Signal Generation And Evaluation Using Digital-to-Analog Converter And Signal Defined Radio

Aakash Choudhury

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Masters of Science
in
Electrical Engineering

Linbo Shao, Chair
Wenjie Xiong
Xiaotang Jia

June 28, 2023
Blacksburg, Virginia

Keywords: DAC, Clock Data Recovery, Signal Integrity, FPGA, SDR.

Signal Generation And Evaluation Using Digital-to-Analog Converter And Signal Defined Radio

Aakash Choudhury

(ABSTRACT)

In contemporary communication systems, Digital-to-Analog Converters (DAC), Signal Defined Radio (SDR) signal creation, and clock data recovery are essential components. DACs convert digital signals to analog signals, creating continuous waveforms. DACs provide versatility in the transmission of SDR by supporting a range of communication protocols. Clock data recovery enables precise signal recovery and synchronization at the receiver end. These elements work together to provide effective and high-quality communication systems across several sectors. With the development of quantum computing, these SDR systems also find extensive use in generating precisely timed signals for controlling components of a quantum computer and also for read-out operations from various specialized instruments. This thesis demonstrates an FPGA (Xilinx vcu118) with a DAC (Analog Devices AD9081) platform. It employs SDR for generating of periodic signals and also stream of bits which are then recovered using a simple Clock Data Recovery technique. The signal integrity of the generated signals and error-rate from the proposed Clock Data Recovery technique is also analyzed.

Signal Generation And Evaluation Using Digital-to-Analog Converter And Signal Defined Radio

Aakash Choudhury

(GENERAL AUDIENCE ABSTRACT)

Communication systems in our networked world depend on key technologies to provide dependable connectivity. By converting digital data into continuous waveforms, Digital-to-Analog Converters (DACs) serve a crucial role in enabling the generation of various analog signals. This makes it possible for Software-Defined Radio (SDR) to produce a variety of modulated signals and enables smooth communication between various hardware and software systems. The Clock and Data Recovery (CDR) algorithms correct for clock fluctuations and phase offsets to provide precise signal recovery and synchronization. Together, these technologies improve communication networks' effectiveness and dependability, allowing seamless connectivity and enhancing our networked experiences. This thesis presents an SDR platform comprising Xilinx FPGA vcu118 and Analog Devices high-speed DAC/ADC AD9081. A CDR algorithm is also proposed to recover data from the signals generated by the DAC, and its effectiveness and error rate is also analyzed.

Dedication

This thesis is dedicated to my family and advisor, Dr. Shao who has been very supportive of me throughout my journey.

Acknowledgments

I want to sincerely thank all of the people who have helped and inspired me while I've been working on my thesis. I want to start out by expressing my sincere gratitude to my advisor Dr. Shao for his unwavering support. His dedication and expertise have been crucial to the success of the project and to bring out the best in me. My gratitude also goes out to the BEARHW lab and Dr. Xiong for providing a productive and comfortable workspace. I would like to express my sincere thanks to the members of my thesis committee (Dr. Shao, Dr. Xiong, Dr. Jia) for their guidance, expertise, and insightful feedback. Their valuable suggestions and constructive criticism have immensely contributed to the quality and refinement of this thesis. I am also thankful to the faculty members and mentors who have imparted their knowledge and expertise, shaping my academic and professional development. I want to express my sincere gratitude to my family for their everlasting support, love, and understanding. My academic success has been greatly influenced by their faith in me and their efforts. I would also like to thank my girlfriend for always having unconditional faith in me and my capabilities. I also want to give a shout-out to my friends for their crucial support and company throughout this difficult undertaking. This journey has been made more fun and unforgettable by their presence, motivation, and common experiences. Finally, I'd want to thank everyone who helped my thesis come to fruition. Your encouragement and support have been extremely important to my academic and personal development. I appreciate everyone's everlasting belief in me and support.

Contents

List of Figures viii

List of Tables x

1 Introduction 1

2 Review Of Literature 4

2.1 Digital-to-Analog Converter (DAC) 4

2.2 Digital Signal Generation Techniques 6

2.2.1 Direct Digital Synthesis (DDS) 6

2.2.2 Delta-Sigma Modulation (DSM) 8

2.2.3 Pulse Width Modulation (PWM) 8

2.2.4 Oversampling DACs 9

2.2.5 Interpolation Techniques 10

2.2.6 Noise Shaping Techniques 11

2.2.7 Software-Defined Radio (SDR) 11

2.3 Clock Data Recovery (CDR) 13

3 Experimental Setup 16

4 LibIIO **20**

5 Digital to Analog Converter (DAC) **28**

5.1 JESD204B/C 28

5.2 Transmit Digital Path 31

6 Generating Signal Using DAC **37**

6.1 Setting Up Channel And Buffer 37

6.2 Generating Periodic Signals 38

6.3 Generating Random Bits 39

7 Clock Data Recovery **43**

7.1 Extracting And Unpacking The Voltage Values 44

7.2 Edge Detection 45

7.3 Recovering Clock From Start Bits 50

7.4 Sampling Data 51

7.5 Error Rate Testing 52

8 Results **53**

8.1 Periodic Signals 53

8.2 Random Bit Stream 54

9 Conclusion **61**

List of Figures

3.1 Block diagram of hardware setup 17

3.2 AD9081 plugin in IIO-Oscilloscope 19

4.1 Libiio Software Stack [9] 21

4.2 IIO devices in the project 22

4.3 Channels for iio:device1. 23

4.4 Heirarchy of different libiio datatypes [9] 24

4.5 Data format for iio:device1 buffer. 24

5.1 Overview of JESD204B/C[15] 30

5.2 Overall Transmit digital datapath [4] 32

5.3 Fine Digital Up-Converter in detail [4] 33

5.4 Fine Digital Up-Converter in detail [4] 34

5.5 Interpolation filter [4] 35

5.6 Mod Mux configuration 0 [4] 36

6.1 Devices shown by the C code using libIIO APIs 39

6.2 Channels shown by C code using LibIIO APIs 40

6.3 Sample sine wave generated by the DAC 41

6.4 Sample square wave generated by the DAC 41

6.5 Sample triangle wave generated by the DAC 42

7.1 Data plotted after unpacking from .FLT file 45

7.2 Edges Detected (includes outliers) 46

7.3 Edges Detected (includes outliers) 47

7.4 Edges after the outliers are removed. 48

7.5 Edges before and after resolving clusters 50

8.1 Sine wave of frequency 1MHz 54

8.2 Square wave of frequency 1MHz 54

8.3 Triangle wave of frequency 1MHz 55

8.4 Sine wave of frequency 10MHz 55

8.5 Square wave of frequency 10MHz 56

8.6 Triangle wave of frequency 10MHz 56

8.7 Sine wave of frequency 25MHz 57

8.8 Square wave of frequency 25MHz 57

8.9 Triangle wave of frequency 25MHz 58

8.10 Bitstream captured by oscilloscope 58

8.11 Plot of Error rate vs Baud rate 59

8.12 Plot of Error rate vs Peak-to-peak Voltage 60

List of Tables

8.1 Error Rate vs Baud Rate 56

8.2 Error Rate vs Peak-to-peak Voltage 59

List of Abbreviations

ADC Analog-to-Digital Converter

APP Averaged Phase Picking

CDR Clock Data Recovery

DAC Digital-to-Analog Converter

DDS Direct Digital Synthesis

DLL Delay-locked loops

DPP Direct Phase Picking

DSM Delta-Sigma Modulation

FPGA Field Programmable Gate Array

LUT Look up Table

PLL Phase-locked loops

PWM Pulse Width Modulation

QAM Quadrature Amplitude Modulation

RF Radio Frequency

SDR Software Defined Radio

SFDR Spurious Free Dynamic Range

DAC is an electronic device that converts digital signals into analog signals.

ADC (Analog-to-Digital Converter) is an electronic device that converts analog signals into digital signals.

FPGA is a reconfigurable integrated circuit that allows users to customize digital logic circuits for various applications.

CDR is a circuit or algorithm that extracts timing information and recovers digital data from a serial data stream.

DDS is a method that produces high-quality analog waveforms by digitally controlling the frequency, phase, and amplitude, allowing for precise and flexible generation of signals in various applications.

PWM (Pulse Width Modulation) is a technique that generates variable-width pulses to control the average power delivered to a load, commonly used in applications such as motor speed control and LED dimming.

RF refers to the range of frequencies within the electromagnetic spectrum typically used for wireless communication and broadcasting applications.

LUT is a digital memory component that stores precomputed values, used for efficient data retrieval and function approximation in various applications.

DSM is a digital signal processing technique that converts analog signals into high-resolution digital signals by oversampling and noise shaping.

A PLL is a type of control system that creates an output signal that is phase and frequency locked to an input reference signal.

DLL is a control system that aligns the phase and frequency of an output signal with a variable delay relative to an input reference signal.

SDR is a radio communication system where traditional hardware functions are replaced or augmented by software, providing flexibility and adaptability in wireless communication.

SFDR is the ratio between the power of the desired signal and the power of the highest spurious signal in a system, measured in decibels (dB).

QAM is a modulation scheme that combines both amplitude and phase modulation to transmit digital information over a carrier wave.

DPP is a technique used to accurately estimate the phase of the incoming data signal for synchronization and clock recovery purposes.

APP is a technique that computes the average phase of the incoming data signal over multiple samples to improve the accuracy of phase estimation for synchronization and clock recovery.

Chapter 1

Introduction

In the rapidly advancing field of digital signal processing and electronic systems, the generation of accurate and reliable analog signals is of utmost importance. Arbitrary Waveform Generators (AWG) are systems that generate a variety of signals. AWGs are often employed in engineering applications, electrical testing, and research and development that call for the creation of complicated waveforms. They offer versatility in generating test signals, and characterising electrical equipment. For this project, a conmbination of a Field Programmable Gate Array (FPGA) and high speed Digital-to-Analog Converter (DAC) has been used as an AWG. Over time, the speed of DACs and communication protocols on board the FPGAs have improved significantly, meaning smoother and more accurate signals can be generated. DACs and FPGAs have emerged as key components in signal generation applications, offering a powerful combination of flexibility, programmability, and precision.

The heart of an AWG lies in its ability to store and reproduce digitally represented waveforms. By employing high-speed DACs, AWGs can convert digital samples into precise analog voltage levels. These voltage levels are then sequentially sent as output to generate the desired waveform shape. The key advantage of AWGs is their ability to generate signals with fine resolution and high sample rates, resulting in an accurate and reliable reproduction of complex waveforms. The applications of arbitrary waveform generators span across numerous disciplines. In the field of telecommunications, AWGs play an important role in the development and testing of communication systems by generating signals as per

requirements, allowing researchers to evaluate system performance under different scenarios. In research laboratories, AWGs facilitate the exploration of signal processing algorithms, device characterization, and waveform analysis. They are utilized in areas such as wireless communications, radar systems, audio and video processing, and medical research. Another upcoming area of application, with the advent of quantum computers, is in quantum measurements and generating pulses to control qubits. DACs can generate analog control signals to precisely manipulate and control the parameters of quantum systems. For example, in experiments involving trapped ions or superconducting qubits, DACs can produce analog voltages that control the strength and duration of laser pulses used for qubit operations or microwave signals for qubit manipulation. DACs play a crucial role in generating complex pulse sequences or arbitrary waveforms for quantum control and measurement. By programming the DAC with a specific sequence of voltage or current values, it can generate precisely timed pulses, modulation patterns, or waveform envelopes required for performing quantum gates, quantum state preparation, or readout operations. Additionally, in quantum systems, it is often necessary to maintain stable conditions or feedback control certain parameters. DACs can be employed in closed-loop control systems to generate analog feedback signals that regulate and stabilize critical parameters, such as laser intensity, cavity length, or magnetic field strength. These feedback signals can be derived from quantum measurements and used to actively maintain the desired experimental conditions. Due to the sensitive nature of quantum processes and the requirement for precise control and manipulation of quantum states, precisely timed signals are of highest significance in quantum systems. In order to execute quantum gates and operations with high accuracy, precisely timed signals are essential. Timing mistakes can ruin the delicate coherence of quantum states. The precision of measurements, for instance, depends on the timing of control pulses and the appropriate coordination of interactions between the quantum system and the target being measured in quantum clocks or atomic magnetometers.

This project's main goal is to create and put into use a system that can produce precise and well-timed analog waveforms utilizing DACs and FPGAs. By leveraging the programmability of FPGAs, the project aims to provide flexibility in waveform generation, allowing us to generate complex and diverse signals that suit the specific requirements of the application. Several waveforms are generated and analyzed using an oscilloscope. The project also extends to processing the waveform to develop an algorithm for Clock Data Recovery (CDR) from the signal sampled by the oscilloscope to recover the bits and check for signal integrity and error rate. Since the communication between the DAC and the oscilloscope is asynchronous, i.e. they do not share a common clock, the bits cannot be directly recovered accurately from the captured waveform and it is necessary to use an algorithm to recover the clock and then the bits. During communication, there might be errors in transmission due to transmission mediums and therefore it is necessary to properly analyze the bits and check against the bits that were generated. For future work, error correction codes can be used to mitigate transmission errors.

Chapter 2

Review Of Literature

Digital-to-Analog Converters (DACs) are electronic devices that convert digital signals into analog signals. They play a crucial role in various applications where digital information needs to be converted into continuous analog waveforms for further processing or direct output. DACs operate by taking the digital input values and producing corresponding analog output voltages. This chapter delves deeper into the background and past work done on DAC and signal generation.

2.1 Digital-to-Analog Converter (DAC)

There are several types of DACs [40], each with its own advantages and limitations. Some common types include:

- Binary-weighted DAC: This type of DAC uses a network of resistors to generate analog voltages based on the digital input. The resistors have weights that are binary multiples, and their values determine the accuracy and resolution of the DAC. An example of a binary-weighted DAC is the AD7524 from Analog Devices. It is a 12-bit voltage-output DAC [1].

- R-2R ladder DAC: This architecture employs a combination of resistors to create voltage references and switches to connect them based on the digital input. It offers good

linearity and is widely used in many applications. The MCP4822 from Microchip Technology is an example of an R-2R ladder DAC which is also a 12-bit DAC [2].

