    : OUT STD_LOGIC_VECTOR(8-1 DOWNTO 0));
END COMPONENT;

COMPONENT mult
  GENERIC (W2  : INTEGER := 17;    -- Multiplier bit width
            W1  : INTEGER := 9;     -- Bit width c+s sum
            W   : INTEGER := 8);    -- Input bit width
  PORT (clk   : STD_LOGIC;  -- Clock for the output register
         x_in, y_in    : IN  STD_LOGIC_VECTOR(W-1 DOWNTO 0);
         c_in, s_in    : IN  STD_LOGIC_VECTOR(W-1 DOWNTO 0);
         r_out, i_out  : OUT STD_LOGIC_VECTOR(W-1 DOWNTO 0));
END COMPONENT;

SIGNAL  count    : integer RANGE 0 TO 15;
TYPE    STATE_TYPE IS (Start, Load, Run);
SIGNAL  state    : STATE_TYPE ;
BEGIN

States: PROCESS -------> State machine
BEGIN
WAIT UNTIL clk = '1';
CASE state IS
WHEN Start => -- Initialization step
state <= Load;
count <= 0;
t1_in <= "00"; -- Reset tables
t2_in <= "00";
WHEN Load => -- Multiply inputs with twiddle factors
IF count = 4 THEN -- Load phase done ?
state <= Run;
ELSE
state <= Load;
t1_in <= t1_in + "01";
END CASE;
END;
END;
END IF;
count <= count + 1;
count <= count;
WHEN Run => -- Multiply filter output with twiddle factors
IF count = 10 THEN -- Run phase done?
state <= Start; -- Output of result
ELSE -- and start again
state <= Run;
t2_in <= t2_in + "01";
END IF;

IF count > 5 AND count < 10 THEN
y_real <= y_r;
y_imag <= y_i;
ELSE
y_real <= (OTHERS => '0');
y_imag <= (OTHERS => '0');
END IF;
count <= count + 1;
count <= count;
END CASE;
END PROCESS States;

Structure: PROCESS
-- Structure of the two FIR
BEGIN -- filters in transposed form
wait until clk='1';
-- Real part of FIR filter in transposed form
realn(6) <= xr90 - xi90; -- beta
realn(5) <= realn(6) - xr127; -- -1
realn(4) <= realn(5) + xr90-xi90; -- beta
realn(3) <= realn(4) + xr127 ; -- 1
realn(2) <= realn(3) + xr90-xi90; -- beta
realn(1) <= realn(2) - xr127; -- -1
realn(0) <= realn(1) + xr90-xi90; -- beta
-- Imaginary part of FIR filter in transposed form
imag(6) <= xr90 + xi90; -- beta
imag(5) <= imag(6) - xi127; -- -1
imag(4) <= imag(5) + xr90 +xi90; -- beta
imag(3) <= imag(4) + xi127; -- 1
imag(2) <= imag(3) + xr90+xi90; -- beta
imag(1) <= imag(2) - xi127; -- -1
imag(0) <= imag(1) + xr90+xi90; -- beta
END PROCESS Structure;

Coeffs: PROCESS(sxtxi,sxtxr) -- Note that all signals
    VARIABLE xr, xi : Integer;

    BEGIN -- are global defined
    -- Compute the filter coefficients and use FFs
    -- Odd coefficients
    xr := CONV_INTEGER(sxtxr);
    xi := CONV_INTEGER(sxtxi);
    xr127 <= xr * 128 - xr; -- Coeff = 1
    xi127 <= xi * 128 - xi;
    -- Even coefficients

    xr90  <= xr * 64 + xr * 16 + xr * 8 + 2 * xr; -- sqrt(2)/2
    xi90  <= xi * 64 + xi * 16 + xi * 8 + 2 * xi;

    END PROCESS Coeffs;
    -- Use table to get INPUT twiddle factor coefficients
    tab_1: diag4b8
        PORT MAP( table_in => t1_in,
re_out => t1_real,
im_out => t1_imag);

-- Multiply inputs with twiddle factor exp(j*phi)
ccmul_1: mult
PORT MAP (clk => clk, -- Clock for the output register
          x_in => x_real, y_in => x_imag,
          c_in => t1_real, s_in => t1_imag,
          r_out => t1_xr, i_out => t1_xi);

Sxt: PROCESS(t1_xr,t1_xi)
BEGIN
  --wait until clk='1';
sxtxr(W-1 DOWNTO 0) <= t1_xr;
sxtxi(W-1 DOWNTO 0) <= t1_xi;
FOR k IN Wfir-1 DOWNTO W LOOP
  sxtxr(k) <= t1_xr(t1_xr'left);
sxtxi(k) <= t1_xi(t1_xi'left);
END LOOP;
END PROCESS Sxt;

-- Use table to get INPUT twiddle factor coefficients

tab_2: diag4b8
PORT MAP( table_in => t2_in,
          re_out => t2_real,
          im_out => t2_imag);

38
-- Multiply inputs with twiddle factor \( \exp(j\phi) \)

\[
\begin{align*}
\text{r0} & \leftarrow \text{conv_std_logic_vector}(\text{realn}(0)/128,8); \\
\text{i0} & \leftarrow \text{conv_std_logic_vector}(\text{imag}(0)/128,8);  \\
r0\_out & \leftarrow \text{r0};  \\
i0\_out & \leftarrow \text{i0};
\end{align*}
\]

ccmul_2: mult

PORT MAP (clk => clk, -- Clock for the output register
\[
\begin{align*}
\text{x\_in} & \rightarrow \text{r0}, \text{y\_in} \rightarrow \text{i0}, \\
\text{c\_in} & \rightarrow \text{t2\_real}, \text{s\_in} \rightarrow \text{t2\_imag}, \\
\text{r\_out} & \rightarrow \text{y\_r}, \text{i\_out} \rightarrow \text{y\_i});
\end{align*}
\]}

END flex;
II.

--Project Name: thesis.npl
--File Name: diag4b8.vhd
--Description: Twiddle factors for 4 point DFT
--Author: Dr. Uwe Meyer-Baese
--Date: 03/30/04

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;

ENTITY diag4b8 IS
  PORT ( table_in : IN STD_LOGIC_VECTOR(2-1 DOWNTO 0);
         re_out    : OUT STD_LOGIC_VECTOR(8-1 DOWNTO 0);
         im_out    : OUT STD_LOGIC_VECTOR(8-1 DOWNTO 0) );
END diag4b8;

ARCHITECTURE LCs OF diag4b8 IS
BEGIN

  -- This is the 8 bit (i.e. -127 ... 127) CASE table for
  -- the 4 point CZT pre and post multiplications
  -- automatic generated with cztgen.exe -- DO NOT EDIT!

  -- Table implements the mapping
  PROCESS (table_in)
  BEGIN
    CASE table_in IS
      WHEN "00" =>    re_out <=  "01111111";       -- = 127 i.e. 1.00
                     im_out <=  (OTHERS => '0');
      WHEN "01" =>    re_out <=  "01011010";       -- =  90 i.e. sqrt(2)/2
                     im_out <=  (OTHERS => '0');
      WHEN "10" =>    re_out <=  "01010001";       -- =  45 i.e. sqrt(2)/4
                     im_out <=  (OTHERS => '0');
      WHEN "11" =>    re_out <=  "01000000";       -- =  22.5 i.e. sqrt(2)/8
                     im_out <=  (OTHERS => '0');
      WHEN OTHERS => re_out <=  "00000000";      -- =  0 i.e. 0.00
                     im_out <=  (OTHERS => '0');
    END CASE;           
  END PROCESS;
END LCs;
im_out <= "10100110"; -- = -90
WHEN "10" => re_out <= "10000001"; -- = 256-128 i.e. -1.00
    im_out <= (OTHERS => '0');
WHEN "11" => re_out <= "01011010"; -- = 90
    im_out <= "10100110"; -- = -90
WHEN OTHERS => re_out <= (OTHERS => '0');
    im_out <= (OTHERS => '0');
END CASE;
END PROCESS;
END LCs;
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity mult is
  GENERIC (W2  : INTEGER := 17;    -- Multiplier bit width
            W1  : INTEGER := 9;     -- Bit width c+s sum
            W   : INTEGER := 8);    -- Input bit width
  Port ( clk : in std_logic;
         x_in : in std_logic_vector(W-1 downto 0);
         y_in : in std_logic_vector(W-1 downto 0);
         s_in : in std_logic_vector(W-1 downto 0);
         c_in : in std_logic_vector(W-1 downto 0);
         r_out,i_out:    out std_logic_vector(W-1 downto 0));
end mult;

architecture Behavioral of mult is
signal x,y,c,s:std_logic_vector (W-1 downto 0);
signal sxtx,sxty,xmy,cps,ems:std_logic_vector(W1-1 downto 0);
signal xmyc,cpsx,cmsy,r,i:std_logic_vector(W2-1 downto 0);

begin
    p1:process--(x_in,y_in,c_in,s_in)
    begin
        wait until clk='1';
        x<=x_in;
        y<=y_in;
        c<=c_in;
        s<=s_in;
        cps<=(c_in(c_in'left)&c_in) + (s_in(s_in'left)&s_in);
        cms<=-(c_in(c_in'left)&c_in)-(s_in(s_in'left)&s_in);
        xmy<=sxtx-sxty;
        xmyc<=xmy*c;
        cmsy<=cms*y;
        cpsx<=cps*x;
        r<=cmsy+xmyc;
        i<=cpsx-xmyc;
    end process p1;

    sxtx<=x(x'left)&x;
    sxty<=y(y'left)&y;
    xmy<=sxtx-sxty;
    xmyc<=xmy*c;
    cmsy<=cms*y;
    cpsx<=cps*x;
    r<=cmsy+xmyc;
    i<=cpsx-xmyc;
    p2:PROCESS(r,i)
    begin
        r_out<=r(W2-3 DOWNTO W-1);
        i_out<=i(W2-3 DOWNTO W-1);
    end process p2;
end Behavioral;
APPENDIX B

VHDL CODES FOR 16 POINT DFT

I.

--Project Name: fir16pt8b.npl
--File Name: fir16pt8b.vhd
--Description: Implementation of 16 point DFT
--Author: Dr. Uwe Meyer-Baese and Hariharan Natarajan
--Date: 03/30/04
--This is the 8 bit and
--16-point CZT FSM and FIR filter
--automatic generated with cztgen.exe -- DO NOT EDIT!
--The filter has 4 real and 4 imaginary different coefficients
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;

ENTITY fir16pt8b IS                      ------> Interface
  GENERIC (Wfir: INTEGER := 19;    -- Filter accu bit width
            Lfir: INTEGER := 31;    -- Filter length = 2*DFTlenght-1
            W1  : INTEGER := 4;    -- log2(DFT length)
            W   : INTEGER := 8);    -- Data/Coeff bit width
  PORT ( clk            : IN  STD_LOGIC;
        x_real_in, x_imag_in : IN  STD_LOGIC_VECTOR(W-1 DOWNTO 0);
        valid_in             : OUT  STD_LOGIC;
        t1_real_out,t1_imag_out, ccmul_r_out,
        ccmul_i_out          : OUT STD_LOGIC_VECTOR(W-1 DOWNTO 0);
        r0_out, i0_out       : OUT STD_LOGIC_VECTOR(W-1 DOWNTO 0);
ARCHITECTURE flex OF fir16pt8b IS

COMPONENT diag16b8
PORT (table_in  : IN  STD_LOGIC_VECTOR(4-1 DOWNTO 0);
      re_out    : OUT STD_LOGIC_VECTOR(8-1 DOWNTO 0);
      im_out    : OUT STD_LOGIC_VECTOR(8-1 DOWNTO 0));
END COMPONENT;

COMPONENT mult
GENERIC (W2  : INTEGER := 17;  -- Multiplier bit width
         W1  : INTEGER := 9;     -- Bit width c+s sum
         W   : INTEGER := 8);    -- Input bit width
PORT (clk  : STD_LOGIC;  -- Clock for the output register
      x_in, y_in    : IN  STD_LOGIC_VECTOR(W-1 DOWNTO 0);
      c_in, s_in    : IN  STD_LOGIC_VECTOR(W-1 DOWNTO 0);
      r_out, i_out  : OUT STD_LOGIC_VECTOR(W-1 DOWNTO 0));
END COMPONENT;

SIGNAL  count    : integer RANGE 0 TO 63;
TYPE    STATE_TYPE IS (Start, Load, Run);
SIGNAL  state    : STATE_TYPE ;
SIGNAL  x_real, x_imag : STD_LOGIC_VECTOR(W-1 DOWNTO 0);

