Goertzel Filter Report

Zaur Gurbanli, Hamida Aliyeva, Farhad Gulizada

13 July, 2024

Introduction

The Goertzel algorithm is a digital signal processing technique which is able to efficiently detect specific frequencies in a signal. It was initially developed as an effective way to compute trigonometric functions, while now it is used to compute discrete-time Fourier transform (DTFT) with only two coefficients.

This article aims to provide information on how to implement Goertzel's algorithm along with the general development process accordingly. To achieve this, the report is organized accordingly to reflect our process. The broad theory of filtering is covered in the beginning, since it is necessary to understand the properties of the algorithm. An overview of the state of the art will be presented next where applications and implementation of various algorithms will be discussed. As a next step, a brief description of how the algorithm is implemented will be shown. The main goal in this section is to propose an efficient, optimized implementation of the Goertzel algorithm that meets the necessary requirements. It also needs to be implemented in a way that allows synthesis on an FPGA/ASIC platform. It will then be explained, and it will be clear how to build a VHDL test bench to test the accuracy and correctness of the Goertzel Filter implementation and how to generate the input stimulus data and expected results of the MATLAB simulation. This simulation served a number of functions, the main ones being to improve algorithm understanding, design decision making, and design correctness verification.

Goertzel algorithm

As mentioned above, the main purpose of the Goertzel algorithm is to detect the frequency in the telephone tone dialing (dual-tone multi-frequency, DTMF). In DTMF, a signal is understood when two of the eight possible frequencies are present simultaneously. [1] The Goertzel algorithm is often called the "Goertzel Filter" because it is thought to take the form of a digital filter, and this explains how it is implemented. The filter is used to perform complex demodulation for any frequency point. On the one hand, its first application was the effective calculation of trigonometric functions. On the other hand, it is now sophisticated enough to provide a productive method for computing the discrete-time Fourier transform (DTFT) with only two coefficients. A discrete N sample signal $x(N)$ can recast its DTFT recursively as follows:

$$y(n) = W_N^m y(n-1) + x(n).$$

where $W_N^m = e^{-\frac{2j\pi mn}{N}}$, $n \in [0,\ N]$, $y(-1) = 0$, $and\ x(N) = 0$.

Thus, when $n = N$:

$$X(m) = y(N).$$

where $X(m)$ is the DTFT of the signal $x(n)$ evaluated at the bin m after N samples. The z-transform of the recursive equation yields the following transfer function:

$$\frac{Y(z)}{X(z)} = \frac{1 - W_N^m z^{-1}}{1 - 2\cos\left(\frac{2\pi m}{N}\right) z^{-1} + z^{-2}}.$$

The equation is realized using the structure shown in the below picture:

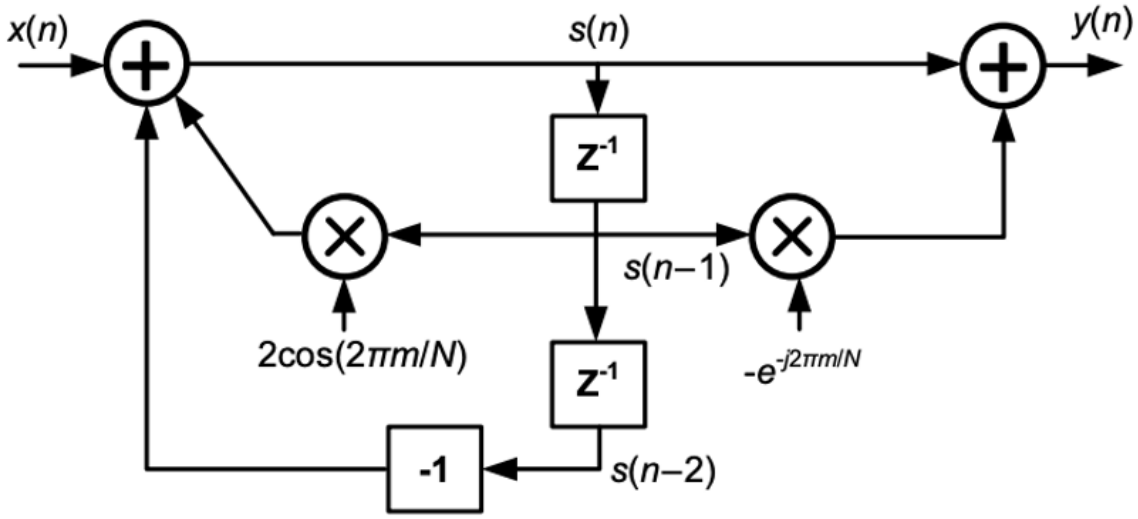

Figure 1. IIR filter structure of the Goertzel algorithm.

