**HO CHI MINH CITY NATIONAL UNIVERSITY**

**HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY**

**FACULTY OF ELECTRICAL ELECTRONICS**

---------------o0o---------------

****

**INTERNSHIP REPORT**

**DESIGN, IMPLEMENTATION OF FINITE IMPULSE RESPONSE (FIR) FILTERS AND WAVEFORM GENERATOR USING FPGA TECHNOLOGY**

**Instructor: Assoc.Prof.Dr. Truong Quang Vinh**

**Student: Nguyen Quoc Tin**

**Student ID: 2213496**

**HO CHI MINH CITY, AUGUST 2025**

***Acknowledgements***

First and foremost, I would like to express my sincere gratitude to Mr. Trương Quang Vinh, lecturer at the Faculty of Electrical and Electronics Engineering, Ho Chi Minh City University of Technology. His support and permission for me to independently choose and pursue a topic in my own direction provided both freedom and motivation. His trust and encouragement were a great source of inspiration that empowered me to explore new ideas, foster creativity, and develop my self-learning ability.

The completion of this project is not only an academic achievement but also a valuable experience that has helped me mature in independent research and problem-solving. I would also like to extend my heartfelt thanks to everyone who, directly or indirectly, contributed to the accomplishment of this work.

**HO CHI MINH CITY, AUGUST 15, 2025**

Student signature

**Project Summary**

The primary objective of this project is to design and implement a versatile waveform generator using an FPGA. The generator should be capable of producing various types of waveforms with adjustable parameters such as frequency, amplitude, and duty cycle. Additionally, the system should be able to inject noise into these waveforms. The final implementation will be demonstrated on an FPGA using the DE10 Standard Development Kit.

The next objective of this project is to design, implement, and test a Finite Impulse Response (FIR) filter using FPGA technology. The goal is to gain a solid understanding of the theory behind FIR filters, apply digital filter design techniques, and acquire hands-on experience in hardware design and implementation.

CONTENTS

[1. INTRODUCTION 6](#_Toc206144712)

[1.1. Overview 6](#_Toc206144713)

[1.2. Assigned Objectives 7](#_Toc206144714)

[2. THEORETICAL BACKGROUND 8](#_Toc206144715)

[2.1. Introduction of the DE10-Standard Board 8](#_Toc206144716)

[2.1.1. Layout and Components 8](#_Toc206144717)

[2.1.2. Block Diagram of the DE10-Standard Board 10](#_Toc206144718)

[2.2. Direct Digital Synthesis (DDS) 12](#_Toc206144719)

[2.3. Phase accumulator and how to generate waveform 15](#_Toc206144720)

[2.4. Linear-feedback shift register (LFSR) 21](#_Toc206144721)

[2.5. I2C protocol 24](#_Toc206144722)

[2.6. I2S communication protocol 30](#_Toc206144723)

[2.7.  Finite impulse response (FIR) filter 32](#_Toc206144724)

[3. HARDWARE DESIGN AND IMPLEMENTATION 39](#_Toc206144725)

[3.1. dds\_accumulator 41](#_Toc206144726)

[3.2. rom\_sin 41](#_Toc206144727)

[3.3. rom\_ecg 42](#_Toc206144728)

[3.4. square\_gen 43](#_Toc206144729)

[3.5. triangle\_gen 44](#_Toc206144730)

[3.6. saw\_gen 45](#_Toc206144731)

[3.7. param\_ctrl 45](#_Toc206144732)

[3.8. lfsr\_noise 47](#_Toc206144733)

[3.9. waveform\_mux 48](#_Toc206144734)

[3.10. noise\_injector 49](#_Toc206144735)

[3.11. amplitude\_scaler 50](#_Toc206144736)

[3.12. symmetric\_fir\_pipelined 51](#_Toc206144737)

[3.13. i2c\_cfg 51](#_Toc206144738)

[3.14. i2s\_tx 53](#_Toc206144739)

[3.15. pll\_12m88 55](#_Toc206144740)

[4. IMPLEMENTATION RESULTS 56](#_Toc206144741)

[5. CONCLUSION AND FUTURE DEVELOPMENT 63](#_Toc206144742)

[6. REFERENCES 69](#_Toc206144743)

TABLE OF FIGURES

[Figure 1. DE10-Standard development board. 8](#_Toc206122933)

[Figure 2. Block diagram of DE10-Standard. 10](#_Toc206122934)

[Figure 3. Functional block diagram of a DDS system. 13](#_Toc206122935)

[Figure 4. Typical DDS architecture and signal path with DAC. 14](#_Toc206122936)

[Figure 5. Digital phase wheel. 16](#_Toc206122937)

[Figure 6. Block diagram of a DDS architecture using LUT and phase truncation. 18](#_Toc206122938)

[Figure 7. Block diagram of a DDS system. 18](#_Toc206122939)

[Figure 8. Fibonacci LFSR. 22](#_Toc206122940)

[Figure 9. Generate a new bit using feedback from the tap bits. 23](#_Toc206122941)

[Figure 10. Generalized I2C Connection Diagram. 25](#_Toc206122942)

[Figure 11. Messages are broken up into frames of data. 25](#_Toc206122943)

[Figure 12. Start Condition And Stop Condition Transitions. 26](#_Toc206122944)

[Figure 13. 2-Wire Serial Interface of WM8731. 29](#_Toc206122945)

[Figure 14. I2S mode. 30](#_Toc206122946)

[Figure 15. Master mode. 31](#_Toc206122947)

[Figure 16. Symmetric Linear-Phase FIR Filter Diagram (8-tap) with Pipelined Architecture. 34](#_Toc206122948)

[Figure 17. Traditional FIR Filter Adder Tree. 36](#_Toc206122949)

[Figure 18. General block diagram of a waveform generator system. 40](#_Toc206122950)

[Figure 19. The block symbol of dds\_accumulator. 41](#_Toc206122951)

[Figure 20. The block symbol of rom\_sin. 42](#_Toc206122952)

[Figure 21. The block symbol of rom\_ecg. 43](#_Toc206122953)

[Figure 22. The block symbol of square\_gen. 44](#_Toc206122954)

[Figure 23. The block symbol of triangle\_gen. 45](#_Toc206122955)

[Figure 24. The block symbol of saw\_gen. 45](#_Toc206122956)

[Figure 25. The block symbol of param\_ctrl. 47](#_Toc206122957)

[Figure 26. The block symbol of lfsr\_noise. 48](#_Toc206122958)

[Figure 27. The block symbol of waveform\_mux. 48](#_Toc206122959)

[Figure 28. The block symbol of noise\_injector. 49](#_Toc206122960)

[Figure 29. Noise-injected signal. 50](#_Toc206122961)

[Figure 30. The block symbol of amplitude\_scaler. 51](#_Toc206122962)

[Figure 31. The block symbol of symmetric\_fir\_pipelined. 51](#_Toc206122963)

[Figure 32. Master mode connection. 52](#_Toc206122964)

[Figure 33. 2-Wire Serial Interface of WM8731. 53](#_Toc206122965)

[Figure 34. The block symbol of i2c\_cfg. 53](#_Toc206122966)

[Figure 35. I2S mode. 54](#_Toc206122967)

[Figure 36. The block symbol of i2s\_tx. 55](#_Toc206122968)

[Figure 37. The block symbol of pll\_12m88. 55](#_Toc206122969)

[Figure 38. Sine Waveform. 56](#_Toc206122970)

[Figure 39. ECG Waveform. 57](#_Toc206122971)

[Figure 40. Square Waveform. 57](#_Toc206122972)

[Figure 41. Triangle Waveform. 58](#_Toc206122973)

[Figure 42. Sawtooth Waveform. 59](#_Toc206122974)

[Figure 43. Inject noise into the sine wave. 60](#_Toc206122975)

[Figure 44. Filter noise from the sine wave. 60](#_Toc206122976)

[Figure 45. Waveform of the I²C protocol. 61](#_Toc206122977)

[Figure 46. Waveform of the I²S protocol. 62](#_Toc206122978)

[Figure 47. Data transmission result. 62](#_Toc206122979)

# INTRODUCTION

## 1.1. Overview

In the field of digital electronics and signal processing, FPGA (Field Programmable Gate Array) technology offers powerful capabilities for high-speed and parallel execution of complex signal processing algorithms. The decision to pursue the topic **"Design and Implementation of Finite Impulse Response (FIR) Filters and a Waveform Generator Using FPGA Technology"** is grounded in two primary objectives: generating test signals and performing digital noise filtering — both of which are fundamental to the analysis and development of signal processing systems.

The first objective is the design of a waveform generator capable of producing adjustable signals in terms of frequency, amplitude, and waveform type. This module serves as a flexible and reusable tool for testing and validating digital signal processing systems. In practical engineering scenarios, having a configurable signal source is essential for simulating real-world conditions and evaluating system performance.

The second objective focuses on the implementation of a Finite Impulse Response (FIR) filter to remove unwanted noise components from input signals. FIR filters are widely used in communications, instrumentation, audio processing, and medical systems due to their linear-phase properties and inherent stability. Designing an FIR filter on FPGA not only provides insight into the theoretical aspects of digital filter design but also offers hands-on experience with hardware-level signal processing.

Combining both signal generation and noise filtering within a single FPGA-based system brings together theory and practice in a cohesive and meaningful way. This integration enhances students’ understanding of system-level design, hardware-software co-development, and real-time digital processing. Moreover, the project helps build essential skills such as HDL programming (e.g., Verilog or VHDL), modular hardware design, timing analysis, and functional verification — all of which are highly valuable in modern embedded and DSP-oriented careers.

## 1.2. Assigned Objectives

* Survey the hardware used in the project; read and study the datasheets of all selected components.
* Implement a top-level system architecture, then decompose it into modular blocks, each responsible for a specific function.
* Study the I²C protocol for WM8731 configuration and the I²S audio protocol for streaming audio data.
* Implement the waveform-generation logic for each waveform type, then provide run-time controls for amplitude, frequency, and duty cycle.
* Extend the system by injecting noise into the waveform and designing/applying an FIR filter to remove the injected noise.
* Develop comprehensive testbenches to verify and simulate each module’s functionality.
* Deploy and validate the design on the target development kit (hardware-in-the-loop) and confirm that measured results match simulation.

# THEORETICAL BACKGROUND

## 2.1. Introduction of the DE10-Standard Board

2.1.1. Layout and Components

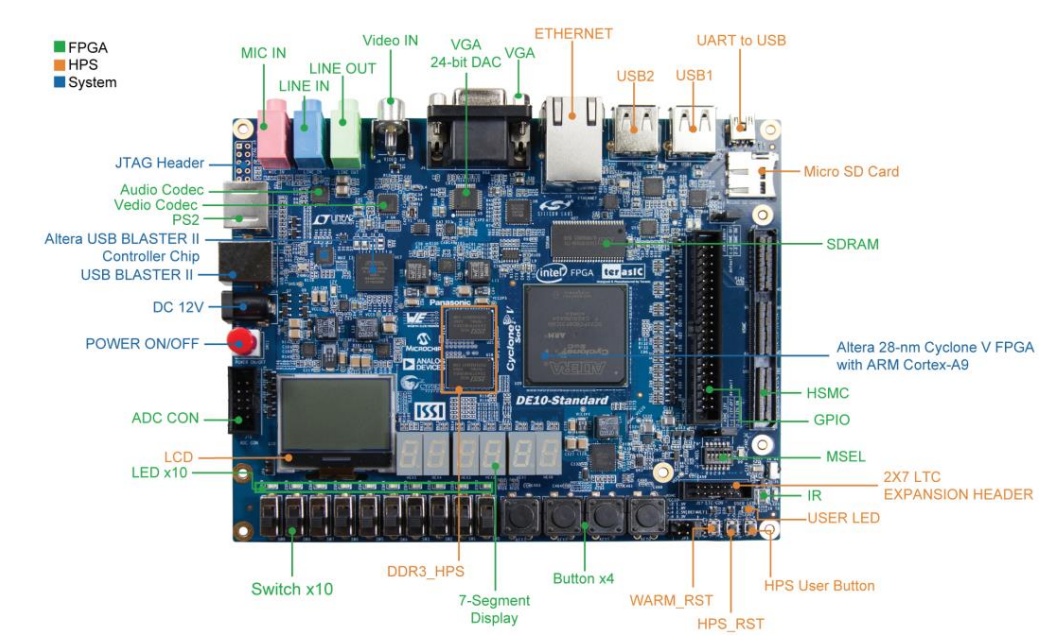


Figure 1. DE10-Standard development board.

The DE10-Standard board has many features that allow users to implement a wide range of designed circuits, from simple circuits to various multimedia projects.

The following hardware is provided on the board:

* + - **FPGA**
      * Altera Cyclone® V SE 5CSXFC6D6F31C6N device
      * Altera serial configuration device – EPCS128
      * USB-Blaster II onboard for programming; JTAG Mode
      * 64MB SDRAM (16-bit data bus)
      * 4 push-buttons
      * 10 slide switches
      * 10 red user LEDs
      * Six 7-segment displays
      * Four 50MHz clock sources from the clock generator
      * 24-bit CD-quality audio CODEC with line-in, line-out, and microphone-in jacks
      * VGA DAC (8-bit high-speed triple DACs) with VGA-out connector
      * TV decoder (NTSC/PAL/SECAM) and TV-in connector
      * PS/2 mouse/keyboard connector
      * IR receiver and IR emitter
      * One HSMC with Configurable I/O standard 1.5/1.8/2.5/3.3
      * One 40-pin expansion header with diode protection
      * A/D converter, 4-pin SPI interface with FPGA
    - **HPS (Hard Processor System)**
      * 800MHz Dual-core ARM Cortex-A9 MPCore processor
      * 1GB DDR3 SDRAM (32-bit data bus)
      * 1 Gigabit Ethernet PHY with RJ45 connector
      * 2-port USB Host, normal Type-A USB connector
      * Micro SD card socket
      * Accelerometer (I2C interface + interrupt)
      * UART to USB, USB Mini-B connector
      * Warm reset button and cold reset button
      * One user button and one user LED
      * LTC 2x7 expansion header
      * 128x64 dots LCD Module with Backlight

### 2.1.2. Block Diagram of the DE10-Standard Board

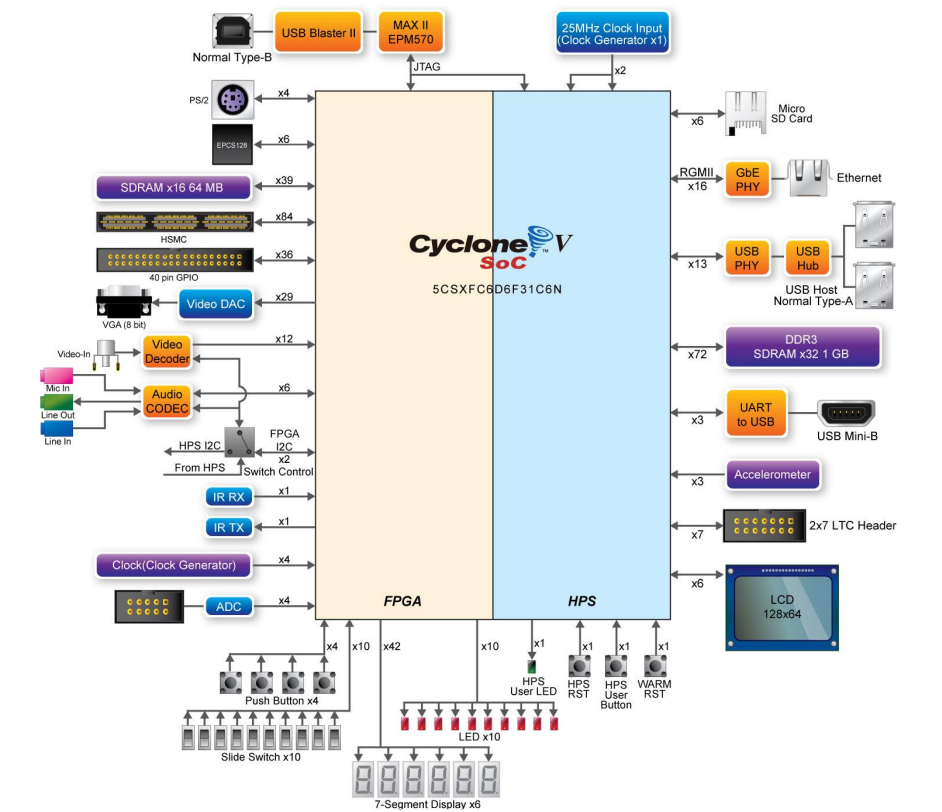


Figure 2. Block diagram of DE10-Standard.

