Group 7

Goertzel Filter

With VHDL Implementation

Microelectronics & HW/SW-Co-Design

Ali Beiti Aydenlou, 7216783, Ali.beitiaydenlou003@stud.fh-dortmund.de

Roghieh Farajialamooti, 7216442, roghieh.farajialamooti001@stud.fh-dortmund.de

Ghazaleh Hadian Ghahfarokhi, 7216547, ghazaleh.hadianghahfarokhi001@stud.fh-dortmund.de

Seda Sensoy, 7216627, seda.sensoy003@stud.fh-dortmund.de

Prof. Dr. Peter Schulz

Summer 2024 Homework

Table of Contents

Introduction

The Goertzel filter is an efficient algorithm for detecting a single frequency component in a signal. The Goertzel algorithm is an efficient method for detecting a single frequency component within a signal. Unlike the Fast Fourier Transform (FFT), which analyzes a broad frequency spectrum, the Goertzel algorithm focuses on a specific frequency, making it computationally efficient for this purpose. The algorithm uses a second-order difference equation to process the input signal and determine the presence of the target frequency. This report presents the design, implementation, and verification of a Goertzel filter using synthesizable VHDL code. The filter is designed to detect a 150 kHz signal within a 4 MHz sample frequency. The process involves designing the filter, implementing it in VHDL, and verifying its performance through extensive testing using various waveforms and frequencies.

The main objective of this report is to verify the Goertzel filter design using a comprehensive testbench. The filter will be tested with various waveforms (sine, square, and triangle) at different frequencies and phases to ensure its accuracy and robustness. The verification process involves generating test stimuli using Octave/MATLAB, applying these stimuli to the Goertzel filter, and comparing the filter's output to expected results. The steps include:

- Generating 12-bit offset binary input data.
- Calculating expected results for each test case.
- Storing input data and expected results in files for use in the VHDL testbench.

The testbench is composed of the following main components:

- Generates test stimuli based on the specified waveforms and frequencies.
- Calculates the expected Goertzel filter output for each test stimulus.
- Handles input and output data storage and retrieval.
- The VHDL implementation of the Goertzel filter.

Goertzel Algorithm

The Goertzel Algorithm is a digital signal processing (DSP) technique utilized for detecting specific frequency components within a signal. It is particularly useful for the efficient computation of individual discrete Fourier transform (DFT) bins. Unlike the Fast Fourier Transform (FFT), which computes the entire frequency spectrum, the Goertzel algorithm is optimized for identifying a small number of frequencies, making it ideal for applications where only a few frequency components are of interest.

The Goertzel algorithm was introduced by Gerald Goertzel. He first presented the algorithm in 1958 in a paper titled "an Algorithm for the Evaluation of Finite Trigonometric Series" published in the American Mathematical Monthly [1].

Key Benefits of Goertzel Algorithm

1- Efficiency in Frequency Detection

The Goertzel Algorithm is designed to detect specific frequencies. This makes it particularly efficient for applications where only a few frequencies are of interest. This is especially important in resource-constrained environments, such as embedded systems, where analyzing a small number of frequencies is crucial. The Goertzel Algorithm is highly beneficial in these scenarios as it can be more computationally efficient than the FFT. This efficiency is due to its reduced computational complexity and lower memory requirements, making it an ideal choice for systems with limited processing power and memory. By concentrating on specific frequencies and minimizing resource usage, the Goertzel Algorithm provides a streamlined and effective solution for frequency detection in various practical applications. The Goerzel algorithm is mainly used as a tone detector in dual tone multi-frequency systems (DTMF). However, its character allows to use the algorithm as a digital filter [2].

2- Simplicity and Ease of Implementation

The Goertzel algorithm is relatively simple to understand and implement. It often requires fewer lines of code compared to FFT and involves simple recursive calculations, which makes it accessible for engineers and developers to deploy in various signal processing tasks. This simplicity makes it easy to understand, code, and debug.

3- Real-time Processing Capabilities

The algorithm has low latency because it does not need to wait for a complete block of data before processing begins. It can process data in real time due to its incremental processing nature with minimal delay. This is crucial for applications like speech prosessing, telecommunications, and other signal processing tasks where timely detection and processing of specific frequencies are essential.

4- Versatility in Applications

The Goertzel algorithm shines with its versatility, finding applications in a wide array of fields. Whether it’s ensuring the smooth operation of TV remotes, smart home devices, factory machines, hearing aids, music and sound processing, or car monitoring systems, this algorithm excels by

quickly detecting the right signals while using minimal power. Its ability to work efficiently across such diverse applications underscores its importance in modern technology, providing reliable performance in both everyday gadgets and specialized equipment. The Goertzel algorithm's efficiency and low power consumption make it an essential tool for optimizing the functionality of a wide range of devices, ensuring they operate effectively and responsively.

5- Flexibility in Applications

The Goertzel algorithm is known for its flexibility in different programming environments and hardware setups. This means it can work well with a wide range of devices, from simple microcontrollers to complex systems. Whether used in low-power embedded systems or high-performance computing platforms, it remains efficient and effective. Additionally, users can easily customize the Goertzel algorithm to meet specific needs, optimizing it for particular tasks and frequencies. This ability to tailor the algorithm makes it very useful in many fields, such as telecommunications, audio processing, and biomedical signal analysis.

Goertzel Algorithm Mathematical Foundation

The Goertzel Algorithm is mathematically grounded in the properties of the DFT. The algorithm computes the DFT for a specific frequency index, k, using a second-order IIR (Infinite Impulse Response) filter. This results in efficient calculation, especially when the number of required frequency components is much smaller than the total number of points in the signal.

Before applying the Goertzel algorithm, some preliminary calculations are necessary. These parameters are essential inputs for setting up and using the Goertzel Filter effectively. They define the characteristics of the signal we're analyzing and how finely we're sampling it in time. First, the number of samples in the data set must be chosen. The number of samples (N) indicates how many data points we must work with. The number of samples in our dataset is 135. We will also need sampling frequency(fs) to determine how often the signal is measured, essentially setting the rate at which data points are collected. In our setup, the sampling frequency is 4 MHz. This high sampling rate allows for accurate representation and analysis of high-frequency components within the signal, ensuring that even rapid changes are captured with precision. The desired frequency (f0) is the specific frequency component within a signal that we aim to detect and analyze. In our case, the signal frequency of interest is 150 kHz. This means that our analysis will focus on identifying and extracting information related to this frequency, ensuring that any relevant characteristics or variations at 150 kHz are accurately captured and studied. Detecting this frequency is crucial for the effectiveness and accuracy of our signal processing tasks.

Algorithm Steps

1- Initialization

The constants α and β are calculated as:

$$\alpha = \frac{2\pi k}{N}$$

$$\beta = \frac{2\pi k(N-1)}{N}$$

These constants represent the angular frequencies for the desired DFT coefficient.

