University of Sarajevo
Faculty of Electrical Engineering
Department of Automation and Electronics

Master's Thesis Documentation

Master's Thesis

- Second Cycle of Studies -

Author:
Tarik Hamedović

Supervisor:
Doc. dr. sc. Nedim Osmić, dipl. ing. el.

Sarajevo,
March 2024

Contents

List of Figures

List of Tables

Chapter 1

Introduction

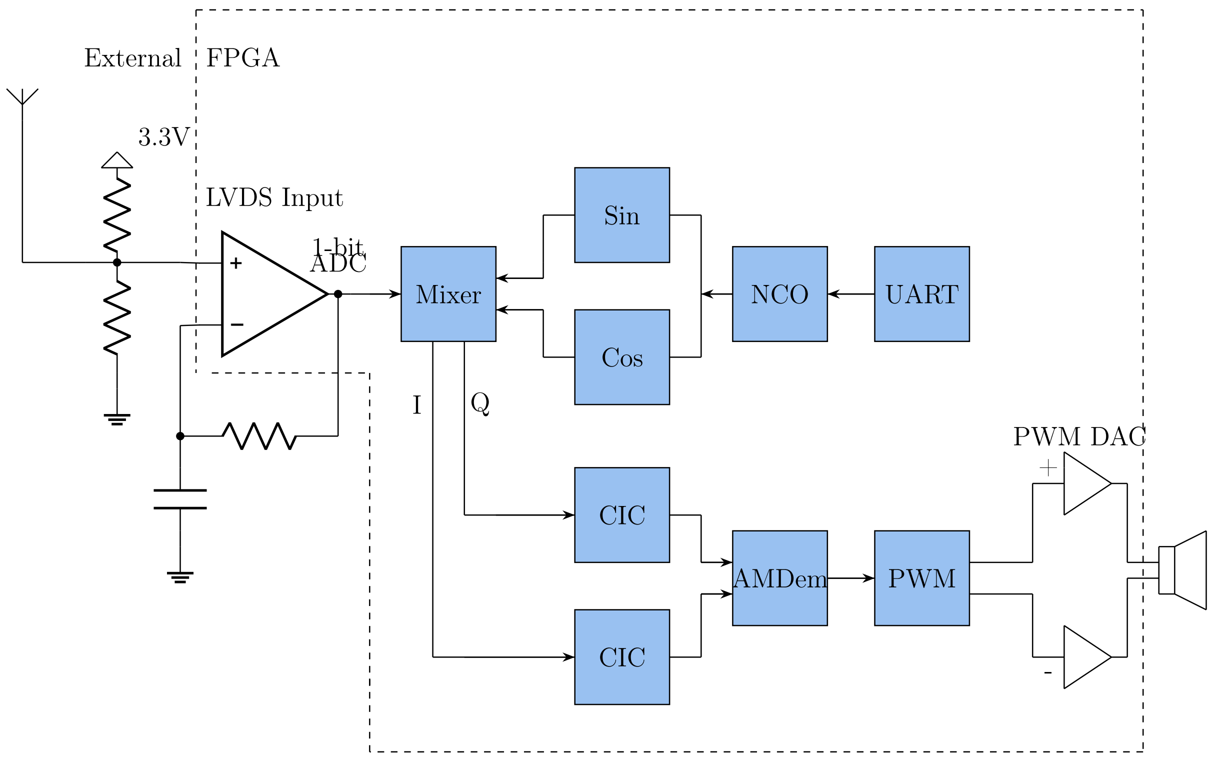
1.1 Overview

The Master's Thesis delves into the development of a Software-Defined Radio (SDR) Receiver, which is conceptualized to operate on an Field Programmable Gate Array(FPGA) platform utilizing High-Level Synthesis (HLS). This approach is inspired by a notable project conducted on a Lattice MACHXO2 Board, which demonstrated the feasibility of receiving AM broadcasts with minimal analog components. The project underscores the shift towards digital processing within the FPGA, highlighting the importance of a robust understanding of Digital Signal Processing (DSP) for successful implementation. HLS comes into play as an advantageous methodology for DSP applications, especially in algorithm-based designs, due to its efficiency in prototyping and testing compared to traditional Verilog-based development.