All the connections are established through the Cyclone V SoC FPGA device to provide maximum flexibility for users. Detailed information about Figure 2:

* + - **FPGA Device**
      * Cyclone V SoC 5CSXFC6D6F31C6N Device
      * Dual-core ARM Cortex-A9 (HPS)
      * 110K programmable logic elements
      * 5,140 Kbits embedded memory
      * 6 fractional PLLs
      * 2 hard memory controllers
      * 3.125G transceivers
    - **Configuration and Debug**
      * Quad serial configuration device – EPCS128 on FPGA
      * Onboard USB-Blaster II (normal type B USB connector)
    - **Memory Device** 
      * 64MB (32Mx16) SDRAM on FPGA
      * 1GB (2x256Mx16) DDR3 SDRAM on HPS
      * Micro SD card socket on HPS
    - **Communication**
      * Two port USB 2.0 Host (ULPI interface with USB type A connector)
      * UART to USB (USB Mini-B connector)
      * 10/100/1000 Ethernet
      * PS/2 mouse/keyboard
      * IR emitter/receiver
      * I2C multiplexer
    - **Connectors**
      * One HSMC (8-channel Transceivers, Configurable I/O standards 1.5/1.8/2.5/3.3V)
      * One 40-pin expansion headers
      * One 10-pin ADC input header
      * One LTC connector (one Serial Peripheral Interface (SPI) Master ,one I2C and one GPIO interface )
    - **Display**
      * 24-bit VGA DAC
      * 128x64 dots LCD Module with Backlight
    - **Audio**

• 24-bit CODEC, Line-in, Line-out, and microphone-in jacks

* + - **Video Input**
      * TV decoder (NTSC/PAL/SECAM) and TV-in connector
    - **ADC**
      * Interface: SPI
      * Fast throughput rate: 500 KSPS
      * Channel number: 8
      * Resolution: 12-bit
      * Analog input range : 0 ~ 4.096
    - **Switches, Buttons, and Indicators**
      * 5 user Keys (FPGA x4, HPS x1)
      * 10 user switches (FPGA x10)
      * 11 user LEDs (FPGA x10, HPS x 1)
      * 2 HPS reset buttons (HPS\_RESET\_n and HPS\_WARM\_RST\_n)
      * Six 7-segment displays
    - **Sensors**
      * G-Sensor on HPS
    - **Power**
      * 12V DC input

**2.2. Direct Digital Synthesis (DDS)**

DDS is a method of generating different Analog waveforms using digital techniques. It operates by saving the points of a waveform in its digital form. Then it reconstructs the waveform by recalling these digital data. It is a technique which uses digital data and analog signal processing blocks to generate signal waveforms that are repetitive in nature. Many years ago, these digital synthesizers were limited by the clock frequency, which would hinder the speed of operation of the digital logic. Now, with increasing frequency, the limits of the DDS are also increasing.

The technique uses digital data processing to generate a frequency- and phase-tunable output related to a fixed frequency reference, or clock source. In a DDS architecture, the reference or system clock frequency is divided down by the scaling factor, set by a programmable binary tuning word. The tuning word is typically 24-48 bits long which enables a DDS implementation to provide superior output frequency tuning resolution.

Today’s cost-competitive, high-performance, functionally-integrated, and small package-sized DDS products are fast becoming an alternative to traditional frequency-agile analog synthesizer solutions. The integration of a high-speed, high-performance, D/A converter and DDS architecture onto a single chip (forming what is commonly known as a Complete-DDS solution) enabled this technology to target a wider range of applications and provide, in many cases, an attractive alternative to analog-based PLL synthesizers. For many applications, the DDS solution holds some distinct advantages over the equivalent agile analog frequency synthesizer employing PLL circuitry.

Simply stated, a direct digital frequency synthesizer translates a train of clock pulses into an analog waveform, typically a sine, triangular, or square wave. As Figure 1 shows, its essential parts are: a *phase accumulator*, which produces a number corresponding to a phase angle of the output waveform, a *phase-to-digital converter(LUT)*, which generates the instantaneous digital fraction of the output amplitude occurring at a particular phase angle, and a *digital-to-analog converter* (DAC), which converts that digital value to a sampled analog data point.

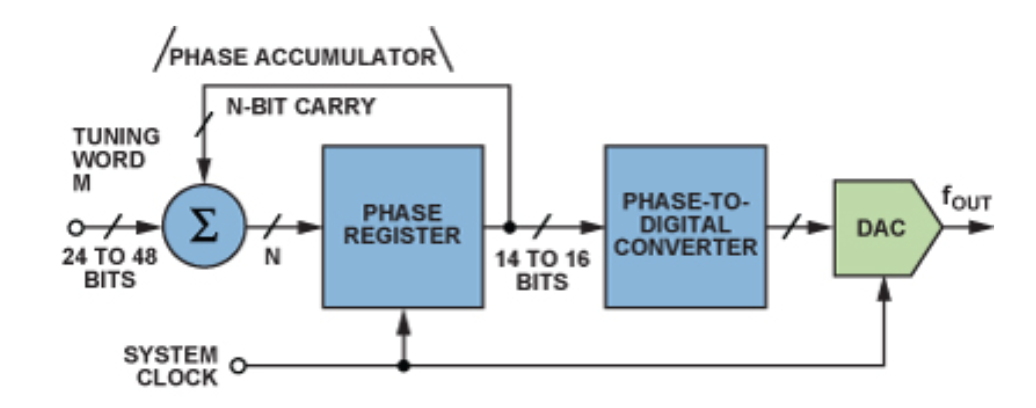


Figure 3. Functional block diagram of a DDS system.

On each clock cycle, the phase accumulator adds a fixed value known as the **frequency tuning word (fcw)**. The result is a digital ramp representing phase over time, which is used to address the LUT. The LUT stores samples of the desired waveform (typically a sine wave), and the output is then converted to analog by the DAC and smoothed by the LPF.

For sine-wave outputs, the phase-to-digital converter is usually a sine lookup table (Figure 4).

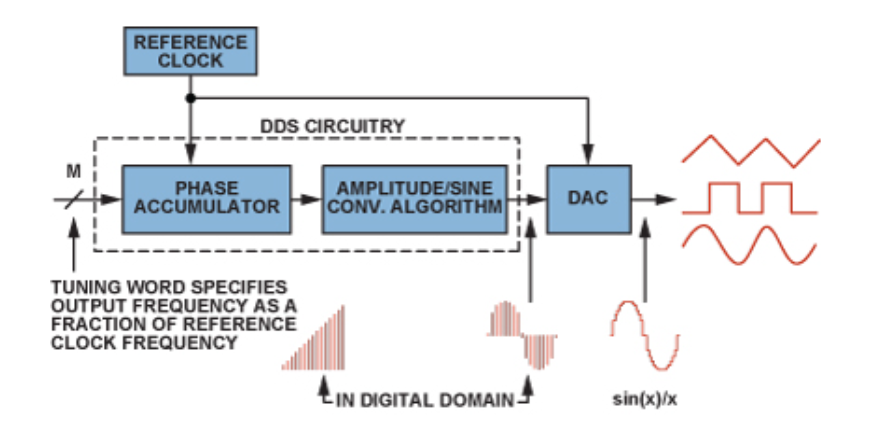


Figure 4. Typical DDS architecture and signal path with DAC.

***DDS advantages:***

* Micro-Hertz tuning resolution of the output frequency and sub-degree phase tuning capability, all under complete digital control.
* Extremely fast “hopping speed” in tuning output frequency (or phase), phase-continuous frequency hops with no over/undershoot or analog-related loop settling time anomalies.
* The DDS digital architecture eliminates the need for the manual system tuning and tweaking associated with component aging and temperature drift in analog synthesizer solutions.
* The digital control interface of the DDS architecture facilitates an environment where systems can be remotely controlled, and minutely optimized, under processor control.
* When utilized as a quadrature synthesizer, DDS afford unparalleled matching and control of I and Q synthesized outputs.

## 2.3. Phase accumulator and how to generate waveform

The **Phase Accumulator** is an *n*-bit register that operates in modulo 2ⁿ arithmetic. At each clock cycle, it adds a fixed value M — known as the **Frequency Control Word (FCW)** or **tuning word** — to its current value. This incremental process controls the phase progression rate, which in turn determines the frequency of the output waveform.

* FCW = M is the phase step added each clock cycle.
* A larger FCW causes the phase to increase more rapidly, resulting in a higher output frequency.
* A smaller FCW results in slower phase accumulation, producing a lower frequency waveform.

**Operating Principle:**

At each rising edge of the system clock, the phase accumulator adds the FCW to its current phase value. This accumulation results in a steady increasing phase signal — forming a digital ramp that cyclically progresses from 0 to 2n-1. When the maximum value is reached, the accumulator overflows and wraps around to 0, similar to a clock hand completing a full revolution. This mechanism creates a “digital phase wheel” as shown in Figure 5.

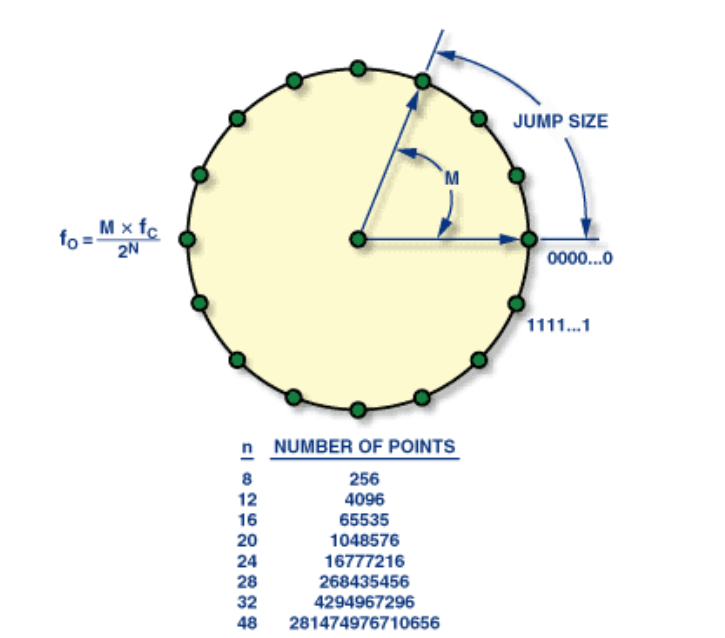


Figure 5. Digital phase wheel.

This sequence of values represents the phase of the waveform, and the rate of phase increment directly determines the output frequency, following the relationship:

fout = fclk

* where:
* *n* is number of bits of phase accumulator (24 bits to 48 bits).
* *M* is Frequency Control Word value.
* fclk is System clock frequency (Hz).
* fout is desired output frequency (Hz).

The output of the phase accumulator, which is the instantaneous phase of the output waveform, is used to drive ***the phase to amplitude converter (LUT).*** The phase to amplitude converter outputs a digital word, the value of which is the amplitude of the sine waveform for the input phase.

For complex waveforms such as sine or ECG, the phase value generated by the phase accumulator (a sequence of numbers) does not directly represent a sine wave. To convert it into a sine waveform, a ***Look-Up Table (LUT)*** is used. The LUT stores precomputed amplitude values corresponding to discrete phase angles of the sine wave.

Each time the phase accumulator updates, the **k most significant bits (MSBs)** of the phase value (typically k ≈ 10 to 16) are extracted and used as the address:

**addr = phase[n-1 : n-k]**

This address is then used to retrieve the corresponding amplitude from the LUT. The resulting value is sent to a ***Digital-to-Analog Converter (DAC)*** to produce the final analog waveform, as previously described.

Note that the number of bits used to control the phase-to-amplitude conversion is less than the number of bits in the phase accumulator. This process is called ***truncation of the phase accumulator***, and is used to reduce chip area and power consumption in the digital stages following the accumulator. Although phase truncation introduces additional spectral components—called truncated pulses—these effects can be minimized through careful system design.

Clipping the lower bits significantly reduces the LUT size while maintaining high frequency resolution. This phase clipping mechanism is illustrated in the block diagram below, which shows the interaction between the phase accumulator, phase clipping, LUT, and DAC.

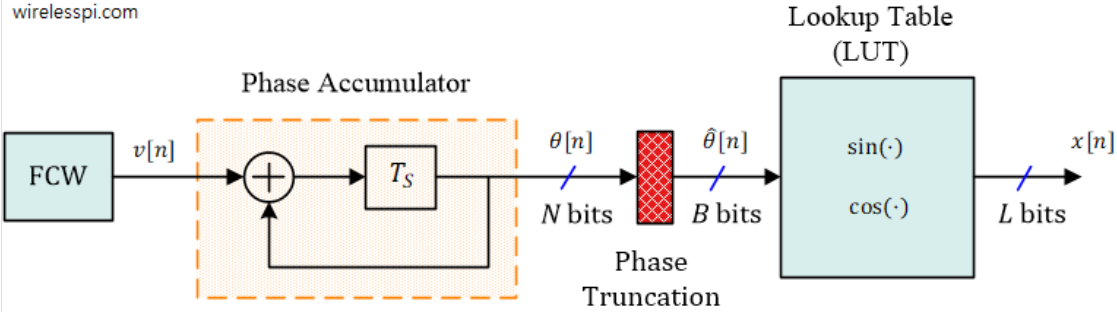
****

Figure 6. Block diagram of a DDS architecture using LUT and phase truncation.