SUBTYPE WORD IS INTEGER RANGE -2**20 TO 2**20-1;
TYPE ARRAY_WORD IS ARRAY (0 to 30) OF WORD;
SIGNAL  realn, imag : ARRAY_WORD:=(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0); -- Tapped delay line array
Real and imaginary filter coefficients:

```vhdl
SIGNAL  xr125, xi125  : WORD:=0;
SIGNAL  xr25, xi25  : WORD:=0;
SIGNAL  xr90, xi90  : WORD:=0;
SIGNAL  xr127, xi127  : WORD:=0;
```

-- Auxiliary signal definitions

```vhdl
SIGNAL  t1_in            : STD_LOGIC_VECTOR(W1-1 DOWNTO 0);
SIGNAL  real_in, imag_in : STD_LOGIC_VECTOR(W-1 DOWNTO 0);
SIGNAL  t1_real, t1_imag : STD_LOGIC_VECTOR(W-1 DOWNTO 0);
SIGNAL  ccmul_r, ccmul_i : STD_LOGIC_VECTOR(W-1 DOWNTO 0);
SIGNAL  sxtxr, sxtxi     : STD_LOGIC_VECTOR(Wfir-1 DOWNTO 0);
SIGNAL  r0, i0           : STD_LOGIC_VECTOR(W-1 DOWNTO 0);
SIGNAL  y_r, y_i         : STD_LOGIC_VECTOR(W-1 DOWNTO 0);
```

BEGIN

States: PROCESS  -----> State machine for RADER filter

BEGIN

WAIT UNTIL clk = '1';