In the forthcoming chapters and sections, we will go into the operational principles of the project, alongside potential enhancements, alterations, and comprehensive discussions on the DSP Modules integral to the SDR Receiver. Additionally, the project's transition to a different platform, specifically the ULX3S board, will be outlined. Furthermore, the performance and functionality of the SDR Receiver will be evaluated through both Verilog and High-Level Synthesis (HLS) testing methodologies.

1.2 1 Bit SDR Project

The block diagram of the aforementioned project is given in Figure ??.



The block diagram for the Software Defined Radio (SDR) Receiver commences with the antenna, serving as the principal input source. To enhance signal detection and acquisition, this configuration adopts an advanced sampling strategy where random RF noise, superimposed on the desired signal, plays a critical role. Notably, the input signal undergoes oversampling, a technique that significantly improves the resolution and quality of the captured signal. This oversampling is synchronized with the FPGA clock signal, which in this scenario, is set at 80 MHz, ensuring a highly precise and efficient signal processing pathway.

The input RF signal from the antenna is succeeded by a voltage divider that feeds into the positive input of a comparator operational amplifier. The comparator's negative input is connected to an integrator, forming a key part of the circuit. The comparator itself is implemented digitally through Low Voltage Differential Signaling (LVDS), and together with the integrator, it constitutes the Sigma-Delta Modulator. This means that the only analog components of the SDR Reciever are the antenna, along with 3 resistors and a capacitor.

The digitized 1-bit data stream from the previous stage is then directed into the Mixer, where it is combined with quadrature signals—specifically, sine and cosine waves. These waves are synthesized by a Numerically Controlled Oscilator (NCO). The NCO excels at producing precise digital representations of sine and cosine waves at specified frequencies, leveraging a phase increment value for this purpose. Control over the NCO's output frequency is facilitated through a UART connection between the board and a PC. This interface allows for dynamic adjustment of the sine/cosine wave frequencies to match the frequency of the target signal—the carrier wave—intended to be received from the antenna. The NCO serves as the digital counterpart to an analog circuit known as the Voltage Controlled Oscillator (VCO).

The Mixer's output consists of two components: the in-phase (I) and quadrature (Q) signals. These signals undergo Binary Phase Shift Keying (BPSK) modulation within the Mixer, a method chosen for its efficiency in conveying data through phase shifts.

Following modulation, the I and Q signals enter the Cascaded Integrator-Comb (CIC) filter. This filter acts as a sophisticated low-pass filter, merging the effects of a moving average filter with a process known as decimation. Decimation effectively reduces the data rate by selecting every 4096th sample, thus achieving a decimation factor of $1/4096$.

After filtering, the signal undergoes Amplitude Modulation (AM) demodulation. The demodulated signal is then directed into a Pulse Width Modulation (PWM) block. Here, the PWM signal is crafted based on the DC value present at its input transforming the demodulated signal into a form suitable for driving audio devices.

Finally, the PWM output signal is amplified to ensure adequate power for audio reproduction. This amplified signal is then conveyed to the speakers, where it is converted into audible sound.

This comprehensive pathway from signal reception to sound output showcases the intricate processes involved in modern digital signal processing within a Software Defined Radio (SDR) receiver, demonstrating a seamless blend of analog and digital techniques to achieve high-quality audio reproduction.

Each of the described blocks constitutes Digital Signal Processing (DSP) Modules, which are thoroughly detailed in Chapter 2.

1.3 ULX3S FPGA Board

The board used for this thesis is the ULX3S FPGA Board. It is a fully open-source development board for LATTICE ECP5 FPGAs. The board has extensive documentation on its GitHub, with manuals, working examples and different projects.