2- Coefficient Calculation

Using β, the trigonometric coefficients are computed:

$$a = \cos(\beta)$$

$$b = -\sin(\beta)$$

The auxiliary coefficients cband d are derived from α:

$$c = \sin(\alpha)\sin(\beta) - \cos(\alpha)\cos(\beta)$$

$$d = \sin(2\pi k)$$

3- Initialization of State Variables:

The initial values for the state variables are set to zero:

$$\omega_0 = \omega_1 = \omega_2 = 0$$

4- Iterative Section:

For each sample in the input signal x[n], the state variables are updated as follows:

$$\omega_0 = x_{in} + 2\cos(\alpha)\omega_1 - \omega_2$$

$$\omega_2 = \omega_1$$

$$\omega_1 = \omega_0$$

This loop runs for all samples in the window, effectively filtering the input signal.

5- Non-Iterative Section:

After processing all samples, the desired DFT coefficient X_k is calculated using the final state variables:

$$X_k = a\omega_1 + c\omega_2 + j(b\omega_1 + d\omega_2)$$

The magnitude and phase of X_k are then computed:

$$|X_k| = \sqrt{(a\omega_1 + c\omega_2)^2 + (b\omega_1 + d\omega_2)^2}$$

$$\Phi(X_k) = \arctan\left(\frac{b\omega_1 + d\omega_2}{a\omega_1 + c\omega_2}\right)$$

Signal Flow Diagram: The signal flow diagram provides a visual representation of the Goertzel Algorithm. State variables (w_0 w_1 w_2) store the state of the filter at each step. They represent the internal memory of the system. It shows the signal path from the input x[n] through the iterative filter stages to the final output $y_k[n]$. The diagram highlights the recursive nature of the algorithm, with feedback loops represented by delay elements z^{-1}.

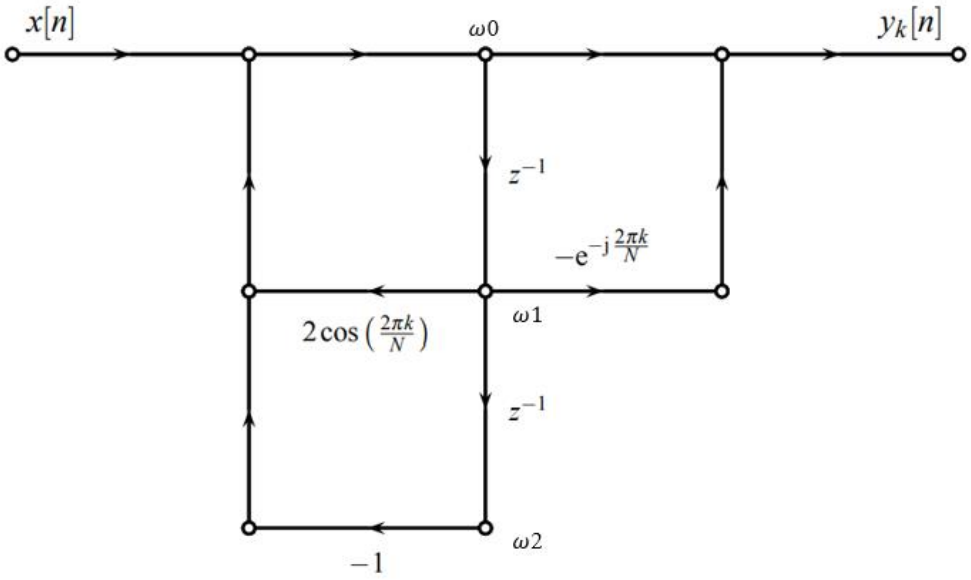

Figure 1 Signal flow graph of second order Goertzel system with indicated state variables [5]

The Goertzel Algorithm offers a focused, efficient alternative to the FFT for applications where only a few frequency components need to be analyzed. By leveraging its iterative nature and reduced computational requirements, the algorithm is well-suited for real-time signal processing tasks, particularly in embedded systems and telecommunication applications. Its implementation, as outlined in the provided steps, highlights the algorithm's simplicity and power in frequency-specific analysis.

Goertzel ASIC and FPGA implementations

Implementations of the Goertzel algorithm can be found on both ASIC (Application-Specific Integrated Circuit) and FPGA (Field-Programmable Gate Array) platforms. The fundamental difference between ASICs and FPGAs lies in their circuitry: ASICs have circuits permanently etched into the silicon, while FPGAs consist of configurable logic blocks that can be programmed. This distinction results in ASICs typically offering superior performance in terms of area, power consumption, and delay. Conversely, FPGAs provide greater flexibility and faster implementation times. Examining the comparative performance metrics of designs implemented on both ASIC and FPGA is a valuable area of study.

An FPGA Implementation of Goertzel Algorithm

The Goertzel Algorithm is a digital signal processing technique used to detect a specific frequency component within a signal. Unlike the Fast Fourier Transform (FFT), the Goertzel Algorithm is more efficient for calculating individual frequency components, making it particularly suitable for applications like DTMF (Dual-Tone Multi-Frequency) signal detection, spectrum analysis, and other areas where real-time frequency detection is crucial. Implementing the Goertzel Algorithm on an FPGA provides an efficient and flexible solution for real-time frequency detection applications. The reconfigurable nature of FPGAs allows for optimizing the algorithm for specific use cases, ensuring efficient resource utilization and high performance.

Field Programmable Gate Arrays (FPGAs) are integrated circuits that can be configured by the user after manufacturing. They offer a flexible and efficient platform for implementing digital signal processing algorithms like the Goertzel Algorithm due to their parallel processing capabilities and reconfigurable nature.

Implementing the Goertzel Algorithm on an FPGA involves the following steps:

- The input signal x[n] is sampled and fed into the FPGA. This can be achieved through an Analog-to-Digital Converter (ADC) interfaced with the FPGA.
- Initialize the recurrence variables s[−1] and s[−2] to zero.
- Implement the recurrence relation using FPGA logic elements. The cosine term can be precomputed and stored in a register or ROM.
    - $2\cos(2\pi k/N)$
- After processing N samples, compute the final value Y[k] using the recurrence values s[N−1] and s[N−2].
- The computed frequency component Y[k] is output from the FPGA, which can be interfaced with other digital systems for further processing or display.

FPGA Goertzel Design Considerations

- Efficient use of FPGA resources such as lookup tables (LUTs), digital signal processing (DSP) slices, and memory blocks is crucial for implementing the Goertzel algorithm.
- Using fixed-point arithmetic instead of floating-point can significantly reduce resource usage and increase processing speed on an FPGA.
- Leveraging the parallel processing capabilities of FPGAs can enhance the performance of the Goertzel Algorithm, enabling real-time signal processing. [3]

ASIC Implementation of Goertzel Filter-Based System

The ASIC implementation of the Goertzel filter-based system has significant implications for the field of EIS (Electrical Impedance Spectroscopy) and related applications. By enabling fast and simultaneous multi-frequency analysis, the proposed system can improve the performance of EIS-based diagnostics and monitoring systems. Potential applications include:

Biomedical Diagnostics: Fast and accurate EIS can enhance the capabilities of diagnostic devices used for monitoring biological tissues and detecting diseases.

Material Characterization: The system can be used to quickly characterize materials and detect changes in their properties, which is valuable in industrial and research settings.