CASE state IS

WHEN Start => -- Initialization step
state <= Load;
count <= 0;
t1_in <= "0000"; -- Reset tables
WHEN Load => -- Multiply inputs with twiddle factors
IF count = 17 THEN -- Load phase done ?
state <= Run;
ELSE
```
state <= Load;
t1_in <= t1_in + "0001";
END IF;

IF count < 16 THEN
t1_in <= t1_in + "0001";
ELSE
    t1_in <= "0000";  -- Reset tables
END IF;
count <= count + 1;

WHEN Run =>  -- Multiply filter output with twiddle factors
    IF count = 47 THEN  -- Run phase done ?
        state <= Start;   -- Output of result
        t1_in <= "0000";   -- Reset tables
    ELSE                -- and start again
        state <= Run;
        t1_in <= t1_in + "0001";
    END IF;
count <= count + 1;
END CASE;

IF count > 18 AND count < 35 THEN
    valid_out <= '1';
y_real <= ccmul_r;
y_imag <= ccmul_i;
ELSE
    valid_out <= '0';
y_real <= (OTHERS => '0');
y_imag <= (OTHERS => '0');
END IF;

IF count < 15 THEN
    valid_in <= '1';
ELSE
    valid_in <= '0';
END IF;
END PROCESS States;

filter: PROCESS  -- Structure of the two FIR
Variable xr,xi : WORD;
BEGIN  -- filters in transposed form
  WAIT UNTIL clk = '1';
  -- FF for input signals
  xr    := CONV_INTEGER(sxtxr);
  xi    := CONV_INTEGER(sxtxi);
  xr125 <=  xr - 4 * xr + 128 * xr;
  xi125 <=  xi - 4 * xi + 128 * xi;
  xr25 <=  xr + 8 * xr + 16 * xr;
  xi25 <=  xi + 8 * xi + 16 * xi;
  xr90 <=  2 * xr + 8 * xr + 16 * xr + 64 * xr + 0;
  xi90 <=  2 * xi + 8 * xi + 16 * xi + 64 * xi + 0;
  xr127 <=  - xr + 128 * xr;
  xi127 <=  - xi + 128 * xi;

  -- Real part of FIR filter in transposed form = xr*Coeff.r - xi*Coeff.i
  realn(0) <= realn(1)  + xr125 - xi25;  -- = 0.981 +0.195 * i
  realn(1) <= realn(2)  + xr90 - xi90;  -- = 0.707 +0.707 * i
  realn(2) <= realn(3)  - xr25 - xi125;  -- = -0.195 +0.981 * i
  realn(3) <= realn(4)  - xr127;  -- = -1.000 +0.000 * i
  realn(4) <= realn(5)  + xr25 + xi125;  -- = 0.195 -0.981 * i
  realn(5) <= realn(6)  + xr90 - xi90;  -- = 0.707 +0.707 * i
  realn(6) <= realn(7)  - xr125 + xi25;  -- = -0.981 -0.195 * i
  realn(7) <= realn(8)  + xr127;  -- = 1.000 -0.000 * i
  realn(8) <= realn(9)  - xr125 + xi25;  -- = -0.981 -0.195 * i
  realn(9) <= realn(10) + xr90 - xi90;  -- = 0.707 +0.707 * i
  realn(10) <= realn(11) + xr25 + xi125;  -- = 0.195 -0.981 * i
  realn(11) <= realn(12) - xr127;  -- = -1.000 +0.000 * i
realn(12) <= realn(13) - xr25 - xi125;  -- = -0.195 +0.981 * i
realn(13) <= realn(14) + xr90 - xi90;  -- = 0.707 +0.707 * i
realn(14) <= realn(15) + xr125 - xi25;  -- = 0.981 +0.195 * i
realn(15) <= realn(16) + xr127;  -- = 1.000 -0.000 * i
realn(16) <= realn(17) + xr125 - xi25;  -- = 0.981 +0.195 * i
realn(17) <= realn(18) + xr90 - xi90;  -- = 0.707 +0.707 * i
realn(18) <= realn(19) - xr25 - xi125;  -- = -0.195 +0.981 * i
realn(19) <= realn(20) - xr127;  -- = -1.000 +0.000 * i
realn(20) <= realn(21) + xr25 + xi125;  -- = 0.195 -0.981 * i
realn(21) <= realn(22) + xr90 - xi90;  -- = 0.707 +0.707 * i
realn(22) <= realn(23) + xr125 + xi25;  -- = -0.981 -0.195 * i
realn(23) <= realn(24) + xr127;  -- = 1.000 -0.000 * i
realn(24) <= realn(25) + xr125 + xi25;  -- = -0.981 -0.195 * i
realn(25) <= realn(26) + xr90 - xi90;  -- = 0.707 +0.707 * i
realn(26) <= realn(27) + xr25 + xi125;  -- = 0.195 -0.981 * i
realn(27) <= realn(28) - xr127;  -- = -1.000 +0.000 * i
realn(28) <= realn(29) - xr25 - xi125;  -- = -0.981 +0.195 * i
realn(29) <= realn(30) + xr90 - xi90;  -- = 0.707 +0.707 * i
realn(30) <= xr125 - xi25;

-- Imaginary part of FIR filter in transposed form = xr*Coeff.i + xi*Coeff.r
imag(0) <= imag(1) + xr25 + xi125;  -- = 0.981 +0.195 * i
imag(1) <= imag(2) + xr90 + xi90;  -- = 0.707 +0.707 * i
imag(2) <= imag(3) +xr125 - xi25;  -- = -0.195 +0.981 * i
imag(3) <= imag(4) - xi127;  -- = -1.000 +0.000 * i
imag(4) <= imag(5) - xr125 + xi25;  -- = 0.195 -0.981 * i
imag(5) <= imag(6) + xr90 + xi90;  -- = 0.707 +0.707 * i
imag(6) <= imag(7) - xr25 - xi125;  -- = -0.981 -0.195 * i
imag(7) <= imag(8) + xi127;  -- = 1.000 -0.000 * i
imag(8) <= imag(9) - xr25 - xi125;  -- = -0.981 -0.195 * i
imag(9) <= imag(10) + xr90 + xi90;  -- = 0.707 +0.707 * i
imag(10) <= imag(11) - xr125 + xi25;  -- = 0.195 -0.981 * i

49
imag(11) <= imag(12) - xi127; -- = -1.000 +0.000 * i
imag(12) <= imag(13) + xr125 - xi25; -- = -0.195 +0.981 * i
imag(13) <= imag(14) + xr90 + xi90; -- = 0.707 +0.707 * i
imag(14) <= imag(15) + xr25 + xi125; -- = 0.981 +0.195 * i
imag(15) <= imag(16) + xi127; -- = 1.000 -0.000 * i
imag(16) <= imag(17) + xr25 + xi125; -- = 0.981 +0.195 * i
imag(17) <= imag(18) + xr90 + xi90; -- = 0.707 +0.707 * i
imag(18) <= imag(19) + xr125 - xi25; -- = -0.195 +0.981 * i
imag(19) <= imag(20) - xi127; -- = -1.000 +0.000 * i
imag(20) <= imag(21) - xr125 + xi25; -- = 0.195 -0.981 * i
imag(21) <= imag(22) + xr90 + xi90; -- = 0.707 +0.707 * i
imag(22) <= imag(23) - xr25 - xi125; -- = -0.981 -0.195 * i
imag(23) <= imag(24) + xi127; -- = 1.000 -0.000 * i
imag(24) <= imag(25) - xr25 - xi125; -- = -0.981 -0.195 * i
imag(25) <= imag(26) + xr90 + xi90; -- = 0.707 +0.707 * i
imag(26) <= imag(27) - xr125 + xi25; -- = 0.195 -0.981 * i
imag(27) <= imag(28) - xi127; -- = -1.000 +0.000 * i
imag(28) <= imag(29) + xr125 - xi25; -- = -0.195 +0.981 * i
imag(29) <= imag(30) + xr90 + xi90; -- = 0.707 +0.707 * i
imag(30) <= xr25 + xi125;
END PROCESS filter;

-- Use table to get INPUT (=OUTPUT!) twiddle factor coefficients
tab_1: diag16b8
    PORT MAP( table_in => t1_in,
        re_out => t1_real,
        im_out => t1_imag);
t1_real_out <= t1_real;
t1_imag_out <= t1_imag;
-- Multiply inputs with twiddle factor exp(j*phi)
ccmul_1: mult
PORT MAP (clk => clk, -- Clock for the output register
    x_in => real_in, y_in => imag_in,
    c_in => t1_real, s_in => t1_imag,
    r_out => ccmul_r, i_out => ccmul_i);
ccmul_r_out <= ccmul_r;
ccmul_i_out <= ccmul_i;

-- Multiplex for the complex multiplier inputs
-- Use DFT input for the first N steps and then the filter outputs
MPX: PROCESS (x_real_in, x_imag_in, r0, i0)
BEGIN
    IF count <= 17 THEN
        real_in <= x_real_in;
        imag_in <= x_imag_in;
    ELSE
        real_in <= r0;
        imag_in <= i0;
    END IF;
END PROCESS MPX;

-- Sign extension for the multiplier output, i.e. filter input
Sxt: PROCESS (ccmul_r, ccmul_i)
BEGIN
    IF count <= 17 THEN
        sxtxr(W-1 DOWNTO 0) <= ccmul_r;
        sxtxi(W-1 DOWNTO 0) <= ccmul_i;
    FOR k IN Wfir-1 DOWNTO W LOOP
        sxtxr(k) <= ccmul_r(ccmul_r'left);
        sxtxi(k) <= ccmul_i(ccmul_i'left);
    END LOOP;
END PROCESS Sxt;
ELSE
    sxtxr <= (OTHERS => '0');
    sxtxi <= (OTHERS => '0');
END IF;
END PROCESS Sxt;
sxtxr_out <= sxtxr;
sxtxi_out <= sxtxi;

-- Multiply filter output with twiddle factor \exp(j*phi)
    r0 <= CONV_STD_LOGIC_VECTOR( (realn(0)/128) , 8);
    i0 <= CONV_STD_LOGIC_VECTOR( (imag(0)/128) , 8);
    r0_out <= r0;
    i0_out <= i0;

-- ccmul_2: cm3a5b8
-- PORT MAP (clk => clk, -- Clock for the output register
--   x_in => r0, y_in => i0,
--   c_in => t2_real, s_in => t2_imag,
--   r_out => y_r , i_out => y_i);
END flex;
-- Project Name: fir16pt8b.npl
-- File Name: diag16b8.vhd
-- Description: Twiddle factor for 16 point DFT
-- Author: Dr. Uwe Meyer-Baese
-- Date: 03/30/04
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;

ENTITY diag16b8 IS
  PORT ( table_in  : IN  STD_LOGIC_VECTOR(4-1 DOWNTO 0);
         re_out    : OUT STD_LOGIC_VECTOR(8-1 DOWNTO 0);
         im_out    : OUT STD_LOGIC_VECTOR(8-1 DOWNTO 0));
END diag16b8;

ARCHITECTURE LCs OF diag16b8 IS
BEGIN

  -- This is the 8 bit (i.e. -127 ... 127) CASE table for
  -- the 16-point CZT pre- and post-multiplications
  -- automatic generated with cztgen.exe -- DO NOT EDIT!

  -- Table implements the mapping
  PROCESS (table_in)
  BEGIN
    CASE table_in IS
      WHEN "0000" => re_out <= "01111111"; -- = 127 i.e. 1.000
      im_out <= "00000000"; -- = 0 i.e. 0.000
  END CASE
END LCs;

WHEN "0001" => re_out <= "01111101";  -- = 125 i.e. 0.981
     im_out <= "11100111";  -- = -25 i.e. -0.195
WHEN "0010" => re_out <= "01011010";  -- = 90 i.e. 0.707
     im_out <= "10100110";  -- = -90 i.e. -0.707
WHEN "0011" => re_out <= "11100111";  -- = -25 i.e. -0.195
     im_out <= "10000011";  -- = -125 i.e. -0.981
WHEN "0100" => re_out <= "10000001";  -- = -127 i.e. -1.000
     im_out <= "00000000";  -- = 0 i.e. -0.000
WHEN "0101" => re_out <= "00011001";  -- = 25 i.e. 0.195
     im_out <= "01111101";  -- = 125 i.e. 0.981
WHEN "0110" => re_out <= "01011010";  -- = 90 i.e. 0.707
     im_out <= "10100110";  -- = -90 i.e. -0.707
WHEN "0111" => re_out <= "10000001";  -- = -125 i.e. -0.981
     im_out <= "00011001";  -- = 25 i.e. 0.195
WHEN "1000" => re_out <= "01111111";  -- = 127 i.e. 1.000
     im_out <= "00000000";  -- = 0 i.e. 0.000
WHEN "1001" => re_out <= "10000001";  -- = -125 i.e. -0.981
     im_out <= "00011001";  -- = 25 i.e. 0.195
WHEN "1010" => re_out <= "01011010";  -- = 90 i.e. 0.707
     im_out <= "10100110";  -- = -90 i.e. -0.707
WHEN "1011" => re_out <= "00011001";  -- = 25 i.e. 0.195
     im_out <= "01111101";  -- = 125 i.e. 0.981
WHEN "1100" => re_out <= "10000011";  -- = -127 i.e. -1.000
     im_out <= "00000000";  -- = 0 i.e. -0.000
WHEN "1101" => re_out <= "11100111";  -- = -25 i.e. -0.195
     im_out <= "10000011";  -- = -125 i.e. -0.981
WHEN "1110" => re_out <= "01011010";  -- = 90 i.e. 0.707
     im_out <= "10100110";  -- = -90 i.e. -0.707
WHEN "1111" => re_out <= "01111101";  -- = 125 i.e. 0.981
     im_out <= "11100111";  -- = -25 i.e. -0.195
WHEN OTHERS => re_out <= (OTHERS => '0');
     im_out <= (OTHERS => '0');
END CASE;
END PROCESS;
END LCs;
APPENDIX C

VHDL CODES FOR 32 POINT DFT

I.

--Project Name: dft32pt.npl
--File Name: czt32sb.vhd
--Description: Implementation of 32 point DFT
--Author: Dr. Uwe Meyer-Baese and Hariharan Natarajan
--Date: 03/30/04
--This is the 8 bit and
--32-point CZT FSM and FIR filter