The main github page for ULX3S serves as the central hub for all resources and information related to the ULX3S ecosystem.

The ULX3S board is available in various models, differentiated primarily by their Look-Up Table (LUT) sizes, including 12F, 25F, 48F, and 85F variants, where the numbers indicate the LUT capacity, such as 85K LUTs, which will be used here. These boards can be synthesized, placed, and routed using either Lattice Diamond or open-source tools. However, for programming, the boards require open-source tools like fujprog or openFPGALoader. The design and layout of the board, along with its peripherals, are showcased in the illustrations referred to as Figures 1.1 and 1.2 for the front and back sides of the board, respectively.

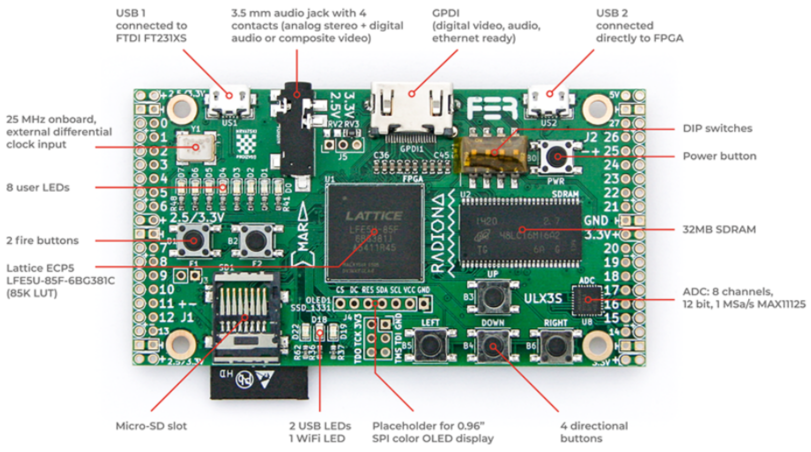

USB 1 connected to FTDI FT231XS
3.5 mm audio jack with 4 contacts (analog stereo + digital audio or composite video)
GPDI (digital video, audio, ethernet ready)
USB 2 connected directly to FPGA
25 MHz onboard, external differential clock input
8 user LEDs
2 fire buttons
Lattice ECP5 LFE5U-85F-6BG381C (85K LUT)
DIP switches
Power button
32MB SDRAM
ADC: 8 channels, 12 bit, 1 MSa/s MAX11125
Micro-SD slot
2 USB LEDs 1 WiFi LED
Placeholder for 0.96" SPI color OLED display
4 directional buttons

Figure 1.1: Front side of ULX3S FPGA Board

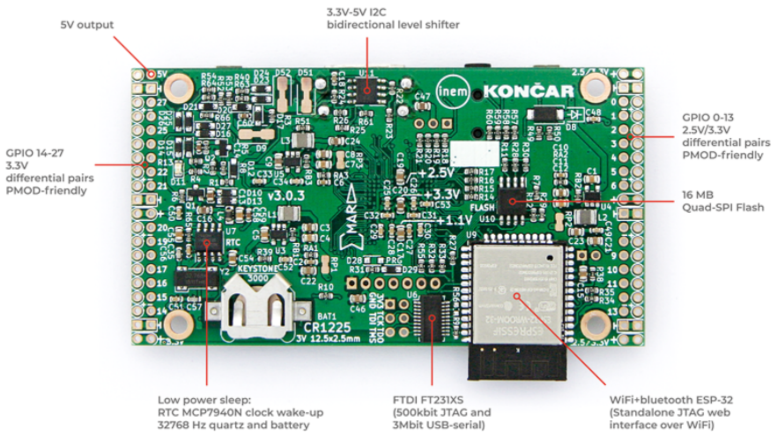

5V output
3.3V-5V I2C bidirectional level shifter
GPIO 14-27 3.3V differential pairs PMOD-friendly
GPIO 0-13 2.5V/3.3V differential pairs PMOD-friendly
16 MB Quad-SPI Flash
Low power sleep: RTC MCP7940N clock wake-up 32768 Hz quartz and battery
FTDI FT231XS (500kbit JTAG and 3Mbit USB-serial)
WiFi+bluetooth ESP-32 (Standalone JTAG web interface over WiFi)