Portable Devices: The low power consumption and compact size of the ASIC make it ideal for integration into portable EIS devices, expanding the range of applications where EIS can be effectively utilized.

The implementation of the Goertzel Algorithm on an ASIC for multi-frequency EIS involves several key design considerations and innovations:

- **Architecture Design:** The ASIC architecture is optimized to perform simultaneous multi-frequency detection using the Goertzel Algorithm. This involves parallel processing paths, each dedicated to a specific frequency component. Such an approach ensures that the system can handle multiple frequencies simultaneously without significant delays.
- **Efficiency and Speed**: By leveraging the hardware capabilities of the ASIC, the system achieves high-speed processing, which is crucial for real-time applications. The hardware implementation allows for the direct and efficient execution of the Goertzel Algorithm, reducing the need for extensive computational resources typically required by software implementations.
- **Resource Utilization:** The ASIC design focuses on optimal resource utilization, ensuring that the system remains power efficient. This is particularly important for portable and battery-operated EIS devices. The design minimizes the silicon area and power consumption while maintaining high performance.
- **Accuracy and Precision:** The Goertzel-based ASIC system is designed to maintain high accuracy and precision in frequency detection. This is critical for EIS applications, where precise impedance measurements are necessary for accurate diagnostics and analysis.
- **Integration and Scalability:** The ASIC design allows for easy integration into existing EIS systems and can be scaled to accommodate additional frequencies if needed. This flexibility makes the system adaptable to various application requirements. [4]

Applications and Benefits of ASIC Implementation

The ASIC implementation of the Goertzel Algorithm for EIS offers several benefits, including:

- **Real-time Processing**: The ability to perform fast, real-time frequency analysis makes the system suitable for dynamic applications where quick response times are essential.
- **Compact and Efficient Design**: The optimized ASIC design results in a compact and efficient solution, ideal for portable devices and applications with stringent power and space constraints.
- **Enhanced Performance**: The dedicated hardware implementation ensures enhanced performance compared to software-based approaches, providing reliable and accurate frequency detection.

Design and Implementation

Based on the mathematical foundations explained in the previous section, design and implement of the Goertzel filter as an entity in VHDL language is explained in this section. The filter as shown in figure 2 has twelve ports for signals, a clock port and a reset port as inputs and 20 output ports.

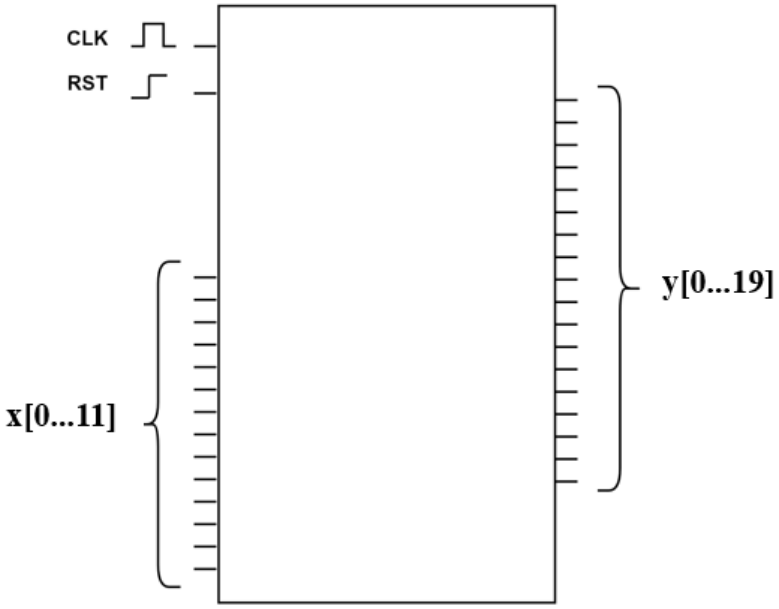

Figure 2 Goertzel Filter

To implement the Goerzel filter, first, the necessary libraries, such as IEEE and IEEE.std_logic_1164, are imported. These libraries are essential for describing digital logic values in VHDL. They facilitate the definition of single-bit and logical vector arrays. Following this, the `goertzel` entity is defined. This entity includes a `clk` port, which handles operations on both the rising and falling edges. Additionally, there is a `rst` port that resets all signals to the value 0 on a rising edge. The entity also features a `x` port, which is a 12-bit unsigned input port used for receiving input signals, and a `y` port, which is a 20-bit signed output port intended for the output signal. This is shown in VHDL code in figure 3.

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


entity goertzel is
port (
    clk: in std_logic; -- operations are performed on Rising and falling edge.
    rst: in std_logic; -- rising edge signal, all internal and external signals are set to 0.
    x : in unsigned(11 downto 0); -- input values have 5v DC bias and multiplied by 100 to remove the negatvie samples
    y : out signed(19 downto 0);
);
end entity;
```

Figure 3 Implemetation of Goerzel Entity

In figure 4, the architecture of the behavior of the filter is illustrated. Initially, the internal constant signals, whose values have been determined based on the mathematical foundation from the previous section, are introduced. These constants, denoted as a_D, b_D, c_D, d_D, and twocosalpha, are 20-bit internal constant signed signals with predefined values. After that, internal signals for intermediate values, labeled w0_D, w1_D, and w2_D, are defined. These signals are also 20-bit signed signals, initialized to zero. Additionally, the signals y_real_D and y_imag_D are designated as 20-bit signed signals to represent the real and imaginary components of the output value, respectively. counter_D is defined for the iterative part of the algorithm. Like the other signals, counter_D is a 20-bit signed signal.

```
architecture behav of goertzel is

  -- internal constant signals that are 20 bit signed
  signal a_D : signed(19 downto 0) := to_signed(973, 20);           -- calculated to be 0.9730
  signal b_D : signed(19 downto 0) := to_signed(309, 20);           -- calculated to be 0.309017
  signal c_D : signed(19 downto 0) := to_signed(-1000, 20);         -- calculated to be -1.0
  signal d_D : signed(19 downto 0) := to_signed(0, 20);             --calculated to be -1.22465e-15
  signal twocosalpha_D : signed(19 downto 0) := to_signed(1946, 20);--calculated to be 1.946

  -- internal signals for intermediate values that are 20 bit signed
  signal w0_D : signed(19 downto 0):= to_signed(0, 20);
  signal w1_D : signed(19 downto 0):= to_signed(0, 20);
  signal w2_D : signed(19 downto 0):= to_signed(0, 20);
  signal y_real_D: signed(19 downto 0); -- real component of the output
  signal y_imag_D: signed(19 downto 0); -- imaginary component of the output