The filter consists of two parts: a feedforward path using a complex multiplication and a second-order infinite impulse response (IIR) filter with a real multiplication. Although the forward pass can only be done once, the filter part for $s(N)$ is executed N times (with $x(N) = 0$). It requires a total of $N + 2$ multiplications and $2N + 1$ additions. With complex multiplication, three coefficients (real and imaginary part coefficients) are stored: one for the recursive component and two for the feedforward section. The computation time when evaluating a frequency point is $(N + 1)T_s$, where T_s is the sample period. Consequently, the Goertzel filter adds a sample period delay after acquiring the N sampled input signal before extracting the demodulated frequency point.

A Goertzel filter requires about half as many multiplications and the same number of additions as for conventional coherent demodulation. Goertzel filtering has a very reasonable memory requirement because it only needs to store three coefficients per bin. To further minimize the memory requirement, only the IIR part can be integrated into the digital circuit. After processing N samples, $s(N)$ and $s(N - 1)$ data are sent to the computer for processing and final complex multiplication. This requires only one coefficient to be stored and reduces the number of samples per box to be broadcast (N samples) to two values (as in coherent modulation and FFT). By adding a correction factor directly after the forward pass path, the Goertzel filter can accurately determine the frequency content for an incomplete box: [1]

$$y(n)_{corrected} = y(n)e^{-j2\pi m}$$

It is important to realize that the filter resonates around the region of interest. Due to this fact there is no harmonic backoff problem in this demodulation approach. [2]

Literature Review

Applications in Various Fields

Before moving into implementation part, literature review will be discussed about Goertzel Filter. There are many applications of Goertzel Filter. Firstly, it is known extremely usable for Dual-Tone Multi-Frequency detection. Dual-tone multi-frequency (DTMF) signaling is a standard in telecommunication systems. When the telephone was initially developed, it was impossible to make direct phone calls. Instead, a telephone operator was located at the opposite end of the line. Each time the phone was picked up, the operator must be told where to connect and they manually forwarded the call to the central switch. There was no way this was going to continue. In addition to what this type of system can handle, people need to communicate with a large number of people. The solution was a control signal method for autonomous routing. With DTMF communication, control signals (such as a destination phone number or special number) are sent outside the voice range. Due to the separate control channels in mobile transmissions, their importance has declined over time, but their continued use in keyboard applications has kept them relevant. Examples include users exploring a business's phone menu, answering questions by pressing buttons on a call panel, and entering account or credit card information during a phone transaction. DTMF detection is used to detect DTMF signals in the presence of speech and dialing tone pulses. One known advantage of Goertzel filter is that it is faster compared to Fourier Transform (FFT) for detecting a tone with a particular frequency in a signal. That is why Goertzel filter is now used for DTMF detection. [3] According Rabiner et al., Goertzel filter is extensively useful for decoding DTMF signals in telecommunication systems. [4] This is accomplished thanks to the filter's strong ability for efficiently detecting the dual-tone frequencies that are generated by telephone keypads.

Biomedical Signal Processing is another widely used application area of the Goertzel filter. The filter in this area is used for analyzing specific frequency bands in Electrocardiography (ECG) and Electroencephalogram (EEG) signals. There are studies that show how effective it is to monitor heart rate and brain wave activities with the help of Goertzel filter where it provides crucial information for medical diagnostics.

Another well-known area where the Goertzel filter is widely used is in industrial applications. In order to be able to monitor machinery vibrations and focus on specific frequencies where mechanical faults are indicated, the filter is utilized. With little computing overhead, real-time frequency analysis can be obtained using the Goertzel filter. This feature of the filter is widely useful for industrial applications.

Finally, the last application area of the Goertzel filter that will be mentioned in this report is adaptive filtering. The Goertzel filter is now included in the advances in adaptive filtering for real-time applications. It is proposed to use an adaptive Goertzel filter for dynamic frequency detection in dynamic signal parameters, which demonstrates the potential of the filter to be used in modern signal processing applications.