Figure 1.2: Back side of ULX3S FPGA Board

1.3.1 Constraint File

The 1-bit SDR project utilizes a Lattice FPGA, although it employs a different board and FPGA chip compared to other setups. To ensure compatibility and proper operation, it's necessary to adjust the Constraint file to align with the system's peripherals. For this purpose, the ULX3S Constraint File, along with its associated schematic plays a crucial role in this customization process.

When examining the schematic, it's crucial to note that all signals within each bank must share the same VCCIO voltage level. Mixing voltage levels across signals in the same bank is not permissible. Most of the pins can be LVCMOS33, just one pin needs to be defined as LVDS25.

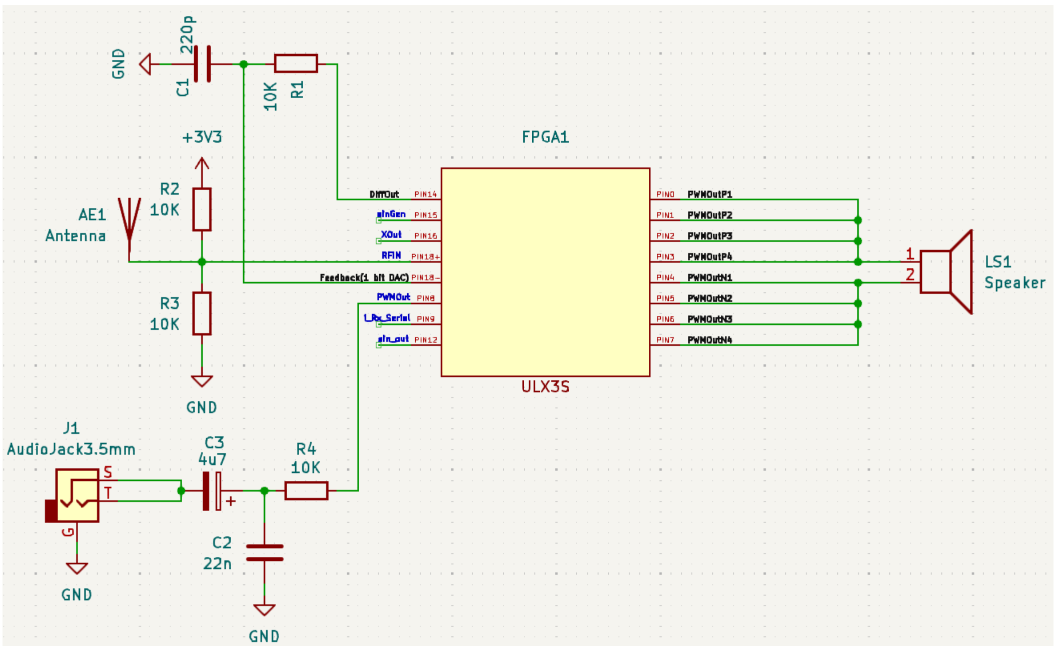


Figure 1.3: ULX3S Schematic for 1-bit SDR

Figure 1.3 illustrates the schematic of a 1-bit SDR system. This schematic focuses exclusively on the external components required for the system's operation, along with the input and output signals and their corresponding pin numbers within the ULX3S FPGA Banks. The working principle of the SDR is going to be explained in detail in later sections.

```
BLOCK RESETPATHS;
BLOCK ASYNCPATHS;
# BLOCK JTAGPATHS; # Added from 1bitSDR Github
## ULX3S v3.1.6 and v3.1.7

# The clock "usb" and "gpdi" sheet
LOCATE COMP "clk_25mhz" SITE "G2";
IOBUF  PORT "clk_25mhz" PULLMODE=NONE IO_TYPE=LVCMOS33;
FREQUENCY PORT "clk_25mhz" 25 MHZ; # FREQUENCY NET?
#CLOCK_TO_OUT ALLPORTS 20.000000 ns CLKNET "clk_25mhz" ;

# JTAG and SPI FLASH voltage 3.3V and options to boot from SPI flash
# write to FLASH possible any time from JTAG:
SYSCONFIG CONFIG_IOVOLTAGE=3.3 COMPRESS_CONFIG=ON MCCLK_FREQ=62
    SLAVE_SPI_PORT=DISABLE MASTER_SPI_PORT=ENABLE SLAVE_PARALLEL_PORT=
    DISABLE;
# write to FLASH possible from user bitstream:
# SYSCONFIG CONFIG_IOVOLTAGE=3.3 COMPRESS_CONFIG=ON MCCLK_FREQ=62
    SLAVE_SPI_PORT=DISABLE MASTER_SPI_PORT=DISABLE SLAVE_PARALLEL_PORT
    =DISABLE;