--automatic generated with cztgen.exe -- DO NOT EDIT!
--The filter has 7 real and 7 imaginary different coefficients

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;

ENTITY czt32s8 IS                      ------> Interface
  GENERIC (Wfir: INTEGER := 22;    -- Filter accu bit width
            Lfir: INTEGER := 63;    -- Filter length = 2*DFTlengh-1
            W1  : INTEGER := 5;    -- log2(DFT length)
            W   : INTEGER := 8);    -- Data/Coeff bit width
  PORT ( clk            : IN  STD_LOGIC;
         x_real_in, x_imag_in : IN  STD_LOGIC_VECTOR(W-1 DOWNTO 0);
         valid_in             : OUT  STD_LOGIC;
         t1_real_out,t1_imag_out, ccmul_r_out,
         ccmul_i_out          : OUT STD_LOGIC_VECTOR(W-1 DOWNTO 0);
      )
END ENTITY czt32s8;
ARCHITECTURE flex OF czt32s8 IS

COMPONENT diag32b8
PORT ( table_in  : IN  STD_LOGIC_VECTOR(5-1 DOWNTO 0);
       re_out    : OUT STD_LOGIC_VECTOR(8-1 DOWNTO 0);
       im_out    : OUT STD_LOGIC_VECTOR(8-1 DOWNTO 0));
END COMPONENT;

COMPONENT mult
GENERIC (W2  : INTEGER := 17; -- Multiplier bit width
          W1  : INTEGER := 9;     -- Bit width c+s sum
          W   : INTEGER := 8);    -- Input bit width
PORT (clk  : STD_LOGIC;  -- Clock for the output register
       x_in, y_in    : IN  STD_LOGIC_VECTOR(W-1 DOWNTO 0);
       c_in, s_in    : IN  STD_LOGIC_VECTOR(W-1 DOWNTO 0);
       r_out, i_out  : OUT STD_LOGIC_VECTOR(W-1 DOWNTO 0));
END COMPONENT;

SIGNAL  count    : integer RANGE 0 TO 127;
TYPE    STATE_TYPE IS (Start, Load, Run);
SIGNAL  state    : STATE_TYPE ;
SIGNAL  x_real, x_imag : STD_LOGIC_VECTOR(W-1 DOWNTO 0);

SUBTYPE WORD IS INTEGER RANGE -2**19 TO 2**19-1;
TYPE ARRAY_WORD IS ARRAY (0 to 62) OF WORD;
SIGNAL  real, imag : ARRAY_WORD:=(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, -- Tapped delay line array
------******* Real and imaginary filter coefficients:

SIGNAL  xr126, xi126 : WORD:=0;
SIGNAL  xr12, xi12  : WORD:=0;
SIGNAL  xr117, xi117  : WORD:=0;
SIGNAL  xr49, xi49  : WORD:=0;
SIGNAL  xr81, xi81  : WORD:=0;
SIGNAL  xr98, xi98  : WORD:=0;
SIGNAL  xr127, xi127  : WORD:=0;

-- Auxiliary signal definitions

SIGNAL  t1_in            : STD_LOGIC_VECTOR(W1-1 DOWNTO 0);
SIGNAL  real_in, imag_in : STD_LOGIC_VECTOR(W-1 DOWNTO 0);
SIGNAL  t1_real, t1_imag : STD_LOGIC_VECTOR(W-1 DOWNTO 0);
SIGNAL  ccmul_r, ccmul_i : STD_LOGIC_VECTOR(W-1 DOWNTO 0);
SIGNAL  sxtxr, sxtxi     : STD_LOGIC_VECTOR(Wfir-1 DOWNTO 0);
SIGNAL  r0, i0           : STD_LOGIC_VECTOR(W-1 DOWNTO 0);
SIGNAL  y_r, y_i         : STD_LOGIC_VECTOR(W-1 DOWNTO 0);

BEGIN

States: PROCESS  -------> State machine for RADER filter
BEGIN

WAIT UNTIL clk = '1';
CASE state IS

WHEN Start =>  -- Initialization step
state <= Load;
count <= 0;
t1_in <= "00000"; -- Reset tables
WHEN Load => -- Multiply inputs with twiddle factors
  IF count = 33 THEN -- Load phase done ?
    state <= Run;
  ELSE
    state <= Load;
    t1_in <= t1_in + "00001";
  END IF;
  IF count < 32 THEN
    t1_in <= t1_in + "00001";
  ELSE
    t1_in <= "00000"; -- Reset tables
  END IF;
count <= count + 1;
WHEN Run => -- Multiply filter output with twiddle factors
  IF count = 95 THEN -- Run phase done ?
    state <= Start; -- Output of result
    t1_in <= "00000"; -- Reset tables
  ELSE -- and start again
    state <= Run;
    t1_in <= t1_in + "00001";
  END IF;
count <= count + 1;
END CASE;
IF count > 34 AND count < 67 THEN
  valid_out <= '1';
  y_real <= ccmul_r;
  y_imag <= ccmul_i;
ELSE
  valid_out <= '0';
y_real <= (OTHERS => '0');
y_imag <= (OTHERS => '0');
END IF;
IF count < 31 THEN
valid_in <= '1';
ELSE
valid_in <= '0';
END IF;
END PROCESS States;

Filter: PROCESS  -- Structure of the two FIR
VARIABLE xr, xi :  WORD;
BEGIN  -- filters in transposed form
  WAIT UNTIL clk = '1';
  -- FF for input signals
  -- FF for input signals
  xr   := CONV_INTEGER(sxtxr);
  xi   := CONV_INTEGER(sxtxi);
  xr126 <=  - 2 * xr + 128 * xr;
  xi126 <=  - 2 * xi + 128 * xi;
  xr12  <=  + 4 * xr + 8 * xr;
  xi12  <=  + 4 * xi + 8 * xi;
  xr117 <=  xr + 4 * xr - 16 * xr + 128 * xr;
  xi117 <=  xi + 4 * xi - 16 * xi + 128 * xi;
  xr49  <=  xr + 16 * xr + 32 * xr;
  xi49  <=  xi + 16 * xi + 32 * xi;
  xr81  <=  xr + 16 * xr + 64 * xr;
  xi81  <=  xi + 16 * xi + 64 * xi;
  xr98  <=  + 2 * xr + 32 * xr + 64 * xr;
  xi98  <=  + 2 * xi + 32 * xi + 64 * xi;
  xr127 <=  - xr + 128 * xr;
  xi127 <=  - xi + 128 * xi;
-- Real part of FIR filter in transposed form = xr*Coeff.r - xi*Coeff.i
real(0) <= real(1) + xr126 - xi12; -- = 0.995 +0.098 * i
real(1) <= real(2) + xr117 - xi49; -- = 0.924 +0.383 * i
real(2) <= real(3) + xr81 - xi98; -- = 0.634 +0.773 * i
real(3) <= real(4) - xi127; -- = 0.000 +1.000 * i
real(4) <= real(5) - xr98 - xi81; -- = -0.773 +0.634 * i
real(5) <= real(6) - xr117 + xi49; -- = -0.924 -0.383 * i
real(6) <= real(7) + xr12 + xi126; -- = 0.098 -0.995 * i
real(7) <= real(8) + xr127; -- = 1.000 -0.000 * i
real(8) <= real(9) - xr12 - xi126; -- = -0.098 +0.995 * i
real(9) <= real(10) - xr117 + xi49; -- = -0.924 -0.383 * i
real(10) <= real(11) + xr98 + xi81; -- = 0.773 -0.634 * i
real(11) <= real(12) - xi127; -- = 0.000 +1.000 * i
real(12) <= real(13) - xr81 + xi98; -- = -0.634 -0.773 * i
real(13) <= real(14) + xr117 - xi49; -- = 0.924 +0.383 * i
real(14) <= real(15) - xr126 + xi12; -- = -0.995 -0.098 * i
real(15) <= real(16) + xr127; -- = 1.000 -0.000 * i
real(16) <= real(17) - xr126 + xi12; -- = -0.995 -0.098 * i
real(17) <= real(18) + xr117 - xi49; -- = 0.924 +0.383 * i
real(18) <= real(19) - xr81 + xi98; -- = -0.634 -0.773 * i
real(19) <= real(20) - xi127; -- = 0.000 +1.000 * i
real(20) <= real(21) + xr98 + xi81; -- = 0.773 -0.634 * i
real(21) <= real(22) - xr117 + xi49; -- = -0.924 -0.383 * i
real(22) <= real(23) - xr12 - xi126; -- = -0.098 +0.995 * i
real(23) <= real(24) + xr127; -- = 1.000 -0.000 * i
real(24) <= real(25) + xr12 + xi126; -- = 0.098 -0.995 * i
real(25) <= real(26) - xr117 + xi49; -- = -0.924 -0.383 * i
real(26) <= real(27) - xr98 - xi81; -- = -0.773 +0.634 * i
real(27) <= real(28) - xi127; -- = 0.000 +1.000 * i
real(28) <= real(29) + xr81 - xi98; -- = 0.634 +0.773 * i
real(29) <= real(30) + xr117 - xi49; -- = 0.924 +0.383 * i
real(30) <= real(31) + xr126 - xi12; -- = 0.995 +0.098 * i

61
real(31) <= real(32) + xr127;  -- = 1.000 -0.000 * i
real(32) <= real(33) + xr126 - xi12;  -- = 0.995 +0.998 * i
real(33) <= real(34) + xr117 - xi49;  -- = 0.924 +0.383 * i
real(34) <= real(35) + xr81 - xi98;  -- = 0.634 +0.773 * i
real(35) <= real(36) - xi127;  -- = 0.000 +1.000 * i
real(36) <= real(37) - xr98 - xi81;  -- = -0.773 +0.634 * i
real(37) <= real(38) - xr117 + xi49;  -- = -0.924 -0.383 * i
real(38) <= real(39) + xr12 + xi126;  -- = 0.098 -0.995 * i
real(39) <= real(40) + xr127;  -- = 1.000 -0.000 * i
real(40) <= real(41) - xr12 - xi126;  -- = -0.098 +0.995 * i
real(41) <= real(42) - xr117 + xi49;  -- = -0.924 -0.383 * i
real(42) <= real(43) + xr98 + xi81;  -- = 0.773 -0.634 * i
real(43) <= real(44) - xi127;  -- = 0.000 +1.000 * i
real(44) <= real(45) - xr81 + xi98;  -- = -0.634 -0.773 * i
real(45) <= real(46) + xr117 - xi49;  -- = 0.924 +0.383 * i
real(46) <= real(47) - xr126 + xi12;  -- = -0.995 -0.098 * i
real(47) <= real(48) + xr127;  -- = 1.000 -0.000 * i
real(48) <= real(49) - xr126 + xi12;  -- = -0.995 -0.098 * i
real(49) <= real(50) + xr117 - xi49;  -- = 0.924 +0.383 * i
real(50) <= real(51) - xr81 + xi98;  -- = -0.634 -0.773 * i
real(51) <= real(52) - xi127;  -- = 0.000 +1.000 * i
real(52) <= real(53) + xr98 + xi81;  -- = 0.773 -0.634 * i
real(53) <= real(54) - xr117 + xi49;  -- = -0.924 -0.383 * i
real(54) <= real(55) - xr12 - xi126;  -- = -0.098 +0.995 * i
real(55) <= real(56) + xr127;  -- = 1.000 -0.000 * i
real(56) <= real(57) + xr12 + xi126;  -- = 0.098 -0.995 * i
real(57) <= real(58) - xr117 + xi49;  -- = -0.924 -0.383 * i
real(58) <= real(59) - xr98 - xi81;  -- = -0.773 +0.634 * i
real(59) <= real(60) - xi127;  -- = 0.000 +1.000 * i
real(60) <= real(61) + xr81 - xi98;  -- = 0.634 +0.773 * i
real(61) <= real(62) + xr117 - xi49;  -- = 0.924 +0.383 * i
real(62) <= xr126 - xi12;
-- Imaginary part of FIR filter in transposed form = xr*Coeff.i + xi*Coeff.r
imag(0) <= imag(1) + xr12 + xi126; -- = 0.995 +0.098 * i
imag(1) <= imag(2) + xr49 + xi117; -- = 0.924 +0.383 * i
imag(2) <= imag(3) + xr98 + xi81; -- = 0.634 +0.773 * i
imag(3) <= imag(4) + xr127; -- = 0.000 +1.000 * i
imag(4) <= imag(5) + xr81 - xi98; -- = -0.773 +0.634 * i
imag(5) <= imag(6) - xr49 - xi117; -- = -0.924 -0.383 * i
imag(6) <= imag(7) - xr126 + xi12; -- = 0.098 -0.995 * i
imag(7) <= imag(8) + xi127; -- = 1.000 -0.000 * i
imag(8) <= imag(9) + xr126 - xi12; -- = -0.995 + 0.995 * i
imag(9) <= imag(10) - xr49 - xi117; -- = -0.924 -0.383 * i
imag(10) <= imag(11) - xr81 + xi98; -- = 0.773 -0.634 * i
imag(11) <= imag(12) + xr127; -- = 0.000 +1.000 * i
imag(12) <= imag(13) - xr98 - xi81; -- = -0.634 -0.773 * i
imag(13) <= imag(14) + xr49 + xi117; -- = 0.924 +0.383 * i
imag(14) <= imag(15) - xr12 - xi126; -- = -0.995 -0.998 * i
imag(15) <= imag(16) + xi127; -- = 1.000 -0.000 * i
imag(16) <= imag(17) - xr12 - xi126; -- = -0.995 -0.998 * i
imag(17) <= imag(18) + xr49 + xi117; -- = 0.924 +0.383 * i
imag(18) <= imag(19) - xr98 - xi81; -- = -0.634 -0.773 * i
imag(19) <= imag(20) + xr127; -- = 0.000 +1.000 * i
imag(20) <= imag(21) - xr81 + xi98; -- = 0.773 -0.634 * i
imag(21) <= imag(22) - xr49 - xi117; -- = -0.924 -0.383 * i
imag(22) <= imag(23) + xr126 - xi12; -- = -0.098 +0.995 * i
imag(23) <= imag(24) + xi127; -- = 1.000 -0.000 * i
imag(24) <= imag(25) - xr126 + xi12; -- = 0.098 -0.995 * i
imag(25) <= imag(26) - xr49 - xi117; -- = -0.924 -0.383 * i
imag(26) <= imag(27) + xr81 - xi98; -- = -0.773 +0.634 * i
imag(27) <= imag(28) + xr127; -- = 0.000 +1.000 * i
imag(28) <= imag(29) + xr98 + xi81; -- = 0.634 +0.773 * i
imag(29) <= imag(30) + xr49 + xi117; -- = 0.924 +0.383 * i