Advantages

There are numerous advantages of Goertzel filter. First of all, it uses less memory. In addition, it is simple so to implement the filter is quite easy. It is also robust to noise. That means the filter is able to isolate and measure specific frequencies even in the presence of noise. Finally, it mainly focuses on individual frequencies, so precise amplitude and phase information for those frequencies are provided

Goertzel Filter implementation in Matlab

Algorithm implementation

This part in the report shows how we implemented Goertzel Filter in Matlab. Below is the Matlab code:

```
function magnitude = Goertzel_Filter(signal, sampleRate, targetFreq)
    w = 2 * pi * targetFreq / sampleRate;
    coef = 2*cos(w);

    q1 = 0;
    q2 = 0;

    for n = 1:length(signal)
        q0 = signal(n) + coef * q1 - q2;
        q2 = q1;
        q1 = q0;
    end

    magnitude = q1^2+q2^2-coef*q2*q1;

end
```

Figure 2. Goertzel Algorithm in Matlab

$function\ magnitude\ =\ Goertzel\_Filter(signal, sampleRate, targetFreq)$ line defines a function named $Goertzel\_Filter$ that calculates the magnitude of a specific target frequency in a given signal. The function takes three inputs: the input signal array, the sampling rate of the signal, and the target frequency to detect in the signal. In order to calculate the angular frequency $w\ =\ 2\ *\ pi\ *\ targetFreq\ /\ sampleRate$ equation is used. The variable $coef$ is the coefficient used in the recurrence relation of the Goertzel algorithm, calculated as $2cos(w)$.
Two state variables, q1 and q2, are initialized to zero. These variables store intermediate values of the recurrence relation. Recurrence Relation Loop processes each sample of the input signal. The recurrence relation is defined as:

- $q0$ is the current state, calculated using the current signal value and previous states $q1$ and $q2$;

- $q2$ is updated to the previous value of $q1$;
- $q1$ is updated to the current state $q0$.

This recurrence relation filters the input signal and isolates the component corresponding to the target frequency.

The magnitude of the target frequency component is calculated using the final values of $q1$ and $q2$. This formula is an optimized version, avoiding the need to calculate the real and imaginary parts separately.

Finally, the Goertzel filter implemented in MATLAB efficiently computes the magnitude of a specific frequency in a given signal. It uses a second-order IIR filter structure defined by a simple recurrence relation, making it computationally efficient for detecting single frequency components. This implementation calculates the magnitude directly from the filter's final states, which is particularly useful in real-time signal processing applications.

Verification

The testing function evaluates the performance of the Goertzel filter across various waveforms (sine, square, and triangle) and frequencies. It generates these waveforms, applies the Goertzel filter, and saves the results for analysis.

Below is the MATLAB initialization setup for testing the Goertzel filter, which includes the definition of test frequencies, waveforms, sampling rate, and other essential parameters. This setup prepares the environment for generating and analyzing the waveforms using the Goertzel filter:

```
freqsin = [5000, 149000, 150000, 151000, 200000];
freqrect = [10000, 16000, 150000, 200000];
freqtriangle = [5000, 149000, 150000, 151000, 200000];
all_frequencies = {freqsin, freqrect, freqtriangle};
degrees = [0, 30, 45, 90, 120];
wave_names = ["Sine", "Square", "Triangle"];
input_bit = 12;

sample_rate = 4000000;
target_freq = 150000;
N = 135;
t = 0:1/sample_rate:(N-1)/sample_rate;
```

Figure 3. Initialization

$freqsin, freqrect, and\ freqtriangle$ are arrays of test frequencies for sine, square, and triangle waves respectively. $'degrees'$ contains phase shifts to test. N is the number of samples and t is the time vector for the generated waveforms.

Below picture shows the magnitude arrays:

```
sinwave_magnitudes = zeros(length(freqsin), length(degrees));
squarewave_magnitudes = zeros(length(freqrect), length(degrees));
trianglewave_magnitudes = zeros(length(freqtriangle), length(degrees));

all_magnitudes = {sinwave_magnitudes, squarewave_magnitudes, trianglewave_magnitudes};
```