  -- internal signal for counter that is 20 bit signed
  signal counter_D : signed(19 downto 0):= to_signed(0, 20);
```

Figure 4 Architecture Behavior of Goertzel Filter

In figure 5, the iterative process of the behavior is displayed. At the beginning, the output value is calculated from the y_real and y_imag signals. Following this, a process based on the clk and rst ports is defined. Initially, the rst port is checked. If the reset port is active, all internal and output values are set to zero. Next, the clk port is checked. On the rising edge of the clk, the counter is examined to determine if it has reached the end of the N window. During this process, the w0_D signal is calculated using $\omega 0 = x\ in + 2\cos(a\omega 1) - \omega 2)$. In this formula, ω0 is the current value of the intermediate variable in the iterative process. x in represents the current input signal value. Cos(a) is the cosine of the angle a, which is related to the frequency being detected by the Goertzel algorithm. $\omega 1$ is the previous value of the intermediate variable. And $\omega 2$ is the value of the intermediate variable from two steps back. At the start, $\omega 1$ and $\omega 2$ are typically initialized to zero. For each new input sample x in, $\omega 0$ is calculated using the current input x in, the previous value $\omega 1$ and the value from two steps back $\omega 2$. 2 cos(a) is a precomputed constant that depends on the frequency of interest. In each iteration, the value of w1_D is updated to the value of w0_D, and the value of w2_D is updated to the value of w1_D. At the end of the iteration window, the real and imaginary components of the output are calculated based on $Xk = a\omega 1 + c\omega 2 + j(b\omega 1 + d\omega 2)$. In this formula Xk is the resulting complex number. a, b, c, and d are constants. $\omega 1$ and $\omega 2$ are intermediate values and j represents the imaginary unit. This formula combines both the real and imaginary components to compute Xk.

```
    begin
            y <= resize(resize((y_real_D*y_real_D),y'length) +  resize( (y_imag_D*y_imag_D),y'length), y'length);

    process(clk,rst)
        begin
        if rst'event and rst='1' then --- resetting all signals to 0
            counter_D <= to_signed(0, counter_D'length);
            w0_D <= to_signed(0, w0_D'length);
            w1_D <= to_signed(0, w1_D'length);
            w2_D <= to_signed(0, w2_D'length);
            y_real_D <= to_signed(0, y_real_D'length);
            y_imag_D <= to_signed(0, y_imag_D'length);

        elsif clk'event and clk='1' then --on rising clock edge
            if counter_D < to_signed(134,counter_D'length) then --if counter is less than 134 then continue with the goertzel algorithm
                w0_D <= resize(signed("0000"&x) - to_signed(500,w0_D'length) + resize((twocosalpha_D * w1_D)/1000,w0_D'length)-
w2_D,w0_D'length);
                counter_D <= counter_D + to_signed(1, 20);
            else -- if counter is > 134 then calculate the real and imaginary components of the output and reset the counter.
                y_real_D <= resize(resize( resize( a_D*w1_D/1000 ,y_real_D'length) + resize( c_D*w2_D/1000 ,y_real_D'length)
,y_real_D'length)/to_signed(135,y_real_D'length),y_real_D'length);
                y_imag_D <= resize(resize( resize( b_D*w1_D/1000 ,y_imag_D'length) + resize( d_D*w2_D ,y_imag_D'length)
,y_imag_D'length)/to_signed(135,y_imag_D'length),y_imag_D'length);
                counter_D <= to_signed(0, 20);

            end if;



        end if;

            w2_D <= w1_D;
            w1_D <= w0_D;
        end process;

    end architecture;
```

Figure 5 Iterative Process of Goertzel Behavior

Verification and Testbench

The verification process involves creating a VHDL testbench that applies various input stimuli to the Goertzel filter and compares the output to expected results. Results from these test cases are analyzed to verify the accuracy and robustness of the Goertzel filter. The filter demonstrated robust performance across various waveforms and frequencies, accurately detecting the target frequency in each test case.

Sine Waves

Test cases for sine waves include frequencies of 150 kHz, 149 kHz, 151 kHz, 5 kHz, and 200 kHz with phase angles of 0°, 30°, 45°, 90°, and 120°.

In MATLAB the steps for generating the stimulus data are as follows:

- The code generates sine wave data at specified frequencies and phases.
- The generated data is scaled and rounded to integer values.
- Instead of using the string function, the code uses num2str and strcat to construct the filename.
- The dlmwrite function writes the data to text files. file is named according to the waveform type, frequency, and phase for easy identification and retrieval.

Figure 6 shows the MATLAB code for sine waveform signals generation.

```
octave:1> sine_freq = [150e3 149e3 151e3 5e3 200e3]; % Different frequencies for sine waves
phases = [0 30 45 90 120]; % Different phases in degrees
t = 0:1/4e6:(135-1)/4e6; % Time vector for 135 samples at 4 MHz sample rate

for i = 1:length(sine_freq)
    for j = 1:length(phases)
        data(:,1) = round((sin(2*pi*sine_freq(i) * t + phases(j) * pi/180) + 5) * 100);

        % Constructing filename without using 'string' function
        freq_str = num2str(sine_freq(i)/1000); % Convert frequency to kHz and string
        phase_str = num2str(phases(j)); % Convert phase to string
        filename = strcat('sine_', freq_str, 'KHz_', phase_str, 'deg.txt');

        dlmwrite(filename, data(:,1), ' ');
    end
end
```

Figure 6 Sine Waveform Signals generation in MATLAB

Figure 7 represents the various sine waveforms at different amplitudes and scales that are used to test the Goertzel filter design.

$$x11 = \sin(2\pi t * 5000)$$

$$x21 = 5 + \sin(2\pi t * 5000)$$

$$x31 = (5 + \sin(2\pi t * 5000)) * 100$$

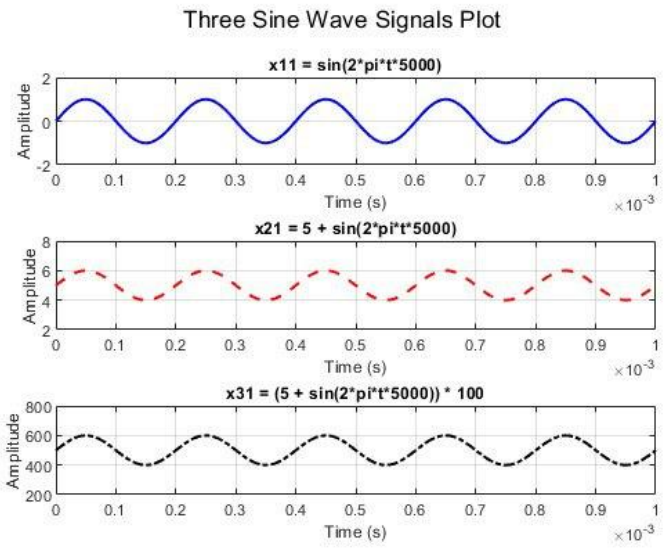

Figure 7 Sine Waveforms used for Test

After generating and saving the signals, they are saved and read in the testbench code. Then, it is applied to the Goertzel filter design.

The VHDL Code below describes the declaration and initialization of signals.

rst: A reset signal of type std_logic.
- clk: A clock signal of type std_logic.
- x: An input signal of type unsigned(11 downto 0).
- y: An output signal of type signed(19 downto 0).
- fileName: A string variable used to store the name of the input file.

Figure 8 shows the signals declaration and initialization in VHDL.