imag(30) <= imag(31) + xr12 + xi126; -- = 0.995 +0.098 * i
imag(31) <= imag(32) + xi127; -- = 1.000 -0.000 * i
imag(32) <= imag(33) + xr12 + xi126; -- = 0.995 +0.098 * i
imag(33) <= imag(34) + xr49 + xi117; -- = 0.924 +0.383 * i
imag(34) <= imag(35) + xr98 + xi81; -- = 0.634 +0.773 * i
imag(35) <= imag(36) + xr127; -- = 0.000 +1.000 * i
imag(36) <= imag(37) + xr81 - xi98; -- = -0.773 +0.634 * i
imag(37) <= imag(38) - xr49 - xi117; -- = -0.924 -0.383 * i
imag(38) <= imag(39) - xr126 + xi12; -- = 0.098 -0.995 * i
imag(39) <= imag(40) + xi127; -- = 1.000 -0.000 * i
imag(40) <= imag(41) + xr126 - xi12; -- = -0.098 +0.995 * i
imag(41) <= imag(42) - xr49 - xi117; -- = -0.924 -0.383 * i
imag(42) <= imag(43) - xr81 + xi98; -- = 0.773 -0.634 * i
imag(43) <= imag(44) + xr127; -- = 0.000 +1.000 * i
imag(44) <= imag(45) - xr98 - xi81; -- = -0.634 -0.773 * i
imag(45) <= imag(46) + xr49 + xi117; -- = 0.924 +0.383 * i
imag(46) <= imag(47) - xr12 - xi126; -- = -0.995 -0.098 * i
imag(47) <= imag(48) + xi127; -- = 1.000 -0.000 * i
imag(48) <= imag(49) - xr12 - xi126; -- = -0.995 -0.098 * i
imag(49) <= imag(50) + xr49 + xi117; -- = 0.924 +0.383 * i
imag(50) <= imag(51) - xr98 - xi81; -- = -0.634 -0.773 * i
imag(51) <= imag(52) + xr127; -- = 0.000 +1.000 * i
imag(52) <= imag(53) - xr81 + xi98; -- = 0.773 -0.634 * i
imag(53) <= imag(54) - xr49 - xi117; -- = -0.924 -0.383 * i
imag(54) <= imag(55) + xr126 - xi12; -- = -0.098 +0.995 * i
imag(55) <= imag(56) + xi127; -- = 1.000 -0.000 * i
imag(56) <= imag(57) - xr126 + xi12; -- = 0.098 -0.995 * i
imag(57) <= imag(58) - xr49 - xi117; -- = -0.924 -0.383 * i
imag(58) <= imag(59) + xr81 - xi98; -- = -0.773 +0.634 * i
imag(59) <= imag(60) + xr127; -- = 0.000 +1.000 * i
imag(60) <= imag(61) + xr98 + xi81; -- = 0.634 +0.773 * i
imag(61) <= imag(62) + xr49 + xi117; -- = 0.924 +0.383 * i

64
imag(62) <= xr12 + xi126;

END PROCESS Filter;

-- Use table to get INPUT (=OUTPUT!) twiddle factor coefficients
	tab_1: diag32b8
	PORT MAP( table_in => t1_in,
				re_out => t1_real,
				im_out => t1_imag);

t1_real_out <= t1_real;
t1_imag_out <= t1_imag;

-- Multiply inputs with twiddle factor exp(j*phi)
	ccmul_1: mult
	PORT MAP (clk => clk,        -- Clock for the output register
		x_in => real_in, y_in => imag_in,
	
c_in => t1_real, s_in => t1_imag,
		r_out => ccmul_r , i_out => ccmul_i);
	ccmul_r_out <= ccmul_r;	ccmul_i_out <= ccmul_i;

-- Multiplex for the complex multiplier inputs
-- Use DFT input for the first N steps and than the filter outputs
	MPX: PROCESS (x_real_in, x_img_in, r0, i0)
	BEGIN
	IF count <= 33 THEN
	real_in <= x_real_in;	imag_in <= x_img_in;
	ELSE
	real_in <= r0;
	ENDIF
imag_in <= i0;
END IF;
END PROCESS MPX;

-- Sign extentions for the multiplier output, i.e. filter input
Sxt: PROCESS (ccmul_r, ccmul_i)
BEGIN
IF count <= 33 THEN
    sxtxr(W-1 DOWNTO 0) <= ccmul_r;
sxtxi(W-1 DOWNTO 0) <= ccmul_i;
    FOR k IN Wfir-1 DOWNTO W LOOP
        sxtxr(k) <= ccmul_r(ccmul_r'left);
sxtxi(k) <= ccmul_i(ccmul_i'left);
    END LOOP;
ELSE
    sxtxr <= (OTHERS => '0');
sxtxi <= (OTHERS => '0');
END IF;
END PROCESS Sxt;
sxtxr_out <= sxtxr;
sxtxi_out <= sxtxi;

-- Multiply filter output with twiddle factor exp(j*phi)
r0 <= CONV_STD_LOGIC_VECTOR( real(0) / 128, 8);
i0 <= CONV_STD_LOGIC_VECTOR( imag(0) / 128, 8);
r0_out <= r0;
i0_out <= i0;
-- ccmul_2: cm3a5b8
-- PORT MAP (clk => clk, -- Clock for the output register
-- x_in => r0, y_in => i0,
-- c_in => t2_real, s_in => t2_imag,
-- r_out => y_r , i_out => y_i);
END flex;
II.

--Project Name: dft32pt.npl
--File Name: diag32b8.vhd
--Description: Twiddle factors for 32 point DFT
--Author: Hariharan Natarajan
--Date : 03/30/04

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY diag32b8 IS
  PORT ( table_in  : IN  STD_LOGIC_VECTOR(5-1 DOWNTO 0);
         re_out    : OUT STD_LOGIC_VECTOR(8-1 DOWNTO 0);
         im_out    : OUT STD_LOGIC_VECTOR(8-1 DOWNTO 0));
END diag32b8;

ARCHITECTURE LCs OF diag32b8 IS
BEGIN

  -- This is the 8 bit (i.e. -127 ... 127) CASE table for
  -- the 32-point CZT pre- and post-multiplications
  -- automatic generated with cztgen.exe -- DO NOT EDIT!

  -- Table implements the mapping
  PROCESS (table_in)
  BEGIN
    CASE table_in IS
      WHEN "00000" => re_out <= "01111111"; -- = 127 i.e. 1.000
                  im_out <= "00000000"; -- = 0 i.e. 0.000
      WHEN "00001" => re_out <= "01111110"; -- = 126 i.e. 0.995
                  im_out <= "11110100"; -- = -12 i.e. -0.098
      \ldots
    END CASE;
  END PROCESS;

END LCs;
WHEN "00010" => re_out <= "01110101"; re rest 117 i.e. 0.924
    im_out <= "11001111"; -- -49 i.e. -0.383
WHEN "00011" => re_out <= "01010001"; re rest 81 i.e. 0.634
    im_out <= "10011110"; -- -98 i.e. -0.773
WHEN "00100" => re_out <= "00000000"; re rest 0 i.e. 0.000
    im_out <= "10000001"; -- -127 i.e. -1.000
WHEN "00101" => re_out <= "10011110"; re rest 98 i.e. -0.773
    im_out <= "10101111"; -- -81 i.e. -0.634
WHEN "00110" => re_out <= "10001011"; re rest 117 i.e. -0.924
    im_out <= "00110001"; -- 49 i.e. 0.383
WHEN "00111" => re_out <= "00001100"; re rest 12 i.e. 0.098
    im_out <= "01111110"; -- 126 i.e. 0.995
WHEN "01000" => re_out <= "01111111"; re rest 127 i.e. 1.000
    im_out <= "00000000"; -- 0 i.e. 0.000
WHEN "01001" => re_out <= "11110100"; re rest -12 i.e. -0.098
    im_out <= "10000010"; -- -126 i.e. -0.995
WHEN "01010" => re_out <= "10001011"; re rest -117 i.e. -0.924
    im_out <= "00110001"; -- 49 i.e. 0.383
WHEN "01011" => re_out <= "01100010"; re rest 98 i.e. 0.773
    im_out <= "01010001"; -- 81 i.e. 0.634
WHEN "01100" => re_out <= "00000000"; re rest 0 i.e. 0.000
    im_out <= "10000001"; -- -127 i.e. -1.000
WHEN "01101" => re_out <= "10101111"; re rest -81 i.e. -0.634
    im_out <= "01100010"; -- 98 i.e. 0.773
WHEN "01110" => re_out <= "01110101"; re rest 117 i.e. 0.924
    im_out <= "11001111"; -- -49 i.e. -0.383
WHEN "01111" => re_out <= "10000010"; re rest -126 i.e. -0.995
    im_out <= "00001100"; -- 12 i.e. 0.098
WHEN "10000" => re_out <= "01111111"; re rest 127 i.e. 1.000
    im_out <= "00000000"; -- 0 i.e. 0.000
WHEN "10001" => re_out <= "10000010"; re rest -126 i.e. -0.995
    im_out <= "00001100"; -- 12 i.e. 0.098

69
WHEN "10010" => re_out <= "01110101"; -- = 117 i.e. 0.924
    im_out <= "11001111"; -- = -49 i.e. -0.383
WHEN "10011" => re_out <= "10101111"; -- = -81 i.e. -0.634
    im_out <= "01100010"; -- = 98 i.e. 0.773
WHEN "10100" => re_out <= "00000000"; -- = 0 i.e. 0.000
    im_out <= "10000001"; -- = -127 i.e. -1.000
WHEN "10101" => re_out <= "01100010"; -- = 98 i.e. 0.773
    im_out <= "01010001"; -- = 81 i.e. 0.634
WHEN "10110" => re_out <= "10001011"; -- = -117 i.e. -0.924
    im_out <= "00110001"; -- = 49 i.e. 0.383
WHEN "10111" => re_out <= "11110100"; -- = -12 i.e. -0.098
    im_out <= "10000010"; -- = -126 i.e. -0.995
WHEN "11000" => re_out <= "01111111"; -- = 127 i.e. 1.000
    im_out <= "00000000"; -- = 0 i.e. 0.000
WHEN "11001" => re_out <= "00001100"; -- = 12 i.e. 0.098
    im_out <= "01111110"; -- = 126 i.e. 0.995
WHEN "11010" => re_out <= "10001011"; -- = -117 i.e. -0.924
    im_out <= "00110001"; -- = 49 i.e. 0.383
WHEN "11011" => re_out <= "10011110"; -- = -98 i.e. -0.773
    im_out <= "10101111"; -- = -81 i.e. -0.634
WHEN "11100" => re_out <= "00000000"; -- = 0 i.e. 0.000
    im_out <= "10000001"; -- = -127 i.e. -1.000
WHEN "11101" => re_out <= "01010001"; -- = 81 i.e. 0.634
    im_out <= "10011110"; -- = -98 i.e. -0.773
WHEN "11110" => re_out <= "01101001"; -- = 117 i.e. 0.924
    im_out <= "11001111"; -- = -49 i.e. -0.383
WHEN "11111" => re_out <= "01111110"; -- = 126 i.e. 0.995
    im_out <= "11110100"; -- = -12 i.e. -0.098
WHEN OTHERS => re_out <= (OTHERS => '0');
    im_out <= (OTHERS => '0');
END CASE;
END PROCESS;
END LCs;
APPENDIX D

VHDL CODES FOR 64 POINT DFT

I.

-- Project Name: dft64pt.npl
-- File Name: czr64s16o0g4.vhd
-- Description: Test bench for czt32s8(Implementation of 32 point DFT)
-- Author: Dr. Uwe Meyer-Baese and Hariharan Natarajan
-- Date: 03/30/04

-- This is the 64-point CZT FSM and FIR filter
-- automatic generated with cztgen.exe -- DO NOT EDIT!
-- The filter has 12 real( and 12 imag(inary different coefficients
-- Coefficient bits = 8       Data bits = 8
-- OVL guard bits = 0    LSB guard bits = 4

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;

ENTITY czr64s16o0g4 IS                ------> Interface
  GENERIC (Wfir: INTEGER := 20;    -- Filter accu bit width
            Lfir: INTEGER := 127;  -- Filter length = 2*DFTlenght-1
            W1 : INTEGER := 6;    -- log2(DFT length)
            W  : INTEGER := 8);   -- Data/Coeff bit width

  PORT ( clk      : IN  STD_LOGIC;
         x_real_in, x_imag_in : IN  STD_LOGIC_VECTOR(W-1 DOWNTO 0);
         valid_in         : OUT STD_LOGIC;

         hotc    : IN  STD_LOGIC;
         outc    : OUT STD_LOGIC;
         opt     : OUT STD_LOGIC;
         ncss    : OUT STD_LOGIC;
         dcss    : OUT STD_LOGIC;
         hcss    : OUT STD_LOGIC;
         zcss    : OUT STD_LOGIC;
         wcss    : OUT STD_LOGIC;
         lcss    : OUT STD_LOGIC;
         ocss    : OUT STD_LOGIC;
         mccs    : OUT STD_LOGIC;
         scs     : OUT STD_LOGIC;
         mcss    : OUT STD_LOGIC;
         mxcss   : OUT STD_LOGIC;
         post    : OUT STD_LOGIC;
         result  : OUT STD_LOGIC;
         y_real_out, y_imag_out : OUT STD_LOGIC_VECTOR(W-1 DOWNTO 0);
         y_valid_out : OUT STD_LOGIC;

    );
ARCHITECTURE flex OF czr64s16o0g4 IS

COMPONENT d64ptb8b
PORT ( table_in  : IN  STD_LOGIC_VECTOR(6-1 DOWNTO 0);
       re_out    : OUT STD_LOGIC_VECTOR(8-1 DOWNTO 0);
       im_out    : OUT STD_LOGIC_VECTOR(8-1 DOWNTO 0));
END COMPONENT;

COMPONENT mult
GENERIC (W2  : INTEGER := 17;  -- Multiplier bit width
          W1  : INTEGER := 9;     -- Bit width c+s sum