- Sigma-Delta DAC: These DACs employ oversampling and noise-shaping techniques to achieve high resolution and excellent signal-to-noise ratio. They are commonly used in audio applications. The PCM1795 from Texas Instruments is an example of a sigma-delta DAC. It is a 24-bit, 192 kHz stereo DAC [3]. These kinds of DACs are particularly useful for audio applications.

- Current Steering DAC: This type of DAC utilizes current sources and switches to generate analog voltages. It offers high-speed operation and is commonly found in high-frequency applications. An example of this kind of DAC is the Analog Devices DAC/ADC AD9081 [4].

- Hybrid DACs: These DACs combine different architectures to achieve a balance between resolution, linearity, speed, and power consumption. The AD5791 from Analog Devices, which is a 20-bit resolution DAC is an example of a hybrid DAC [5].

Several performance metrics dictate the overall performance of a DAC and the domain of applications [29, 34]:

- Resolution: It indicates the number of discrete output levels that can be generated by the DAC and is typically measured in bits. Higher resolution leads to finer granularity and improved accuracy.

- Linearity: This parameter quantifies the DAC's ability to produce output voltages that are proportional to the digital input values. Non-linearities can introduce distortion and affect the fidelity of the generated signals.

- Signal-to-Noise Ratio (SNR): SNR represents the ratio of the desired power of signal to the power of noise in the output of the DAC. A higher SNR indicates better dynamic range and signal fidelity.

- Settling Time: Settling time refers to the time taken by the DAC output to stabilize within a specified accuracy after a change in the input value.

- Power Consumption: DACs consume power during operation, and optimizing power efficiency is essential for many applications, particularly portable devices.

- Bandwidth: The bandwidth of a DAC refers to its ability to accurately reproduce high-frequency signals. It depends on the DAC's internal circuitry and the sampling rate at which the digital input is provided.

2.2 Digital Signal Generation Techniques

Depending on the application and available resources there are multiple methods of signal generation:

2.2.1 Direct Digital Synthesis (DDS)

Direct Digital Synthesis (DDS) is a technique that generates waveforms digitally by manipulating the phase and frequency of a digital signal. It utilizes a phase accumulator, which generates a digital phase value that corresponds to the desired frequency. The value of phase is converted to analog signals using a DAC. DDS provides precise control over frequency, phase, and amplitude, making it suitable for generating sine waves, square waves, and other complex waveforms [21, 22]. It is widely used in applications such as communication sys-

tems, signal synthesis, and frequency generation. A spur is an unwanted signal that appears in the output frequency spectrum. Spurs are caused by harmonic imperfections in the generated waveform due to aliasing. They can interfere with the desired output frequency and reduce the overall performance of the DDS. Therefore, it is important to use spur reduction techniques to minimize or eliminate spurs in DDSs. Some techniques that are used to reduce spurs are spurless fractional divider [44, 42], Wheatley jitter injection DDS [43], randomized DAC DDS, and nonuniform clock DDS [44, 42, 43, 26]. The possible effects of a recently created chirp synthesizer on waveform generation and local oscillator topologies for next-generation radar designs are explored by Adler et al in their study [17]. Radar performance can be enhanced thanks to the synthesizer's adaptability in waveform control.

Juokko Vanka [54] discusses methods of compressing the quarter-wave memory, using trigonometric identities, the Nicholas method, Taylor series, and the CORDIC algorithm. Memory compression along with algorithmic techniques have effects on the DDS output. The paper simulates these effects and also analyzes 14-bit phase to 12-bit amplitude mapping. The chirp synthesizer is capable of generating a variety of waveforms for several applications. With the development of quantum computer grabbing pace, signal generation using DACs for controlling quantum measurements and synchronizing different signals is very important. Using Direct Digital Synthesis (DDS), Petrov et al [39] investigate a novel technique for producing quantum frequency standard output signal. The authors show how this technique may improve spectral properties, widen the output frequency range, and minimize output frequency tuning steps. Additionally, they go over DDS's accuracy and stability as benefits over alternative frequency synthesis techniques. DDS systems are immune to aging effects and temperature drift, which can cause instability in other methods. Additionally, the technique offers a chance to improve the output signal's spectral properties, broaden the output frequency range, and minimize output frequency tuning steps. These developments may

result in quantum frequency standards that are more accurate and reliable.

2.2.2 Delta-Sigma Modulation (DSM)

Delta-Sigma Modulation (DSM) is a technique used in oversampling Analog-to-Digital Converters (ADCs) and DACs. In the context of signal generation, DSM-based DACs are utilized to generate high-resolution analog waveforms. The technique involves oversampling the input signal, quantizing it to a single bit, and then using a digital filter to shape the resulting noise. The filtered signal is converted to an analog waveform using a DAC [51]. DSM provides high resolution, excellent linearity, and low distortion, making it popular in audio applications where high fidelity is required. Kinyua in his patent [32] describes a continuous-time delta-sigma DAC which has less signal-to-noise ratio. For signals with moderate bandwidth, delta-sigma modulators often use noise-shaping and oversampling to conduct digital-to-analog conversion at a relatively higher resolution. The quantization noise is pushed farther into the frequency spectrum and away from the signal baseband. With this innovative method, better resolution may be attained in the same space as sigma-delta DACs of the past. The complexity difficulties with internal flash ADCs are resolved by the novel architecture for a multibit single-stage delta-sigma ADC shown in [20]. The complexity of the huge internal quantizer is reduced by the segmented design that is suggested with coarse/fine DEM and DAC.

2.2.3 Pulse Width Modulation (PWM)

Pulse Width Modulation (PWM) is a technique where the width of a pulse in a train of pulses is varied to represent an analog signal. In PWM-based signal generation, the input signal is compared with a reference value, and the resulting error determines the width of

the pulses. The pulse train is then passed through a low-pass filter to obtain the desired analog waveform [50]. Its common applications include motor control, power electronics, and audio amplification, where precise control over voltage or current levels is required. An appropriate digital sequence can be sent to a DAC, for instance, to create a sinewave. The DAC output looks like a piecewise approximation of a sinusoidal wave. In other words, it looks an approximation of a sine wave made up of steps The sampling harmonics are then removed from the DAC output, resulting in a smooth sinewave. The resolution, amount of steps required to generate the waveform, conversion accuracy, linearity, and dc reference value affect the precision and any possible deformity of the resulting sinewave[55]. A PWM decoder in a DAC has been presented by Rowan et al [46] that translates the digital input code into a sequence of pulses, which are then utilized by the output-driver circuit to produce an output signal that consists of pulses with high and intermediate logic levels. A filter then integrates this signal to produce an analog output value that matches the digital input code. This variable pulse approach performs better than other PWM methods because it enables higher speed and resolution. The resolution can be enhanced by expanding the number of pulse periods incorporated into a certain conversion period.

2.2.4 Oversampling DACs

Oversampling DACs utilize sampling rates that are greater than the Nyquist rate to produce improved resolution and dynamic range. These DACs employ sophisticated digital signal processing techniques, such as interpolation and noise shaping, to generate high-resolution analog waveforms [53, 30]. By oversampling the input signal and employing advanced digital filtering, oversampling DACs can achieve higher effective resolutions and enhanced signal quality. Andrew et al [35] have integrated an oversampling DAC into ADSL central office modems using 0.18 µm CMOS technology. Signal reconstruction is done using their proposed

multi-bit semi-digital filter. They have also used a digital pre-emphasis filter to improve Signal-to-Noise ratio. A technique called dynamic matching is used to improve performance of oversampling DACs. This technique reduces harmonic distortion components by switching DACs in a specific pattern. It also improves linearity and accuracy by correcting distortions of output signal in real-time [33].

2.2.5 Interpolation Techniques

Interpolation techniques are used to generate intermediate data points between existing samples to improve waveform quality and achieve higher effective resolutions. Various interpolation methods, such as linear interpolation, polynomial interpolation, and sinc interpolation, can be applied to DACs to fill in the gaps between samples and reconstruct the waveform with increased accuracy and smoothness. This technique gives a lot of flexibility in signal generation. By inserting additional data points, the interpolated waveform can be upsampled to a higher sampling rate or downsampled to a lower sampling rate while maintaining the desired signal characteristics. This flexibility is particularly useful in applications where the DAC output needs to be synchronized with other components or when frequency translation is required. Sinusoidal interpolation gives a much smoother signal tan linear interpolation and this technique can be used in combination with oscillatory pulses combining DAC with an up-converter. The time domain interpolation can approximate the oscillatory signal to modulate the final DAC output to a higher Nyquist range [47]. The DDS and non-linear interpolation concepts can be combined to improve the spurious free dynamic range (SFDR), while simultaneously keeping the size of the DAC unchanged [38].

2.2.6 Noise Shaping Techniques

Noise shaping techniques are employed to shape the quantization noise in DACs, pushing it to higher frequencies where it is less perceptible or can be effectively filtered out. By carefully designing the digital filter in the feedback loop of the DAC, noise shaping techniques improve the signal-to-noise ratio and overall audio quality in audio DACs or the spectral purity in RF DACs. Improvements in noise shaping and SNR of around 10 dB can obtained by inserting a signal processing block before the Demodulator which is a filter that will shape the quantization noise and one block after the DAC which provides loop feedback to correct quantization error [19]. A fifth-order cascade of resonators feedback Delta-Sigma Modulation (DSM) with a multibit quantizer and the ”phase-switching” harmonic distortion cancellation technique can be employed to generate sine waves with low noise and high linearity. By leveraging the inherent oversampling nature of DSM, the ”phase-switching” technique addresses the reduction in bandwidth requirements, resulting in minimal additional hardware overhead. The required modifications primarily involve updating the data stored in the Look-Up Table (LUT), making the hardware impact negligible [28].

2.2.7 Software-Defined Radio (SDR)

Software-Defined Radio (SDR) is a technique where the modulation, encoding, and other signal processing functionalities are implemented in software running on a digital processing device, such as an FPGA or a computer. SDR enables the generation and reception of a wide range of waveforms by leveraging software algorithms for signal-processing tasks. It allows for the flexibility to change modulation schemes, adjust frequencies, and implement real-time signal processing tasks using programmable software [36, 48, 23]. SDR has gained popularity in wireless communication systems, radio astronomy, and research, as it offers versatility,

adaptability, and the ability to support multiple standards and waveforms through software configuration. FPGAs due to their reprogrammability has emerged as the primary platforms for SDR based signal generation. FPGA devices are primarily utilized for implementing digital baseband and intermediate frequency (IF) functionalities in radio communication systems. However, to meet the requirements of radio communication, a considerable number of analog and RF components are still necessary. RF signals are generated in the digital domain by an all-digital transmitter [57], which eliminates the need for a majority of the analog and RF components typically required. It comprises a QAM and RF pulse width modulator (RFPWM). The utilization of Scattering Parameters for characterizing RF DACs provides RF engineers with insights into the deviations between a particular device and its ideal performance.

An approach that generates a model resembling S-Parameters that captures the response from the spectral shape produced by the DAC and the reflection coefficient of its output impedance is proposed in [45]. This model can be seamlessly integrated into commercial RF/microwave simulators, enabling accurate prediction of the actual behavior of the RF DAC. As a result, the design effort is significantly reduced, allowing for more efficient development processes. Using SDR for generating signals is the most commonly used technique used in wireless communications. In [24], the proposed system contains a baseband processing unit operating at a low intermediate frequency (IF), an RF front-end for transmit (TX) and receive (RX) ends, and a millimeter wave RF front-end. The RF up-conversion and down-conversion procedures are fully independent of the creation and demodulation of the low-IF waveform. The prototype radio system's adaptability was increased by using this modular strategy, which enables signal transfers across many frequency bands. In other words, by solely modifying the RF front-end, the millimeter wave carrier frequency can be adjusted to align with the standards of other RF communication systems. SDR is

also used in long-range space communications.

Jet Propulsion Laboratory has developed a Universal Space Transponder [25], which allows simultaneous, multiband operations with data rates up to 37.5 Mbps RX and 300 Mbps TX. This is possible due to the high configurability of SDR systems, enabling only one device to perform all kinds of radio transmissions, thus reducing the number of devices and components on a spaceship. Apart from communication systems, SDR can also be used for quantum measurements. In the fields of quantum physics and nuclear science, Metallic Magnetic Calorimeter arrays are used for detecting signals. Read out of these arrays is a challenging task that can be done by SDR [49]. I and Q signal modulation using SDR allows more information to be carried which needs high fidelity and high bandwidth. The amplitude and phase of femtosecond laser pulses can be modulated to generate high-power signals with high-frequency carrier waves [27]. Microwave Kinetic Inductance detectors are another kind of high-precision devices which are used in quantum computing that requires SDR for read out operations [52].

2.3 Clock Data Recovery (CDR)

Clock Data Recovery (CDR) is a vital component in modern high-speed communication systems that ensures the accurate extraction of data from incoming signals. With the ever-increasing demand for faster and more reliable data transmission, CDR plays a crucial role in recovering the clock signal embedded in the received data stream, allowing for reliable and accurate data retrieval.

In high-speed communication systems, data is typically transmitted in the form of bits, where each bit is represented by specific voltage levels or symbols. The data stream is accompanied by a clock signal that indicates the timing of each bit. However, during transmission, various

factors such as noise, interference, and channel impairments can cause deviations in the received clock signal, leading to timing errors and loss of synchronization.

Clock Data Recovery addresses these challenges by extracting the clock signal from the incoming data stream and using it as a reference to correctly sample and recover the original data. The CDR circuitry employs advanced algorithms and techniques to track and synchronize with the received signal's clock frequency, phase, and amplitude variations.

The fundamental principle of CDR involves phase-locked loops (PLLs) or delay-locked loops (DLLs), which are key building blocks in the recovery process. These loops generate an internal clock signal that closely matches the timing of the incoming data, enabling accurate sampling and recovery. The PLL or DLL continuously adjusts its parameters to align with the variations in the received signal, ensuring proper synchronization and data extraction.

CDR techniques are employed in a wide range of high-speed communication systems, including optical fiber networks, wireless communication systems, high-speed serial links, and digital data storage devices. They are critical in enabling reliable data transmission over long distances, mitigating the effects of noise and signal impairments, and ensuring error-free data recovery.