```
signal rst : std_logic := '0';
signal clk : std_logic := '0';
signal x : unsigned(11 downto 0);
signal y : signed(19 downto 0);
signal fileName : string(1 to 22) := to_string("sine_149kHZ_120deg.txt");
```

Figure 8 VHDL Code for signal declaration and initialization

The DUT is instantiated using a component named Goertzel, with ports for reset, clock, input, and output signals. It's shown in figure 9.

```
dut: goertzel port map (
    rst => rst,
    clk => clk,
    x => x,
    y => y
);
```

Figure 9 DUT declaration

The VHDL Code below in figure 10 continuously reads lines from the file, converts the data to the appropriate format, and applies it to the DUT. It reads each line from the file, converts the data to unsigned, and assigns it to x. Clock (clk) is toggled ('0' and '1') to simulate the operation of the DUT. The process counts the number of data points processed (count) and reports the output after processing 135 samples. The report includes the filename, the power of the output (to_integer(y)), and the amplitude calculated as the square root of the integer value of y.

```
process
    variable testfile : text;
    variable linestr : line;
    variable readval : integer;
    variable count : natural := 0;
begin
    file_open(testfile, fileName, read_mode);
    while not endfile(testfile) loop
        readline(testfile, linestr);
        read(linestr, readval);
        x <= unsigned(to_unsigned(readval, x'length));
        clk <= '0';
        wait for 1 ns;
        clk <= '1';
        wait for 1 ns;
        count := count + 1;
        if count = 135 then
            report fileName & " -> power: " & to_string(to_integer(y)) &
                   " Amplitude: " & to_string(sqrt(real(to_integer(y))), "ff") & ".";
            exit;
        end if;
    end loop;
    file_close(testfile);
end process;
```

Figure 10 VHDL Code to read signals and apply to DUT

Square Waves

Test cases for square waves include frequencies of 150 kHz, 16 kHz, 10 kHz, and 200 kHz with phase angles of 0°, 30°, 45°, 90°, and 120°. Figure 11 shows the MATLAB code to generate square signals.

```
octave:13> square_freq = [150e3 16e3 10e3 200e3]; % Different frequencies for square waves
phases = [0 30 45 90 120]; % Different phases in degrees
t = 0:1/4e6:(135-1)/4e6; % Time vector for 135 samples at 4 MHz sample rate

for i = 1:length(square_freq)
    for j = 1:length(phases)
        data(:,1) = round((square(2*pi*square_freq(i) * t + phases(j) * pi/180) + 5) * 100);

        % Constructing filename
        freq_str = num2str(square_freq(i)/1000); % Convert frequency to kHz and string
        phase_str = num2str(phases(j)); % Convert phase to string
        filename = strcat('square_', freq_str, 'KHz_', phase_str, 'deg.txt');

        dlmwrite(filename, data(:,1), ' ');
    end
end
```

Figure 11 Square Waveform Signals generation in MATLAB

Three of the square signals generated are represented in figure 12.

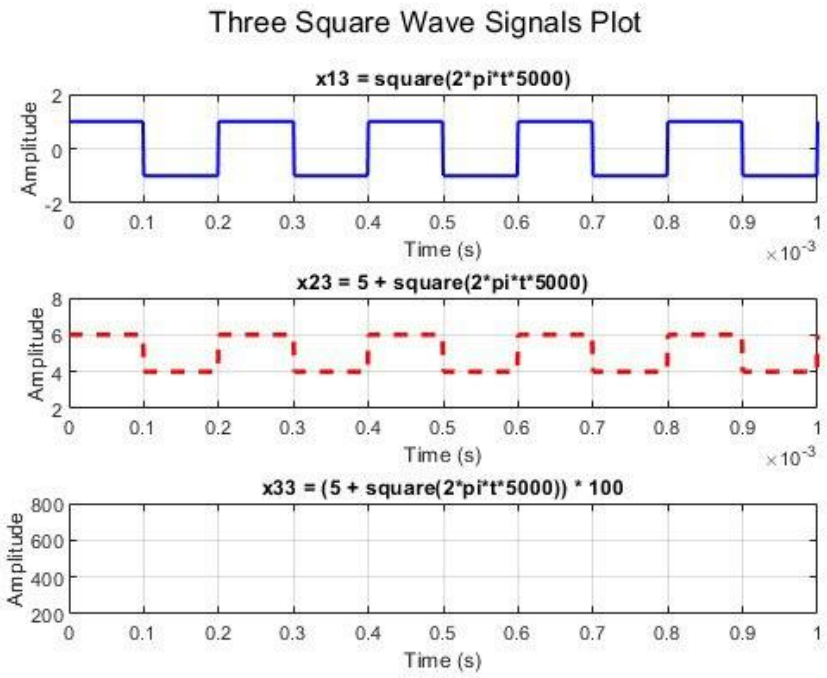

Figure 12 Square waveforms used for Test

$$x13 = \text{square}(2\pi t * 5000)$$

$$x23 = 5 + \text{square}\,(2\pi t * 5000)$$

$$x33 = (5 + \text{square}\,(2\pi t * 5000)) * 100$$

The square signal alternates between a high value – one - and a low value – zero - with a 50% duty cycle. Therefore, it spends an equal amount of time in the high state and the low state within each period.

After this, the same steps as Sine waves are repeated for this waveform.

Triangle Waves

Test cases for triangle waves include frequencies of 150 kHz, 149 kHz, 151 kHz, 5 kHz, and 200 kHz with phase angles of 0°, 30°, 45°, 90°, and 120°. Figure 13 shows the MATLAB code to generate square signals.

```
octave:9> triangle_freq = [150e3 149e3 151e3 5e3 200e3]; % Different frequencies for triangle waves
phases = [0 30 45 90 120]; % Different phases in degrees
t = 0:1/4e6:(135-1)/4e6; % Time vector for 135 samples at 4 MHz sample rate

for i = 1:length(triangle_freq)
    for j = 1:length(phases)
        data(:,1) = round((sawtooth(2*pi*triangle_freq(i) * t + phases(j) * pi/180, 0.5) + 5) * 100);

        % Constructing filename
        freq_str = num2str(triangle_freq(i)/1000); % Convert frequency to kHz and string
        phase_str = num2str(phases(j)); % Convert phase to string
        filename = strcat('triangle_', freq_str, 'KHz_', phase_str, 'deg.txt');