Figure 4. Magnitude Arrays

These arrays store the magnitudes calculated by the Goertzel filter for each frequency and phase shift combination.
Main Processing Loop is displayed in the picture below:

```
for k = 1:length(all_frequencies)
    frequencies = all_frequencies{k};
    wave_magnitudes = all_magnitudes{k};
    wave_name = wave_names(k);

    for i = 1:length(degrees)
        for j = 1:length(frequencies)

            deg = degrees(i);
            freq = frequencies(j);

            sinewave = sin(2*pi*freq*t + deg2rad(deg))*(2^input_bit-1);
            squarewave = square(2*pi*freq*t + deg2rad(deg))*(2^input_bit-1);
            trianglewave = sawtooth(2*pi*freq*t + deg2rad(deg))*(2^input_bit-1);


            all_waves = {sinewave, squarewave, trianglewave};

            wave_magnitudes(j, i) = Goertzel_Filter( all_waves{k}, sample_rate, target_freq);
            save_to_file("input", wave_name, freq, deg, all_waves{k});
            save_to_file("result", wave_name, freq, deg, wave_magnitudes(j, i));

        end
    end

    subplot(3,1,k)
    bar(wave_magnitudes, 'grouped');
    hold on
    title("Magnitudes for " + wave_name + " Waves");
    legendInfo = arrayfun(@(deg) [num2str(deg), ' Degree'], degrees, 'UniformOutput', false);
    legend(legendInfo, "Location", "bestoutside");
    xticklabels(arrayfun(@num2str, frequencies, 'UniformOutput', false));
end
```

Figure 5. Main Processing Loop

Outer Loop iterates over the different waveforms (sine, square, triangle) while middle loop iterates over the different phase shifts and inner loop iterates over the different test frequencies. Wave Generation generates sine, square, and triangle waves for each frequency and phase shift. Each generated waveform (sine, square, and triangle) is scaled by $2^{input\_bit} - 1$. By doing so, the amplitude of the waveforms is scaled to fit the range of a 12-bit digital system, which ranges from 0 to 4095. This scaling ensures that the waveforms are represented in a manner consistent with how they would be processed in a real-world digital signal processor.

Then the Goertzel filter is applied to each waveform and stores the result in $wave\_magnitudes$. There is a $save\_to\_file$ function which saves both the input waveform and the resulting magnitude. Finally, bar plots for the magnitudes of each waveform type are created. $save\_to\_file$ function, shown in the picture below, saves data to a text file. It ensures the folder exists, deletes any existing file with the same name, and writes the data to the file. The $uint16$ function converts the data to a 16-bit unsigned integer format, ensuring that any negative values are set to 0.

```
function save_to_file(folder_name, wave_name, freq, degree, data)

    if ~exist(folder_name, 'dir')
        mkdir(folder_name);
    end

    filename = sprintf('%s/%s_%ddegree_%dHz.txt', folder_name, lower(wave_name), degree, freq);

    if exist(filename, 'file')
        delete(filename);
    end

    fileID = fopen(filename, 'w');
    fprintf(fileID, '%d\n', uint16(round(data)));
    fclose(fileID);

end
```

Figure 6. Save to File Function

The *testing* function systematically evaluates the Goertzel filter across various waveforms, frequencies, and phase shifts. It generates waveforms, applies the filter, saves the inputs and results, and plots the magnitudes. This thorough testing approach ensures that the Goertzel filter's performance can be analyzed for different scenarios, making it an effective tool for frequency detection in digital signal processing.

Below simulation results illustrate the magnitude of sine waves, square waves, and triangle waves as detected by the Goertzel filter across various frequencies and phase shifts. Each bar chart provides a visual representation of the filter's performance, displaying magnitudes for every combination of degree and frequency. This allows for a detailed comparison of how the Goertzel filter responds to different waveforms under various conditions:

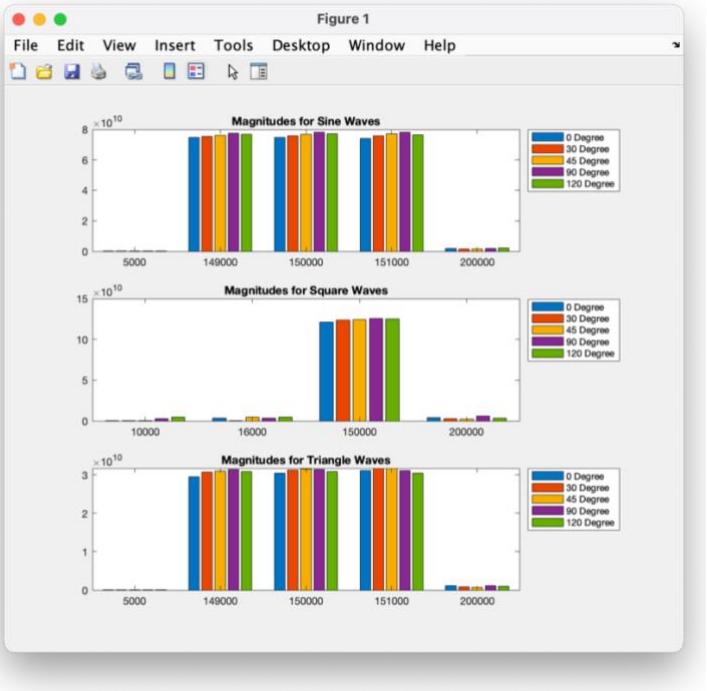

Figure 7. Results of MATLAB simulation

Goertzel Filter implementation in VHDL

The Goertzel filter has been implemented in VHDL to detect specific frequencies within a digital signal. This implementation utilizes fixed-point arithmetic to ensure efficient use of hardware resources, avoiding the complexity and resource demands of floating-point operations.

Below picture shows how entity is declared:

```
entity goertzel_filter is
    generic (
        coef : signed(19 downto 0) := to_signed(19447, 20);
        coef_div : signed(19 downto 0) := to_signed(10000, 20)
    );
    port (
        input_signal : in unsigned (11 downto 0);
        clk : in STD_LOGIC;
        rst : in STD_LOGIC;
        magnitude : out signed(19 downto 0)
    );
end entity goertzel_filter;
```

Figure 8. Entity Declaration

The entity $goertzel\_filter$ defines the filter's interface, which includes generics and ports. The generics are used to set the coefficients ($coef$) and a division factor ($coef\_div$) that were precomputed using MATLAB. These coefficients are critical for the filter's frequency detection capabilities. The ports include the input signal, a clock signal (clk), a reset signal (rst), and the output magnitude.

Below picture shows how the architecture is declared:

```
architecture behavioural of goertzel_filter is
    signal q0, q1, q2 : signed(19 downto 0);
    signal x : signed(11 downto 0);
    signal result : signed(19 downto 0);
begin
    process (clk, rst)
    begin
        if rst = '1' then

            q0 <= (others => '0');
            q1 <= (others => '0');
            q2 <= (others => '0');
            result <= (others => '0');

        elsif rising_edge(clk) then

            x <= signed(resize(input_signal, x'length));
            q0 <= x + resize(coef * q1 / coef_div, q0'length) - q2;
            q2 <= q1;
            q1 <= q0;

            --Optimized Goertzel
            result <= resize(q1 * q1 + q2 * q2 - resize(coef * q2 * q1 / coef_div, result'length),
result'length);

        end if;
    end process;

    magnitude <= result;
end architecture behavioural;
```

Figure 9. Architecture Declaration

The architecture behavioral describes the internal workings of the filter. It uses several signals to store the state of the filter ($q0, q1, q2$) and the result.

Process Block is triggered on the rising edge of the clock or when the reset signal is active. When reset, all state signals, and the result are initialized to zero. On each clock cycle, the process block updates the state signals based on the Goertzel algorithm. This includes converting the input signal to a signed format, updating the filter states, and calculating the magnitude of the detected frequency using the optimized Goertzel formula.

Fixed-Point Arithmetic: To maintain precision and efficiency, the algorithm uses fixed-point arithmetic. The $coef$ and $coef\_div$ values are used to scale the operations appropriately, ensuring the calculations fit within the fixed-point representation without requiring floating-point hardware.

The coefficients used in the VHDL code were calculated in MATLAB. Since these coefficients are originally in floating-point format, they are converted to a fixed-point representation suitable for hardware implementation. The division factor ($coef\_div$) helps in this conversion, allowing the multiplication and division operations to be performed accurately within the fixed-point domain.

In conclusion, The VHDL implementation of the Goertzel filter is designed for efficient frequency detection in digital signals. By leveraging fixed-point arithmetic and precomputed coefficients, the implementation minimizes hardware resource usage while maintaining accurate frequency detection capabilities. This approach ensures that the filter can be used effectively in various digital signal processing applications.

Testbenchmarking

After implementing the Goertzel filter in Matlab and generating stimuli data the next step was to implement the filter in the VHDL. The aim is to make sure that the generated input gives the same result in the VHDL implementation of the Goertzel filter as in Matlab implementation. In order to verify that the results are the same in both implementations a test benchmark has been implemented and it has been written in the VHDL.

The implemented benchmarking code sets up a testbench for a Goertzel filter, which involves generating a clock signal, reading input signals and expected results from files generated in matlab, instantiating the filter, and comparing the filter's output to the expected results.