          W   : INTEGER := 8);    -- Input bit width
PORT (clk  : STD_LOGIC;  -- Clock for the output register
       x_in, y_in    : IN  STD_LOGIC_VECTOR(W-1 DOWNTO 0);
       c_in, s_in    : IN  STD_LOGIC_VECTOR(W-1 DOWNTO 0);
       r_out, i_out  : OUT STD_LOGIC_VECTOR(W-1 DOWNTO 0));
END COMPONENT;

SIGNAL  count    : integer RANGE 0 TO 255;
TYPE    STATE_TYPE IS (Start, Load, Run);
SIGNAL  state    : STATE_TYPE ;
SIGNAL  x_real, x_imag : STD_LOGIC_VECTOR(W-1 DOWNTO 0);

SUBTYPE WORD IS INTEGER RANGE -2**19 TO 2**19-1;
TYPE ARRAY_WORD IS ARRAY (0 to 126) OF WORD;

SIGNAL real, imag : ARRAY_WORD:=(0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,
BEGIN

States: PROCESS    ------> State machine for CZT filter

BEGIN

WAIT UNTIL clk = '1';

CASE state IS

WHEN Start =>       -- Initialization step
    state <= Load;
    count <= 0;
    t1_in <= "000000";       -- Reset tables

WHEN Load =>       -- Multiply inputs with twiddle factors
    IF count = 65 THEN    -- Load phase done ?
        state <= Run;
    ELSE
        state <= Load;
        t1_in <= t1_in + "000001";
    END IF;

END CASE;

    IF count < 64 THEN
t1_in <= t1_in + "000001";
ELSE
  t1_in <= "000000"; -- Reset tables
END IF;

count <= count + 1;

WHEN Run => -- Multiply filter output with twiddle factors
  IF count = 191 THEN -- Run phase done ?
    state <= Start; -- Output of result
    t1_in <= "000000"; -- Reset tables
  ELSE -- and start again
    state <= Run;
    t1_in <= t1_in + "000001";
  END IF;

END IF;

END CASE;

IF count > 67 AND count < 132 THEN
  valid_out <= '1';
  y_real <= ccmul_r;
  y_imag <= ccmul_i;
ELSE
  valid_out <= '0';
  y_real <= (OTHERS => '0');
  y_imag <= (OTHERS => '0');
END IF;

END PROCESS States;

-- Note that all signals: xrXXX, xiXXX are global defined
Filter: PROCESS

VARIABLE xr, xi : WORD;
BEGIN
  wait until clk='1';
-- Compute the filter coefficients but use no FFs
  xr := CONV_INTEGER(sxtxr);
  xi := CONV_INTEGER(sxtxi);
-- rag: Generate even coefficients by shift of fundamentals
  xr1 <= xr;
  xi1 <= xi;
  xr90 <= xr45*2;
  xi90 <= xi45*2;
  xr116 <= xr29*4;
  xi116 <= xi29*4;
  xr94 <= xr47*2;
  xi94 <= xi47*2;
-- rag: Generate coefficient by multiplying fundamentals with power-of-2
  xr25 <= 16*xr1 + 8*xr1 + 1*xr1;
  xi25 <= 16*xi1 + 8*xi1 + 1*xi;
  xr29 <= xr25 + 4*xr1;
  xi29 <= xi25 + 4*xi;
  xr45 <= xr29 + xr1*16;
  xi45 <= xi29 + xi1*16;
  xr47 <= xr45 + xr1*2;
  xi47 <= xi45 + xi1*2;
  xr85 <= xr1*64 + xr25 - xr1*4;
  xi85 <= xi1*64 + xi25 - xi1*4;
  xr115 <= xr85 + xr29 + xr1*1;
  xi115 <= xi85 + xi29 + xi1*1;
  xr127 <= xr1*128-xr1;
  xi127 <= xi1*128-xi1;
-- Comp. Time= 44.000000 sec.
-- Results adder: rag = 46  mag = 80  csd = 96
-- Resources: length extfund = 115  summat = 10419
-- RAG with Add/Sub
-- linput= 12  lcoefft = 11  mag table used = 18 bits

-- real( and imag(inary complex filter coefficients
-- real( part = xr*Coeff.r - xi*Coeff.i
-- imag(inary part = xr*Coeff.i + xi*Coeff.r

-- real( part of FIR filter in transposed form = xr*Coeff.r - xi*Coeff.i
real(0) <= real(1) + x127 - x16; --0.999 +0.049 * i
real(1) <= real(2) + x125 - xi25; --0.981 +0.195 * i
real(2) <= real(3) + x115 - xi54; --0.904 +0.428 * i
real(3) <= real(4) + x90 - xi90; --0.707 +0.707 * i
real(4) <= real(5) + x43 - xi116; --0.337 +0.942 * i
real(5) <= real(6) - x25 - xi125; --0.195 +0.981 * i
real(6) <= real(7) - x94 - xi85; --0.741 +0.672 * i
real(7) <= real(8) - x127; --1.000 +0.000 * i
real(8) <= real(9) - x85 + xi94; --0.672 -0.741 * i
real(9) <= real(10) + x25 + xi125; --0.195 -0.981 * i
real(10) <= real(11) + x116 + xi43; --0.942 -0.337 * i
real(11) <= real(12) + x90 - xi90; --0.707 +0.707 * i
real(12) <= real(13) - x54 - xi115; --0.428 +0.904 * i
real(13) <= real(14) - x125 + xi25; --0.981 -0.195 * i
real(14) <= real(15) + x6 + xi127; --0.049 -0.999 * i
real(15) <= real(16) + x127; --1.000 +0.000 * i
real(16) <= real(17) - x6 - xi127; --0.049 +0.999 * i
real(17) <= real(18) - x125 + xi25; --0.981 -0.195 * i
real(18) <= real(19) + x54 + xi115; --0.428 -0.904 * i
real(19) <= real(20) + x90 - xi90; --0.707 +0.707 * i
real(20) <= real(21) - xr116 - xi43;--;--=> -0.942 +0.337 * i
real(21) <= real(22) + xr25 + xi125;--;--=> 0.195 -0.981 * i
real(22) <= real(23) + xr85 - xi94;--;--=> 0.672 +0.741 * i
real(23) <= real(24) - xr127;--;--=> -1.000 +0.000 * i
real(24) <= real(25) + xr94 + xi85;--;--=> 0.741 -0.672 * i
real(25) <= real(26) - xr25 - xi125;--;--=> -0.195 +0.981 * i
real(26) <= real(27) - xr43 + xi116;--;--=> -0.337 -0.942 * i
real(27) <= real(28) + xr90 - xi90;--;--=> 0.707 +0.707 * i
real(28) <= real(29) - xr115 + xi54;--;--=> -0.904 -0.428 * i
real(29) <= real(30) + xr125 - xi25;--;--=> 0.981 +0.195 * i
real(30) <= real(31) - xr127 + xi6;--;--=> -0.999 -0.049 * i
real(31) <= real(32) + xr127;--;--=> 1.000 -0.000 * i
real(32) <= real(33) - xr127 + xi6;--;--=> -0.999 -0.049 * i
real(33) <= real(34) + xr125 - xi25;--;--=> 0.981 +0.195 * i
real(34) <= real(35) - xr115 + xi54;--;--=> -0.904 -0.428 * i
real(35) <= real(36) + xr90 - xi90;--;--=> 0.707 +0.707 * i
real(36) <= real(37) - xr43 + xi116;--;--=> -0.337 -0.942 * i
real(37) <= real(38) - xr25 - xi125;--;--=> -0.195 +0.981 * i
real(38) <= real(39) + xr94 + xi85;--;--=> 0.741 -0.672 * i
real(39) <= real(40) - xr127;--;--=> -1.000 +0.000 * i
real(40) <= real(41) + xr85 - xi94;--;--=> 0.672 +0.741 * i
real(41) <= real(42) + xr25 + xi125;--;--=> 0.195 -0.981 * i
real(42) <= real(43) - xr116 - xi43;--;--=> -0.942 +0.337 * i
real(43) <= real(44) + xr90 - xi90;--;--=> 0.707 +0.707 * i
real(44) <= real(45) + xr54 + xi115;--;--=> 0.428 -0.904 * i
real(45) <= real(46) - xr125 + xi25;--;--=> -0.981 -0.195 * i
real(46) <= real(47) - xr6 - xi127;--;--=> -0.049 +0.999 * i
real(47) <= real(48) + xr127;--;--=> 1.000 +0.000 * i
real(48) <= real(49) + xr6 + xi127;--;--=> 0.049 -0.999 * i
real(49) <= real(50) - xr125 + xi25;--;--=> -0.981 -0.195 * i
real(50) <= real(51) - xr54 - xi115;--;--=> -0.428 +0.904 * i
real(51) <= real(52) + xr90 - xi90;--;--=> 0.707 +0.707 * i
real(52) <= real(53) + xr116 + xi43;--=> 0.942 -0.337 * i
real(53) <= real(54) + xr25 + xi125;--=> 0.195 -0.981 * i
real(54) <= real(55) - xr85 + xi94;--=> -0.672 -0.741 * i
real(55) <= real(56) - xr127;--=> -1.000 -0.000 * i
real(56) <= real(57) - xr94 - xi85;--=> -0.741 +0.672 * i
real(57) <= real(58) - xr25 - xi125;--=> -0.195 +0.981 * i
real(58) <= real(59) + xr43 - xi116;--=> 0.337 +0.942 * i
real(59) <= real(60) + xr90 - xi90;--=> 0.707 +0.707 * i
real(60) <= real(61) + xr115 - xi54;--=> 0.904 +0.428 * i
real(61) <= real(62) + xr125 - xi25;--=> 0.981 +0.195 * i
real(62) <= real(63) + xr127 - xi6;--=> 0.999 +0.049 * i
real(63) <= real(64) + xr127;--=> 1.000 +0.000 * i
real(64) <= real(65) + xr127 - xi6;--=> 0.999 +0.049 * i
real(65) <= real(66) + xr125 - xi25;--=> 0.981 +0.195 * i
real(66) <= real(67) + xr115 - xi54;--=> 0.904 +0.428 * i
real(67) <= real(68) + xr90 - xi90;--=> 0.707 +0.707 * i
real(68) <= real(69) + xr43 - xi116;--=> 0.337 +0.942 * i
real(69) <= real(70) - xr25 - xi125;--=> -0.195 +0.981 * i
real(70) <= real(71) - xr94 - xi85;--=> -0.741 +0.672 * i
real(71) <= real(72) - xr127;--=> -1.000 -0.000 * i
real(72) <= real(73) - xr85 + xi94;--=> -0.672 -0.741 * i
real(73) <= real(74) + xr25 + xi125;--=> 0.195 -0.981 * i
real(74) <= real(75) + xr116 + xi43;--=> 0.942 -0.337 * i
real(75) <= real(76) + xr90 - xi90;--=> 0.707 +0.707 * i
real(76) <= real(77) - xr54 - xi115;--=> -0.428 +0.904 * i
real(77) <= real(78) - xr125 + xi25;--=> -0.981 -0.195 * i
real(78) <= real(79) + xr6 + xi127;--=> 0.049 -0.999 * i
real(79) <= real(80) + xr127;--=> 1.000 +0.000 * i
real(80) <= real(81) - xr6 - xi127;--=> -0.049 +0.999 * i
real(81) <= real(82) - xr125 + xi25;--=> -0.981 -0.195 * i
real(82) <= real(83) + xr54 + xi115;--=> 0.428 -0.904 * i
real(83) <= real(84) + xr90 - xi90;--=> 0.707 +0.707 * i
real(84) <= real(85)  \ - xr116 - xi43;--\Rightarrow -0.942 +0.337 * i
real(85) <= real(86)  \ + xr25 + xi125;--\Rightarrow 0.195 -0.981 * i
real(86) <= real(87)  \ + xr85 - xi94;--\Rightarrow 0.672 +0.741 * i
real(87) <= real(88)  \ - xr127;--\Rightarrow -1.000 -0.000 * i
real(88) <= real(89)  \ + xr94 + xi85;--\Rightarrow 0.741 -0.672 * i
real(89) <= real(90)  \ - xr25 - xi125;--\Rightarrow -0.195 +0.981 * i
real(90) <= real(91)  \ - xr43 + xi116;--\Rightarrow -0.337 -0.942 * i
real(91) <= real(92)  \ + xr90 - xi90;--\Rightarrow 0.707 +0.707 * i
real(92) <= real(93)  \ - xr115 + xi54;--\Rightarrow -0.904 -0.428 * i
real(93) <= real(94)  \ + xr125 - xi25;--\Rightarrow 0.981 +0.195 * i
real(94) <= real(95)  \ - xr127 + xi6;--\Rightarrow -0.999 -0.049 * i
real(95) <= real(96)  \ + xr127;--\Rightarrow 1.000 +0.000 * i
real(96) <= real(97)  \ - xr127 + xi6;--\Rightarrow -0.999 -0.049 * i
real(97) <= real(98)  \ + xr125 - xi25;--\Rightarrow 0.981 +0.195 * i
real(98) <= real(99)  \ - xr115 + xi54;--\Rightarrow -0.904 -0.428 * i
real(99) <= real(100) \ + xr90 - xi90;--\Rightarrow 0.707 +0.707 * i
real(100) <= real(101) \ - xr43 + xi116;--\Rightarrow -0.337 -0.942 * i
real(101) <= real(102) \ - xr25 - xi125;--\Rightarrow -0.195 +0.981 * i
real(102) <= real(103) \ + xr94 + xi85;--\Rightarrow 0.741 -0.672 * i
real(103) <= real(104) \ - xr127;--\Rightarrow -1.000 -0.000 * i
real(104) <= real(105) \ + xr85 - xi94;--\Rightarrow 0.672 +0.741 * i
real(105) <= real(106) \ + xr25 + xi125;--\Rightarrow 0.195 -0.981 * i
real(106) <= real(107) \ - xr116 - xi43;--\Rightarrow -0.942 +0.337 * i
real(107) <= real(108) \ + xr90 - xi90;--\Rightarrow 0.707 +0.707 * i
real(108) <= real(109) \ + xr54 + xi115;--\Rightarrow 0.428 -0.904 * i
real(109) <= real(110) \ - xr125 + xi25;--\Rightarrow -0.981 -0.195 * i
real(110) <= real(111) \ - xr6 - xi127;--\Rightarrow -0.049 +0.999 * i
real(111) <= real(112) \ + xr127;--\Rightarrow 1.000 +0.000 * i
real(112) <= real(113) \ + xr6 + xi127;--\Rightarrow 0.049 -0.999 * i