        dlmwrite(filename, data(:,1), ' ');
    end
end
```

Figure 13 Triangle Waveform Signals generation in MATLAB

Three of the triangle signals generated are represented in figure 14.

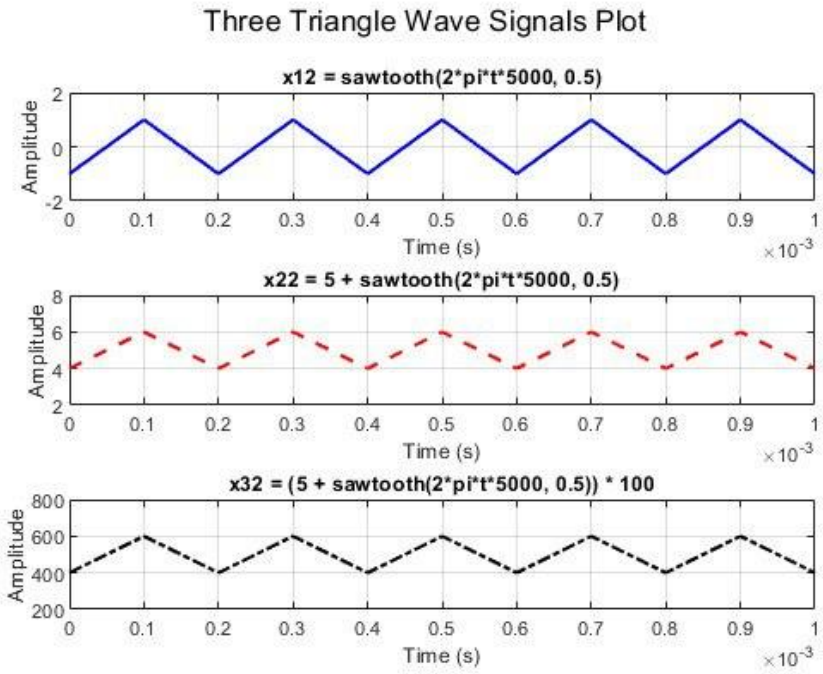

Figure 14 Triangle waveforms used for Test

$$x12 = \text{sawtooth}(2\pi t * 5000, 0.5)$$

$$x22 = 5 + \text{sawtooth}\,(2\pi t * 5000, 0.5)$$

$$x32 = (5 + \text{sawtooth}\,(2\pi t * 5000, 0.5)) * 100$$

In the first parameter, 5000 indicates the angular frequency of the sawtooth wave. This corresponds to a linear frequency of 5000 Hz (or 5 kHz). The second parameter 0.5 specifies the shape of the sawtooth wave. For a typical sawtooth wave, this parameter can control the duty cycle or the position where the wave resets. A value of 0.5 means that the wave rises and falls symmetrically.

After this step, the same process as Sine waves is repeated for this waveform.

Results

EPWave Results

This report analyzes the simulation results of the VHDL test bench designed for testing a digital signal processing module (DUT). The waveform results are captured using the EPWave, an advanced waveform viewer and simulation tool integrated into EDA Playground, showcasing the behavior of various signals over time. The purpose of this analysis is to verify the correct functionality of the DUT by examining the simulation waveforms. Figure 15 shows a VHDL code to convert and assign values within a clock cycle.