## LED indicators "blinkey" and "gpio" sheet
LOCATE COMP "led[7]" SITE "H3";
LOCATE COMP "led[6]" SITE "E1";
LOCATE COMP "led[5]" SITE "E2";
LOCATE COMP "led[4]" SITE "D1";
LOCATE COMP "led[3]" SITE "D2";
LOCATE COMP "led[2]" SITE "C1";
LOCATE COMP "led[1]" SITE "C2";
LOCATE COMP "led[0]" SITE "B2";
IOBUF  PORT "led[0]" PULLMODE=NONE IO_TYPE=LVCMOS33 DRIVE = 4;
IOBUF  PORT "led[1]" PULLMODE=NONE IO_TYPE=LVCMOS33 DRIVE = 4;
IOBUF  PORT "led[2]" PULLMODE=NONE IO_TYPE=LVCMOS33 DRIVE = 4;
IOBUF  PORT "led[3]" PULLMODE=NONE IO_TYPE=LVCMOS33 DRIVE = 4;
IOBUF  PORT "led[4]" PULLMODE=NONE IO_TYPE=LVCMOS33 DRIVE = 4;
IOBUF  PORT "led[5]" PULLMODE=NONE IO_TYPE=LVCMOS33 DRIVE = 4;
IOBUF  PORT "led[6]" PULLMODE=NONE IO_TYPE=LVCMOS33 DRIVE = 4;
IOBUF  PORT "led[7]" PULLMODE=NONE IO_TYPE=LVCMOS33 DRIVE = 4;


# GPIO (default single-ended) "gpio", "ram", "gpdi" sheet
# Pins enumerated gp[0-27], gn[0-27].
# With differential mode enabled on Lattice,
# gp[] (+) are used, gn[] (-) are ignored from design
# as they handle inverted signal by default.
# To enable differential, rename LVCMOS33->LVCMOS33D
# FEMALE ANGLED (90 deg PMOD) on TOP or
# MALE VERTICAL ( 0 deg pins) on BOTTOM and flat cable

######################## 3.3V   ########################
######################## BANK 0 ########################
```

```
# PIN B11 From Bank 0 [Pin 0 ON BOARD]
LOCATE COMP "PWMOutP1" SITE "B11";
IOBUF PORT  "PWMOutP1" PULLMODE=NONE IO_TYPE=LVCMOS33 DRIVE = 4;

# PIN A10 From Bank 0 [Pin 1 ON BOARD]
LOCATE COMP "PWMOutP2" SITE "A10";
IOBUF PORT  "PWMOutP2" PULLMODE=NONE IO_TYPE=LVCMOS33 DRIVE = 4;

# PIN A9 From Bank 0 [Pin 2 ON BOARD]
LOCATE COMP "PWMOutP3" SITE "A9";
IOBUF PORT  "PWMOutP3" PULLMODE=NONE IO_TYPE=LVCMOS33 DRIVE = 4;

# PIN B9 From Bank 0 [Pin 3 ON BOARD]
LOCATE COMP "PWMOutP4" SITE "B9";
IOBUF PORT  "PWMOutP4" PULLMODE=NONE IO_TYPE=LVCMOS33 DRIVE = 4;

# PIN A7 From Bank 0 [Pin 4 ON BOARD]
LOCATE COMP "PWMOutN1" SITE "A7";
IOBUF PORT  "PWMOutN1" PULLMODE=NONE IO_TYPE=LVCMOS33 DRIVE = 4;