real(113) <= real(114) \ - xr125 + xi25;--\Rightarrow -0.981 -0.195 * i
real(114) <= real(115) \ - xr54 - xi115;--\Rightarrow -0.428 +0.904 * i
real(115) <= real(116) \ + xr90 - xi90;--\Rightarrow 0.707 +0.707 * i
real(116) <= real(117) + xr116 + xi43;--=> 0.942 -0.337 * i  
real(117) <= real(118) + xr25 + xi125;--=> 0.195 -0.981 * i  
real(118) <= real(119) - xr85 + xi94;--=> -0.672 -0.741 * i  
real(119) <= real(120) - xr127;--=> -1.000 -0.000 * i  
real(120) <= real(121) - xr94 - xi85;--=> -0.741 +0.672 * i  
real(121) <= real(122) - xr25 - xi125;--=> -0.195 +0.981 * i  
real(122) <= real(123) + xr43 - xi116;--=> 0.337 +0.942 * i  
real(123) <= real(124) + xr90 - xi90;--=> 0.707 +0.707 * i  
real(124) <= real(125) + xr115 - xi54;--=> 0.904 +0.428 * i  
real(125) <= real(126) + xr125 - xi25;--=> 0.981 +0.195 * i  
real(126) <= xr127 - xi6;

-- imaginary part of FIR filter in transposed form = xr*Coeff.i + xi*Coeff.r  
imag(0) <= imag(1) + xr6 + xi127;--=> 0.999 +0.049 * i  
imag(1) <= imag(2) + xr25 + xi125;--=> 0.981 +0.195 * i  
imag(2) <= imag(3) + xr54 + xi115;--=> 0.904 +0.428 * i  
imag(3) <= imag(4) + xr90 + xi90;--=> 0.707 +0.707 * i  
imag(4) <= imag(5) + xr116 + xi43;--=> 0.337 +0.942 * i  
imag(5) <= imag(6) + xr125 - xi25;--=> -0.195 +0.981 * i  
imag(6) <= imag(7) + xr85 - xi94;--=> -0.741 +0.672 * i  
imag(7) <= imag(8) - xi127;--=> -1.000 +0.000 * i  
imag(8) <= imag(9) - xr94 - xi85;--=> -0.672 -0.741 * i  
imag(9) <= imag(10) - xr125 + xi25;--=> 0.195 -0.981 * i  
imag(10) <= imag(11) - xr43 + xi116;--=> 0.942 -0.337 * i  
imag(11) <= imag(12) - xr90 + xi90;--=> 0.707 +0.707 * i  
imag(12) <= imag(13) + xr115 - xi54;--=> -0.428 +0.904 * i  
imag(13) <= imag(14) - xr25 - xi125;--=> -0.981 -0.195 * i  
imag(14) <= imag(15) - xr127 + xi6;--=> 0.049 -0.999 * i  
imag(15) <= imag(16) + xi127;--=> 1.000 +0.000 * i  
imag(16) <= imag(17) + xr127 - xi6;--=> -0.049 +0.999 * i  
imag(17) <= imag(18) - xr25 - xi125;--=> -0.981 -0.195 * i  
imag(18) <= imag(19) - xr115 + xi54;--=> 0.428 -0.904 * i
imag(19) <= imag(20) + xi90 + xi90;--=> 0.707 +0.707 * i
imag(20) <= imag(21) + xi43 - xi116;--=> -0.942 +0.337 * i
imag(21) <= imag(22) - xr125 + xi25;--=> 0.195 -0.981 * i
imag(22) <= imag(23) + xr94 + xi85;--=> 0.672 +0.741 * i
imag(23) <= imag(24) - xi127;--=> -1.000 +0.000 * i
imag(24) <= imag(25) - xr85 + xi94;--=> 0.741 -0.672 * i
imag(25) <= imag(26) + xr125 - xi25;--=> -0.195 +0.981 * i
imag(26) <= imag(27) - xr116 - xi43;--=> -0.337 -0.942 * i
imag(27) <= imag(28) + xr90 + xi90;--=> 0.707 +0.707 * i
imag(28) <= imag(29) - xr54 - xi115;--=> -0.904 -0.428 * i
imag(29) <= imag(30) + xr25 + xi125;--=> 0.981 +0.195 * i
imag(30) <= imag(31) - xr6 - xi127;--=> -0.999 -0.049 * i
imag(31) <= imag(32) + xi127;--=> 1.000 -0.000 * i
imag(32) <= imag(33) - xr6 - xi127;--=> -0.999 -0.049 * i
imag(33) <= imag(34) + xr25 + xi125;--=> 0.981 +0.195 * i
imag(34) <= imag(35) - xr54 - xi115;--=> -0.904 -0.428 * i
imag(35) <= imag(36) + xr90 + xi90;--=> 0.707 +0.707 * i
imag(36) <= imag(37) - xr116 - xi43;--=> -0.337 -0.942 * i
imag(37) <= imag(38) + xr125 - xi25;--=> -0.195 +0.981 * i
imag(38) <= imag(39) - xr85 + xi94;--=> 0.741 -0.672 * i
imag(39) <= imag(40) - xi127;--=> -1.000 +0.000 * i
imag(40) <= imag(41) + xr94 + xi85;--=> 0.672 +0.741 * i
imag(41) <= imag(42) - xr125 + xi25;--=> 0.195 -0.981 * i
imag(42) <= imag(43) + xr43 - xi116;--=> -0.942 +0.337 * i
imag(43) <= imag(44) + xr90 + xi90;--=> 0.707 +0.707 * i
imag(44) <= imag(45) - xr115 + xi54;--=> 0.428 -0.904 * i
imag(45) <= imag(46) - xr25 - xi125;--=> -0.981 -0.195 * i
imag(46) <= imag(47) + xr127 - xi6;--=> -0.049 +0.999 * i
imag(47) <= imag(48) + xi127;--=> 1.000 +0.000 * i
imag(48) <= imag(49) - xr127 + xi6;--=> 0.049 -0.999 * i
imag(49) <= imag(50) - xr25 - xi125;--=> -0.981 -0.195 * i
imag(50) <= imag(51) + xr115 - xi54;--=> -0.428 +0.904 * i
\[ \text{imag}(51) \leq \text{imag}(52) + \text{xr}90 + \text{xi}90; \Rightarrow 0.707 + 0.707 \times i \]
\[ \text{imag}(52) \leq \text{imag}(53) - \text{xr}43 + \text{xi}116; \Rightarrow 0.942 - 0.337 \times i \]
\[ \text{imag}(53) \leq \text{imag}(54) - \text{xr}125 + \text{xi}25; \Rightarrow 0.195 - 0.981 \times i \]
\[ \text{imag}(54) \leq \text{imag}(55) - \text{xr}94 - \text{xi}85; \Rightarrow -0.672 - 0.741 \times i \]
\[ \text{imag}(55) \leq \text{imag}(56) - \text{xi}127; \Rightarrow -1.000 - 0.000 \times i \]
\[ \text{imag}(56) \leq \text{imag}(57) + \text{xr}85 - \text{xi}94; \Rightarrow -0.741 + 0.672 \times i \]
\[ \text{imag}(57) \leq \text{imag}(58) + \text{xr}125 - \text{xi}25; \Rightarrow -0.195 + 0.981 \times i \]
\[ \text{imag}(58) \leq \text{imag}(59) + \text{xr}116 + \text{xi}43; \Rightarrow 0.337 + 0.942 \times i \]
\[ \text{imag}(59) \leq \text{imag}(60) + \text{xr}90 + \text{xi}90; \Rightarrow 0.707 + 0.707 \times i \]
\[ \text{imag}(60) \leq \text{imag}(61) + \text{xr}54 + \text{xi}115; \Rightarrow 0.904 + 0.428 \times i \]
\[ \text{imag}(61) \leq \text{imag}(62) + \text{xr}25 + \text{xi}125; \Rightarrow 0.981 + 0.195 \times i \]
\[ \text{imag}(62) \leq \text{imag}(63) + \text{xr}6 + \text{xi}127; \Rightarrow 0.999 + 0.049 \times i \]
\[ \text{imag}(63) \leq \text{imag}(64) + \text{xi}127; \Rightarrow 1.000 + 0.000 \times i \]
\[ \text{imag}(64) \leq \text{imag}(65) + \text{xr}6 + \text{xi}127; \Rightarrow 0.999 + 0.049 \times i \]
\[ \text{imag}(65) \leq \text{imag}(66) + \text{xr}25 + \text{xi}125; \Rightarrow 0.981 + 0.195 \times i \]
\[ \text{imag}(66) \leq \text{imag}(67) + \text{xr}54 + \text{xi}115; \Rightarrow 0.904 + 0.428 \times i \]
\[ \text{imag}(67) \leq \text{imag}(68) + \text{xr}90 + \text{xi}90; \Rightarrow 0.707 + 0.707 \times i \]
\[ \text{imag}(68) \leq \text{imag}(69) + \text{xr}116 + \text{xi}43; \Rightarrow 0.337 + 0.942 \times i \]
\[ \text{imag}(69) \leq \text{imag}(70) + \text{xr}125 - \text{xi}25; \Rightarrow -0.195 + 0.981 \times i \]
\[ \text{imag}(70) \leq \text{imag}(71) + \text{xr}85 - \text{xi}94; \Rightarrow -0.741 + 0.672 \times i \]
\[ \text{imag}(71) \leq \text{imag}(72) - \text{xi}127; \Rightarrow -1.000 - 0.000 \times i \]
\[ \text{imag}(72) \leq \text{imag}(73) - \text{xr}94 - \text{xi}85; \Rightarrow -0.672 - 0.741 \times i \]
\[ \text{imag}(73) \leq \text{imag}(74) - \text{xr}125 + \text{xi}25; \Rightarrow 0.195 - 0.981 \times i \]
\[ \text{imag}(74) \leq \text{imag}(75) - \text{xr}43 + \text{xi}116; \Rightarrow 0.942 - 0.337 \times i \]
\[ \text{imag}(75) \leq \text{imag}(76) + \text{xr}90 + \text{xi}90; \Rightarrow 0.707 + 0.707 \times i \]
\[ \text{imag}(76) \leq \text{imag}(77) + \text{xr}115 - \text{xi}54; \Rightarrow -0.428 + 0.904 \times i \]
\[ \text{imag}(77) \leq \text{imag}(78) - \text{xr}25 - \text{xi}125; \Rightarrow -0.981 - 0.195 \times i \]
\[ \text{imag}(78) \leq \text{imag}(79) - \text{xr}127 + \text{xi}6; \Rightarrow 0.049 - 0.999 \times i \]
\[ \text{imag}(79) \leq \text{imag}(80) + \text{xi}127; \Rightarrow 1.000 + 0.000 \times i \]
\[ \text{imag}(80) \leq \text{imag}(81) + \text{xr}127 - \text{xi}6; \Rightarrow -0.049 + 0.999 \times i \]
\[ \text{imag}(81) \leq \text{imag}(82) - \text{xr}25 - \text{xi}125; \Rightarrow -0.981 - 0.195 \times i \]
\[ \text{imag}(82) \leq \text{imag}(83) - \text{xr}115 + \text{xi}54; \Rightarrow 0.428 - 0.904 \times i \]
imag(83) <= imag(84) + xr90 + xi90;--=> 0.707 +0.707 * i
imag(84) <= imag(85) + xr43 - xi116;--=> -0.942 +0.337 * i
imag(85) <= imag(86) - xr125 + xi25;--=> 0.195 -0.981 * i
imag(86) <= imag(87) + xr94 + xi85;--=> 0.672 +0.741 * i
imag(87) <= imag(88) - xi127;--=> -1.000 -0.000 * i
imag(88) <= imag(89) - xr85 + xi94;--=> 0.741 -0.672 * i
imag(89) <= imag(90) + xr125 - xi25;--=> -0.195 +0.981 * i
imag(90) <= imag(91) - xr116 - xi43;--=> -0.337 -0.942 * i
imag(91) <= imag(92) + xr90 + xi90;--=> 0.707 +0.707 * i
imag(92) <= imag(93) - xr54 - xi115;--=> -0.904 -0.428 * i
imag(93) <= imag(94) + xr25 + xi125;--=> 0.981 +0.195 * i
imag(94) <= imag(95) - xr6 - xi127;--=> -0.999 -0.049 * i
imag(95) <= imag(96) + xi127;--=> 1.000 +0.000 * i
imag(96) <= imag(97) - xr6 - xi127;--=> -0.999 -0.049 * i
imag(97) <= imag(98) + xr25 + xi125;--=> 0.981 +0.195 * i
imag(98) <= imag(99) - xr54 - xi115;--=> -0.904 -0.428 * i
imag(99) <= imag(100) + xr90 + xi90;--=> 0.707 +0.707 * i
imag(100) <= imag(101) - xr116 - xi43;--=> -0.337 -0.942 * i
imag(101) <= imag(102) + xr125 - xi25;--=> -0.195 +0.981 * i
imag(102) <= imag(103) - xr85 + xi94;--=> 0.741 -0.672 * i
imag(103) <= imag(104) - xi127;--=> -1.000 -0.000 * i
imag(104) <= imag(105) + xr94 + xi85;--=> 0.672 +0.741 * i
imag(105) <= imag(106) - xr125 + xi25;--=> 0.195 -0.981 * i
imag(106) <= imag(107) + xr43 - xi116;--=> -0.942 +0.337 * i
imag(107) <= imag(108) + xr90 + xi90;--=> 0.707 +0.707 * i
imag(108) <= imag(109) - xr115 + xi54;--=> 0.428 -0.904 * i
imag(109) <= imag(110) - xr25 - xi125;--=> -0.981 -0.195 * i
imag(110) <= imag(111) + xr127 - xi6;--=> -0.049 +0.999 * i
imag(111) <= imag(112) + xi127;--=> 1.000 +0.000 * i
imag(112) <= imag(113) - xr127 + xi6;--=> 0.049 -0.999 * i
imag(113) <= imag(114) - xr25 - xi125;--=> -0.981 -0.195 * i
imag(114) <= imag(115) + xr115 - xi54;--=> -0.428 +0.904 * i
imag(115) <= imag(116) + xr90 + xi90;--=> 0.707 +0.707 * i
imag(116) <= imag(117) - xr43 + xi116;--=> 0.942 -0.337 * i
imag(117) <= imag(118) - xr125 + xi25;--=> 0.195 -0.981 * i
imag(118) <= imag(119) - xr94 - xi85;--=> -0.672 -0.741 * i
imag(119) <= imag(120) - xi127;--=> -1.000 -0.000 * i
imag(120) <= imag(121) + xr85 - xi94;--=> -0.741 +0.672 * i
imag(121) <= imag(122) + xr125 - xi25;--=> -0.195 +0.981 * i
imag(122) <= imag(123) + xr116 + xi43;--=> 0.337 +0.942 * i
imag(123) <= imag(124) + xr90 + xi90;--=> 0.707 +0.707 * i
imag(124) <= imag(125) + xr54 + xi115;--=> 0.904 +0.428 * i
imag(125) <= imag(126) + xr25 + xi125;--=> 0.981 +0.195 * i