In cases of video signal transmissions, as in the case of televisions, the communication is asynchronous, and has rail-to-rail non-return to zero (NRZ) signals. This requires a PLL using half clock frequency which can work at transmission rates of 1 Gbps, which has been built using 0.5 micrometer CMOS [41]. The use of half frequency reduces power consumption. A CDR with even higher frequency capabilities of 10 Gbps have been by Jafar et al [56]. By using an interpolated Voltage Controlled Oscillator (VCO), the performance is better than [41].

In contrast to the majority of other Clock Data Recovery (CDR) techniques that continu-

ously adjust the retrieved clock signal to align with the center of the data eye pattern, Blind Oversampling CDR operates differently. Instead, the incoming data stream is blindly sampled at a frequency, which is a multiple of the bit rate, choosing the most likely sample out of all of them to represent the right data. After collecting the samples, blind oversampling CDRs detect phase and recover data only in the digital domain [31]. Two commonly used BO CDR techniques are Direct phase picking (DPP) and Averaged phase picking (APP). However, the disadvantage is that these algorithms are not flexible in terms of oversampling rate. Travis et al [18] have proposed a delay window based CDR which is more flexible. The size of the window is adapted according to input data stream. A new window is started when an edge is detected. Another kind of clock data recovery algorithm is the Feed Forward CDR which operates by predicting the ideal sampling instant for each data bit based on past information and using this prediction to align the sampling clock with the incoming data [37].

In the following chapters, the usage of the AD9081 high-speed DAC/ADC will be discussed in addition to how the experimental setup was achieved and the steps to generate signals using AD9081 and then finally recovering the signal using a simple Clock Data Recovery algorithm.

Chapter 3

Experimental Setup

For our experimental setup, we are using the AMD-Xilinx FPGA called vcu118 and Analog Devices high speed ADC/DAC (AD9081). These will be the main components of the delay line memory. Another piece of hardware used is the Rhode & Shwarz 5 GSa/s oscilloscope (RTM3004) to analyze signal generated by the DAC of AD9081. The pre-requisite software for the setup are Xilinx Vivado 2019.1 (later versions can also be used), Xilinx SDK 2019.1 or later, a UART terminal like MobaXterm and IIO-Oscilloscope, which for Windows can be downloaded and launched directly but for Linux, many dependencies need to be installed separately. Figure 3.1 shows an overall block diagram of the hardware setup. The VCU118 FPGA is connected over a PCIe connection to the AD 9081, which is communicating with the FPGA through the JESD204B protocol. The FPGA is connected to the host machine through a serial JTAG connection and also through the LibIIO backend with the help of an ethernet connection. The host machine is running the IIO-Oscilloscope application which can connect to the FPGA using the LibIIO backend. The DAC0 port of AD9081 is connected to the RTM 3004 Oscilloscope, which writes the acquired data into .flt file. This file is read using the CDR algorithm and the bits are extracted after recovering the clock.

The first step is to clone the Analog Devices HDL project repository from github [6]. Since, Windows does not support GNU make, a separate software called Cygwin needs to be installed, which provides a UNIX like environment along with a command line for Windows. Once Cygwin is installed, we added the FPGA design tools paths to the PATH environment

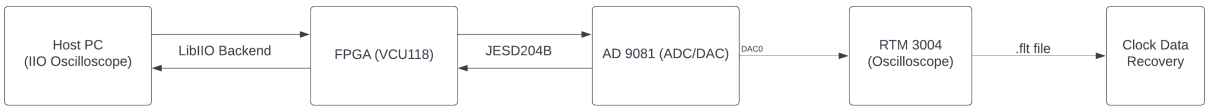


Figure 3.1: Block diagram of hardware setup

variable which was done by modifying the bashrc file. Now the analog devices hdl repository can be cloned into a new directory and the hdl_2019_r2 branch was checked out. This repository contains Analog Devices HDL libraries which can be used for various reference design and prototyping systems. Analog Devices also provides Tcl scripts for building an FPGA design. After checking out the suggested branch, we navigate to the projects directory, we find many folders for different platforms and hardware. We navigate to the ad9081_fmca_ebz folder and then into the vcu118 folder, where there is a Makefile and we run make. Make will build all the libraries needed for the project and then the project. The licenses for all the software need to be setup correctly otherwise, make will fail.

The next step is to build the Linux kernel for the Xilinx Microblaze Softcore processor on the vcu118. We clone the analog devices Linux repository [7] and checkout the master branch. Some environmental variables also need to be set. The variable ARCH is given the value microblaze and CROSS_COMPILE is given the value microblazeel-xilinx-linux-gnu. We then configure the kernel by running make adi_mb_defconfig. Now, we download the root file system rootfs.cpio.gz which must be placed in the root of the kernel tree (/linux/rotfs.cpio.gz). This contains everything needed to support a full Linux file system. There are several device tree source (dts) files in /linux/arch/microblaze/boot which will be used to build our kernel. The dts files provide hardware description to the Linux OS. Next, we build the kernel using the command make simpleImage.<dts file name> which results in a .strip file which needs to be loaded onto the FPGA. We make a working directory and then copy the system_top.bit which has been built in <hdl_rep_dir>/projects/ad9081_fmca_ebz_vcu118.runs/impl_1

when we built the hdl project. This file is a bitstream which has the configuration information for the FPGA. We also copy the .strip we generated into the same directory. Now to load the .strip file onto the FPGA, we use the xsdb (Xilinx software debugger) or xsct (Xilinx software command line tool) terminal. Both xsdb and xsct provides a terminal and is an interactive and scriptable command line interface to Xilinx software development tools. They both use the Xilinx hw_server as the underlying debug engine. Either the run.tcl script can be sourced in the xsct or xsdb terminal. The tcl script contains code to first connect to the fpga and then load the bitstream (system_top.bit). After that the bitstream is mapped, the microblaze processor can then be the target for the .strip file, which will then load the linux image built onto the microblaze processor. In order to view the linux boot messages and give commands to the linux running on the microblaze processor, we use MobaXterm, where a serial port communication with the appropriate COM port is established at 115200 baud rate. After the boot load messages we can finally log in with the username root and password analog. To view the hardware connected to the FPGA, we use the command iio_info | grep iio:device. This will show three devices connected to the FPGA – namely iio:device0: hmc7044, iio:device1: axi-ad9081-tx-hpc (buffer capable) and iio:device2: axi-ad9081-rx-hpc (buffer capable). The JESD connection can also be checked with the command jesd_status -s which shows the device clock and lane rates and status of the lanes. Now, the IP of the fpga is found using ifconfig and this will be used to remotely connect IIO-Oscilloscope software with the FPGA. IIO-Oscilloscope, developed by Analog Devices Inc., is a cross platform GUI platform and lets us interface different evaluation boards from within a Linux system. The application can be used to capture and view waveforms in both time and frequency domains. It allows connection of both local and remote devices. We will be using remote connection and connect to the device using its IP address. There are multiple tabs and plugins available on the main window [8] as shown in figure 3.2.

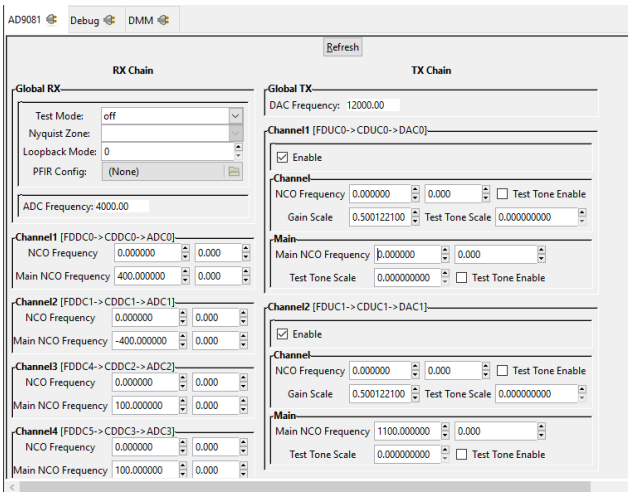


Figure 3.2: AD9081 plugin in IIO-Oscilloscope

- DMM plugin/tab – This is a digital multimeter and can be used to display data about the device selected from the same tab. In our case, the device is axi-ad9081-rx-hpc and the data we can view is the temperature of the device.

- Debug plugin/tab – This tab allows debugging and changing and viewing SPI or AXI register values of the selected device. The devices available in our case are axi-ad9081-rx-hpc (receiver or ADC), axi-ad9081-tx-hpc (transmitter or DAC) and hmc7044 (clock).

- Device specific tab/plugin – In our case this tab is for the AD9081 and it allows changing of the Nyquist zone, test mode, NCO frequencies of the different channels of both DAC and ADC. It also allows us to alter the gain scale, test tone scale of different channels and load pre-defined waveforms into the DAC buffer.

The capture window is the window where we can view the waveform or data captured by the ADC or the axi-ad9081-rx-hpc. It lets us select which channel's data needs to be viewed.

Chapter 4

LibIIO

Libiio, developed by Analog Devices Inc., facilitates the development of software that interfaces with Linux Industrial I/O (IIO) devices. By abstracting the hardware's low-level features and offering a complete programming interface that can be utilized for complex applications, Libiio makes the development process simpler. Libiio serves as an interface to the IIO Subsystem [9, 10]. This subsystem provides support for various devices that involve analog-to-digital or digital-to-analog conversion, such as ADCs, DACs, accelerometers, gyros, pressure sensors, temperature sensors, and RF transceivers, among others. Users can utilize libiio in two ways: by directly using it on an embedded Linux target (local mode) or by remotely communicating with the target using libiio from a host Linux, Windows, or MAC system via USB, Ethernet, or Serial connections.

While Analog Devices Inc. developed libiio primarily, it is an open-source library with contributions from numerous individuals. It can be used on any processor, FPGA, or SoC from any vendor that can control any peripheral device locally or remotely under the terms of the GNU Lesser General Public License (LGPL), version 2.1 or later, which permits users to use the library in both commercial and non-commercial applications. However, users must adhere to the LGPL license obligations, freedoms, and restrictions. Additionally, the GNU General Public License (GPL), version 2.0 or any later version, is available separately for the libiio (iio-utils) examples and test programs. One high-level API and a number of backends, including the ”local” backend and the ”network” backend, make up the library.

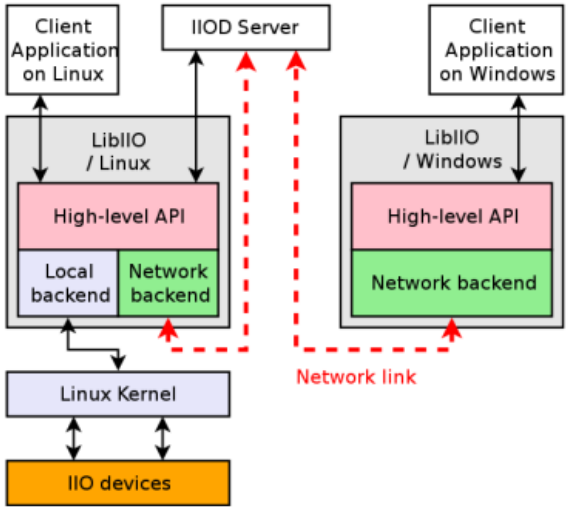


Figure 4.1: Libiio Software Stack [9]

The sysfs virtual filesystem is used to interface the ”local” backend of Libiio with the Linux kernel, allowing users to access and control IIO devices locally. This backend is particularly useful for applications that require low latency and high bandwidth. The ”network” backend of Libiio, on the other hand, communicates with the iiod server over a network link, allowing for the remote access and management of IIO devices. This backend is ideal for distributed applications that need to access IIO devices from multiple locations. One example of an application that utilizes Libiio is the IIO Daemon (IIOD) server, which is a software component that provides access to IIO devices on a network. The IIOD server creates a Libiio context that uses the ”local” backend to communicate with the Linux kernel and control IIO devices locally. It then shares this context on the network using the ”network” backend, allowing client applications to remotely access and control IIO devices. The software stack for a network connection is shown in figure 4.1

The API of Industrial Input Output subsystem is exposed through the sysfs in the path /sys/bus/iio/devices/*. Libiio is responsible for :

Figure 4.2: IIO devices in the project

- Identifying any IIO device connected to the system attributes of the device and channel as shown in figure 4.2.
- Finding channels for the device: There are multiple channels in iio:device1 as shown in figure 4.3. For example for out_voltage0_i_calibphase, the id of the channel is voltage0 and calibphase is the name of the attribute. Out specifies the direction of the channel and i is the name of the channel. Scan elements is a folder which has all the channels which can support streaming data, i.e. capturing samples from hardware or sending samples to hardware
- Identifying attributes specific to a channel as mentioned above.
- Identifying shared attributes – Here for example, the attribute sampling frequency is shared by all out_altvoltage channels.

The Libiio API is built majorly out of 4 datatypes:

- iio_context
- iio_device
- iio_channel
- iio_buffer

The hierarchy of the datatypes is as shown in figure 4.4. iio_context can have number of devices ranging from 0 to any number. iio_device can have 0 or 1 buffer and 0 or multiple

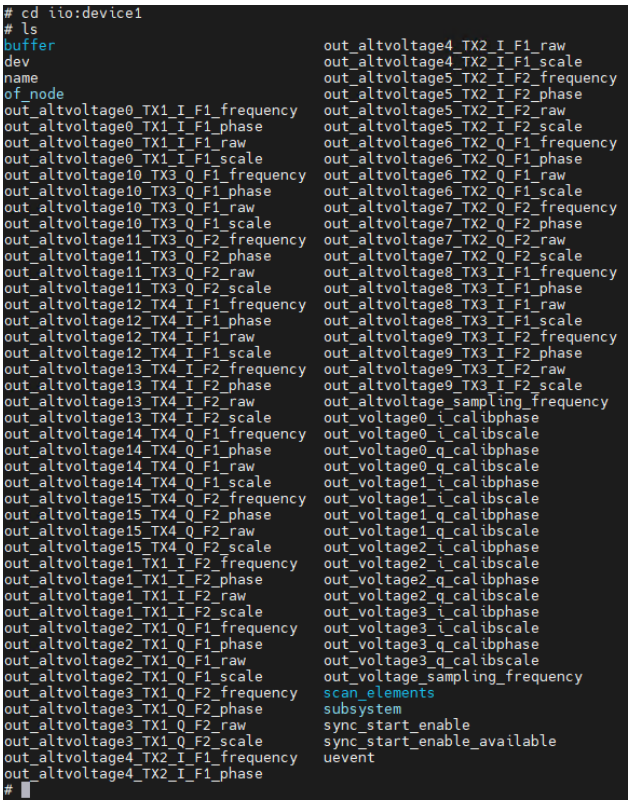


Figure 4.3: Channels for iio:device1.

channels. Each of the iio_context, iio_device, iio_channel and iio_buffer can have multiple attributes.