Moreover, for simple waveforms such as square, triangle, or sawtooth waves, the use of a LUT is not required. Instead, the waveform can be computed directly from the phase value generated by the phase accumulator. Here are some ways to generate a few types of waves

**Common Waveforms Generated by DDS**

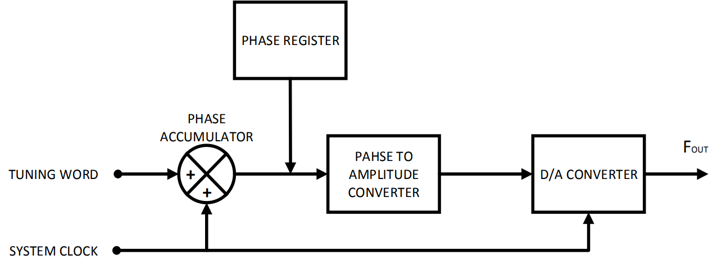


Figure 7. Block diagram of a DDS system.

Basically, waveform generation using the DDS principle follows the block diagram shown in Figure 6 or as previously discussed. Below is the method I used to generate each waveform type:

* **Sine Wave**:

The sine wave is generated by storing sampled sine values in a Look-Up Table (LUT) and using a phase accumulator as the address index. At each clock cycle, the phase value increases by a fixed step M, pointing to a new amplitude value in the LUT. This approach leverages the symmetry of the sine function to reduce LUT size, yet still requires storing and reading amplitude values. The LUT method is chosen because real-time sine computation in hardware is complex and resource-intensive. As a result, the output sine wave is smooth and highly accurate. This method also allows easy adjustments of phase and amplitude by changing control words and data frames.

* **Square Wave**:

Generating a square wave simply involves converting the phase accumulator value into alternating high/low logic levels. Specifically, the most significant bit (MSB) of the phase accumulator or the sine amplitude value can be used to determine the output as either +1 or -1 (or 0 and Vmax). For example, many DDS chips such as the AD9833 output square waves directly from the MSB of the DAC. Therefore, generating a square wave requires no LUT—just a simple control bit suffices for the two-level signal. However, square waves contain a rich harmonic spectrum and require significant filtering to avoid signal distortion. Theoretically, square wave frequencies should remain below half the reference clock frequency to limit high-order harmonics. Despite their simplicity, the noisy harmonic content of square waves makes them suitable mainly when used with proper filtering.

The method for generating a square wave is quite straightforward: it involves producing two levels — high and low — by comparing the value of the phase accumulator with a defined threshold (duty cycle). If the accumulated phase value is less than the threshold, the output is set to a high level, allowing for easy adjustment of the duty cycle. Otherwise, the output is set to a low level.

* **Sawtooth Wave**:

As the phase accumulator increases linearly and resets, feeding its full n-bit output directly to the DAC results in a sawtooth waveform.

This method requires no LUT—the DAC input is simply the scaled phase value. Sawtooth waves linearly ramp in amplitude from one peak to the next, directly reflecting the phase progression. The greater the number of bits n, the smoother the waveform, limited only by DAC resolution.

Specifically, this method takes advantage of the fact that with each clock cycle, the phase accumulator is linearly incremented by a fixed value M, gradually increasing until it reaches 2n-1. Due to this linear increase, the resulting values form a ramp-like waveform. Once the accumulator exceeds 2n-1, it immediately overflows and wraps back to 0. At this point, a sawtooth wave is generated — characterized by a linear rise to the maximum value followed by a sudden drop to zero. Repeating this process continuously produces a periodic sawtooth waveform.

* **Triangle Wave:**

A triangle wave is essentially a bidirectional version of a sawtooth wave — it linearly ramps up and then ramps down. Therefore, the method for generating a triangle wave is quite similar to that of a sawtooth wave: the phase accumulator is used to linearly increase up to 2n-1, after which its output is either bitwise inverted or processed using arithmetic operations to count downward, producing the falling edge.

Some DDS chips (e.g., the AD9833) generate triangle waves directly by sequentially processing the phase accumulator in an up-then-down pattern (from 0 to 2n-1 and back to 0 in a repeating cycle). Like square waves, triangle waves can also be generated without the use of a LUT, relying instead on arithmetic logic. However, the output quality is highly dependent on the DAC’s bandwidth and linearity..

* **Arbitrary Waveform (e.g., ECG Signal)**:

Complex waveforms such as ECG (electrocardiogram) signals do not follow simple mathematical formulas. To reproduce them, sampled waveform data must be stored in a LUT. For instance, an ECG signal contains P, QRS, and T components with varying amplitudes and durations. Therefore, an ECG-simulating DDS usually stores representative samples of these components in a LUT. During operation, a microcontroller or FPGA sequentially reads these samples from the LUT and feeds them to the DAC, forming a complete ECG cycle. This method allows the generation of any periodic waveform, provided there is sufficient memory for storing the samples. Thus, arbitrary signals (e.g., ECG, audio samples) are best generated using LUTs due to the lack of efficient mathematical representations.

* **Summary – Why Some Waveforms Use LUTs and Others Don’t**:

Each waveform type has its own characteristics. Sine and arbitrary waveforms require precise amplitude data and therefore are usually stored in LUTs. In contrast, basic waveforms like square, sawtooth, and triangle can be efficiently generated using simple arithmetic or bitwise operations, eliminating the need for LUTs. Avoiding LUTs saves memory but may result in less smooth output (due to harmonic distortion) or frequency limitations, as previously discussed.

## 2.4. Linear-feedback shift register (LFSR)

A **linear-feedback shift register** (**LFSR**) is a shift register whose input bit is a linear function of its previous state. The most used linear function of single bits is exclusive-or (XOR). Thus, an LFSR is most often a shift register whose input bit is driven by the XOR of some bits of the overall shift register value.

The initial value of the LFSR is called the seed, and because the operation of the register is deterministic, the stream of values produced by the register is completely determined by its current (or previous) state. Likewise, because the register has a finite number of possible states, it must eventually enter a repeating cycle. However, an LFSR with a well-chosen feedback function can produce a sequence of bits that appear at random and have a very long cycle.

Applications of LFSRs include generating pseudo-random numbers, pseudo-noise sequences, fast digital counters, and whitening sequences...Both hardware and software implementations of LFSRs are common.

There are two types of LFSR: Fibonacci LFSR (External Feedback) and Galois LFSR (Internal Feedback). In practice, we often refer to both simply as LFSR, without specifying the type. **This time, I used the Fibonacci pattern.**

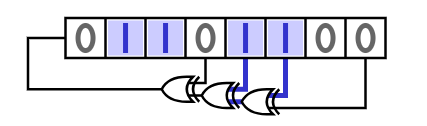


Figure 8. Fibonacci LFSR.

It's clear that the linear feedback operations are performed outside the shift register.

When data is shifted left or right, the new bits are updated based on the previous bits. At the same time, the bit selected as the output is taken as the MSB of the PRBS, and the shift register effectively loses one bit.

To compensate, a new bit (the feedback bit) is generated and inserted into the input of the register by XORing or XNORing the selected tap bits outside the register (i.e., bits chosen to perform XOR/XNOR).

What distinguishes a Fibonacci LFSR is that it uses only one common XOR/XNOR gate to process all tap bits at once. For example, consider two 3-bit XOR based LFSRs with different tap selections (Figure 9).

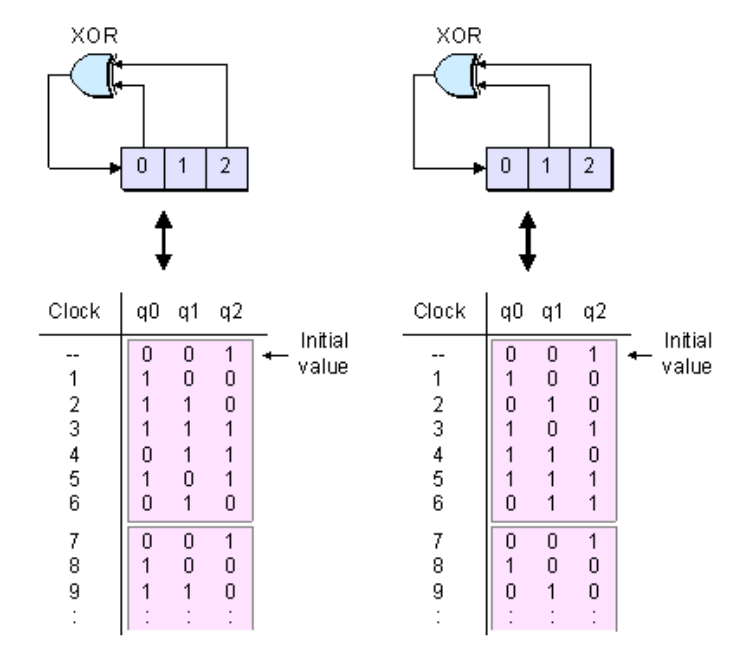


Figure 9. Generate a new bit using feedback from the tap bits.

We can only achieve a maximum sequence length of 2n - 1 and not 2n. The reason lies in two special cases:

1. The register contains **all 0s**, and XOR is used.
2. The register contains **all 1s**, and XNOR is used.

When XORing with 0 or XNORing with 1, the result is always the same as the input bit.

Therefore, if the seed (register value) is initialized with all bits as 0 (with an XOR gate) or all bits as 1 (with an XNOR gate), the result will still be the same — it becomes an **infinite loop**.

Hence, we conclude that **regardless of tap bit selection**, these initial conditions will always produce infinite repetition. The solution is simple:

* Always initialize with a non-zero value for XOR feedback.
* Always initialize with a non-one value for XNOR feedback.

In this project, a Fibonacci LFSR is used to generate noise.

## 2.5. I2C protocol

I2C stands for **Inter-Integrated Circuit.**It is a bus interface connection protocol incorporated into devices for serial communication. It was originally designed by Philips Semiconductor in 1982. Recently, it is a widely used protocol for short-distance communication. It is also known as Two Wired Interface (TWI).

It uses only 2 bi-directional open-drain lines for data communication called SDA and SCL. Both these lines are pulled high.

**Serial Data (SDA):**Transfer of data takes place through this pin.

**Serial Clock (SCL):** It carries the clock signal.

I2C operates in 2 modes:

* Master mode
* Slave mode

According to I2C protocols, the data line cannot change when the clock line is high, it can change only when the clock line is low. The 2 lines are open drains. Hence a pull-up resistor is required so that the lines are high since the devices on the I2C bus are active low.

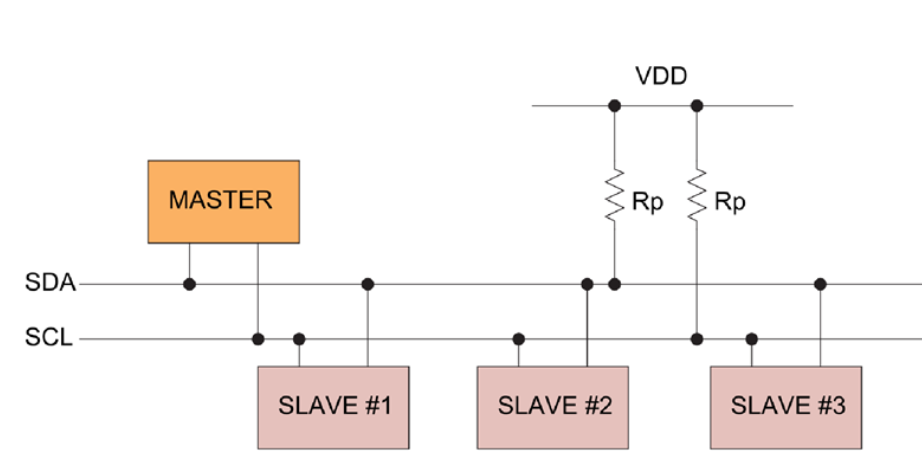


Figure 10. Generalized I2C Connection Diagram.

**How I2C Works:**

With I2C, data is transferred to ***messages****.*Messages are broken up into ***frames***of data. Each message has an address frame that contains the binary address of the slave, and one or more data frames that contain the data being transmitted. The message also includes start and stop conditions, read/write bits, and ACK/NACK bits between each data frame:

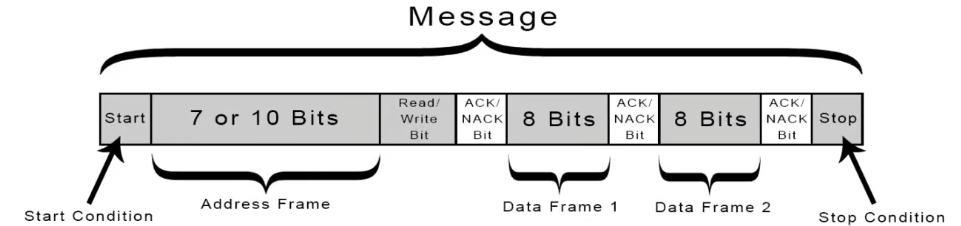


Figure 11. Messages are broken up into frames of data.

* **Start Condition:** The SDA line switches from a high voltage level to a low voltage level *before* the SCL line switches from high to low.
* **Stop Condition:** The SDA line switches from a low voltage level to a high voltage level *after* the SCL line switches from low to high.

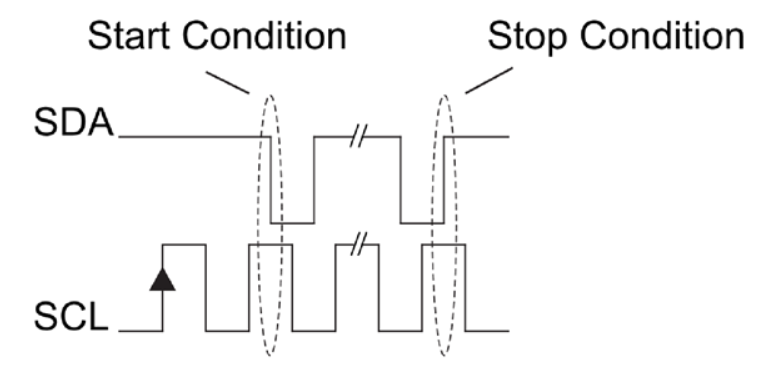


Figure 12. Start Condition And Stop Condition Transitions.

* **Address Frame:** A 7- or 10-bit sequence unique to each slave that identifies the slave when the master wants to talk to it.
* **Read/Write Bit:**A single bit specifying whether the master is sending data to the slave (low voltage level) or requesting data from it (high voltage level).
* **ACK/NACK Bit:** Each frame in a message is followed by an acknowledge/no-acknowledge bit. If an address frame or data frame was successfully received, an ACK bit is returned to the sender from the receiving device.