```
x <= unsigned(to_signed(readVal, x'length));
clk <= '0';
wait for 1 ns;
clk <= '1';
wait for 1 ns;
```

Figure 15 Converting and assigning values within a clock cycle

Figures 16 and 17 represent the EPWave waveforms for sin(2π5000t),

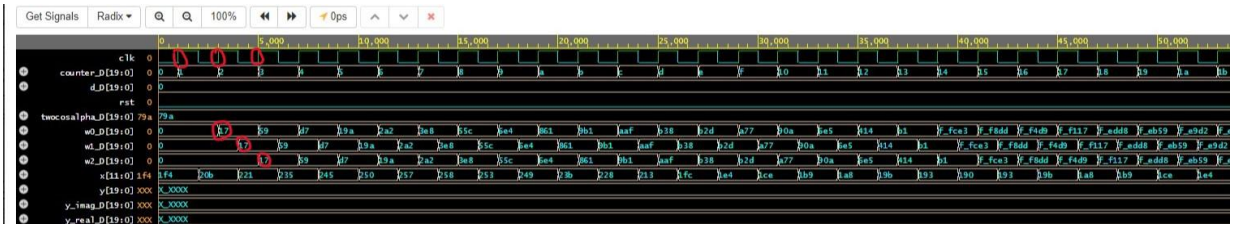

Figure 16 EPWave waveforms for sin (2π5000t)

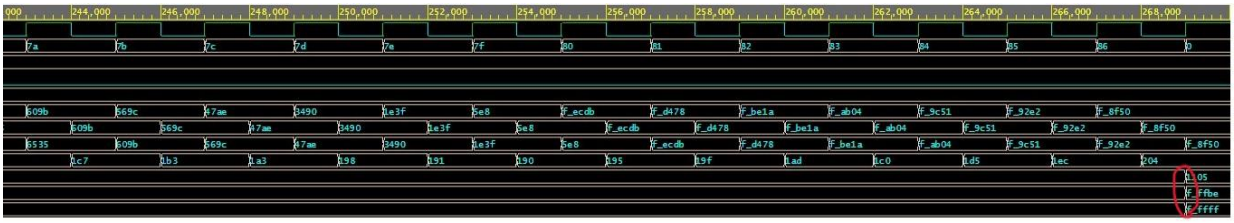

Figure 17 EPWave waveforms for sin (2π5000t)

The provided simulation waveform in figures 16 and 17 captures several signals, including the clock (clk), reset (rst), input signals (x), internal signals (w0, w1, w2), and output signal (y_real, y_imag). The simulation spans a time frame of 270,000 ns. The clock signal is toggling consistently with a period of 2 ns, indicating a frequency of 500 MHz. This high-frequency clock is used to drive the simulation. The counter increments on the rising edge of clk sequentially, reflecting the progression of the simulation time. Each increment represents a new test cycle within the simulation. The d signal holds the data being processed by the DUT. The values change at each clock cycle, corresponding to the input data read from the test file. The reset signal is initially low (0) and remains low throughout the simulation, ensuring the DUT is not reset during the test cycles. These signals represent intermediate calculations within the DUT. The values change dynamically as the input signal x is processed. The input signal x shows the data read from the input file. The

values correspond to the test cases specified in the test bench. Each value is applied to the DUT for a duration of 2 ns, aligning with the clock period. The output signals represent the real and imaginary parts of the DUT's output. Initially, these signals show undefined values (XXXX), which stabilize as the simulation progresses. The changes in these signals indicate the DUT's response to the input signal x. The clock signal is consistent and drives the simulation as expected. Each clock cycle corresponds to a new test cycle. The input data (x) is processed correctly by the DUT, as indicated by the changes in internal signals (w0, w1, w2) and the output signals (y_real, y_imag). The output signals initially show undefined values but stabilize as the simulation progresses. This behavior is typical as the DUT takes a few cycles to process the input data and produce meaningful output. The intermediate values (w0, w1, w2) change in response to the input data, reflecting the internal calculations within the DUT. The simulation waveform indicates that the VHDL test bench and DUT are functioning correctly. The clock signal is stable, and the input data is processed appropriately, resulting in expected changes in the output signals. The DUT demonstrates correct behavior under the given test cases, and the simulation provides valuable insights into its internal workings. The stability of the output signals after initial undefined values confirms the DUT's proper operation over time. At the end of 135 sampling cycles, the output value is calculated shown in figure 13. This means the algorithm computes one output for every given sampling number.

MATLAB vs VHDL

Figure 18 shows the results that the magnitude of 150Khz inside a 5Khz signal is non-existent, and thus our VHDL implementation produces a value of near zero magnitude, matching that of MATLAB at 2.43 magnitude with a negligible margin of error. For 200Khz input signal, our VHDL code also does not detect any 150khz signal matching that of MATLAB. However, we get very similar results for input with frequency of 149 kHz, 150 kHz, and 151 kHz. That is due to the small window size of 135 elements, neither MATLAB nor VHDL were able to distinguish these three frequencies apart.

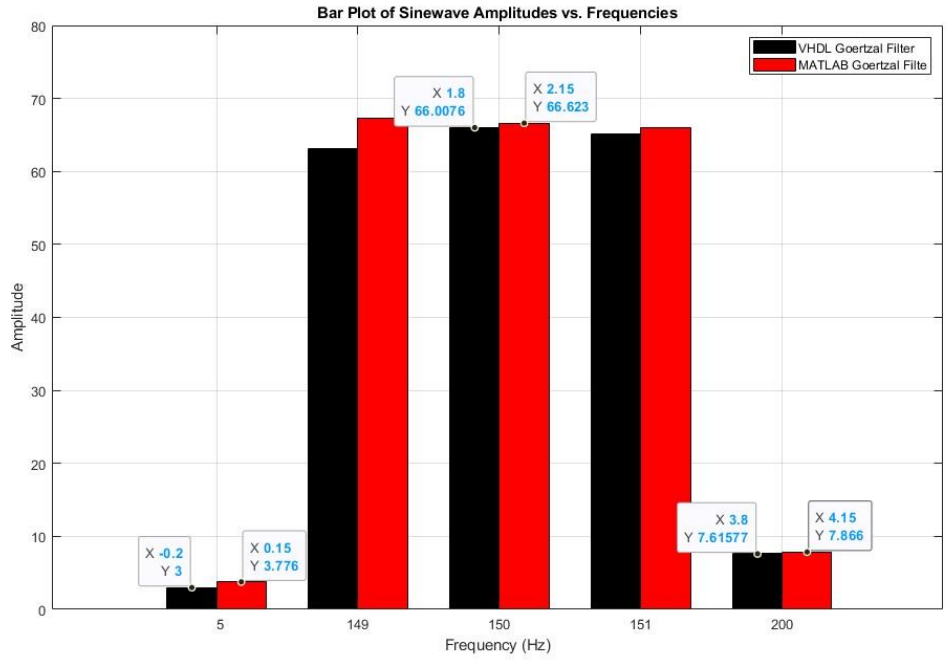

Figure 18 150 kHz signal Detection of sine wave testcase in MATLAB and VHDL

Similarly, in figure 19 square wave testcases were used in the same procedure. despite the same value for VHDL and MATLAB results for the square wave, we can clearly see that 150 kHz frequency exists in nonnegligible magnitudes in almost all the frequency components. This is due to the jump in magnitude in the square wave signal where one jump contains all the frequencies and thus affecting the magnitude.

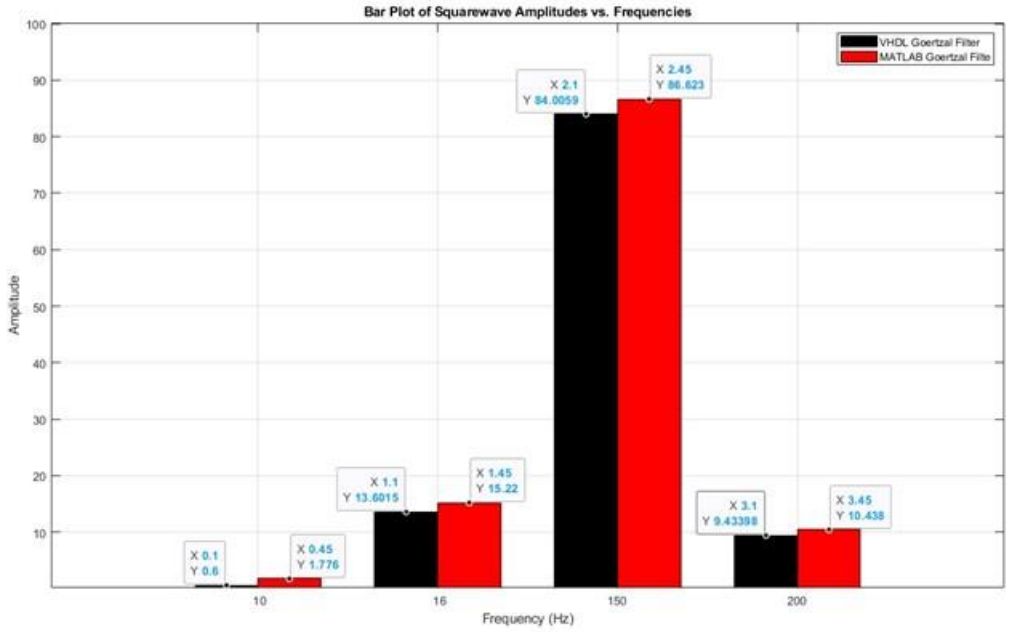

Figure 19 150 kHz signal Detection of Square waves testcase in MATLAB and VHDL

Conclusion

The Goertzel Algorithm is a digital signal processing technique used for efficient and fast calculation of individual DFT coefficients. It provides a computationally efficient alternative to the Fast Fourier Transform (FFT) algorithm when the calculation of a specific frequency component is required rather than the entire spectrum. In this project, a brief introduction of Goertzel algorithm as well as literature research on the state-of-the-art Goertzel ASIC and FPGA implementations are provided. In practical stage, a Goertzel algorithm for the given criteria of target frequency 150kHz and sampling cycle of 135 is designed and implemented in VHDL using EDA Playground. It is also verified through stimuli data of three different signals in a range of frequencies close and far from the target frequency in both VHDL and MATLAB. As the results show the algorithm identifies the frequencies around 150kHz very well while cannot recognize those far away of 150kHz like 5kHz, which proves the performance of the algorithm as theoretically expected.

REFERENCES

[1] G. Goertzel, "An Algorithm for the Evaluation of Finite Trigonometric Series," *The American Mathematical Monthly.*

[2] M. T. L. V. Y. R. M. A. De Jesús, "Nonuniform Discrete Short-Time Fourier Transform A Goertzel Filter Bank versus a FIR Filtering Approach," *IEEE.*

[3] T. Dulik, "An FPGA Implementation of Goertzel Algorithm.," *Springer.*

[4] L. Regnacq, Y. Wu, N. Neshatvar, D. Jiang and A. Demosthenous, "A Goertzel Filter-Based System for Fast Simultaneous Multi-Frequency EIS," *IEEE.*

[5] P. Sysel and P. Rajmic, "Goertzel algorithm generalized to non-integer multiples of fundamental frequency," *EURASIP Journal on Advances in Signal Processing.*

Appendix

VHDL Goertzel Design

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


entity goertzel is
port (
     clk: in std_logic; -- operations are performed on Rising and falling edge.
     rst: in std_logic; -- rising edge signal, all internal and external signals
are set to 0.
     x : in unsigned(11 downto 0); -- input values have 5v DC bias and multiplied
by 100 to remove the negatvie samples
     y : out signed(19 downto 0);
    );
end entity;


architecture behav of goertzel is

  -- internal constant signals that are 20 bit signed
         signal a_D : signed(19 downto 0) := to_signed(973, 20);           --
  calculated to be 0.9730
         signal b_D : signed(19 downto 0) := to_signed(309, 20);           --
  calculated to be 0.309017
         signal c_D : signed(19 downto 0) := to_signed(-1000, 20);         --
  calculated to be -1.0
  signal d_D : signed(19 downto 0) := to_signed(0, 20);             --calculated to
  be -1.22465e-15
  signal twocosalpha_D : signed(19 downto 0) := to_signed(1946, 20);--calculated to
  be 1.946

   -- internal signals for intermediate values that are 20 bit signed
  signal w0_D : signed(19 downto 0):= to_signed(0, 20);
  signal w1_D : signed(19 downto 0):= to_signed(0, 20);
  signal w2_D : signed(19 downto 0):= to_signed(0, 20);
  signal y_real_D: signed(19 downto 0); -- real component of the output
  signal y_imag_D: signed(19 downto 0); -- imaginary component of the output

  -- internal signal for counter that is 20 bit signed
  signal counter_D : signed(19 downto 0):= to_signed(0, 20);


begin
                y <= resize(resize((y_real_D*y_real_D),y'length) +  resize(
(y_imag_D*y_imag_D),y'length), y'length);

process(clk,rst)
   begin
   if rst'event and rst='1' then --- resetting all signals to 0
         counter_D <= to_signed(0, counter_D'length);
         w0_D <= to_signed(0, w0_D'length);
         w1_D <= to_signed(0, w1_D'length);
```

```
            w2_D <= to_signed(0, w2_D'length);
            y_real_D <= to_signed(0, y_real_D'length);
            y_imag_D <= to_signed(0, y_imag_D'length);

    elsif clk'event and clk='1' then --on rising clock edge
       if counter_D < to_signed(134,counter_D'length) then --if counter is less than
       134 then continue with the goertzel algorithm
            w0_D <= resize(signed("0000"&x) - to_signed(500,w0_D'length) +
            resize((twocosalpha_D * w1_D)/1000,w0_D'length)- w2_D,w0_D'length);
            counter_D <= counter_D + to_signed(1, 20);
       else -- if counter is > 134 then calculate the real and imaginary components
       of the output and reset the counter.
            y_real_D <= resize(resize( resize( a_D*w1_D/1000 ,y_real_D'length) +
            resize( c_D*w2_D/1000 ,y_real_D'length)
            ,y_real_D'length)/to_signed(135,y_real_D'length),y_real_D'length);
            y_imag_D <= resize(resize( resize( b_D*w1_D/1000 ,y_imag_D'length) +
            resize( d_D*w2_D ,y_imag_D'length)
            ,y_imag_D'length)/to_signed(135,y_imag_D'length),y_imag_D'length);
            counter_D <= to_signed(0, 20);

               end if;



      end if;

            w2_D <= w1_D;
            w1_D <= w0_D;
      end process;

  end architecture;
```