The buffer data format represented as text as reported by kernel is shown in figure 4.5. le symbolizes that the data is stored in little endian format. 16 bits is the sample size and the "»0" symbolizes that the full 16 bits is used to represent the data sample. S16 shows that the value is 16 bit signed.

The following are a few important libiio APIs that have been used in the project [11, 12, 13, 14]:

- iio_create_default_context – Used to create a context from local or remote IIO devices. It returns a pointer to a iio_context structure.

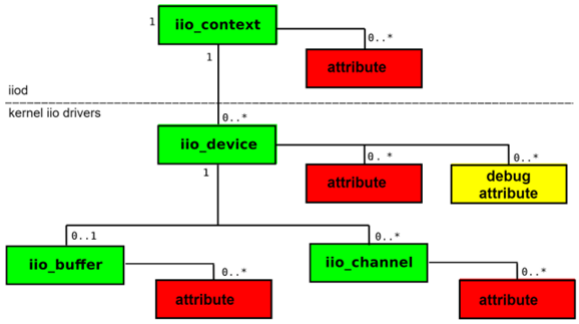


Figure 4.4: Heirarchy of different libiio datatypes [9]

```
#  cat out_voltage0_i_type
le:s16/16>>0
#
```

Figure 4.5: Data format for iio:device1 buffer.

- unsigned int iio_context_get_det_device_count (const struct iio_context *ctx) – It will return the number of devices associated with to the passed context structure.
- char * iio_context_get_name (const struct iio_context *ctx) – Used to return a string which has the name of the passed context structure.
- struct iio_device * iio_context_get_device (const struct iio_context *ctx, unsigned int index) – This API takes an iio_context structure and an index as arguments and returns a pointer to the iio_device structure at that index
- void iio_context_destroy (struct iio_context *ctx) – This is used at the end of the program to destroy the context .
- struct iio_device * iio_context_find_device (const struct iio_context *ctx, const char *name) – Similar to the previous one, this function takes an iio_context structure and name of the device to return the pointer to the corresponding iio_device structure.
- const struct iio_context* iio_device_get_context(const struct iio_device * dev) –

Used to return the iio_context structure pointer of which the passed iio_device structure is a part of.

- struct iio_channel* iio_device_get_channel(const struct iio_device * dev, unsigned int index) – This API is used to get the iio_channel structure pointer associated with the passed iio_device structure.
- unsigned int iio_device_get_channels_count(const struct iio_device * dev) – returns the number of channels that the iio_device structure has.
- const char* iio_device_get_name(const struct iio_device * dev) – Used to return the name of the device
- const char* iio_device_get_id(const struct iio_device * dev) – This function returns the id of the device . For example iio:device0.
- struct iio_channel * iio_device_find_channel (const struct iio_device *dev, const char *name, bool output) – This function takes an iio_device structure, name of the channel and 1 for if the channel is output or 0 otherwise and returns the pointer to the channel structure.
- const char * iio_channel_get_id (const struct iio_channel *chn) – It is used to get the id of the iio_channel structure passed. Example - voltage0.
- const char* iio_channel_get_name(const struct iio_channel * chn) - Returns the name of the channel.
- void iio_channel_enable (struct iio_channel * chn) – Used to enable a particular channel.
- void iio_channel_disable (struct iio_channel * chn) – Used to disable a channel.

- struct iio_buffer * iio_device_create_buffer (const struct iio_device *dev, size_t samples_count, bool cyclic) – This API is used to create a buffer for input or output connected to the specified device. It takes a pointer to the device structure, the number of samples to be stored and true or false to enable cyclic mode.

- void* iio_buffer_start(const struct iio_buffer * buf) – This is used to return a pointer which points to the starting address of the buffer.

- void* iio_buffer_first(const struct iio_buffer * buf, const struct iio_channel * chn) -This is used to find the first sample in the buffer. The function will return a pointer to the first sample of the buffer.

- void* iio_buffer_end(const struct iio_buffer * buf) – This returns a pointer to the address that is right after the last element in the buffer.

- ptrdiff_t iio_buffer_step(const struct iio_buffer * buf) – Used to get the size of the step between 2 consecutive samples in a buffer

- ssize_t iio_buffer_push(struct iio_buffer * buf) – This is used to send or push samples from the buffer to the hardware. The buffer should be an output buffer. It returns the number of bytes pushed.

- ssize_t iio_buffer_push_partial(struct iio_buffer * buf, size_t ,samples_count) – It works similar to iio_buffer_push but here the number of samples to be sent can be specified.

- ssize_t iio_buffer_refill (struct iio_buffer * buf) – This is used for input buffers to get samples from the hardware and fill the buffer.

- void iio_buffer_destroy (struct iio_buffer * buf) – This is called at the end of the code after the streaming of data is complete to destroy the buffer. This is usually followed

by disabling the channel and destroying the context.

Chapter 5

Digital to Analog Converter (DAC)

The AD9081 comes with 4 DAC channels and a maximum sampling rate of 12 GSPS if used with JESD204C interface with the FPGA. The device also supports SERDES JESD204B/C interface with 16 lanes up to a speed of 24.75 Gbps. AD9081 provides DAC ADC sample ratios of 1,2,3 and 4. AD9081 also provides 8 fine Digital Up Converters (FDUC) and 4 Coarse Digital Up Converters (CDUC) each of which can be configured depending on the users' requirements [4]. In this chapter, the general working and the utilization of libiio APIs to generate signals using the DAC will be explained.

5.1 JESD204B/C

With the increasing speed of the ADC/DACs, it is necessary to match this speed through a high speed data transfer protocol between the FPGAs and ADC/DAC. JESD204B/C is a high-speed serial interface standard widely adopted in the field of data converters, providing a reliable and efficient means of transferring data between ADCs or DACs and digital signal processing devices. To create a uniform framework and minimize the amount of data inputs/outputs between high-speed data converters and other devices like FPGAs, the JEDEC committee created the JESD204, JESD204A, JESD204B, and JESD204C data converter serial interface standards[15]. This standardization aims to simplify system layout, enabling smaller form factors without compromising overall performance. These character-

istics are particularly valuable in addressing size and cost limitations in various high-speed ADC applications. These applications include wireless infrastructure technologies (such as GSM, EDGE, W-CDMA, LTE, CDMA2000, WiMAX, TD-SCDMA), transceiver architectures, software-defined radios, portable instrumentation, medical ultrasound equipment, as well as Mil/Aero applications like radar and secure communications.

The JESD204B version introduced significant enhancements compared to its predecessor, JESD204A. It introduced a new coding scheme, robust synchronization mechanisms, and improved performance in terms of data rates and signal integrity. JESD204B supports data rates ranging from a few megabits per second (Mbps) to several gigabits per second (Gbps), making it suitable for both low- and high-speed applications. With the release of JESD204C, the standard further evolved to meet the increasing demands of high-performance systems. JESD204C added features such as higher data rates, deterministic latency, and support for multiple devices within a single interface link. It also introduced additional flexibility in terms of lane assignment and reduced overhead in terms of line coding.

The JESD204B/C interface is divided into 4 layers as shown in figure 5.1.

- Physical Layer (PHY): The PHY layer is responsible for the physical transmission of data and controls information over high-speed serial links. It handles the encoding and decoding of data, clock synchronization, and lane alignment. The PHY layer operates at the electrical level, taking care of voltage levels, timing, and signal integrity.
- Data Link Layer (DLL): The DLL layer ensures reliable and error-free transmission of data between the transmitter and receiver. It manages frame alignment, error detection, and error correction using cyclic redundancy check (CRC) codes. The DLL layer also handles flow control, buffering, and scrambling to optimize data transfer.
- Transport Layer (TL): The TL layer deals with higher-level functions, including frame

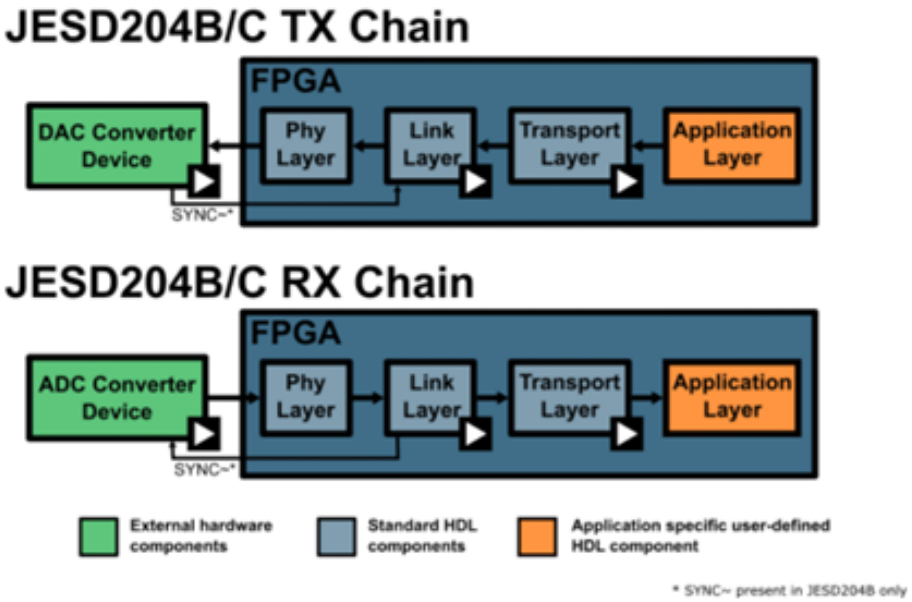


Figure 5.1: Overview of JESD204B/C[15]

structure, synchronization, and multi-frame alignment. It facilitates the organization and management of data frames and multi-frames, allowing for efficient data transfer and synchronization across the interface.

- Application layer: The application layer in JESD204B/C is where additional protocols, processing algorithms, and system-specific functionalities are implemented to facilitate the integration and interaction between the JESD204B/C interface and the rest of the system. It includes the software and firmware components that handle the processing and utilization of the data transferred over the JESD204B/C interface.

There are a few parameters that need to be adjusted for the seamless performance of the interface:

- Lanes (L): The number of lanes, denoted by L, represents the quantity of high-speed serial links used for data transmission. Lanes carry both data and control information,

and their count can range from 1 to 32. Increasing the number of lanes allows for higher data throughput and parallel transmission.

- Converter Samples per Frame (M): M refers to the number of converter samples contained within each frame. Frames organize data into discrete chunks, and M can vary from 1 to 16,384, accommodating different sample rates and system configurations. The value of M impacts the frame size and overall data rate.

- Frames per Multi-Frame (N): N signifies the total number of frames within a multi-frame synchronization sequence. Multi-frame synchronization ensures accurate data alignment and synchronization between the data converter and receiving device. N ranges from 1 to 256 and influences synchronization patterns and initialization time.

The JESD204B/C configuration can be controlled or viewed using the JESD_MODE register (0x01FE [5:0]). These values are set while loading the linux image onto the FPGA. The JESD mode for the project is mode 9 (M=8, L=4, N=16).

5.2 Transmit Digital Path

Figure 5.2 shows the whole transmit digital datapath. As mentioned before, the main digital path (CDUC) and the channelizer data path (FDUC) can be configured or bypassed by changing the values of some registers which will be discussed in more detail. The link between the FDUC and the CDUC is an 8x8 crossbar multiplexer which maps the output of the FDUC to the main data path. In case the channelizer path is bypassed, a 4x4 crossbar multiplexer is used to map the data to the CDUC. There are selectable interpolation filters that can be used to achieve the interpolation, the user needs. The total interpolation is the product of the main path interpolation MTX and channelizer interpolation NTX. To

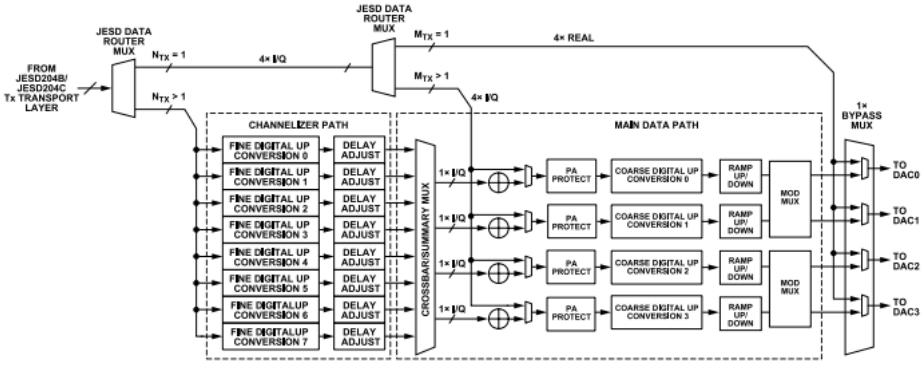


Figure 5.2: Overall Transmit digital datapath [4]

bypass the main and channelizer data path, the interpolation should be 1. The values to which MTX and NTX can be set to can be restricted because certain JESDB/C settings won't allow certain combinations of MTX and NTX. To change the interpolation settings, the value of the register 0x01FF needs to be changed, where the 4 Most Significant Bits is the coarse interpolation select (N) and the lower 4 bits is the fine interpolation select.

When the channelizer data path is enabled, every I/Q data point is sent to a pair of virtual converters – one for I data and the other for Q data. Assignment of the converters is done in sequential fashion. For instance, the I/Q data samples going into the Virtual converter 0 and 1 are mapped to FDUC0. Virtual converters 2 and 3 will be mapped to FDUC1 and so on. In case of dual link JESD operation, samples from the first link are mapped to FDUC0 through FDUC3 and the samples from the second link are mapped to FDUC4 through FDUC7. The 8x8 crossbar multiplexer can be configured to map output from FDUC to any of the four CDUC.