**Addressing**

I2C doesn’t have slave select lines like SPI, so it needs another way to let the slave know that data is being sent to it, and not another slave. It does this by *addressing*. The address frame is always the first frame after the start bit in a new message.

The master sends the address of the slave it wants to communicate with to every slave connected to it. Each slave then compares the address sent from the master to its own address. If the address matches, it sends a low voltage ACK bit back to the master. If the address doesn’t match, the slave does nothing, and the SDA line remains high.

A slave address is sent in 8-bit byte format, MSB first, but the last bit signifies whether the transaction will be read or write to the slave. In effect, the upper 7 bits constitute the slave address, while the 8th bit serves as a ***READ/WRITE*** command bit. Thus, there is an address space of 128 unique addresses for addressing up to 128 slaves. Often times.

**Acknowledge and Not Acknowledge Bits (ACK/NACK)**

As a form of feedback, after every byte transmission the receiving device sends an Acknowledge or Not Acknowledge bit. An Acknowledge bit is generated by the receiver by holding the SDA line low during a HIGH SCL period, while a Not Acknowledge bit is generated when the receiver leaves the SDA line passively pulled HIGH and does not respond in any way. This fact implies that in response to an address byte, all unmatched SLAVEs send a Not Acknowledge bit by not responding.

An ACK is used to denote that a byte (address or data) was transmitted and received successfully and that the transmission can continue to the next byte transfer, a stop condition or a repeated start. A NACK is generally used by the receiver to indicate whether an error occurred somewhere in the data transmission. This is used to signal to the transmitting device to terminate the transmission immediately or to make another attempt by sending a repeated start.

**WM8731 I2C Data Transmission Steps:**

1. The master sends the start condition to every connected slave by switching the SDA line from a high voltage level to a low voltage level *before* switching the SCL line from high to low.

2. The master sends each slave the 7-bit address (for the WM8731, the **7-bit I²C slave address** is typically 0x1A) of the slave it wants to communicate with, along with the read/write bit.

The WM8731 supports a 2-wire MPU serial interface. The device operates as a slave device only. The WM8731 has one of two slave addresses that are selected by setting the state of pin 10, (CSB).

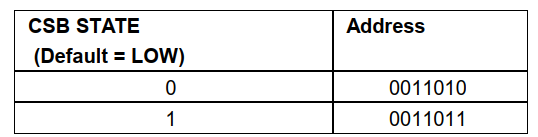


Table 1. 2-Wire MPU Interface Address Selection (Select Slave address to communicate.)

3. Each slave compares the address sent from the master to its own address. If the address matches, the slave returns an ACK bit by pulling the SDA line low for one bit. If the address from the master does not match the slave’s own address, the slave leaves the SDA line high.

4. The master sends or receives the data frame.

5. After each data frame has been transferred, the receiving device returns another ACK bit to the sender to acknowledge successful receipt of the frame.

6. After successfully receiving the ACK (confirming that the SLAVE is ready to communicate), SDA will continue sending the configuration register DATA, which consists of the 7-bit register address (DATA B15–B9) and the 9-bit configuration value for that register (DATA B8–B0), for a total of 16 bits divided into 2 frames.

In the first transmission, the frame containing the register address is sent after the ACK has been received. Since SDA sends data in 8-bit frames, the register address frame must take the MSB from the 9-bit register configuration value as the LSB of the address frame and send it first (DATA B15–B8).

7. When the data transmission is completed, the process continues by waiting to check whether the ACK bit is returned. If successful, proceed to send the remaining frame, which contains the 8-bit configuration data (DATA B7–B0).

8. After transmission, wait for the ACK bit response to verify whether the DATA has been successfully sent.

9. After confirming that the ACK has been successfully received, proceed to the stop process. To stop the data transmission, the master sends a stop condition to the slave by switching SCL high before switching SDA high.

10. When the STOP condition is successfully met, it means the configuration data for one register has been fully transmitted.  
To perform the same operation for other registers, simply repeat all the steps from the beginning up to the STOP condition.

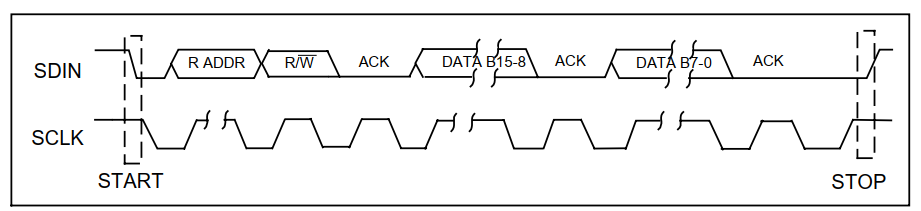


Figure 13. 2-Wire Serial Interface of WM8731.

The I2C communication protocol is a simple and effective way for devices to communicate with each other. It allows multiple devices to connect using just two wires, making it easy to add new components to a system. I2C is popular in various applications because it supports multiple devices, is relatively easy to implement, and requires less wiring compared to other protocols. Overall, I2C is a reliable choice for connecting sensors, displays, and other peripherals in electronic projects.

## 2.6. I2S communication protocol

The protocol which is used to transmit digital audio data from one device to another is known as I2S or Inter-IC Sound protocol. This protocol transmits PCM (pulse-code modulated) audio data from one IC to another within an electronic device. I2S plays a key role in transmitting audio files which are pre-recorded from an MCU to a DAC or amplifier. This protocol can also be utilized to digitize audio using a microphone. There is no compression within I2S protocols, so you cannot play OGG or MP3 or other audio formats that condense the audio, however, you can play WAV files.

In the WM8731 datasheet, it is specified that **I²S mode** operates as follows:

***I²S mode is where the MSB (most-significant bit) is available on the 2nd rising edge of BCLK (Bit Clock) following an LRCLK/DACLRC (Left/Right Clock) transition*** — meaning the MSB is **delayed by one BCLK** compared to the left-justified format. This is the standard method to synchronize left/right channel data with the LRCLK signal, as illustrated in figure 14 below.

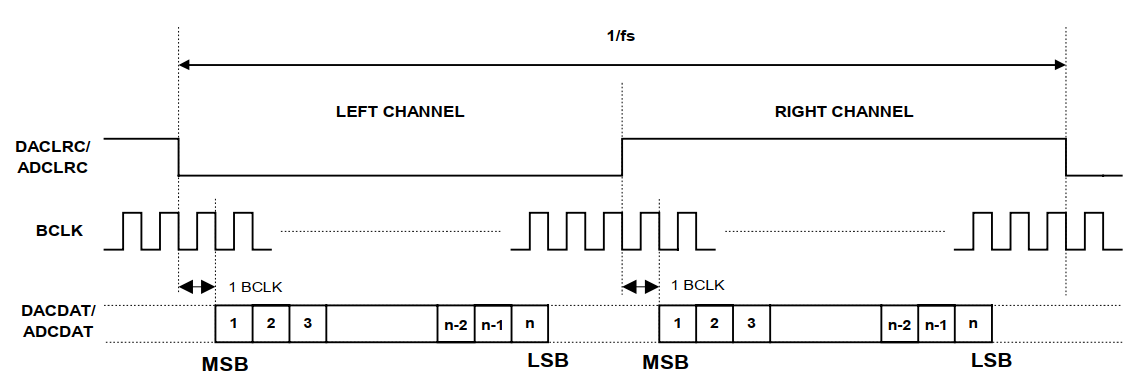


Figure 14. I2S mode.

LRCLK must always **transition on the** **falling edge of BCLK**, as stated in the datasheet. This requirement must be met when interfacing devices in both master and slave modes. *Each data bit is changed/shifted on the high-to-low transition of BCLK and sampled on the low-to-high transition of BCLK.*

According to the provided I²C register configuration, the WM8731 is set to:

**Slave Mode:** When WM8731 operates in Slave mode, both BCLK and DACLRC are inputs, as shown in the figure.

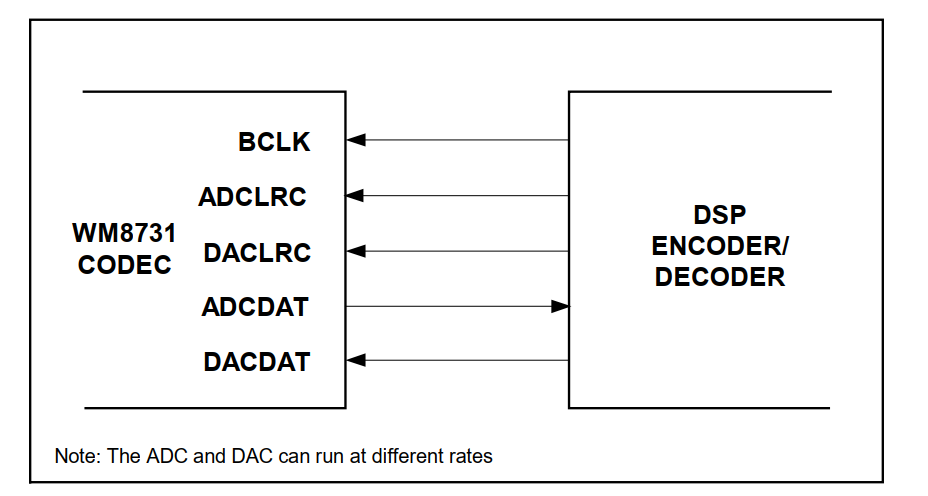


Figure 15. Slave mode.

**Audio Data Format Select**: I²S format.

**Input Audio Data Bit Length Select:** 24-bits.

**USB Mode** (sample rate: fs = 48 kHz = LRCLK).

**MCLK** **(Master Clock):** The WM8731 operates from a Master Clock (MCLK), which sets the sample rate fs​ and generates the subsidiary clocks (BCLK, LRCLK). In USB mode with a ratio, the relationship is: .

To achieve this on the FPGA, use the PLL to synthesize from the on-board reference clock.

In I²S, the group chooses a common framing is 25 bits per channel (slot length), stereo (2 channels), **Bit Clock formula:**

*BCLK = LRCLK × bits per channel × number of channels*

*= 48 kHz x 25 bits/channel x 2 = 2.4 Mhz*.

***Therefore, when configuring the codec and FPGA, verify***: MCLK=12 MHz (for fs = 48 kHz, 250 x fs ​); LRCLK = fs =48 kHz; BCLK = 2.4 MHz (assuming 25-bit slots × 2 channels).

## 2.7.  Finite impulse response (FIR) filter

The FIR (Finite Impulse Response) filter is a digital filter with a finite impulse response, implemented by multiplying input samples with fixed coefficients and summing the results. Its typical structure consists of a delay register chain, multiplier blocks, and adder blocks. The general equation of an FIR filter of order N−1 is:

where h(k) represents the filter coefficients (taps).

A major advantage of FIR filters is their inherent stability (due to the absence of feedback) and ease of design for linear phase response (constant group delay). Specifically, an FIR filter can achieve a linear phase response, meaning all frequency components are delayed by the same amount (proportional to the number of samples), thus avoiding phase distortion in the output signal. Additionally, FIR filters allow precise control of the frequency response (due to the fixed impulse response length) and straightforward consideration of quantization noise effects. However, the drawback of FIR filters is that, as the filter length (number of taps) increases, the number of multiply–accumulate (MAC) operations also increases, leading to higher hardware cost and greater processing delay.

**Symmetry in Linear-Phase FIR Filters:**

A key property of a linear-phase FIR filter is that its impulse response is either symmetric or anti-symmetric. Theory shows that an FIR filter has a linear phase if its impulse response h[n] satisfies either:

**h[k] = h[N-1−k**] (symmetric) or **h[k] = −h[N-1−k]** (anti-symmetric).

Here, M represents the central delay, typically M = .

This symmetry property allows combining pairs of symmetric inputs from the delay line before multiplication by the coefficient, thereby reducing both the number of multiplications and the coefficient storage requirements. By exploiting coefficient symmetry, significant computational savings can be achieved: symmetric coefficient pairs (*e.g., h₀ with h₇, h₁ with h₆*) are summed prior to multiplication with the corresponding signal sample, requiring only a single multiplication for each pair. As a result, the number of coefficient multiplications is reduced by approximately half compared to the conventional FIR structure. For example, an 8-tap symmetric FIR filter (h₀…h₇) can be expressed as:

**y[n]=h0[x(n)+x(n−7)]+h1[x(n−1)+x(n−6)]+h2[x(n−2)+x(n−5)]+h3[x(n−3)+x(n−4)]y[n].**

Only half of the coefficients need to be stored, as the remaining ones can be derived through symmetry. This way, the symmetric filter achieves both a linear-phase response and reduced computational and storage requirements.

**Symmetric Linear-Phase FIR Filter Diagram (8-tap) with Pipelined Architecture.**

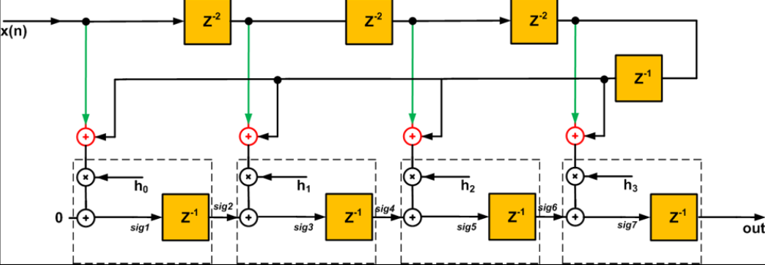


Figure 16. Symmetric Linear-Phase FIR Filter Diagram (8-tap) with Pipelined Architecture.

The input signal x(n) passes sequentially through delay elements and multiplier blocks (x) with coefficients hi. Each pair of symmetric samples (e.g., x[n]+x[n-7]) is summed before being multiplied by the shared coefficient, leveraging coefficient symmetry to minimize multiplications. The blocks in each stage (dashed boxes) are modular and can be repeated, making it easy to scale to any number of taps. Compared to a conventional FIR filter, this structure requires only half the number of multiplications due to symmetry while still maintaining a linear-phase response.

**Pipeline Architecture and Adder Tree**

In digital filters, a pipelined architecture divides the computation into multiple sequential stages, each capable of processing data concurrently and separated by buffer registers. By inserting registers between operations, the logic delay per stage is reduced, allowing the system to operate at a higher clock frequency (increased throughput).  
The trade-off is increased overall latency, as each inserted register adds one clock cycle of delay. Specifically, inserting *P* pipeline registers into the filter typically adds *P* cycles of latency in exchange for a significantly higher achievable clock rate.  
This property is especially valuable for systems requiring high real-time performance and high processing speed.

The adder tree is a parallel addition method that reduces the depth of addition operations. Instead of sequential accumulation (which has a depth of roughly *N–1* addtions for *N* taps), a binary tree structure reduces the addition depth to approximately log2.