MATLAB Signal Generation

```
clc
clear all;
close all;

fs = 4e6;    % Sampling frequency
f0 = 150e3; % Desired frequency
N = 135;     % Number of samples

sine_freq = [150 149 151 5 200] ;  % Different frequnecies for sine waves
rect_freq = [150 16 10 200];       % Different frequencies for rectangular waves
tria_freq = [150 149 151 5 200];   % Different phases for triangular waves
phases = [0 30 45 90 120] ;        % Different phases in radians
t = (0:N-1)*(1/fs);                % timesteps for the desired number of samples

for i = 1:length(sine_freq)
    for j = 1:length(phases)
    data(:,1) = round((sin(2*pi*sine_freq(i)* 1000 *t + phases(j)* pi/180) + 5)
*100); % Generate 12-bit offset binary data with phase shift
    filename = 'sine_'+ string(sine_freq(i)) + 'KHz_' + string(phases(j)) +
'deg.txt';
    %dlmwrite(filename, data(:,1), ' ');
```

```
    end
    subplot(3,1,1)
    stem(data(:,1),t );
end

for i = 1:length(rect_freq)
    for j = 1:length(phases)
    data(:,1) = round((square(2*pi*rect_freq(i)* 1000 *t + phases(j)* pi/180) + 5)
*100); % Generate 12-bit offset binary data with phase shift
    filename = 'rect_' + string(rect_freq(i)) + 'KHz_' + string(phases(j)) +
'deg.txt';
    dlmwrite(filename, data(:,1), ' ');
    end
    subplot(3,1,2)
    stem(data(:,1),t );
end

for i = 1:length(tria_freq)
    for j = 1:length(phases)
    data(:,1) = round((sawtooth(2*pi*tria_freq(i)* 1000 *t + phases(j)* pi/180 ,1)
+5) *100); % Generate 12-bit offset binary data with phase shift
    filename = 'tria_' + string(tria_freq(i)) + 'KHz_' + string(phases(j)) +
'deg.txt';
    dlmwrite(filename, data(:,1), ' ');
    end
    subplot(3,1,3)
    stem(data(:,1),t );
end
```

```
k = round(N * f0/fs) ; %frequency index
%k= 6;
alpha = 2 * pi * k / N;
beta = 2 * pi * k * (N-1)/N;
a = cos(beta);
b = -1 * sin(beta);
c = sin(alpha) * sin(beta) - cos(alpha) * cos(beta);
d = sin(2 * pi * k);
2 * cos(alpha);
```

VHDL Testbench

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.math_real.all;
use std.textio.all;
use IEEE.STD_LOGIC_TEXTIO.all;

entity goertzel_tb is
end entity;

architecture behav of goertzel_tb is
```

```
  -- Component declaration
  component goertzel
      port (
      clk: in std_logic;
      rst: in std_logic;
      x : in unsigned(11 downto 0);
      y : out signed(19 downto 0)
      );
  end component;

  -- Signal declarations
  signal   rst      :  std_logic := '0';
  signal   clk      :  std_logic := '0';
  signal   x        :  unsigned(11 downto 0);
  signal   y        :  signed(19 downto 0);
  signal   fileName : string(1 to 22) := to_string("sine_149KHz_120deg.txt"); ---
sample files name with 22 bits

begin -- architecture

  -- Instantiate the DUT (Device Under Test)
  dut: goertzel port map (
      rst => rst,
      clk => clk,
      x   => x,
      y   => y
      );

  process
      file testFile : text;
      variable lineStr : line;
      variable readVal : integer;
      variable count   : natural := 0;
  begin
      file_open(testFile, fileName, read_mode);

      while not endfile(testFile) loop

          readline(testFile, lineStr);
          read(lineStr, readVal);
          x <= unsigned(to_unsigned(readVal, x'length));
          clk <= '0';
          wait for 1 ns;
          clk <= '1';
          wait for 1 ns;
          count := count + 1;


          if count = 135 then
            report  fileName&" -> power "& to_string(to_integer(y)) &"-> Amplitude:
" & to_string(sqrt(real(to_integer(y))),"%f") & ".";
            exit;
          end if;

       end loop;

      file_close(testFile);
```

```
      wait;
  end process;

end architecture;
```