Goertzel Filter Specifications

- Number of samples - 135
- Input data - 12 bit
- Internal data - 20 bit
- Sample frequency - 4MHz
- Signal frequency to detect - 150 KHz

The variables assigned to these will be provided in the following.

Library and Package Inclusions

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;
```

Figure 10.Used Library and Package Declaration

This section includes the necessary libraries and packages. The *ieee.std_logic_1164* package is for definitions for standard logic types. The *ieee.numeric_std* package provides arithmetic operations on these types, and *std.textio* handles text file input/output operations so that to read the stimuli data and expected data from files.

Entity Declaration

```
entity tb_goertzel_filter is
end entity tb_goertzel_filter;
```

Figure 11. Script for Entity Declaration

The *tb_goertzel_filter* entity is defined here, as the container for the benchmarking. No ports are declared as this is a self-contained testbench.

Architecture Declaration

```
architecture tb_arch of tb_goertzel_filter is
```

Figure 12. Declaration Snippet for Architecture

This line initiates the definition for architecture named **tb_arch** for the **tb_goertzel_filter** entity. It is to encapsulate the structure of the testbench and internal workings of the testbench.

Constants and Signal Declarations

```
    constant INPUT_FILE_PATH : string := "input_signal.txt";
    constant EXPECTED_FILE_PATH : string := "expected_magnitude.txt";

    signal clk : std_logic := '0';
    signal rst : std_logic := '0';
    signal input_signal : std_logic_vector(11 downto 0);
    signal magnitude : std_logic_vector(19 downto 0);
    signal expected_magnitude : std_logic_vector(19 downto 0);

    signal input_signal_unsigned : unsigned(11 downto 0);
    signal magnitude_signed : signed(19 downto 0);

    file input_signal_file : text open read_mode is INPUT_FILE_PATH;
    file expected_magnitude_file : text open read_mode is EXPECTED_FILE_PATH;
```

Figure 13. Constants and Signals

This section declares constants for file paths, signals for the clock, reset, input and output signals, and file handles for reading input and expected data. Constants *INPUT_FILE_PATH* and *EXPECTED_FILE_PATH* specify the paths to the corresponding the input signal and expected magnitude files, respectively. Various signals are declared for managing:

- the clock (*clk*)
- reset (*rst*)
- input signal (*input_signal*)
- computed magnitude (*magnitude*)
- expected magnitude (*expected_magnitude)*

Clock Process

```
    process
    begin
        wait for 10 ns;
        clk <= not clk;
    end process;
```

Figure 14. Code for Clock Process

The process logic is to generate a clock signal with a period of 20 nanoseconds by toggling the clk signal every 10 nanoseconds.

Reading Input and Expected Data

```
    process
        variable input_line : line;
        variable expected_line : line;
        variable input_value : integer;
        variable expected_value : integer;
    begin
        while not endfile(input_signal_file) loop
            readline(input_signal_file, input_line);
            read(input_line, input_value);
            input_signal <= std_logic_vector(to_unsigned(input_value,
input_signal'length));
            input_signal_unsigned <= to_unsigned(input_value,
input_signal_unsigned'length);
            wait until rising_edge(clk);
        end loop;

        while not endfile(expected_magnitude_file) loop
            readline(expected_magnitude_file, expected_line);
            read(expected_line, expected_value);
            expected_magnitude <= std_logic_vector(to_signed(expected_value,
magnitude'length));
            wait until rising_edge(clk);
        end loop;

        wait;
    end process;
```

Figure 15. File I/O Handling

As mentioned above, the snippet handles file operations, and the process reads data from the input signal file and expected magnitude file. In each line in the *input_signal_file***,** it reads an integer value, converts it to an unsigned type, and assigns it to *input_signal* and *input_signal_unsigned*. Also, it reads the expected magnitude values, converts them to a signed type, and assigns them to *expected_magnitude*. The process waits for a rising clock edge after reading each line.

Instantiating the Goertzel Filter

```
    goertzel_inst : entity work.goertzel_filter
        port map (
            clk => clk,
            rst => rst,
            input_signal => input_signal_unsigned,
            magnitude => magnitude_signed
        );
```

Figure 16. Goertzel Filter Instance

Here, an instance of the entity *goertzel_filter* has been initialized and mapped to the signals respectively. The filter processes the input *input_signal_unsigned* and produces a magnitude *magnitude_signed*.

Comparison Process

```
    process
    begin
        -- Wait for filter to settle
        wait for 100 ns;