The initial additions (at the tree leaves) are performed in parallel, and intermediate sums are subsequently added layer-by-layer until the final result is produced. The main advantage of the adder tree is reduced combinational path delay per cycle, enabling higher clock speeds. The drawback is the increased number of parallel adder blocks and more complex interconnections, which result in higher logic resource usage and power consumption.

In the traditional approach, the adders in the programmable logic (PL) are usually the performance bottleneck. The number of adders needed and the associated routing depends on the size of the filter. The depth of the adder tree scales as the log2 of the number of taps in the filter. Using the adder tree structure shown in the figure above could also increase the cost, logic resources, and power.

**Example Diagram of an Adder Tree in an 8-Tap Pipelined FIR Filter.**

In this configuration, each multiplier block (**×**) receives appropriately delayed signal samples. The multiplication results from the first stage are pairwise summed at addition nodes (**+**), and these intermediate sums are subsequently combined in later stages. Pipeline registers (labeled “18” in the figure) are inserted between stages to break long combinational paths, thus enabling higher operating clock frequencies.

This design illustrates hierarchical parallel addition: the total addition depth is only log2, which improves processing speed; however, it requires more adder blocks and complex interconnects. According to Xilinx application notes, adders in FPGA FIR implementations often become performance and resource bottlenecks when the tap count is large (adder tree depth grows with log2 of the number of taps).

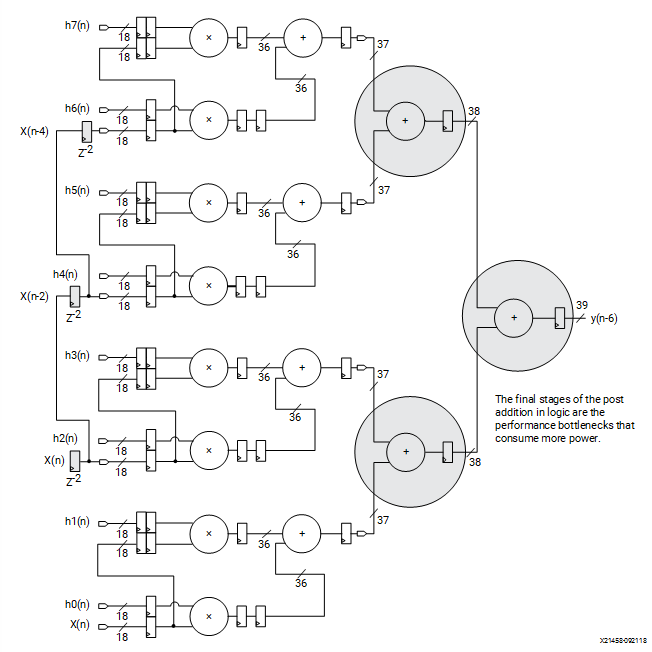


Figure 17. Traditional FIR Filter Adder Tree.

In the illustrated structure, delayed input samples are first fed into **pre-adders** that sum symmetric pairs, and the results are then multiplied by a single shared coefficient. The multiplier outputs are fed into the **adder tree**, which accumulates them to produce the final filter output.  
Thanks to this mathematical arrangement, the design only requires approximately multiplications to implement a symmetric FIR filter of length N.

**Performance Analysis – Advantages and Drawbacks**

The pipelined symmetric FIR filter offers several clear advantages:

* By leveraging both pipelining and coefficient symmetry, the filter achieves very high throughput and a linear-phase response (no signal phase distortion).
* Using only about multiplications—compared to N in a conventional design—significantly reduces DSP block usage.
* Pipelining allows the filter to run at substantially higher clock frequencies than non-pipelined architectures, making it suitable for high-speed real-time applications.

However, there are trade-offs:

* Additional pipeline registers increase **latency**: each inserted register adds one clock cycle to the output delay. For example, inserting *P* pipeline registers increases the total computational delay by *P* samples.
* Hardware usage increases, with more registers and parallel adders required.
* While the adder tree shortens the execution path, it requires extra adder blocks and interconnects, increasing logic resource usage and power consumption.
* In terms of resources, a symmetric FIR filter with this architecture needs only about DSP blocks for multiplications (significantly fewer than N in a standard FIR) but at the cost of additional registers and logic.
* Hardware complexity (LUT/FF usage in FPGA) depends on the chosen pipeline depth and the trade-off between throughput and latency.

Overall, the pipelined-adder-tree FIR structure is optimized for throughput (high output sample rate) and maximum clock frequency, but with increased signal latency compared to a straightforward non-pipelined implementation.

**Practical Applications and Performance of Pipelined Symmetric FIR Filters**

Pipelined symmetric FIR filters are well-suited for high-throughput, low phase-distortion signal processing tasks, such as:

* Digital audio processing – Linear-phase FIR filters preserve waveform fidelity (no phase distortion across frequency components), which is critical in music and speech applications.
* Biomedical signal processing (e.g., ECG) – ECG signals require noise removal (e.g., 60 Hz power line noise, muscle artifacts) without distorting the QRS waveform; a symmetric FIR ensures this while pipelining enables real-time performance.
* Digital communications (DSP modems) – FIR filters are widely used for channel shaping (e.g., raised-cosine filtering) and equalization; symmetry ensures stability and phase linearity, while pipelining supports continuous processing of high-speed data streams.
* Image processing – Many spatial filtering operations (e.g., edge detection, blur) involve convolutions with symmetric kernels, making the pipelined symmetric FIR architecture suitable for real-time video or image pipelines.

In summary, the advantages of a pipelined symmetric FIR include:

* ~50% reduction in multiplications and coefficient storage.
* Guaranteed linear-phase and inherent stability.
* Increased operating frequency and throughput via pipelining.

These characteristics make it a high-performance digital filter architecture—ideal for electrical engineering and telecommunications students designing FPGA-based DSP systems where speed and phase accuracy are critical.

# HARDWARE DESIGN AND IMPLEMENTATION

The overall waveform generator system design consists of many important modules with different roles.

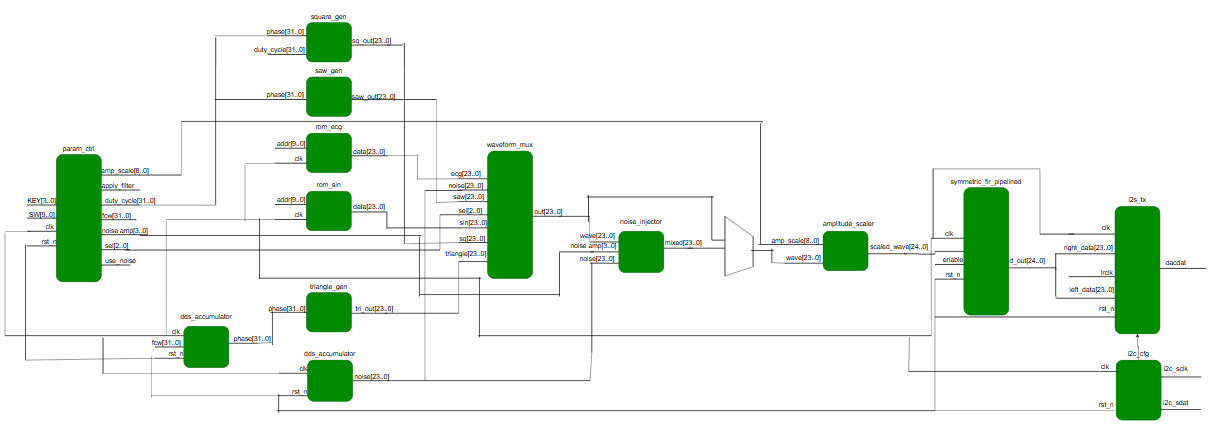


Figure 18. General block diagram of a waveform generator system

## 3.1. dds\_accumulator

As introduced in the theoretical section, this is a phase accumulator (32-bit) in the Direct Digital Synthesis (DDS) technique. The implementation concept is to maintain an n-bit phase variable and add a Frequency Control Word (FCW) to it every clock cycle. If the clock frequency is fclk the generated frequency (at the phase step output or the ROM index) is:

fout = fclk

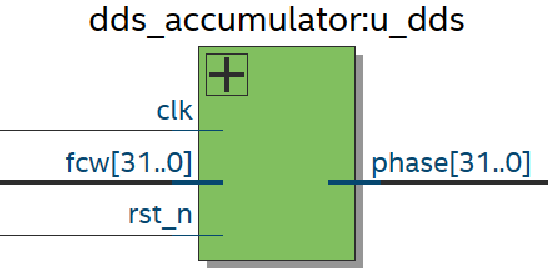


Figure 19. The block symbol of dds\_accumulator.

Because the phase wraps around modulo (: PHASE\_WIDTH = 32) , we have the capability to generate continuous frequencies. Application: generating phase indices to feed into the lookup ROM (sine, triangle, sawtooth, etc.), synchronizing digital waveform generators, and similar systems.

## 3.2. rom\_sin

**Design objective:** Whose depth matches the DAC sample depth to avoid resolution loss.

Use a ROM with 1024 samples per sine period Choose ADDR = 10 (10-bit address bus).

addr is the index used to read from the ROM. When addr is applied, the ROM returns data = mem[addr]. The address is derived from the phase accumulator by taking its MSBs: addr = phase[31:22]. As phase increments, addr changes accordingly, and the LUT value is presented on data.

Phase wrap-around: The phase accumulator wraps modulo 232 → a full wrap corresponds to one electrical cycle (0–360°). With addr = phase[31:22], each 0–360° sweep of phase maps to addr running from 0 to 1023. At the same time, each addr fetches the corresponding ROM value (data = mem[addr]), which is the output amplitude. In other words, every address value converts the instantaneous phase to the sine amplitude stored at the LUT location that addr points to.

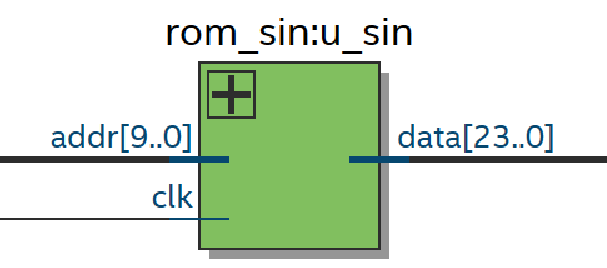


Figure 20. The block symbol of rom\_sin.

## 3.3. rom\_ecg

The design is identical to rom\_sin, differing only in the ROM contents (ECG waveform file). Purpose: generate ECG samples.

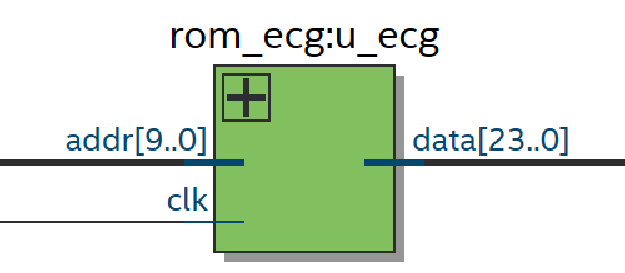


Figure 21. The block symbol of rom\_ecg.

## 3.4. square\_gen

phase and duty\_cycle share the same WIDTH (32 bits). duty\_cycle is a fixed-point value in the phase domain (e.g., 50% → 32'h8000\_0000).

**Implementation idea:** Compare phase < duty\_cycle using the modulo phase accumulator to mark the initial “high” portion and thus realize the duty cycle. Let phase sweep from 0 to 232-1, if phase lies within the first duty\_cycle interval, the output is High, otherwise Low.

**Output scaling:**

* 24'h7FFFFF = + (223 -1) → max positive for 24-bit signed using sign magnitude (here two's complement max positive = 0x7FFFFF).
* -24'h800000 = most negative for 24-bit (-223) output full-scale ±1 LSB.

Expressing the duty cycle in the same WIDTH as phase simplifies control and comparison. Instead of checking only phase[31] (a single bit), comparing the full accumulator supports duty control from 0 to 232-1.

If duty\_cycle = 0 always low; or full-range always high. As noted earlier, using phase < duty\_cycle (or ≤) is an approximation: because the accumulator advances by fcw each clock, the instantaneous phase may never equal duty\_cycle exactly. This method is therefore practical but not mathematically exact.

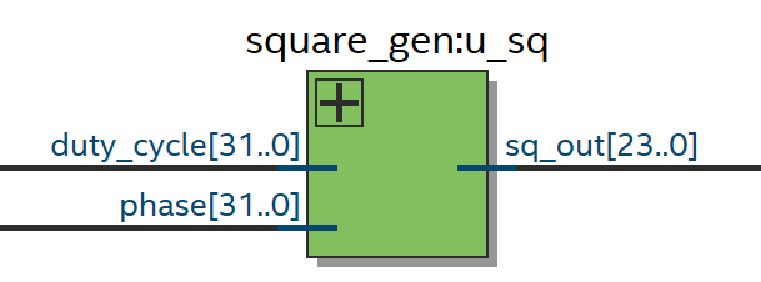


Figure 22. The block symbol of square\_gen.

## 3.5. triangle\_gen

**Implementation idea:** Define raw = phase[31:8], taking the 24 MSB of the 32-bit phase as the coarse “hour hand” representing amplitude. The remaining 8 LSB serve as the “minute hand,” enabling fine sub-step resolution, thus reducing jitter (undesired timing variation of pulses/samples that can cause noise floor elevation, bit errors, or signal distortion).

* If phase[31] == 0 (first half of the cycle): tri\_out = raw (linear ramp-up).
* If phase[31] == 1 (second half of the cycle): tri\_out = maxv – raw (linear ramp-down).

A 24-bit resolution triangle amplitude ramping up then down, matched to the WM8731 DAC resolution.

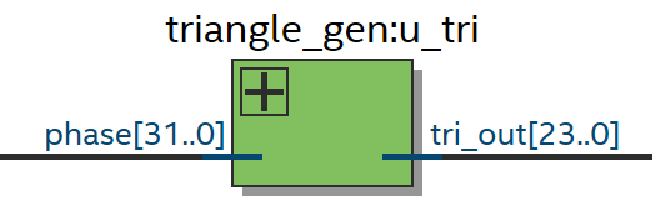


Figure 23. The block symbol of triangle\_gen.

## 3.6. saw\_gen

Directly assign saw\_out = phase[31:8] (24 MSB), producing a linear ramp from 0 → max with wrap-around at overflow.

**Implementation idea:** The phase value increases linearly due to accumulation of fcw on each clock cycle. The wrap-around property ensures immediate reset to 0 at overflow, creating a continuous sawtooth. Output is kept at 24 bits to match the WM8731 DAC amplitude resolution.