imag(126) <= xr6 + xi127;
END PROCESS Filter;

-- Use table to get INPUT (=OUTPUT!) twiddle factor coefficients
    tab_1: d64ptb8b
        PORT MAP( table_in => t1_in,
                   re_out => t1_real,
                   im_out => t1_imag);
t1_real_out <= t1_real;
t1_imag_out <= t1_imag;

-- Multiply inputs with twiddle factor exp(j*phi)
    ccmul_1: mult
        PORT MAP (clk => clk, -- Clock for the output register
                   x_in => real_in, y_in => imag_in,
                   c_in => t1_real, s_in => t1_imag,
                   r_out => ccmul_r, i_out => ccmul_i);
    ccmul_r_out <= ccmul_r;
    ccmul_i_out <= ccmul_i;

-- Multiplex for the complex multiplier inputs
-- Use DFT input for the first N steps and than the filter outputs
MPX: PROCESS (x_real_in, x_imag_in, r0, i0)
BEGIN
  IF count <= 65 THEN
    real_in <= x_real_in;
    imag_in <= x_imag_in;
  ELSE
    real_in <= r0;
    imag_in <= i0;
  END IF;
END PROCESS MPX;

-- Sign extentsion for the multiplier output, i.e. filter input
Sxt: PROCESS (ccmul_r, ccmul_i)
BEGIN
  IF count <= 65 THEN
    sxtxr(W-1 DOWNTO 0) <= ccmul_r;
    sxtxi(W-1 DOWNTO 0) <= ccmul_i;
    FOR k IN Wfir-1 DOWNTO W LOOP
      sxtxr(k) <= ccmul_r(ccmul_r'left);
      sxtxi(k) <= ccmul_i(ccmul_i'left);
    END LOOP;
  ELSE
    sxtxr <= (OTHERS => '0');
    sxtxi <= (OTHERS => '0');
  END IF;
END PROCESS Sxt;

sxtxr_out <= sxtxr;
sxtxi_out <= sxtxi;

-- Multiply filter output with twiddle factor exp(j*phi)
r0 <= CONV_STD_LOGIC_VECTOR( real(0)/128 , 8);
i0 <= CONV_STD_LOGIC_VECTOR( imag(0)/128 , 8);

r0_out <= r0;
i0_out <= i0;

-- ccmul_2: cm3a5b16
-- PORT MAP (clk => clk, -- Clock for the output register
-- x_in => r0, y_in => i0,
-- c_in => t2_real, s_in => t2_imag,
-- r_out => y_r, i_out => y_i);

END flex;
II.

--Project Name: dft64pt.npl
--File Name: d64ptb8b.vhd
--Description: Twiddle factors for 64 point multiplier
--Author: Hariharan Natarajan
--Date : 03/30/04

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;

ENTITY d64ptb8b IS
  PORT ( table_in  : IN  STD_LOGIC_VECTOR(6-1 DOWNTO 0);
          re_out    : OUT STD_LOGIC_VECTOR(8-1 DOWNTO 0);
          im_out    : OUT STD_LOGIC_VECTOR(8-1 DOWNTO 0));
END d64ptb8b;

ARCHITECTURE LCs OF d64ptb8b IS
BEGIN

-- This is the 8 bit (i.e. -127 ... 127) CASE table for
-- the 64-point CZT pre- and post-multiplications
-- automatic generated with cztgen.exe -- DO NOT EDIT!

-- Table implements the mapping
PROCESS(table_in)
BEGIN
-- WAIT UNTIL clk = '1'; -- For benchmark only
CASE table_in IS
-- CASE t_in IS -- For benchmark only
  WHEN "000000" => re_out <= "01111111"; -- = 127 i.e. 1.000
          im_out <= "00000000"; -- = 0 i.e. 0.000
  WHEN "000001" => re_out <= "01111111"; -- = 127 i.e. 0.999
          im_out <= "11111010"; -- = -6 i.e. -0.049
  WHEN "000010" => re_out <= "01110011"; -- = 125 i.e. 0.981
          im_out <= "11100111"; -- = -25 i.e. -0.195
  WHEN "000011" => re_out <= "01011010"; -- = 90 i.e. 0.707
          im_out <= "11000011"; -- = -90 i.e. -0.707
  WHEN "000100" => re_out <= "01011010"; -- = 90 i.e. 0.707
          im_out <= "10100110"; -- = -90 i.e. -0.707
  WHEN "000101" => re_out <= "00101011"; -- = 43 i.e. 0.337
          im_out <= "10001000"; -- = -120 i.e. -0.942
  WHEN "000110" => re_out <= "11100111"; -- = -25 i.e. -0.195
          im_out <= "10000011"; -- = -125 i.e. -0.981
  WHEN "000111" => re_out <= "10100010"; -- = -94 i.e. -0.741
END CASE;
END PROCESS;
END LCs;
im_out <= "10101011";  -- = -85 i.e. -0.672
WHEN "001000" => re_out <= "10000001";  -- = -127 i.e. -1.000
im_out <= "00000000";  -- = 0 i.e. 0.000
WHEN "001001" => re_out <= "10101011";  -- = -85 i.e. -0.672
im_out <= "01011110";  -- = 94 i.e. 0.741
WHEN "001010" => re_out <= "00011001";  -- = 25 i.e. 0.195
im_out <= "01111101";  -- = 125 i.e. 0.981
WHEN "001011" => re_out <= "01111000";  -- = 120 i.e. 0.942
im_out <= "00101011";  -- = 43 i.e. 0.337
WHEN "001100" => re_out <= "01011010";  -- = 90 i.e. 0.707
im_out <= "10100110";  -- = -90 i.e. -0.707
WHEN "001101" => re_out <= "11001010";  -- = -54 i.e. -0.428
im_out <= "10001101";  -- = -115 i.e. -0.904
WHEN "001110" => re_out <= "10000011";  -- = -125 i.e. -0.981
im_out <= "00011001";  -- = 25 i.e. 0.195
WHEN "001111" => re_out <= "00000110";  -- = 6 i.e. 0.049
im_out <= "01111111";  -- = 127 i.e. 0.999
WHEN "010000" => re_out <= "01111111";  -- = 127 i.e. 1.000
im_out <= "00000000";  -- = 0 i.e. 0.000
WHEN "010001" => re_out <= "11110101";  -- = -6 i.e. -0.049
im_out <= "10000001";  -- = -127 i.e. -0.999
WHEN "010010" => re_out <= "10000011";  -- = -125 i.e. -0.981
im_out <= "00011001";  -- = 25 i.e. 0.195
WHEN "010011" => re_out <= "01101010";  -- = 90 i.e. 0.707
im_out <= "10100110";  -- = -90 i.e. -0.707
WHEN "010100" => re_out <= "10001000";  -- = -120 i.e. -0.942
im_out <= "11010101";  -- = -43 i.e. -0.337
WHEN "010101" => re_out <= "11100111";  -- = -25 i.e. -0.195
im_out <= "01100001";  -- = 125 i.e. 0.981
WHEN "010110" => re_out <= "10000001";  -- = 127 i.e. 1.000
im_out <= "01100000";  -- = 0 i.e. 0.000
WHEN "010111" => re_out <= "00110110";  -- = 25 i.e. 0.195
im_out <= "01111001";  -- = 120 i.e. 0.942
WHEN "011000" => re_out <= "01101010";  -- = 90 i.e. 0.707
im_out <= "10100110";  -- = -90 i.e. -0.707
WHEN "011001" => re_out <= "11100010";  -- = -43 i.e. -0.337
im_out <= "01110000";  -- = 125 i.e. 0.981
WHEN "011010" => re_out <= "11000111";  -- = -25 i.e. -0.195
im_out <= "10000001";  -- = -125 i.e. -0.981
WHEN "011011" => re_out <= "11010101";  -- = -43 i.e. -0.337
im_out <= "01111000";  -- = 120 i.e. 0.942
WHEN "011100" => re_out <= "01011001";  -- = 90 i.e. 0.707
im_out <= "10100110";  -- = -90 i.e. -0.707
WHEN "011101" => re_out <= "10001101";  -- = -115 i.e. -0.904
im_out <= "10001000";  -- = 54 i.e. 0.428
im_out <= "11010101";  -- = -43 i.e. -0.337
WHEN "011111" => re_out <= "10000001";  -- = -127 i.e. -0.999
im_out <= "00000110"; -- = 6 i.e. 0.049
WHEN "100000" => re_out <= "01111111"; -- = 127 i.e. 1.000
im_out <= "00000000"; -- = 0 i.e. 0.000
WHEN "100001" => re_out <= "10000001"; -- = -127 i.e. -0.999
im_out <= "00000001"; -- = 6 i.e. 0.049
WHEN "100010" => re_out <= "01111101"; -- = 125 i.e. 0.981
im_out <= "11001111"; -- = -25 i.e. -0.195
WHEN "100011" => re_out <= "10001101"; -- = -115 i.e. -0.904
im_out <= "00110110"; -- = 54 i.e. 0.428
WHEN "100100" => re_out <= "01011010"; -- = 90 i.e. 0.707
im_out <= "10100110"; -- = -90 i.e. -0.707
WHEN "100101" => re_out <= "11010101"; -- = -43 i.e. -0.337
im_out <= "01111000"; -- = 120 i.e. 0.942
WHEN "100110" => re_out <= "11100111"; -- = -25 i.e. -0.195
im_out <= "10000001"; -- = -125 i.e. -0.981
WHEN "100111" => re_out <= "01011110"; -- = 94 i.e. 0.741
im_out <= "01010101"; -- = 85 i.e. 0.672
WHEN "101000" => re_out <= "10000011"; -- = -127 i.e. -1.000
im_out <= "00000000"; -- = 0 i.e. -0.000
WHEN "101001" => re_out <= "01010101"; -- = 85 i.e. 0.672
im_out <= "10100010"; -- = -94 i.e. -0.741
WHEN "101010" => re_out <= "00011101"; -- = 25 i.e. 0.195
im_out <= "01111101"; -- = 125 i.e. 0.981
WHEN "101011" => re_out <= "10001000"; -- = -120 i.e. -0.942
im_out <= "11010101"; -- = -43 i.e. -0.337
WHEN "101100" => re_out <= "01011110"; -- = 90 i.e. 0.707
im_out <= "10100110"; -- = -90 i.e. -0.707
WHEN "101101" => re_out <= "00110110"; -- = 54 i.e. 0.428
im_out <= "01110011"; -- = 115 i.e. 0.904
WHEN "101110" => re_out <= "10000011"; -- = -125 i.e. -0.981
im_out <= "00011101"; -- = 25 i.e. 0.195
WHEN "101111" => re_out <= "11111101"; -- = -6 i.e. -0.049
im_out <= "10000001"; -- = -127 i.e. -0.999
WHEN "110000" => re_out <= "01111111"; -- = 127 i.e. 1.000
im_out <= "00000000"; -- = 0 i.e. -0.000
WHEN "110001" => re_out <= "00000110"; -- = 6 i.e. 0.049
im_out <= "01111111"; -- = 127 i.e. 0.999
WHEN "110010" => re_out <= "10000011"; -- = -125 i.e. -0.981
im_out <= "00011101"; -- = 25 i.e. 0.195
WHEN "110011" => re_out <= "11001010"; -- = -54 i.e. -0.428
im_out <= "10001101"; -- = -115 i.e. -0.904
WHEN "110100" => re_out <= "01011010"; -- = 90 i.e. 0.707
im_out <= "10100110"; -- = -90 i.e. -0.707
WHEN "110101" => re_out <= "01111000"; -- = 120 i.e. 0.942
im_out <= "00101011"; -- = 43 i.e. 0.337
WHEN "110110" => re_out <= "00011001"; -- = 25 i.e. 0.195
im_out <= "01111101"; -- = 125 i.e. 0.981
WHEN "110111" => re_out <= "10101011"; -- = -85 i.e. -0.672
im_out <= "01011111"; -- = 94 i.e. 0.741
WHEN "111000" => re_out <= "100000001"; -- = -127 i.e. -1.000
  im_out <= "00000000"; -- = 0 i.e. 0.000
WHEN "111001" => re_out <= "10100001"; -- = -94 i.e. -0.741
  im_out <= "10101011"; -- = -85 i.e. -0.672
WHEN "111010" => re_out <= "11100111"; -- = -25 i.e. -0.195
  im_out <= "10000011"; -- = -125 i.e. -0.981
WHEN "111011" => re_out <= "00101011"; -- = 43 i.e. 0.337
  im_out <= "10001000"; -- = -120 i.e. -0.942
WHEN "111100" => re_out <= "01011010"; -- = 90 i.e. 0.707
  im_out <= "10100110"; -- = -90 i.e. -0.707
WHEN "111101" => re_out <= "01110011"; -- = 115 i.e. 0.904
  im_out <= "11001010"; -- = -54 i.e. -0.428
WHEN "111110" => re_out <= "01111101"; -- = 125 i.e. 0.981
  im_out <= "11100111"; -- = -25 i.e. -0.195
WHEN "111111" => re_out <= "01111111"; -- = 127 i.e. 0.999
  im_out <= "11111010"; -- = -6 i.e. -0.049
WHEN OTHERS => re_out <= (OTHERS => '0');
  im_out <= (OTHERS => '0');
END CASE;
-- t_in := table_in; -- For benchmark only
END PROCESS;
END LCs;
REFERENCES


[10] Xilinx Corporation, “Xilinx ISE 5.2”


BIOGRAPHICAL SKETCH

Hariharan Natarajan was born on 11\textsuperscript{th} February 1980, in Chennai, India. After finishing high school in Hyderabad, India. He graduated from Madras University with BS degree in Instrumentation and Control engineering. He started his Masters of Science programme at Florida State University in fall 2001 and graduated in Summer 2004. His area of specialization is digital electronics and ASIC design.