In case the FDUC shown in figure 5.3 is bypassed and the main data path, that is the CDUC, shown in figure 5.4 is enabled, the mapping is similar to what happens in the channelizer data path. The first pair of I/Q samples are sent to CDUC0 and then the next pair to CDUC2 and son on till CDUC3. In case of dual link operation, the data from the first link are mapped to

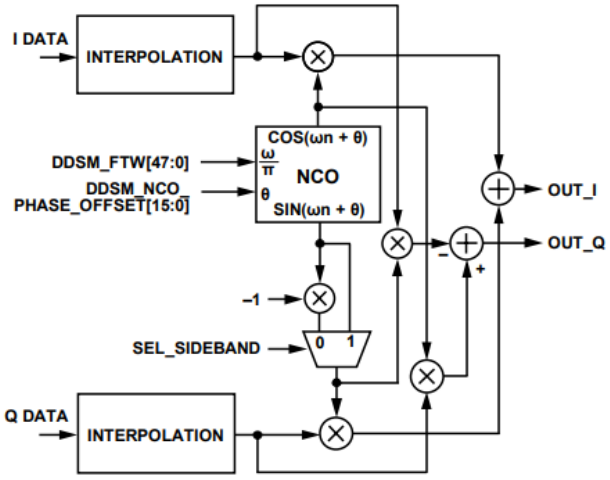


Figure 5.3: Fine Digital Up-Converter in detail [4]

CDUC0 and CDUC1 and data from second link is mapped to CDUC2 and CDUC3. Mapping of the output from CDUC to the DAC cores is also sequential. For instance, CDUC0 maps to DAC0, CDUC1 to DAC1 and so on. This mapping is also configurable and can be done by configuring the Modulator Mulitplexer. If both the channelizer and the main data paths are bypassed, the mapping from JESD204 B/C link is again sequential as mentioned before.

The I/Q data coming from the JESD204 B/C link need not be in quadrature for proper functioning of the device. When the NCO for both the main and channelizer data paths is bypassed, it can be useful to send any data in the I/Q datapath. The absence of NCO means that the signal will keep its original form and will undergo only interpolation. This is the case which is used for this project. The two data streams from CDUC can then be sent to 2 DAC cores by splitting them into 2 streams or only one stream can be kept and transmitted to a DAC core. For this project I/Q quadrature data samples are not used. Instead, only I data is sent and the NCO is kept at zero.

While mapping to the main datapath's CDUC, usually the default sequential mapping is sufficient. However, a 4x4 crossbar multiplexer is also provided so that the user can control the mapping. For example, the user is using 2 DACs and to achieve maximum isolation between

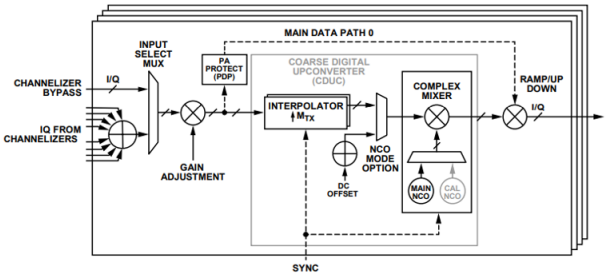


Figure 5.4: Fine Digital Up-Converter in detail [4]

the 2 data paths, the 4x4 multiplexer can be used to map the data to CDUC0 and CDUC3. The DACPAGE_MSK register (0x001B) is used for this purpose. The bitfield corresponding to the CDUC needs to be set. The bit 7 of the MAINDP_DAC_1XXX_ENABLES register (0x01C8) can be set to enable remapping. To configure which pair of I/Q data goes to the selected CDUC, set the corresponding bit in Bits[3:0] of the same register (0x01C8).

The channelizer data path gets input from the JESD204 B/C link if fine interpolation factor is set to greater than 1. The DACCHAN_MSK bit field or register 0x001C can be used to configure different parameters like the NCO, gain and skew delay for each channelizer path. Each bit in the aforementioned register corresponds to the number of the channelizer path. These parameters can also be modified from the AD9081 plugin tab of the IIO-Oscilloscope which is easier than changing the register values in binary or hex because we can directly input the number in decimal to plugin.

The register 0x01FF is used to change the interpolation settings for the data. The Bits [3:0] or FINE_INTERP_SEL bit field can be modified and according to the values the interpolation filter, as shown in figure 5.5, can be selected. The maximum fine interpolation possible is 12.

The DDSM_FTW 48 bit register is used to change the value of the NCO frequency. The PHASE_OFFSET 16 bit register is to change the phase offset of the Numerically Con-

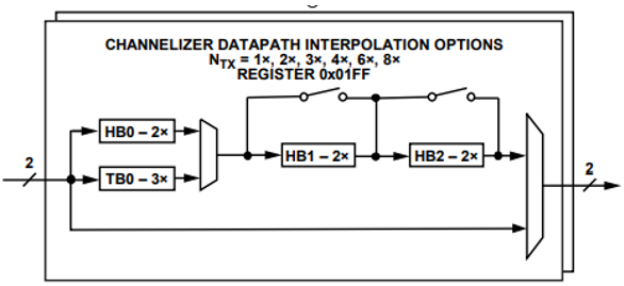


Figure 5.5: Interpolation filter [4]

trolled Oscillator. The polarity of the NCO carrier frequency can be changed by setting the DDSC_SEL_SIDEBAND bit to 1. This is analogous to swapping the I and Q data. To get a resolution of more than 48 bits for the NCO frequency, the NCO dual modulus mode of operation can be used and can be set by setting the DDSC_MODULUS_EN bit of register 0x01A0 (bit 2) to 1.

The main data path is similar to the channelizer path in terms of functionality with a few extra blocks. It has an additional input summation block, a power amplification protection block, and also provides additional 32-bit calibration for the CDUC NCO. The coarse interpolation can be set using the COARSE_INTERP_SEL bit of register 0x01FF (bits [7:4]). The maximum coarse interpolation is 12. Gain adjustment can be done through a 12-bit word which spans over bits [11:8] of register 0x20D4 and bits [7:0] of register 0x20D3. It can also be changed from the AD9081 plugin tab in IIO-Oscilloscope.

The modulator multiplexer (Mod Mux) maps the output of the main datapath to the DAC cores for final output. There 2 of these multiplexers. The MODS_MSK bit field of register 0x001D (bits[3:2]) is used to select between the two multiplexers. Bit 3 selects mod mux 1 and bit 2 selects mod mux 0. The AD9081 provides 6 possible mod mux configurations.

The configuration of the mod mux is dependent on the users' end application. For example, configurations 1 or 3 can be used when the application does not need any additional upcon-

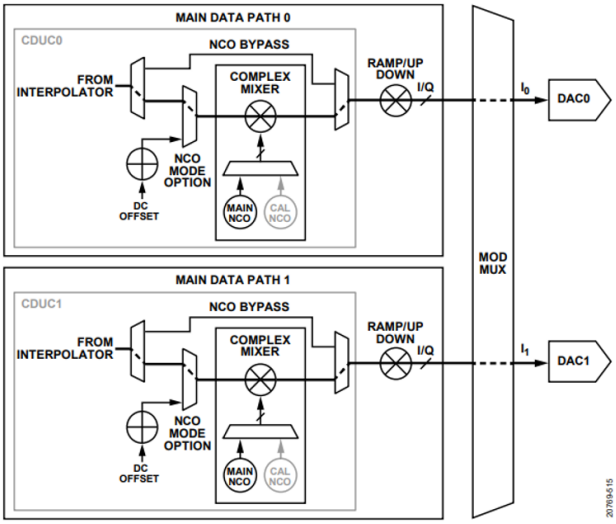


Figure 5.6: Mod Mux configuration 0 [4]

version stages. For applications where I and Q have been reconsctructed, sent to two DAC cores for output and needs upconversion using an I/Q modulator. The summation of two CDCUC outputs is also an option. This is more prevalent in multiband applications where the two CDUC are operated as a wideband channel. Since there can be a problem of digital clipping when the two CDUC outputs are added, the outputs are scaled by 0.5 as seen in configurations 1,3 and 3A. For the project, the configuration 0 shown in figure 5.6 is used as the signals are non-IQ signals and the streams are interpreted as 2 separate streams and need the datapath only for interpolation. Here there is only I data and so the output from DAC0 needs to be captured. The register that dictates these configurations is called the DDSM_DATAPATH_CFG register (0x01C9).

Chapter 6

Generating Signal Using DAC

DACs provide a lot of flexibility in generating precisely controlled and timed signals which can be used for a plethora of applications. They can be controlled by an FPGA which runs C code to control the operation of the DAC. This chapter discusses in detail how libiio is used to generate different kinds of signals ranging from sinusoidal waves to square waves and triangular waves to a random stream of bits. As mentioned in the previous chapter, the libiio library can be used to interact with the Industrial Input Output (IIO) ports of any linux device, in this case, the Microblaze processor running on the FPGA. The DAC is attached to the FPGA using the PCIE connections to the IIO ports which makes it easier to use the libiio APIs rather than using the API developed by Analog Devices for AD9081. Through the libiio daemon backend, it is also possible to connect the ADC/DAC to IIO-Oscilloscope which provides a comprehensive GUI to configure and debug the DAC.

6.1 Setting Up Channel And Buffer

The header files required for the project are stdbool.h, stdint.h, signal.h, stdio.h, iio.h, math.h, unistd.h, time.h. After including the headers it is also necessary to define the size of the buffer. The receiver and the transmitter buffer sizes are defined in the hdl files that were used to build the linux image. In this case, the size is defined as 64*1024 samples. The initial setup also includes setting all the device context pointers, receiver and transmitter

device and channel pointers (for DAC operation only transmitter channels are required) and the buffer pointers to NULL. The flow of operation goes according to the hierarchy of libiio structures:

- Creating context – This is done using the libiio function called iio_create_default_context() which returns the pointer to the newly created local context. The name of the context can be displayed using iio_context_get_name() function as shown in figure 6.1.
- Getting device pointer – The device pointer is extracted using iio_context_get_device() function. The number of devices and the name of the respective devices, shown in figure 6.1, can be found using the iio_context_get_devices_count() and iio_device_get_name() functions respectively as shown in figure 1. Here there will be 3 devices – HMC744 (clock), axi-ad9081-tx-hpc (DAC), axi-ad9081-rx-hpc (DAC).
- Getting channel pointers – Now that the particular device pointers are extracted, the pointer for channels associated with the DAC needs to be found using iio_device_get_channel(). The number of channels, their names and id can be found using the iio_device_get_channels_count() and iio_channel_get_id() respectively, shown in figure 6.2. After choosing which channel to use, the channel is enabled using the iio_channel_enable() API. As seen in figure 2, there are 17 channels in the DAC device.
- Creating buffers – Buffers can be created using the iio_device_create_buffer() API, which will return a pointer to the buffer

6.2 Generating Periodic Signals

Periodic signals can be generated using the built-in functions of the C math library, for instance sine, square, and triangle waves. These signals were used initially to test the

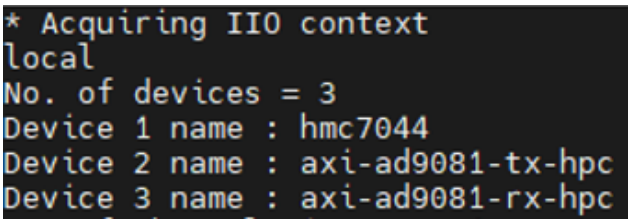


Figure 6.1: Devices shown by the C code using libIIO APIs

working of the DAC. The sampling rate and amplitude need to be defined. Here the sampling frequency is defined as 250 MHz since the linux image that was used, set the sampling frequency to that value. The amplitude needs to be less than 32767, since the each buffer memory address stores 32 bits, 16 for I data and 16 for Q data. We are sending only I data into voltage0_i channel, so for signed integers, 32767 is the maximum value that can be stored in 16 bits. If it goes beyond that value, there will be overflow and the signal will be inverted (180 degrees out of phase) and with much less amplitude. Since these are periodic signals, all 1024*64 samples are not generated. Samples for only one time period are generated and the rest of the samples are filled by cycling over the samples of one time period. The samples are then put into the buffer created for the DAC device and sent to the hardware using iio_buffer_push(). The waveforms captured by the oscilloscope are shown in figures 6.3, 6.4 and 6.5.

6.3 Generating Random Bits

After completing the tests on periodic signals, a random stream of bits was generated and sent to the DAC hardware, from which the bits were later recovered using Clock Data Recovery discussed in detail in the next chapter. To save processing time, the bits were generated using C code on the local machine. The stream is 32*1024 samples long (half buffer). The first 800 bits are defined as start bits which consist of alternating 20 high (+amplitude) and

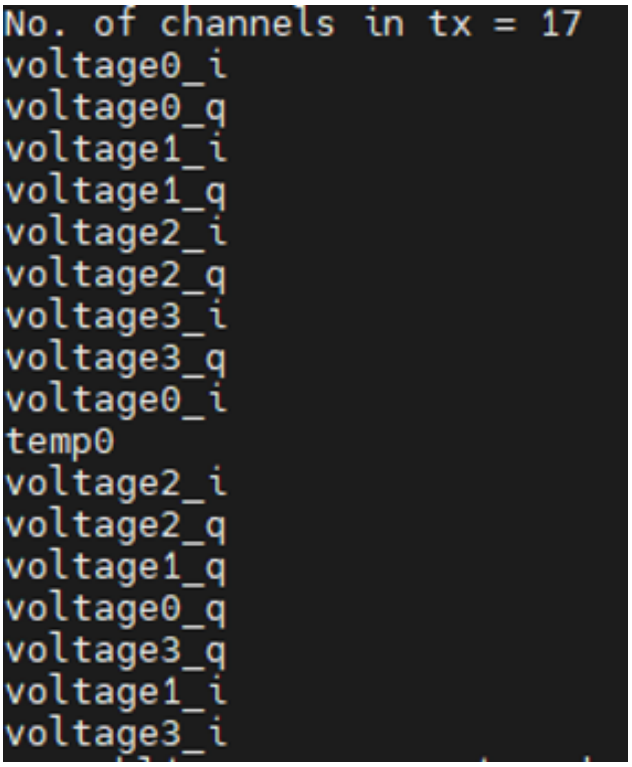


Figure 6.2: Channels shown by C code using LibIIO APIs

20 low bits (-amplitude). 20 bits are chosen to give the voltage enough time to settle and give clear edges so that the clock can be recovered easily. Several different amplitudes were tested, the maximum being 30000, to keep the buffer from overflowing. After the first 800 bits, 200 bits were set to 0. This was done to signify the end of the start bits, which will be later used to recover the clock, and the beginning of the main data bits. Different baud rates or bits per symbol were tested, which will be discussed later in the results section.