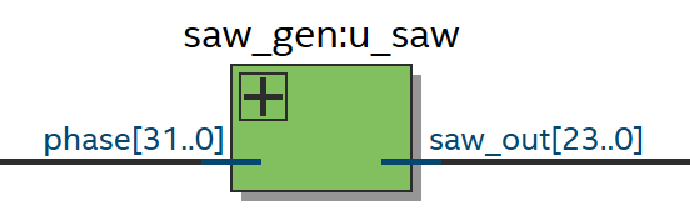


Figure 24. The block symbol of saw\_gen.

## 3.7. param\_ctrl

**Design objective:** Centralized control of parameters—fcw, duty\_cycle, amp\_scale, sel, noise\_amp, use\_noise, and apply\_filter—based on the state of the board’s SW and KEY inputs.

From the output frequency formula, fcw is set so that fout ranges from ~1Hz (min) đến ~20kHz (max).

* foutmin = fclk = 1 HzMmin = fcwmin =

= 85.899. Choose fcwmin = 86.

* foutmax = fclk = 20 kHzMmax = fcwmax

= = 1717986.918. Choose fcwmax = 1717987.

Chosen range rationale:

* Matches the operating bandwidth of the WM8731 audio codec.
* According to Nyquist, fs 2fmax fmax = = 24 kHz. Selecting 20 kHz ensures safety margin and full compliance.
* Covers the entire human-audible spectrum, allowing real-world evaluation of distortion, frequency response, filtering, and fidelity.

**Test Case Coverage**

* Low frequency (~1 Hz): observe DC drift and long-term stability.
* Mid frequency (~1 kHz): evaluate baseband frequency response.
* High frequency (10–20 kHz): test roll-off and aliasing effects.
* Very high (>100 kHz) is impractical for standard oscilloscopes (difficult to view clean cycles), while extremely low takes excessive simulation time—thus 1 Hz to 20 kHz is the optimal practical range.

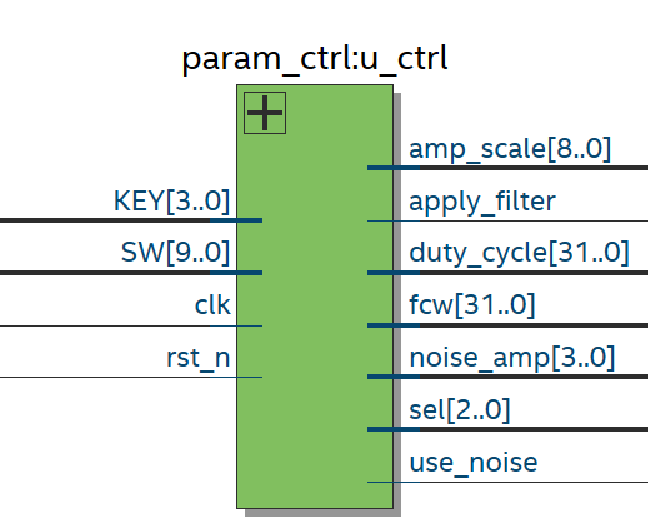


Figure 25. The block symbol of param\_ctrl.

## 3.8. lfsr\_noise

**Implementation idea:**

* Generate a 24-bit random seed (noise) as the starting value.
* The seed must be non-zero, because the feedback uses XOR of selected tap bits—if all bits were zero, the sequence would stall.
* Perform XOR on the selected tap bits to create the feedback bit, which is then shifted in as the LSB of the noise register.
* This mechanism implements a Linear Feedback Shift Register (LFSR), producing a pseudo-random noise pattern.

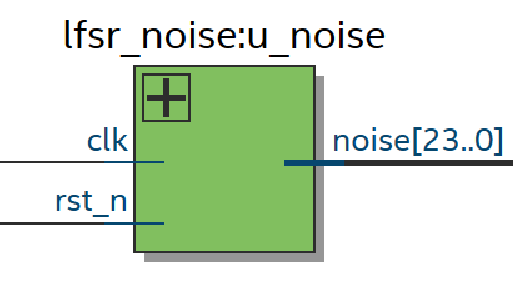


Figure 26. The block symbol of lfsr\_noise.

## 3.9. waveform\_mux

**Implementation idea:** The param\_ctrl module reads the board’s SW states to select which waveform is routed through a multiplexer (mux) for amplitude adjustment.

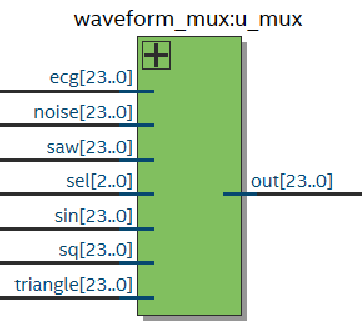


Figure 27. The block symbol of waveform\_mux.

## 3.10. noise\_injector

It is employed to introduce controlled noise into a pristine waveform for research and validation purposes—specifically, to evaluate signal integrity degradation when the clean waveform is combined with pseudo-random noise generated by the previously implemented mux and lfsr\_noise modules. This setup further enables the development and assessment of digital filter architectures aimed at effectively suppressing the injected noise.

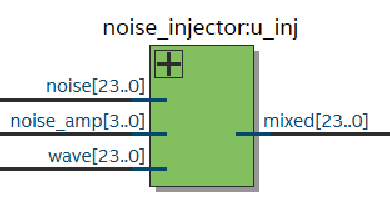


Figure 28. The block symbol of noise\_injector.

In addition, beyond the purpose of injecting noise, the design also aims to allow the user to flexibly select between a clean signal and a noise-injected signal for amplitude adjustment via the multiplexer, as illustrated in Figure 29.

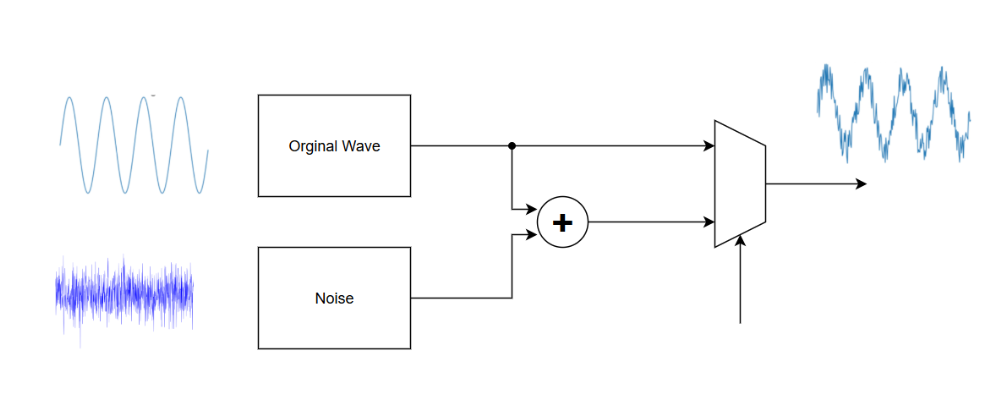


Figure 29. Noise-injected signal.

## 3.11. amplitude\_scaler

The target is used to increase/decrease the amplitude of clean or noisy waves for observation on broadcast waves.

amp\_scale is initialized as a 9-bit value but the project currently uses only the lower 8 bits (range 0–255). The 9-bit width is kept for possible future expansion, allowing higher gain if needed. Because waveform samples have WIDTH = 24 and are signed, while amp\_scale is unsigned, amp\_scale is extended to 10 bits ([9:0]) with the MSB fixed to 0—ensuring a positive multiplier.

The multiplication is thus 24-bit signed × 10-bit unsigned, yielding up to a 34-bit result. In practice, only 33 bits are kept since the top bit is unused in typical viewing conditions.

Default amp\_scale = 128 (unity gain). Right-shifting the 33-bit result by 7 bits achieves the equivalent of division by 128 for normalization.

Although a 24-bit output is sufficient for amp\_scale = 255, a 25-bit intermediate (scaled\_wave) is used to allow scaling up to 256 (×2 gain) without clipping.

In the final step, the result is truncated back to 24 bits to match the WM8731 DAC input format.

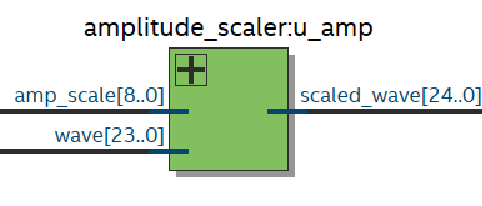


Figure 30. The block symbol of amplitude\_scaler.

## 3.12. symmetric\_fir\_pipelined

Proceed with the FIR filter implementation in accordance with the theoretical foundation described in the preceding section.

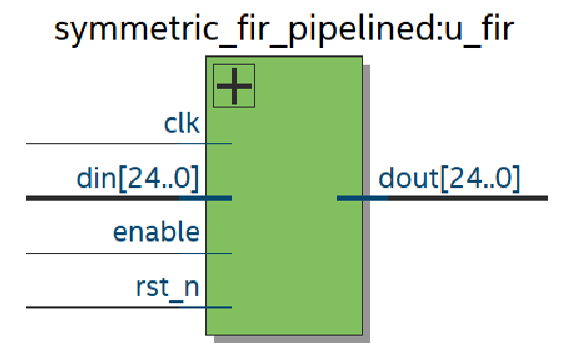


Figure 31. The block symbol of symmetric\_fir\_pipelined.

## 3.13. i2c\_cfg

**Function**: Sends a predefined sequence of configuration commands to the WM8731 Audio CODEC using the I²C protocol.

Enables the FPGA-generated digital waveform to be converted to analog by the CODEC without requiring an external processor. Acts as a fully automatic configuration circuit, ensuring the audio system initializes and operates independently right after reset.

**Implementation idea:** Configure WM8731 in **MASTER mode**, 48 kHz sampling rate, DAC enabled, I²S mode, 32-bit per left/right channel. Automatically initialize the CODEC after reset—no CPU or external control required—by writing to 9 configuration registers.

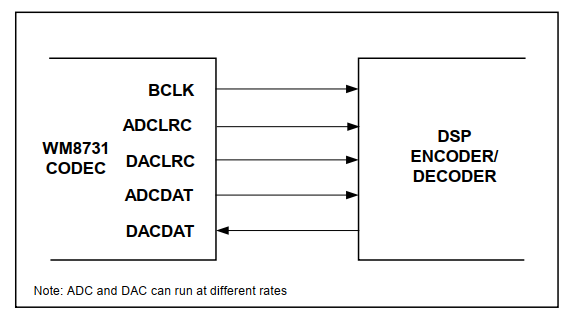


Figure 32. Master mode connection.

Each configuration register is 16 bits:

* Upper 7 bits (MSB) → Register address.
* Lower 9 bits → Register data.
* When transmitting via I²C (8-bit frames), the MSB of the 9-bit data field becomes the LSB of the 7-bit address field for transfer formatting.
* The I²C frame transmission sequence follows the WM8731 datasheet requirements.

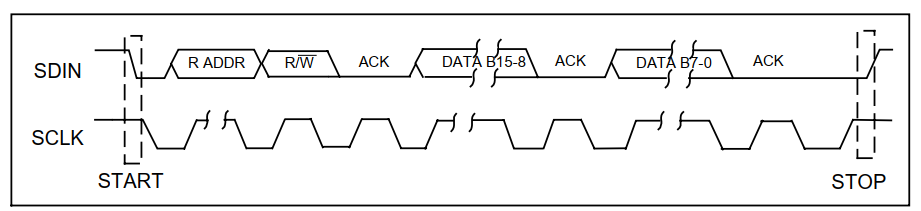
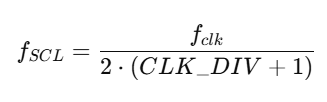


Figure 33. 2-Wire Serial Interface of WM8731.

* Finite State Machine (FSM) controls the start condition, data shifting, acknowledge handling, and stop condition.
* SCL frequency set to 200 kHz by dividing the 50 MHz system clock—toggle SCL every 125 system clock cycles.



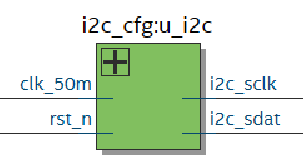


Figure 34. The block symbol of i2c\_cfg.

## 3.14. i2s\_tx

**Function:**

* Transmits left and right channel audio samples from FPGA to WM8731 DAC using the I²S protocol.
* Inputs: left\_data and right\_data (each **WORD\_BITS** bits, default 24 bits).
* Output: Serial data on DACDAT, clocked by BCLK and framed by LRCLK.

**Operation:**

Samples left\_data/right\_data into staging registers on posedge BCLK (per I²S standard).

Detects falling edge of LRCLK to identify the start of the left channel frame.

On negedge BCLK, loads and shifts out the MSB-first audio data, with a one-BCLK delay before the first MSB to meet I²S timing.

Supports 24-bit data within a 32-bit slot (MSB-aligned).

Safe for CODEC master mode (BCLK and LRCLK provided by WM8731).

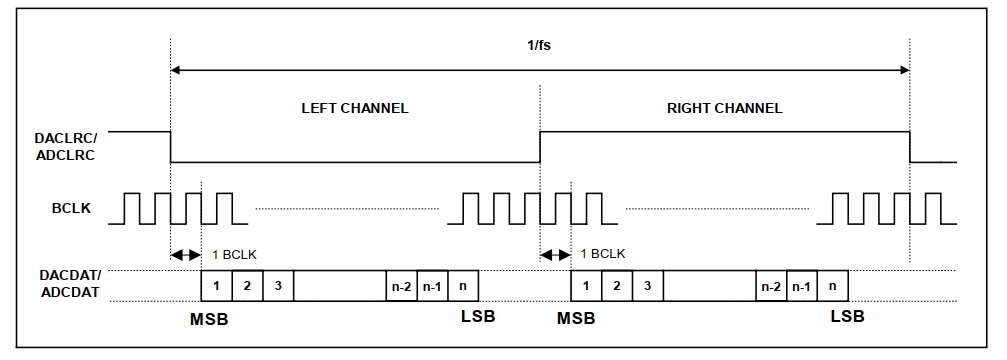


Figure 35. I2S mode.

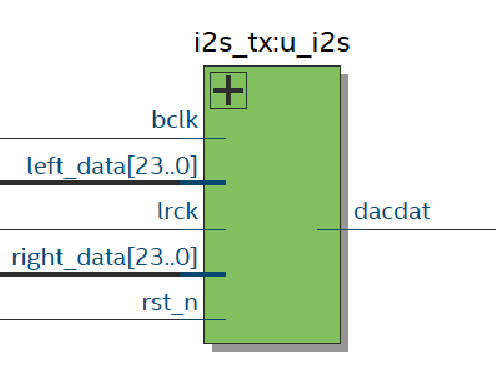


Figure 36. The block symbol of i2s\_tx.

## 3.15. pll\_12m88

A PLL inside the FPGA generates a stable 12.288 MHz master clock (MCLK) to drive the WM8731 at 48 kHz sample rate (12.288 MHz ÷ 256 = 48 kHz).