        -- Compare magnitudes
        assert std_logic_vector(magnitude_signed) = expected_magnitude
            report "Mismatch between calculated and expected magnitude!"
            severity error;

         assert false report "Test done." severity note;

        wait;
    end process;
```

Figure 17. Comparison of Expected and Obtained Results

In this snippet the process waits for the goertzel filter to settle and later compares the magnitude with the expected magnitude. If the values mismatch, an error is reported. At the end of the code, it asserts a note meaning that the testing is done, and it waits indefinitely.

End of Architecture

```
end architecture tb_arch;
```

Figure18. End of Testbench Architecture

This line marks the end of the architecture definition.

Simulation Process

The last step is realized on EDA playground where GHDL (version 3.0) for simulation and VHDL language for testbench and design is used. The result of the simulation process is illustrated below:

```
[2024-07-10 12:40:22 UTC] ghdl -i  design.vhd testbench.vhd  && ghdl -m  goertzel_filter && ghdl -r  goertzel_filter
analyze testbench.vhd
elaborate goertzel_filter
Done
```

Figure19. Result of simulation

References

[1] P. Sysel and P. Rajmic, "Goertzel algorithm generalized to non-integer multiples of fundamental frequency," *EURASIP Journal on Advances in Signal Processing,* 2012.

[2] R. G. Lyons and R. G., "Understanding Digital Signal Processing," *Addison Wesley Pub. Co,* 1997.

[3] Q. Chaudhari, "Goertzel Algorithm – Evaluating DFT without DFT," Wireless Pi, [Online]. Available: https://wirelesspi.com/goertzel-algorithm-evaluating-dft-without-dft/.

[4] L. R. Rabiner and B. Gold, Theory and Application of Digital Signal Processing, 1975.

Appendix

A) Goertzel_Filter.m

```
function magnitude = Goertzel_Filter(signal, sampleRate, targetFreq)
    w = 2 * pi * targetFreq / sampleRate;
    coef = 2*cos(w);

    q1 = 0;
    q2 = 0;

    for n = 1:length(signal)
        q0 = signal(n) + coef * q1 - q2;
        q2 = q1;
        q1 = q0;
    end

    % real = q1 - q2*cos(w);
    % imag = q2*sin(w);
    % magnitude2 = real^2 + imag^2;

    %Optimized Goertzel
    magnitude = q1^2+q2^2-coef*q2*q1;

end
```

B) Testing.m

```
function testing()

    freqsin = [5000, 149000, 150000, 151000, 200000];
    freqrect = [10000, 16000, 150000, 200000];
    freqtriangle = [5000, 149000, 150000, 151000, 200000];
    all_frequencies = {freqsin, freqrect, freqtriangle};
    degrees = [0, 30, 45, 90, 120];
    wave_names = ["Sine", "Square", "Triangle"];
    input_bit = 12;

    sample_rate = 4000000;
    target_freq = 150000;
```

```
N = 135;
t = 0:1/sample_rate:(N-1)/sample_rate;



sinwave_magnitudes = zeros(length(freqsin), length(degrees));
squarewave_magnitudes = zeros(length(freqrect), length(degrees));
trianglewave_magnitudes = zeros(length(freqtriangle), length(degrees));

all_magnitudes = {sinwave_magnitudes, squarewave_magnitudes, trianglewave_magnitudes};




for k = 1:length(all_frequencies)
   frequencies = all_frequencies{k};
   wave_magnitudes = all_magnitudes{k};
   wave_name = wave_names(k);

   for i = 1:length(degrees)
      for j = 1:length(frequencies)

         deg = degrees(i);
         freq = frequencies(j);

         sinewave = sin(2*pi*freq*t + deg2rad(deg))*(2^input_bit-1);
         squarewave = square(2*pi*freq*t + deg2rad(deg))*(2^input_bit-1);
         trianglewave = sawtooth(2*pi*freq*t + deg2rad(deg))*(2^input_bit-1);


         all_waves = {sinewave, squarewave, trianglewave};

         wave_magnitudes(j, i) = Goertzel_Filter( all_waves{k}, sample_rate, target_freq);
         save_to_file("input", wave_name, freq, deg, all_waves{k});
         save_to_file("result", wave_name, freq, deg, wave_magnitudes(j, i));

      end
   end

   subplot(3,1,k)
   bar(wave_magnitudes, 'grouped');
   hold on
   title("Magnitudes for " + wave_name + " Waves");
   legendInfo = arrayfun(@(deg) [num2str(deg), ' Degree'], degrees, 'UniformOutput', false);
   legend(legendInfo, "Location", "bestoutside");
   xticklabels(arrayfun(@num2str, frequencies, 'UniformOutput', false));
```