While testing the operation of the DAC, it was observed that when a full buffer is used, there is an overlap of two consecutive buffer pushes to the hardware. The reason behind this is that the libiio software is not able to keep up with the push to the hardware. There is some mismatch in speed in populating the buffer and sending the data out by the hardware. Due to the speed mismatch between the hardware and software, there is an overlap of the data and hence only the later half the buffer is filled with the bit stream while the rest of

Figure 6.3: Sample sine wave generated by the DAC

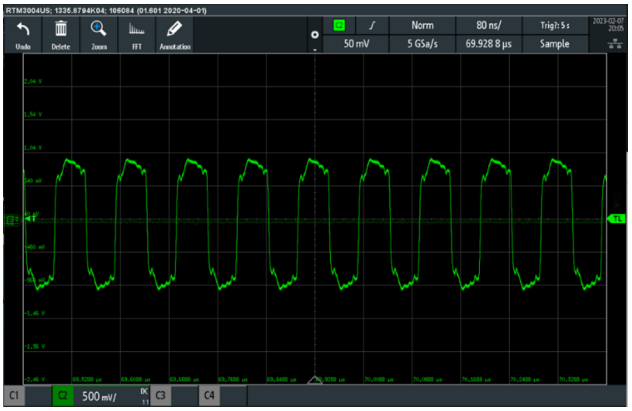


Figure 6.4: Sample square wave generated by the DAC

the buffer has 0. So, even if there is an overlap it will be in the first half of the buffer, and the later part where the data is, will be intact and unaffected.

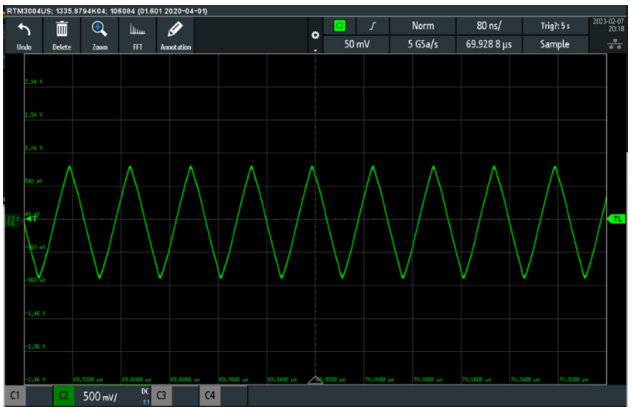


Figure 6.5: Sample triangle wave generated by the DAC

Chapter 7

Clock Data Recovery

Recovering the bits sent by the DAC and acquired by the oscilloscope is required for testing the operation of the DAC. The oscilloscope and the DAC do not share a clock. Therefore, the communication between the DAC and the oscilloscope is asynchronous, which means the data sampling by the oscilloscope from the serial data needs to be synchronized with the clock of the DAC. For this purpose, we need to get the clock of the DAC to correctly recover the bits from the data stream sampled by the oscilloscope.

The RTM3004 oscilloscope stores the acquisition data in multiple formats - .txt, .csv, .flt and .bin. The .txt and .csv are much larger in size as compared to the .bin and .flt formats. The documentation of the RTM3004 oscilloscope does not provide much information about the .bin file. Therefore, the .flt file format was chosen for processing the data. The .flt format stores the streamed data samples in IEEE 754 floating point 32 bit format, i.e. 4 bytes is used to store one sample. The processing is done in python since python provides multiple libraries which can be used to unpack streaming data and for visualizing the acquired data. The entire algorithm can be divided into the following major steps:

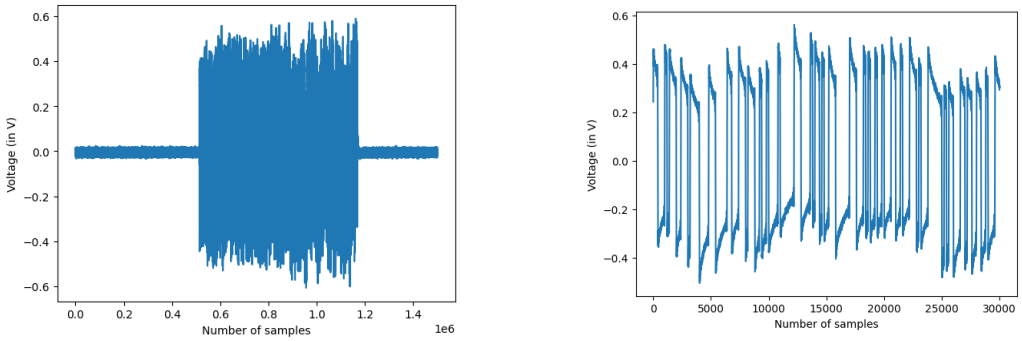
- Extracting and unpacking the voltage values from the .flt file
- Edge detection
- Recovering the clock from the start bits

- Sampling the data according to the positive clock edges

7.1 Extracting And Unpacking The Voltage Values

The struct library in python is used to unpack data from the .flt file and convert the values from IEEE 754 floating point binary format to decimal format. The struct module in python is a standard module used for dealing with data streams. It contains different functions which are used to convert values between C structs, which are represented as python strings and python values. The struct module is used to handle binary data from data streams, databases, and network connections. Format specifies can be used to indicate to the module which format the data needs to be converted to. The endianness of the byte can also be specified.

Here the unpack function from the struct module is used to convert the binary data to big endian float values in python. First the data in binary is read from the .flt file using the read_bytes function of python and stored in a list. Since in the specified IEEE 754 floating point format, one sample value is stored in 4 bytes, a packet of 4 consecutive bytes is unpacked using the unpack() function in the form big endian float and appended to a different list. This gives a list that has all the samples acquired by the oscilloscope. The oscilloscope acquisition memory was set at 1.5 million samples to capture a sufficient amount of data. After extracting the data into a list, it was plotted using Matplotlib, and the output is shown in figure 7.1.



(a) Entire data

(b) Data zoomed in

Figure 7.1: Data plotted after unpacking from .FLT file

7.2 Edge Detection

The next step is to identify the edges in the data stream. This is necessary since the edge signifies change in value from high to low or low to high and this means that there was a positive clock edge of the DAC at that point. This gives a starting point for reconstructing the clock. The simplest way to find a transitioning edge in the data stream is to use the slope of two consecutive data points. When the slope between 2 consecutive points is small, that signifies that there are no transitions or edges. Conversely, there is a transition or edge when the slope is high. This was the main principle behind edge detection. The steps used to identify the edges are as follows:

- Traverse the list with the samples, calculate the difference between consecutive values, and store the difference in a separate list.
- Plot the list using the matplotlib library's plotting tools. This makes it easier to visualize and determine the threshold to classify a slope value as a rising or falling edge. There should be a positive and a negative threshold to classify a rising and falling edge.
- Whenever a slope value exceeds the threshold value, append 1 into a separate list

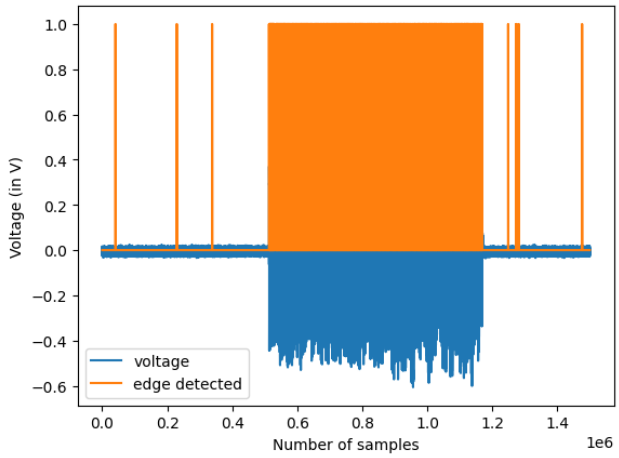


Figure 7.2: Edges Detected (includes outliers)

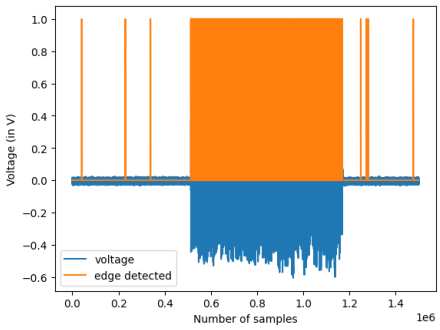
(called slope_threshold), otherwise append 0. This list will help to keep track of the indices where edges can be present.

After completing these steps, the slope_threshold list was plotted along with the voltage values extracted from the file, as shown in figure 7.3.

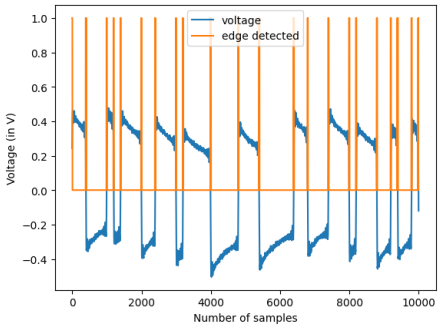
As observed from figure 7.3, the edge detection is not completely correct. The above steps can detect edges but it also detects certain cases where the noise may have caused the slope values to go beyond the threshold. These outliers need to be removed. The logic used for this purpose is that the correct edges are present where the value of the voltage itself is beyond a certain threshold. The steps used are as follows:

- Run a loop traversing the entire list of samples and generate windows of 1000 samples to check if all the voltage samples in that window are within the voltage range of the noise.

- For all such windows, change any 1s in the slope_threshold list within the indices of the window to 0, since they are not actually edges.



(a) All edges detected



(b) Zoomed view of edges detected

Figure 7.3: Edges Detected (includes outliers)

These steps initially took a lot of computation time (more than 2 minutes), which is why the slope_threshold list and the voltage list were changed to numpy arrays, which improved the computation time to less than 30 seconds. As seen in figure 7.4, the above steps remove the outliers, and the edges detected are within the main data.

Now that the edges are correctly detected, the indices of the edges are added to another list called slope_index which will be useful later on in the algorithm. The updated slope_threshold numpy array will have clusters of 1 which will correspond to one edge. These clusters, as shown in figure 7.5a need to be replaced with 1 at the middle index of the cluster, and the rest of the 1s in the cluster need to be replaced with 0. This is achieved using the following algorithm:

- Traverse the slope_index list and find the difference between consecutive values.

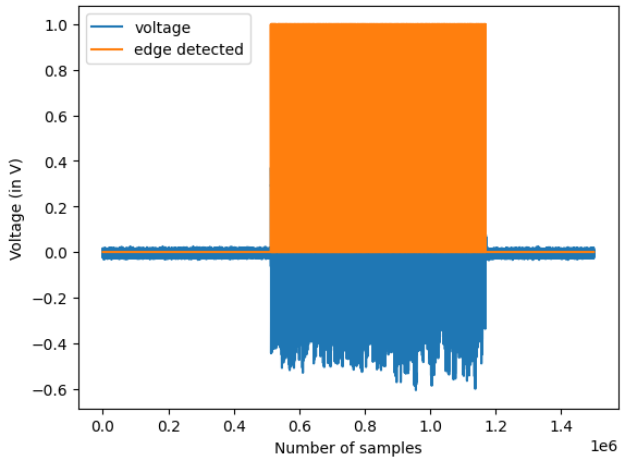


Figure 7.4: Edges after the outliers are removed.

- Check if the difference is more than 50 (this signifies the end of one cluster). The middle element of the cluster has to be 1 and the rest of the elements need to be set to 0.

$First\_index \leftarrow slope\_index[0]$

$Last\_index \leftarrow 0$

$i \leftarrow 0$

while $i \leq length\ of\ slope\_index–2$ **do**

$slope\_of\_index \leftarrow slope\_index[i+1] - slope\_index[i]$

if $slope\_of\_index > 50$ **then**

$last\_index \leftarrow slope\_index\_one[i]$

$mid\_index \leftarrow round((last\_index + first\_index)/2)$

$slope\_threshold[mid\_index] \leftarrow 1$

$j \leftarrow first\_index$

while $j \leq last\_index$ **do**

if $j \neq mid\_index$ **then**

$slope\_threshold[j] \leftarrow 0$

end if

$j \leftarrow j + 1$

end while

$first\_index \leftarrow slope\_index[i + 1]$

end if

if $i = len(slope\_index) - 2$ **then**

$last\_index \leftarrow slope\_index\_one[i + 1]$

$mid\_index \leftarrow round((last\_index + first\_index)/2)$

$slope\_threshold[mid\_index] \leftarrow 1$

$j \leftarrow first\_index$

while $j \leq last\_index$ **do**

if $j \neq mid\_index$ **then**

$slope\_threshold[j] \leftarrow 0$

end if

$j \leftarrow j + 1$

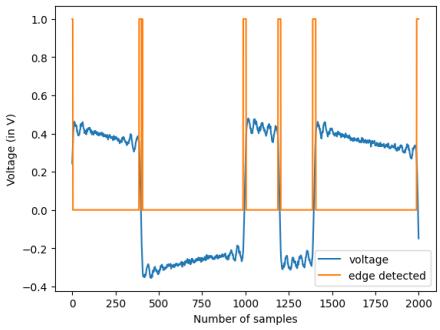
end while

end if

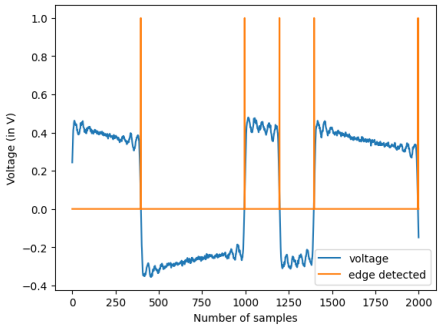
$i \leftarrow i + 1$

end while

After the aforementioned steps and updating the slope_threshold numpy array, the indices of 1s are updated in the slope_index list. The plot of edges and voltages are shown in figure 7.5b and it can be seen that the edges are correctly identified.