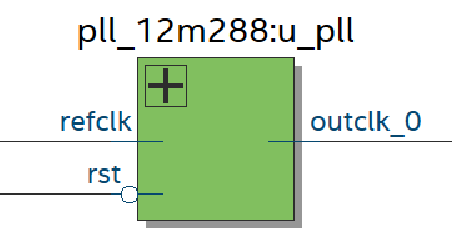


Figure 37. The block symbol of pll\_12m88.

# IMPLEMENTATION RESULTS

|  |  |  |  |
| --- | --- | --- | --- |
| Variable name | Test case | Desire | Result |
| sin\_w(scaled24) | Read ROM with incrementing address (phase) | Output a 24-bit sine wave in standard form without overflow when scaled | PASS |

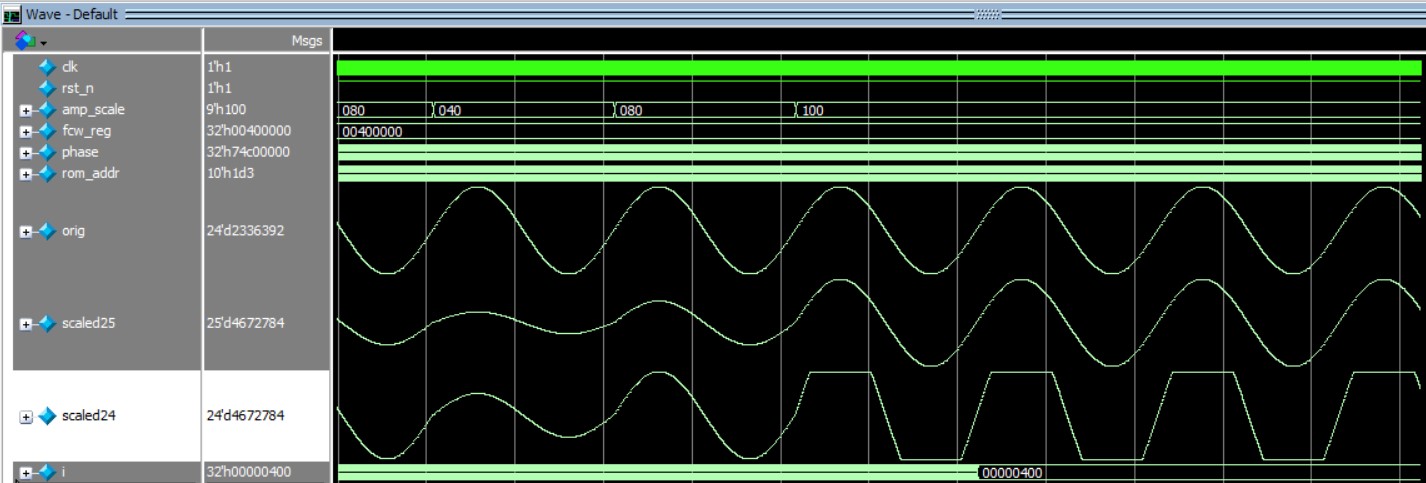


Figure 38. Sine Waveform.

|  |  |  |  |
| --- | --- | --- | --- |
| Variable name | Test case | Desire | Result |
| ecg\_w(scaled24) | Read ROM with incrementing address (phase) | Output a 24-bit ecg wave in standard form without overflow when scaled | PASS |

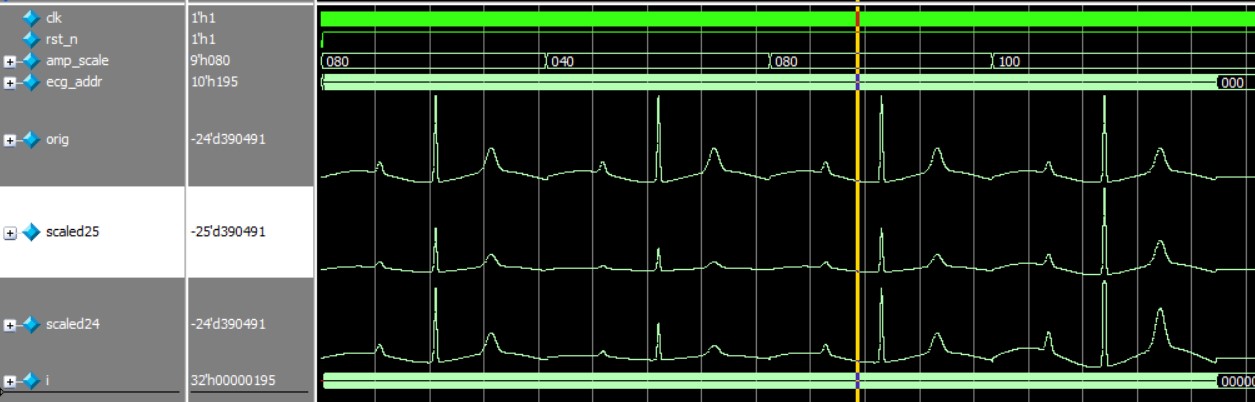


Figure 39. ECG Waveform.

|  |  |  |  |
| --- | --- | --- | --- |
| Variable name | Test case | Desire | Result |
| sq\_w(scaled24) | Choose duty\_cycle = 50% and scale the output waveform | Square wave with the correct corresponding duty\_cycle and successfully scaled | PASS |

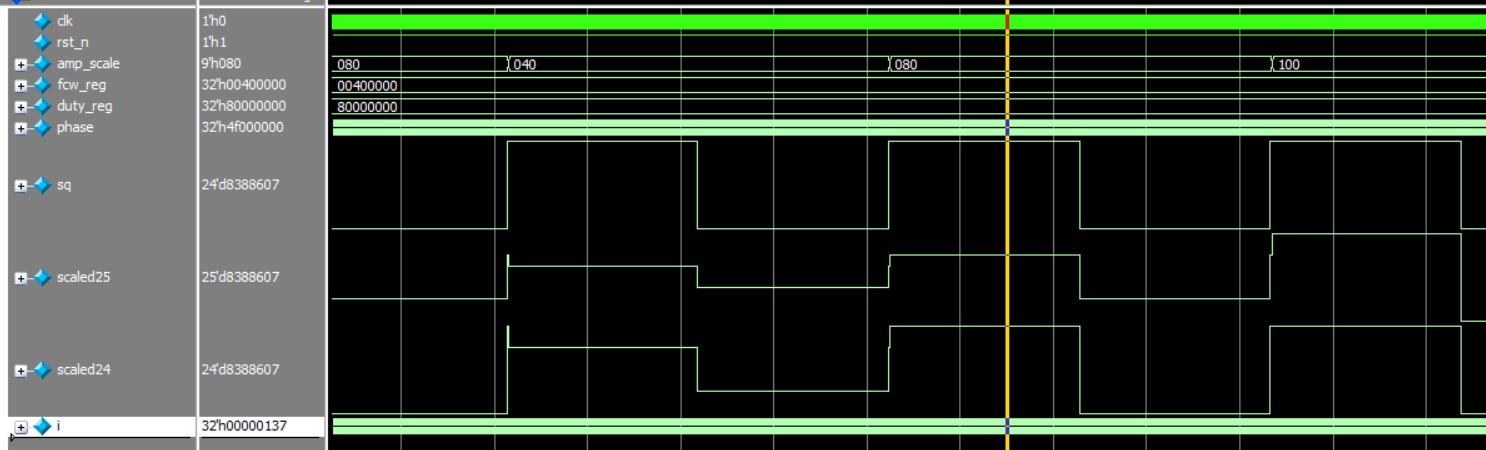


Figure 40. Square Waveform.

|  |  |  |  |
| --- | --- | --- | --- |
| Variable name | Test case | Desire | Result |
| tri\_w(scaled24) | Increase the phase gradually, resulting in a wrap-around phenomenon | The output waveform matches the computed logic, exhibiting two-sided | PASS |

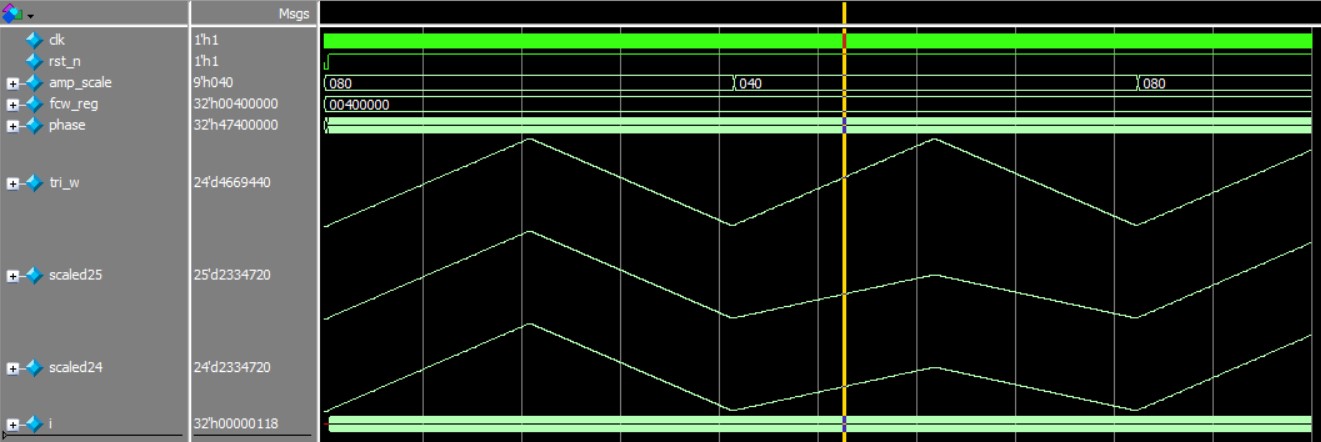


Figure 41. Triangle Waveform.

|  |  |  |  |
| --- | --- | --- | --- |
| Variable name | Test case | Desire | Result |
| saw\_w(scaled24) | Increase the phase gradually, resulting in a wrap-around phenomenon | The output waveform matches the computed logic, exhibiting one-sided slopes before returning to zero due to the wrap-around effect." | PASS |

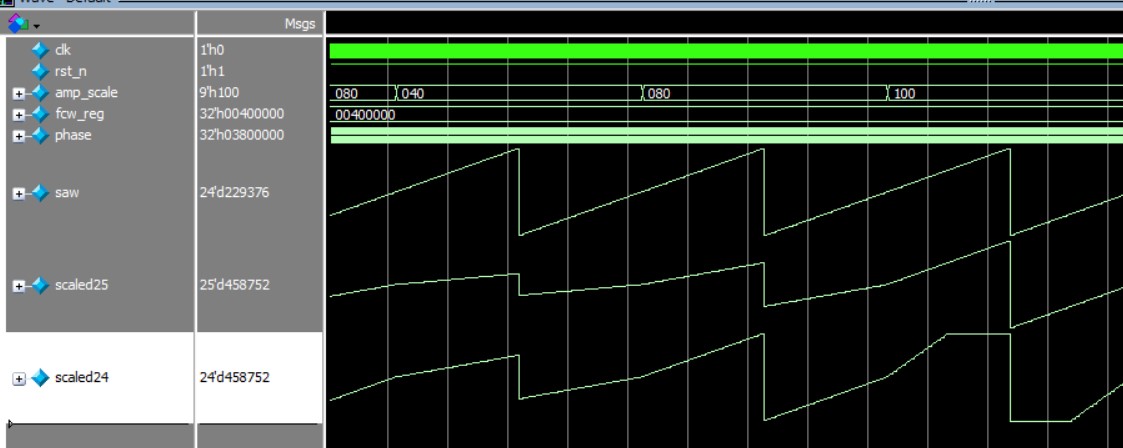


Figure 42. Sawtooth Waveform.

|  |  |  |  |
| --- | --- | --- | --- |
| Variable name | Test case | Desire | Result |
| noise | Perform XOR for linear feedback to generate pseudo-random values as noise. | Appearance of noise with varying amplitudes and different values | PASS |
| sin\_w(noisy\_scaled24) | Generate a sine wave and inject noise to produce a noise-contaminated sine wave | A sine wave appears, but its original smooth form is distorted by noise, causing the amplitude to vary irregularly | PASS |

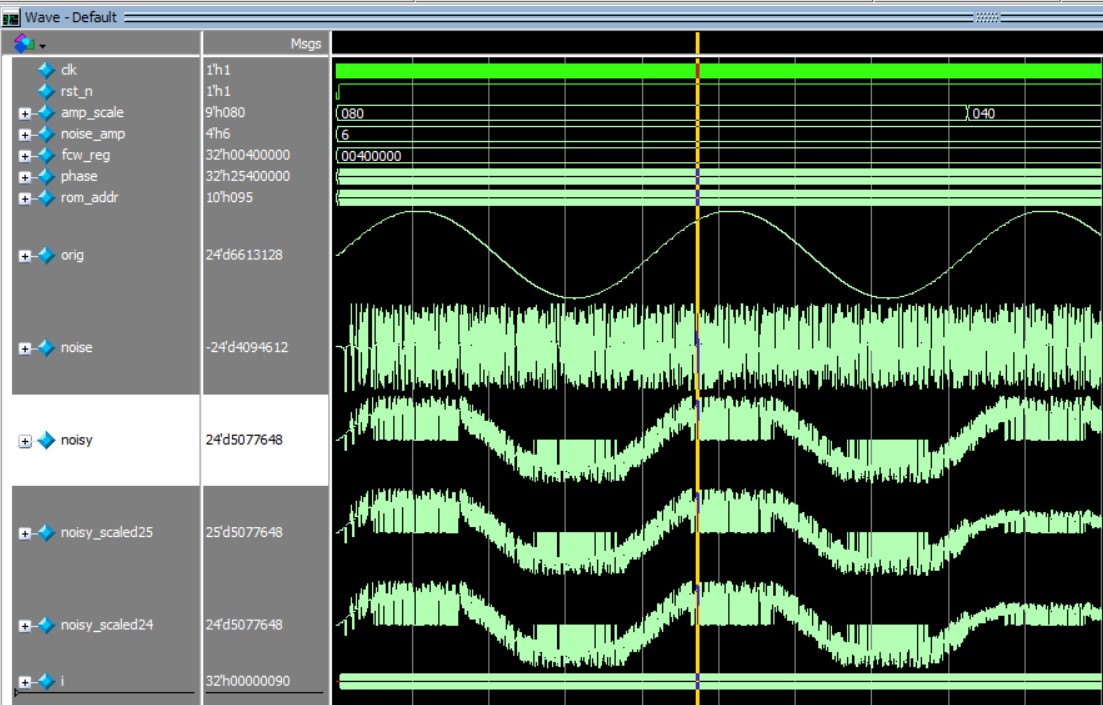


Figure 43. Inject noise into the sine wave.