```
    end
end



function save_to_file(folder_name, wave_name, freq, degree, data)

    if ~exist(folder_name, 'dir')
        mkdir(folder_name);
    end

    filename = sprintf('%s/%s_%ddegree_%dHz.txt', folder_name, lower(wave_name), degree,
freq);

    if exist(filename, 'file')
        delete(filename);
    end

    fileID = fopen(filename, 'w');
    fprintf(fileID, '%d\n', uint16(round(data)));
    fclose(fileID);

end
```

C) Design.vhd:

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

entity goertzel_filter is
    generic (
        coef : signed(19 downto 0) := to_signed(19447, 20);
        coef_div : signed(19 downto 0) := to_signed(10000, 20)
    );
    port (
        input_signal : in unsigned (11 downto 0);
        clk : in STD_LOGIC;
        rst : in STD_LOGIC;
        magnitude : out signed(19 downto 0)
    );
end entity goertzel_filter;
```

```
architecture behavioural of goertzel_filter is
    signal q0, q1, q2 : signed(19 downto 0);
    signal x : signed(11 downto 0);
    signal result : signed(19 downto 0);
begin
    process (clk, rst)
    begin
        if rst = '1' then

            q0 <= (others => '0');
            q1 <= (others => '0');
            q2 <= (others => '0');
            result <= (others => '0');

        elsif rising_edge(clk) then

            x <= signed(resize(input_signal, x'length));
            q0 <= x + resize(coef * q1 / coef_div, q0'length) - q2;
            q2 <= q1;
            q1 <= q0;

            --Optimized Goertzel
            result <= resize(q1 * q1 + q2 * q2 - resize(coef * q2 * q1 / coef_div, result'length),
result'length);

        end if;
    end process;

    magnitude <= result;
end architecture behavioural;
```

D) Testbench.vhd:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;

entity tb_goertzel_filter is
end entity tb_goertzel_filter;
```

```
architecture tb_arch of tb_goertzel_filter is
    -- Constants for file paths and scaling factors
    constant INPUT_FILE_PATH : string := "input_signal.txt";
    constant EXPECTED_FILE_PATH : string := "expected_magnitude.txt";

    signal clk : std_logic := '0';
    signal rst : std_logic := '0';
    signal input_signal : std_logic_vector(11 downto 0);
    signal magnitude : std_logic_vector(19 downto 0);
    signal expected_magnitude : std_logic_vector(19 downto 0);

    signal input_signal_unsigned : unsigned(11 downto 0);
    signal magnitude_signed : signed(19 downto 0);

    file input_signal_file : text open read_mode is INPUT_FILE_PATH;
    file expected_magnitude_file : text open read_mode is EXPECTED_FILE_PATH;
begin
    -- Clock process
    process
    begin
        wait for 10 ns;
        clk <= not clk;
    end process;

    -- Read input signal and expected magnitude
    process
        variable input_line : line;
        variable expected_line : line;
        variable input_value : integer;
        variable expected_value : integer;
    begin
        while not endfile(input_signal_file) loop
            readline(input_signal_file, input_line);
            read(input_line, input_value);
            input_signal <= std_logic_vector(to_unsigned(input_value, input_signal'length));
            input_signal_unsigned <= to_unsigned(input_value, input_signal_unsigned'length);
            wait until rising_edge(clk);
        end loop;

        while not endfile(expected_magnitude_file) loop
            readline(expected_magnitude_file, expected_line);
            read(expected_line, expected_value);
            expected_magnitude <= std_logic_vector(to_signed(expected_value, magnitude'length));
            wait until rising_edge(clk);
        end loop;
```

```
        wait;
    end process;

    -- Instantiate Goertzel filter
    goertzel_inst : entity work.goertzel_filter
        port map (
            clk => clk,
            rst => rst,
            input_signal => input_signal_unsigned,
            magnitude => magnitude_signed
        );

    -- Compare output with expected magnitude
    process
    begin
        -- Wait for filter to settle
        wait for 100 ns;

        -- Compare magnitudes
        assert std_logic_vector(magnitude_signed) = expected_magnitude
            report "Mismatch between calculated and expected magnitude!"
            severity error;

         assert false report "Test done." severity note;

        wait;
    end process;

end architecture tb_arch;
```