(a) Edges before clusters are resolved



(b) Edges after clusters are resolved

Figure 7.5: Edges before and after resolving clusters

7.3 Recovering Clock From Start Bits

In UART communication protocol, the start bits are used to train and extract the clock of the transmitter. The same principle will be used for extracting the DAC clock from the start bits. The start bits are modeled in such a way that there will be alternating 20 high and low bits for 40 cycles (20 high and 20 low). This means that there will be 41 edges from the start till the end of the start bits. These edges are extracted from the slop_threshold numpy array and stored in a separate list called clock. The indices of these edges are also stored in a separate list called clock_index. Since we know that there are 20 bits in between the edges, we need to find the rising edges of the clock in between these already-defined edges.

$i \leftarrow 0$

while $i \leq length\ of\ clock\_index - 2$ **do**

$gap \leftarrow clock\_index[i-1] - clock\_index[i]$

$interval \leftarrow round(gap/20)$

$j \leftarrow 1$

while $j \leq 19$ **do**

$Clock[clock\_index[i] + interval * j] \leftarrow 1$

$j \leftarrow j + 1$

end while

$i \leftarrow i + 1$

end while

The above steps will generate the positive clock edges. The average gap between the indices of the updated edges is calculated, which is actually the period after which the voltage needs to be sampled.

7.4 Sampling Data

Now that the positive edges of the clock have been generated and the interval between two consecutive edges has been calculated, the start and end of the main data need to be identified. The index where the main data starts is the 42^{nd} element of the slope_index list and the index at the end of the main data is the last element of the slope_index list. The positive edges of the clock needs to be overlayed on the main data, so that the bits can be recovered. The data is sampled at the middle of 2 positive clock edges since that is where the value is most stable. The following steps are used:

$clock\_index\_main\_data \leftarrow empty\ list$

$recovered\_bits \leftarrow empty\ list$

$i \leftarrow first\_index\_of\_main\ data$

while $i \leq last\_index\_of\_main\_data - average\_gap$ **do**

$clock\_index\_main\_data[] \leftarrow i$

$i \leftarrow i + average\_gap$

end while

$i \leftarrow 0$

while $i \leq length\ of\ clock\_index\_main\_data - 1$ **do**

$data \leftarrow voltage[clock\_index\_main\_data[i] + round(average\_gap/2)]$

if $data > 0$ **then**

$recovered\_bits[] \leftarrow 1$

else

$recovered\_bits[] \leftarrow 0$

end if

end while

The list recovered_bits now has the bits sampled after performing clock data recovery

7.5 Error Rate Testing

The bits that have been recovered need to be verified against the bits that were originally sent by the DAC to check for the correctness of the developed algorithm. The original .flt file that the DAC used to generate the bits is opened in the Jupyter notebook. The values are then put into a list. These values are then checked against the recovered bits to check for errors and the percentage error is calculated.

Chapter 8

Results

This chapter is divided into multiple sections discussing the results at different stages of the development of the platform.

8.1 Periodic Signals

The periodic signals tested for the DAC (sampling rate = 250 MHz) are sine wave, square wave, and triangle wave. The signals were analyzed using the RTM3004 oscilloscope. Three different frequencies of the periodic waves were tested – 1MHz, 10 MHz, and 25 MHz. Figures 8.1, 8.2, 8.3 show the 1 MHz, figures 8.4, 8.5, 8.6 show the signals 10 MHz, and figures 8.7, 8.8, 8.9 show the signals 25 MHz.

As seen in figures 8.1, 8.2, and 8.3, the time period and frequency measured by the oscilloscope as expected (1 microsecond and 1MHz respectively). The shape of the square wave and triangle wave, however are not accurate, and the voltage is not stable and that is why the shape of the signals are morphed. A higher frequency is used to check the performance of the DAC.

As seen in figures 8.4 to 8.9, the performance of the DAC for the square and triangle waves is much better. This is because the DC voltage has to stay stable for much lesser time for higher frequency square waves and triangle waves than for lower frequency waves. It is easier

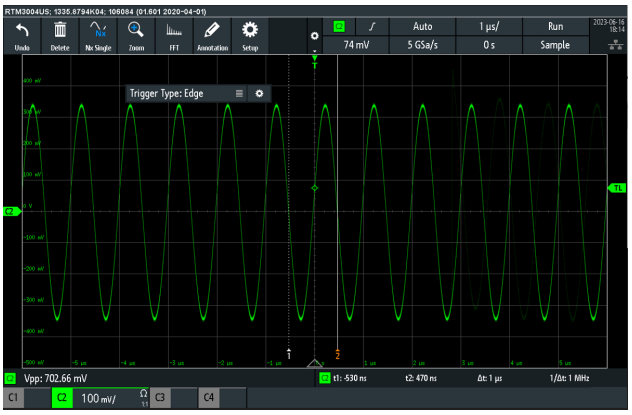


Figure 8.1: Sine wave of frequency 1MHz

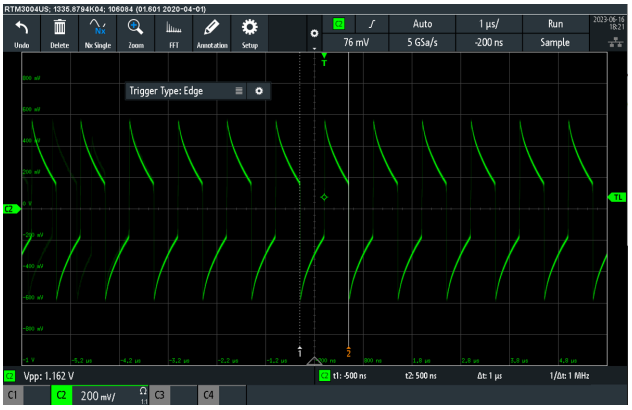


Figure 8.2: Square wave of frequency 1MHz

for the output capacitors to maintain the voltage for a short period of time.

8.2 Random Bit Stream

The random bits were generated using a C code on the host machine instead of generating them on the FPGA to save computation time. The bits generated by the C code was stored in a .bin file. The .bin file was uploaded to the FPGA along with the C code to control the DAC. The file was read and the values were put into the buffer and pushed to hardware. The signal captured by the oscilloscope is as shown in figure 8.10.

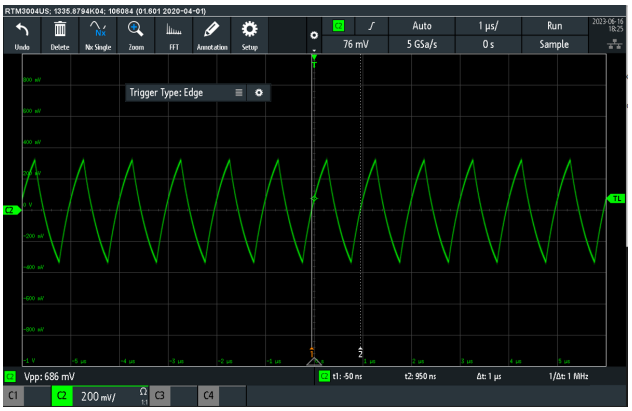


Figure 8.3: Triangle wave of frequency 1MHz

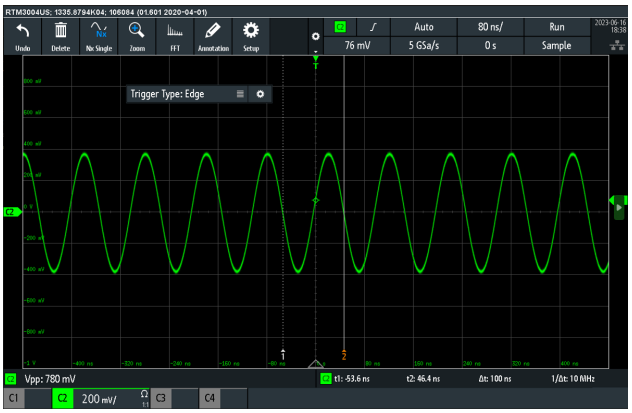


Figure 8.4: Sine wave of frequency 10MHz

The captured data was further analyzed using the CDR algorithm. The acquisition data of the oscilloscope is stored in a .flt file. This file is read in a Jupyter notebook and processed using the algorithm discussed in the previous chapter. The recovered bits were compared against the bits in the .bin file that was used to generate the bits. Different bits per symbol resulting in different baud rates were tested and the accuracies in the different cases were tabulated and plotted. Tests were done for 2, 5, 10, 15 and 20 bits per baud or symbol. Equation 8.1 [16] was used to translate the bits per baud or symbol to baud rate. The error rates and baud rate are tabulated in Table 8.1 and plotted in figure 8.11 .

$$baudrate = bitrate/bitsperbaud \tag{8.1}$$

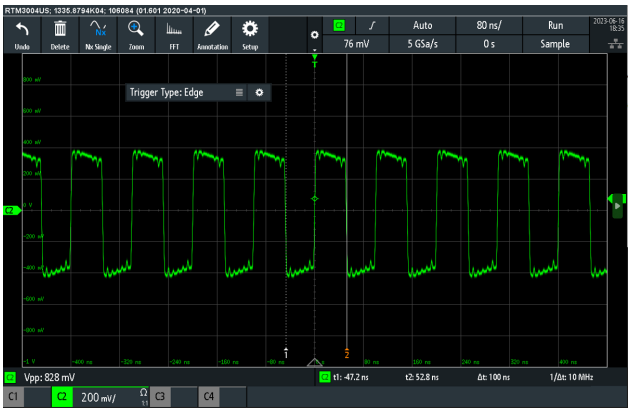


Figure 8.5: Square wave of frequency 10MHz

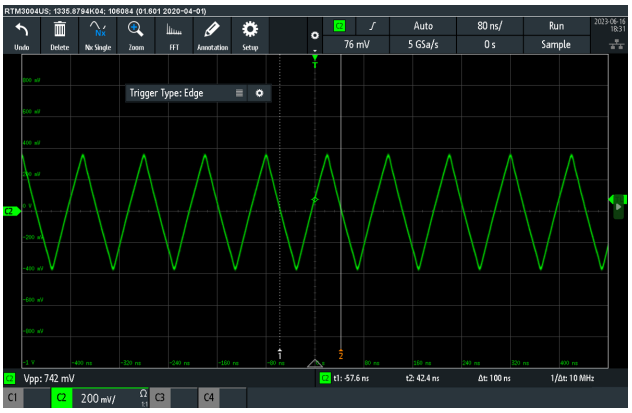


Figure 8.6: Triangle wave of frequency 10MHz

In equation 8.1, bit rate is the number of bits produced per second. In this case, the sampling rate is 250 MHz, which means 250,000,000 bits are produced per second and hence, bit rate is also 250 MHz. Bits per baud is the number of bits that represent one baud or symbol.

Bits per symbol	Baud rate(x 10^6)	Error (%)
20	12.5	0.09
15	16.67	0.1
10	25	0.11
5	50	0.43
2	125	0.62

Table 8.1: Error Rate vs Baud Rate

As seen from figure 8.11, the error rate of recovering the bits increases with increase in baud

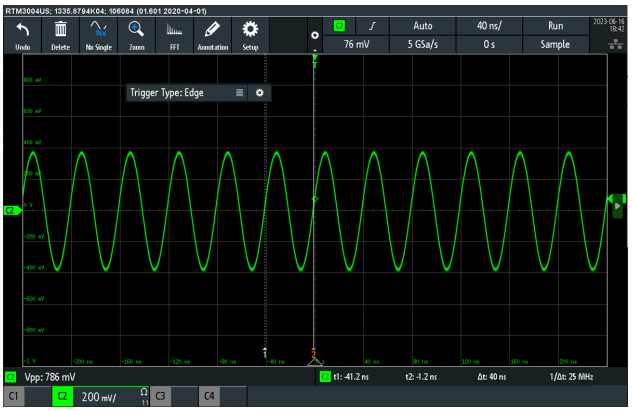


Figure 8.7: Sine wave of frequency 25MHz

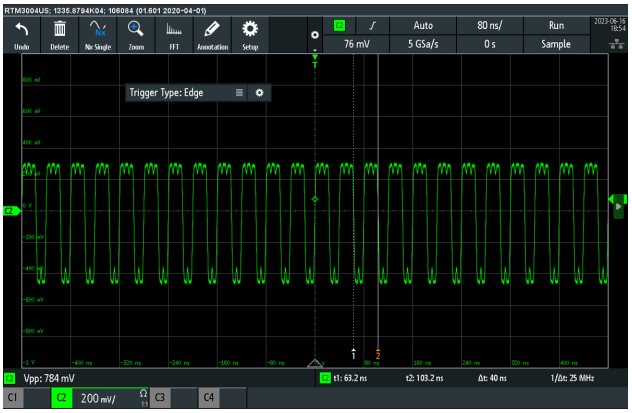


Figure 8.8: Square wave of frequency 25MHz

rate, that is the error rate increases with decrease in bits per symbol. By increasing the bits per symbol, more bits make up a symbol and there are less transitions from high to low and vice versa. The edges become cleaner and easier to detect. Hence, the CDR algorithm is able to detect the edges more accuartely, which translates to a lower error rate.

Error rates for different voltage swings were also analyzed. The voltage swing can be increased or decreased by changing the gain value from IIO-Oscilloscope. The gain ranges from 0 to 1. On increasing the gain, the peak-to-peak voltage increases, which is better for edge detection since higher voltage swings give higher slope values. However, it must be kept in mind that by increasing the gain beyond a certain limit, the sample value stored in

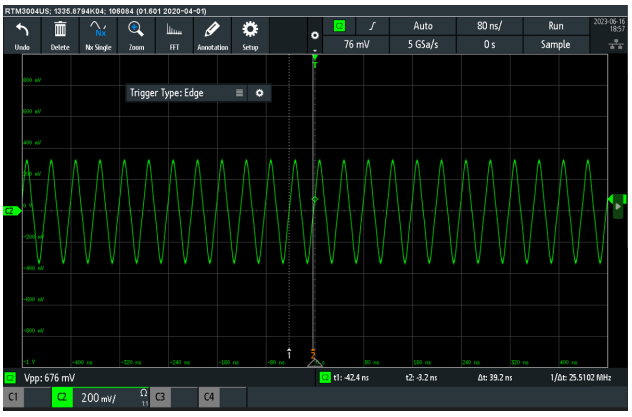


Figure 8.9: Triangle wave of frequency 25MHz

Figure 8.10: Bitstream captured by oscilloscope

the buffer might overflow which results in inverted bits. The buffer stores the sample values as 16-bit signed integers. Therefore, the value cannot exceed +32,767 and -32,767. If by increasing the gain, the value exceeds the aforementioned values, there will be overflow and the peak-to-peak voltage will decrease and the bits will be inverted. Table 8.2 shows the readings for error rate with change in peak to peak voltage and is plotted in figure 8.12.