|  |  |  |  |
| --- | --- | --- | --- |
| Variable name | Test case | Desire | Result |
| use\_noise(fir\_enable) | Enable or disable to allow selective application or bypass | Enable noise filtering when on, and disable when off. | PASS |
| sin\_w(fir\_out24) | The sine wave is noise-filtered when enabled. | The wave undergoes noise filtering when fir\_enable = 1 | PASS |

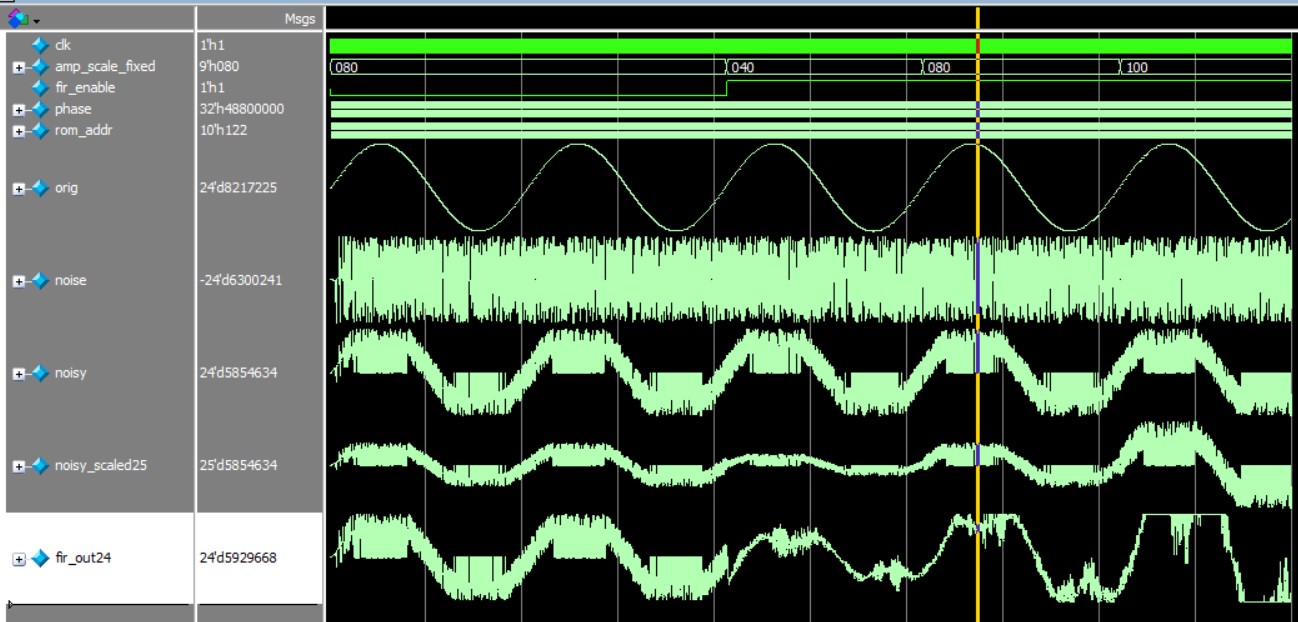


Figure 44. Filter noise from the sine wave.

|  |  |  |  |
| --- | --- | --- | --- |
| Variable name | Test case | Desire | Result |
| I2C \_sclk | Use the 50 MHz clock divided down to approximately 200 kHz to match the I²C protocol. | The clock responds each time a bit is transmitted and is used to identify states such as start, stop, and so on. | PASS |
| I2C\_sdat | Combined with s\_clk to detect start or stop conditions, and perform data transmission by frame. | The frame transmits the correct data and accurately identifies the states to start and stop the transmission. | PASS |
| ack\_drive | Send an ACK after **s\_dat** has finished transmitting a frame. | The ACK indicates that **s\_dat** has been successfully transmitted. | PASS |

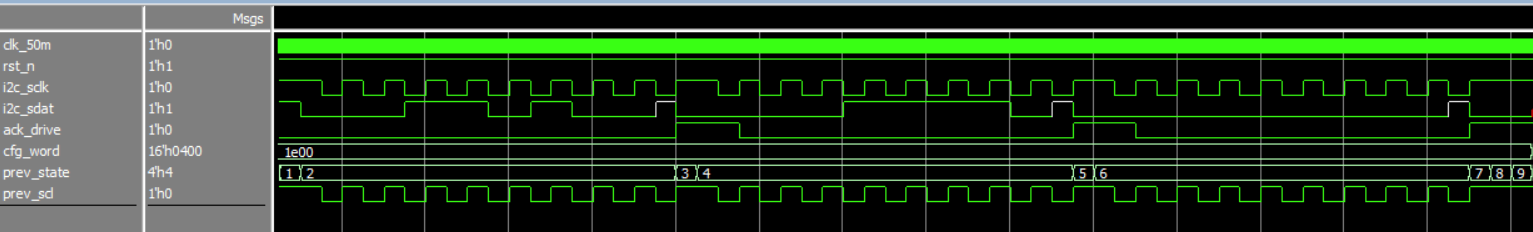


Figure 45. Waveform of the I²C protocol.

|  |  |  |  |
| --- | --- | --- | --- |
| Variable name | Test case | Desire | Result |
| LRCLK | Check whether the channel switch edge coincides with the falling edge of BCLK according to the I2S standard. | Switch channels exactly on the falling edge of BCLK as specified by the standard. | PASS |
| DACDAT | Check whether the transmission of the MSB starts exactly on the second rising edge of BCLK after the channel switch. | Begin sending data (MSB first) precisely on the second rising edge of BCLK, with each bit sampled on the rising edge of BCLK. | PASS |

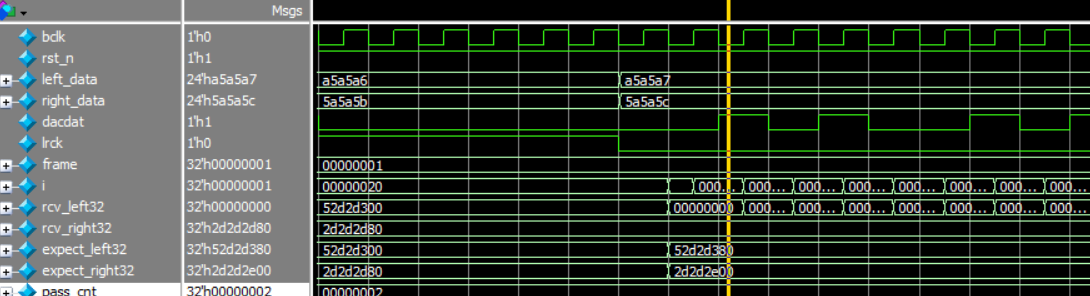


Figure 46. Waveform of the I²S protocol.

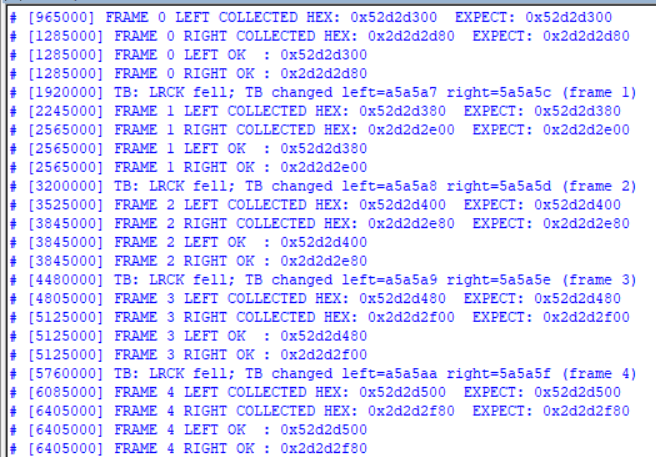


Figure 47. Data transmission result.

# CONCLUSION AND FUTURE DEVELOPMENT

**a. General Conclusion**

The **“Digital Waveform Generator on FPGA”** project has successfully implemented the main functional blocks: Phase generator (DDS-dds\_accumulator), sample ROM (sine, ECG), basic waveform generators (square/triangle/saw), noise generator (LFSR), waveform selector (waveform\_mux), noise injection (noise\_injector), amplitude scaling (amplitude\_scaler), pipelined multi-tap symmetric FIR filter (symmetric\_fir\_pipelined), asynchronous FIFO, and I²S / I²C interfaces.  
The overall system architecture is well-organized: the control block (param\_ctrl) provides real-time parameters, audio data is buffered into a FIFO, read-side prefetch and skid buffer ensure synchronization with BCLK/LRCK for I²S, and a PLL generates MCLK for the codec.

**Key strengths:**

* Clear modularization: Each function is encapsulated in an independent module, making testing and reuse straightforward.
* Practical real-time performance: FIR pipelining and adder tree allow high throughput; asynchronous FIFO enables clock-domain bridging (50 MHz ↔ audio BCLK).
* Feature richness: Multiple waveform types, noise injection, filter selection, amplitude & duty-cycle control, automatic I²C configuration for the CODEC.
* ROM preload readiness: Utilizes altsyncram megafunction with .mif files to preload sample data (sine.mif, ecg.mif).

**Limitations / observations:**

* Incomplete timing closure (timing-critical warnings). Requires SDC constraints and pipeline/routing optimization to meet timing targets.
* Bit-width and unused port warnings: Includes truncations, unused gates, unconnected PLL locked port — should be reviewed to avoid unexpected hardware behavior.
* High end-to-end latency (from phase → I²S) due to FIR/pipeline stages — must be documented for applications with strict latency requirements.
* Lack of hardware test measurements: THD, SNR, jitter need to be measured and verified on physical hardware.

**Summary:** The core architecture and features are complete at a demo/IP-packaging level. However, further optimization, hardware verification, and a refined build/deployment process are required before it can be considered board-ready.

**b. Development Directions (Practical and Detailed)**

Prioritized into: short-term (immediate), mid-term (within the project scope), long-term (research/productization).

**\*Short-Term — Bug Fixes and Board-Ready Stability**

**Complete constraints / timing**

* Create and add wrapper.sdc (or equivalent) describing clock domains (50 MHz, MCLK, BCLK/LRCK) and timing requirements.
* Declare false\_path between unrelated domains (e.g., between aud\_bclk and clk\_50m if using FIFO).
* Connect pll\_locked to ensure audio clocks are only used after PLL lock (avoiding glitches).
* Run timing reports and fix failing paths (add pipeline registers, balance I/O paths, move heavy logic to DSP blocks).

**Standardize ROM / MIF files**

* Verify sine.mif and ecg.mif exist in the project. If using megafunction, set INIT\_FILE correctly.
* Add a build script to automatically package .mif files into the project (ensuring reproducibility).

**Resolve code warnings**

* Remove unused signals, explicitly handle truncations (casting/resizing), ensure all signals have drivers.
* Connect and document debug ports (pll\_locked, FIFO status) for on-board debugging.

**Basic hardware verification**

* Load bitstream, check XCK/BCLK/LRCK on oscilloscope — verify stable MCLK, confirm I²C init sequence (ACK response), and validate I²S DAC output (line-out → oscilloscope / sound card).

**\*Mid-Term — Performance and Signal Quality Optimization**

**Pipeline & DSP block optimization**

* Map multipliers and large adders into FPGA DSP blocks (e.g., DSP48) to save LUT/FF and improve timing.
* Adjust FIR pipeline depth (symmetric\_fir\_pipelined) to balance latency vs. achievable clock rate.

**Filter improvements**

* Support selectable windows and coefficient design from offline tools (MATLAB/Python) with .mif loading.
* Add runtime coefficient reloading.
* Extend to multi-rate structures (decimation/interpolation) if sample-rate conversion is required.

**Signal quality measurement**

* Develop testbenches and hardware tests to measure THD, SINAD, SNR, spur levels, and jitter.
* Compare hardware output with MATLAB/Python simulations for accuracy verification.

**User interface**

* Add UART/USB control for real-time parameter updates (FCW, waveform select, amplitude, noise).
* Display status indicators (PLL lock, FIFO fill level) on LEDs or 7-seg display.

**\*Long-Term — Feature Expansion and Productization**

**Multi-channel / MIMO**

* Extend to multi-channel audio (stereo with independent sources, multi-tone generation).

**Automated calibration**

* Integrate ADC feedback (if available) for automatic amplitude tuning, DC offset compensation, or frontend response measurement.

**Resource and power optimization**

* Share DSP/filter hardware between channels; reuse arithmetic units to reduce resource usage.

**Packaging & IP core**

* Wrap into an IP core (Avalon/AXI) with a register map for SoC integration (Nios II / MicroBlaze).

**Advanced applications**

* Add real-time FFT/spectrogram, digital modulation patterns, or codec test signal generation.

**c. Experimental Validation Methodology**

**Functional verification**

* Unit testbench for each module (DDS, ROM, mux, noise injector, amp scaler, FIR) comparing SystemVerilog simulation vs. expected waveforms.

**Jitter and stability**

* Oscilloscope measurement of aud\_xck (MCLK) and aud\_bclk jitter; compare before/after PLL lock.

**Audio quality**

* Capture line-out via audio interface → FFT → measure THD+N, SNR across test tones (1 kHz, 10 kHz, full sweep).

**FIFO & timing stress test**

* Simulate FIFO underrun/overrun scenarios to validate prefill and skid-buffer logic.

**Resource & power profiling**

* Analyze LUT/FF/DSP/RAM usage and estimate power consumption (Power Analyzer).

**d. Summary & Short Roadmap**

**Summary:** The system is functionally operational with a strong foundation for further development into a robust digital waveform generator. The next focus should be on timing closure, signal quality measurement, and hardware validation.

**Roadmap:**

1. Complete constraints & connect pll\_locked.
2. Resolve all synthesis warnings and validate .mif loading.
3. Perform on-board I²S/I²C verification.
4. Measure THD/SNR and optimize FIR mapping to DSP blocks..
5. Add real-time control interface (UART/USB) and package as IP.

# 6. REFERENCES

[1] B. Cronin, "DDS Devices Generate High-Quality Waveforms Simply, Efficiently, and Flexibly," Page 1.

[2] A. Pini, "The Basics of Direct Digital Synthesizers (DDSs) and How to Select and Use Them," Contributed By DigiKey's North American Editors, 2019.

[3] geeksforgeeks, "I2C Communication Protocol," geeksforgeeks, 2025.

[4] M.io Staff, "What is the I2S Communication Protocol?," digikey, 2023.

[5] W. Microelectronics, "WM8731 Datasheet," alldatasheet, 2001.

[6] F. Bruekers, "Symmetry and Efficiency in Complex FIR Filters," Technische Universiteit Eindhoven, 2009.

[7] RAMIJABI SHAIK1, ANJANEYULU YERUGUDIDINNE2, PATTAN AYUB KHAN3, Dr.M.M.RAGHAVENDRA4, Dr.G.LAKSHMINARAYANA5, "FPGA IMPLEMENTATION OF SYMMETRIC SYSTOLIC FIR FILTER USING MULTI-CHANNEL TECHNIQUE," ENGINEERING COLLEGE, 2022.