As seen from figure 8.12, the error rate goes down as the peak-to-peak voltage increases. This is because, with higher voltage swings, the slope between consecutive samples will be higher and the algorithm to detect the edges will be less prone to error. Decreasing the peak-to-peak voltage further will induce much more error in the edge detection algorithm.

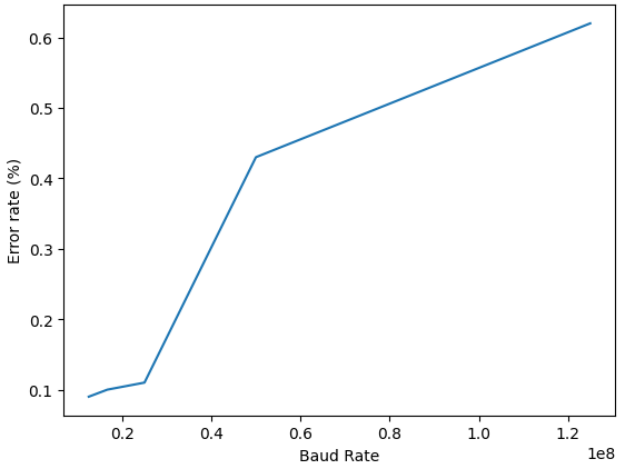


Figure 8.11: Plot of Error rate vs Baud rate

Peak-to-peak Voltage (in V)	Error (%)
0.94	3.3
0.98	2.9
1.02	1.2
1.06	0.8
1.10	0.57
1.14	0.23
1.18	0.14

Table 8.2: Error Rate vs Peak-to-peak Voltage

With higher voltage swings, a higher threshold value to determine edges can be set. If this value is low, sometimes fluctuations in noise can also be detected as edges.

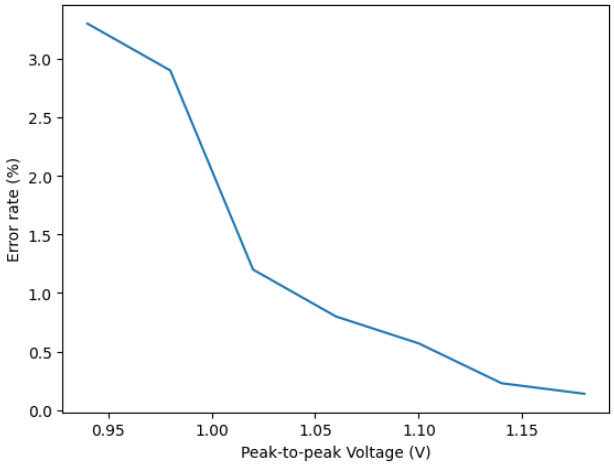


Figure 8.12: Plot of Error rate vs Peak-to-peak Voltage

Chapter 9

Conclusion

We demonstrated the implementation of DAC signal generation using SDR. By leveraging the capabilities of DACs and the flexibility of SDR, we have achieved the generation of high-quality signals which can be used in different applications in a plethora of domains ranging from communication systems to quantum computers for quantum measurements and for generating precisely timed control signals. Through the use of CDR, the data from the signals were recovered with a low error rate establishing the integrity of the transmitted signal.

The integration of DAC signal generation in SDR has showcased the potential for enhanced signal flexibility and adaptability in modern communication systems. The ability to generate modulated signals on-the-fly has allowed for rapid deployment and reconfiguration, making it suitable for dynamic and evolving communication scenarios. Furthermore, the combination of DAC signal generation with SDR has provided a cost-effective and scalable solution, eliminating the need for dedicated hardware for each communication standard.

One possible improvement to the system can be the design and usage of dedicated memory modules in the FPGA, which will make the generation of signals more efficient. The problem with the current method of using C code to generate and transmit signals using libiio, is that the speed of generating and populating the buffer is slower than the push to hardware. There is an overlap in the first half portion of the buffer. This is the reason why the buffer was filled with data to half capacity, while the first half of the buffer is filled with zero,

such that there is no loss of data. The possible future architecture can include a memory module that is controlled by a memory controller module. The memory module can store all the data samples for the signal that will send samples to the IIO port through a FIFO of sufficient depth. This should be a more seamless approach than the use of C code and libiio.

This project's findings and implementation highlight the significance of DAC signal generation in advancing communication systems. The utilization of SDR and DACs opens up possibilities for future research and development in the field, ranging from improved wireless communication protocols to advanced signal processing techniques. As technology continues to evolve, the integration of DAC signal generation with SDR will play a vital role in shaping the future of communication systems, enabling efficient, adaptable, and high-quality connectivity for various applications and domains.

Bibliography

[1] https://www.analog.com/media/en/technical-documentation/data-sheets/ad7524.pdf.

[2] https://ww1.microchip.com/downloads/aemDocuments/documents/OTH/ProductDocuments/DataSheets/20002249B.pdf.

[3] https://www.ti.com/lit/ds/symlink/pcm1795.pdf?ts=1685200614560&ref_.

[4] https://www.analog.com/media/en/technical-documentation/user-guides/ad9081-ad9082-ug-1578.pdf.

[5] https://www.analog.com/media/en/technical-documentation/user-guides/eval-ad5791sdz-ug-1152.pdf.

[6] https://www.analog.com/media/en/technical-documentation/data-sheets/ad7524.pdf.

[7] https://github.com/analogdevicesinc/linux.git.

[8] https://wiki.analog.com/resources/tools-software/linux-software/ad9081_plugin.

[9] https://wiki.analog.com/resources/tools-software/linux-software/libiio.

[10] https://github.com/analogdevicesinc/libiio.

[11] https://analogdevicesinc.github.io/libiio/v0.4/group__Context.html.

[12] https://analogdevicesinc.github.io/libiio/v0.8/group__Device.html.

[13] https://analogdevicesinc.github.io/libiio/v0.8/group__Channel.html.

[14] http://analogdevicesinc.github.io/libiio/master/libiio/group__Buffer.html.

[15] https://wiki.analog.com/resources/fpga/peripherals/jesd204.

[16] https://www.geeksforgeeks.org/difference-btween-bit-rate-and-baud-rate/.

[17] Eric D Adler et al. "Direct digital synthesis applications for radar development". In: *Proceedings International Radar Conference*. IEEE. 1995, pp. 224–226.

[18] Travis Bartley et al. "Delay window blind oversampling clock and data recovery algorithm with wide tracking range". In: *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2015, pp. 1598–1601.

[19] Jianzhong Chen and Yong Ping Xu. "A novel noise-shaping DAC for multi-bit sigma-delta modulator". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 53.5 (2006), pp. 344–348.

[20] Yongjie Cheng et al. "Multibit delta-sigma modulator with two-step quantization and segmented DAC". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 53.9 (2006), pp. 848–852.

[21] Lionel Cordesses. "Direct digital synthesis: A tool for periodic wave generation (part 1)". In: *IEEE Signal processing magazine* 21.4 (2004), pp. 50–54.

[22] Lionel Cordesses. "Direct digital synthesis: a tool for periodic wave generation (part 2)". In: *IEEE Signal Processing Magazine* 21.5 (2004), pp. 110–112.

[23] Pedro Cruz, Nuno Borges Carvalho, and Kate A Remley. "Designing and testing software-defined radios". In: *IEEE Microwave magazine* 11.4 (2010), pp. 83–94.

[24] Luis Duarte et al. "A Software-Defined Radio for Future wireless communication systems at 60 GHz". In: *Electronics* 8.12 (2019), p. 1490.

[25] Luis Duarte et al. “A Software-Defined Radio for Future wireless communication systems at 60 GHz”. In: *Electronics* 8.12 (2019), p. 1490.

[26] Mohammad Elmi, Martin Lee, and Kambiz Moez. “An Ultra-Low-Power Non-Uniform Derivative-Based Sampling Scheme With Tunable Accuracy”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* (2023).

[27] AS Gowda. *Photonic Software Defined Radio*. Tech. rep. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2019.

[28] Jiajie Huang et al. “Low-Noise, High-Linearity Sine-Wave Generation Using Noise-Shaping Phase-Switching Technique”. In: *IEEE Transactions on Instrumentation and Measurement* 71 (2021), pp. 1–7.

[29] Walt Kester. “Evaluating high speed dac performance”. In: *Analog Device, MT-013, January* (2006).

[30] Walt Kester. “Oversampling interpolating dacs”. In: *MT-017, tutorial documents, Analog Devices, Inc. available at http://www. analog. com/static/imported-files/tutorials/MT-017. pdf* (2009).

[31] Jaeha Kim and Deog-Kyoon Jeong. “Multi-gigabit-rate clock and data recovery based on blind oversampling”. In: *IEEE Communications Magazine* 41.12 (2003), pp. 68–74.

[32] Martin Kinyua. *Method and circuit for continuous-time delta-sigma DAC with reduced noise*. US Patent 8,325,074. Dec. 2012.

[33] Bosco Leung. *Multi-bit oversampled DAC with dynamic element matching*. US Patent 5,406,283. Apr. 1995.

[34] Xing Li and Lei Zhou. “A survey of high-speed high-resolution current steering DACs”. In: *Journal of Semiconductors* 41.11 (2020), p. 111404.

[35] Andrew CY Lin et al. “A CMOS oversampled DAC with multi-bit semi-digital filtering and boosted subcarrier SNR for ADSL central office modems”. In: *IEEE journal of solid-state circuits* 41.4 (2006), pp. 868–875.

[36] Raquel G Machado and Alexander M Wyglinski. “Software-defined radio: Bridging the analog–digital divide”. In: *Proceedings of the IEEE* 103.3 (2015), pp. 409–423.

[37] Patrick Matalla et al. “Hardware comparison of feed-forward clock recovery algorithms for optical communications”. In: *2021 Optical Fiber Communications Conference and Exhibition (OFC)*. IEEE. 2021, pp. 1–3.

[38] AL McEwan and S Collins. “Efficient, ROM-less DDFS using non-linear interpolation and non-linear DAC”. In: *Analog Integrated Circuits and Signal Processing* 48 (2006), pp. 231–237.

[39] AA Petrov, VV Davydov, and NM Grebennikova. “On the potential application of direct digital synthesis in the development of frequency synthesizers for quantum frequency standards”. In: *Journal of Communications Technology and Electronics* 63 (2018), pp. 1281–1285.

[40] LF Rahman et al. “The evolution of digital to analog converter”. In: *2016 International Conference on Advances in Electrical, Electronic and Systems Engineering (ICAEES)*. IEEE. 2016, pp. 151–154.

[41] M Rau et al. “Clock/data recovery PLL using half-frequency clock”. In: *IEEE Journal of Solid-State Circuits* 32.7 (1997), pp. 1156–1159.

[42] Nicolas Regimbal. “Study of fractional frequency synthesizers for high data rate applications”. PhD thesis. Bordeaux 1, 2011.

[43] Victor S Reinhardt. “Direct digital synthesizers”. In: *Proceedings of the 17th Annual Precise Time and Time Interval Systems and Applications Meeting*. 1985, pp. 345–374.

[44] Victor S Reinhardt. "Spur reduction techniques in direct digital synthesizers". In: *1993 IEEE International Frequency Control Symposium*. IEEE. 1993, pp. 230–241.

[45] Diogo C Ribeiro et al. "Automatic characterization of RF DACs for software defined radio applications". In: *2014 IEEE MTT-S International Microwave Symposium (IMS2014)*. IEEE. 2014, pp. 1–4.

[46] Bryan Scott Rowan, Louis Joseph Serrano, and Mantle Man-Hon Yu. *Variable pulse PWM DAC method and apparatus*. US Patent 6,362,766. Mar. 2002.

[47] M Reza Sadeghifar and J Jacob Wikner. "A higher Nyquist-range DAC employing sinusoidal interpolation". In: *NORCHIP 2010*. IEEE. 2010, pp. 1–4.

[48] Mathew NO Sadiku and Cajetan M Akujuobi. "Software-defined radio: a brief overview". In: *Ieee Potentials* 23.4 (2004), pp. 14–15.

[49] Oliver Sander et al. "Software-defined radio readout system for the ECHo experiment". In: *IEEE Transactions on Nuclear Science* 66.7 (2019), pp. 1204–1209.

[50] MB Sandler. "Digital-to-analogue conversion using pulse width modulation". In: *Electronics & Communication Engineering Journal* 5.6 (1993), pp. 339–348.

[51] Richard Schreier, Gabor C Temes, et al. *Understanding delta-sigma data converters*. Vol. 74. IEEE press Piscataway, NJ, 2005.

[52] Mehdi Shafiee et al. "A readout system for microwave kinetic inductance detectors using software defined radios". In: *Journal of Instrumentation* 16.07 (2021), P07015.

[53] RW Stewart and E Pfann. "Oversampling and sigma-delta strategies for data conversion". In: *Electronics & Communication Engineering Journal* 10.1 (1998), pp. 37–47.

[54] Jouko Vankka. "Methods of mapping from phase to sine amplitude in direct digital synthesis". In: *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control* 44.2 (1997), pp. 526–534.

[55] Paul S Wright and John R Pickering. "An AC voltage standard based on a PWM DAC". In: *IEEE transactions on instrumentation and measurement* 48.2 (1999), pp. 457–461.

[56] Ching-Yuan Yang, Yu Lee, and Cheng-Hsing Lee. "A CMOS clock and data recovery circuit with a half-rate three-state phase detector". In: *IEICE transactions on electronics* 89.6 (2006), pp. 746–752.

[57] Zhuan Ye, John Grosspietsch, and Gokhan Memik. "An FPGA based all-digital transmitter with radio frequency output for software defined radio". In: *2007 Design, Automation & Test in Europe Conference & Exhibition*. IEEE. 2007, pp. 1–6.