# PIN C8 From Bank 0 [Pin 5 ON BOARD]
LOCATE COMP "PWMOutN2" SITE "C8";
IOBUF PORT  "PWMOutN2" PULLMODE=NONE IO_TYPE=LVCMOS33 DRIVE = 4;

# PIN C6 From Bank 0 [Pin 6 ON BOARD]
LOCATE COMP "PWMOutN3" SITE "C6";
IOBUF PORT  "PWMOutN3" PULLMODE=NONE IO_TYPE=LVCMOS33 DRIVE = 4;

# PIN A6 From Bank 0 [Pin 7 ON BOARD]
LOCATE COMP "PWMOutN4" SITE "A6";
IOBUF PORT  "PWMOutN4" PULLMODE=NONE IO_TYPE=LVCMOS33 DRIVE = 4;

######################## BANK 7 ########################
# PIN A4 From Bank 7 [Pin 8 ON BOARD]
LOCATE COMP "PWMOut" SITE "A4";
IOBUF PORT  "PWMOut" PULLMODE=NONE IO_TYPE=LVCMOS33 DRIVE = 4;

# Pin A2 From Bank 7 [PIN 9 ON BOARD]
LOCATE COMP "i_Rx_Serial"  SITE "A2";
IOBUF PORT  "i_Rx_Serial"  PULLMODE=NONE IO_TYPE=LVCMOS33; # [deleted
    DRIVE=4 and changed PULLMODE=UP]

# PIN C4 From Bank 7 [PIN 10 ON BOARD] (NOT USED)
#LOCATE COMP "o_Tx_serial"  SITE "C4"; # PCLK
#IOBUF PORT  "o_Tx_serial"  PULLMODE=UP IO_TYPE=LVCMOS33;

# PIN G3 From Bank 7 [Pin 12 on BOARD]
LOCATE COMP "sin_out"  SITE "G3";
IOBUF PORT  "sin_out"  IO_TYPE=LVCMOS33;

######################## 2.5V   ########################
```

```
######################## BANK 3 ########################
# PIN U18 From BANK 3 [PIN 14 ON BOARD]
LOCATE COMP "DiffOut"  SITE "U18";
IOBUF PORT  "DiffOut"  SLEWRATE=SLOW IO_TYPE=LVCMOS25;
OUTPUT PORT "DiffOut" LOAD 0.000000 pF;

# PIN N17 From Bank 3 [PIN 15 ON BOARD]
LOCATE COMP "sinGen"  SITE "N17";
IOBUF PORT  "sinGen"  PULLMODE=NONE IO_TYPE=LVCMOS25;

# PIN N16 From Bank 3 [Pin 16 on BOARD] (NOT USED)
LOCATE COMP "XOut"  SITE "N16";
IOBUF PORT "XOut" IO_TYPE=LVCMOS25 SLEWRATE=SLOW HYSTERESIS=NA ;

######################### LVDS  ########################
######################## BANK 2 ########################
# LVDS (Differential signal on +- Pins on Board)
# PIN H18 From Bank 2 [Positive Pin 18 ON BOARD]
LOCATE COMP "RFIn"  SITE "H18";
IOBUF PORT "RFIn" IO_TYPE=LVDS HYSTERESIS=NA CLAMP=ON DIFFRESISTOR=
   OFF PULLMODE=NONE OPENDRAIN=OFF; # LVDS25 NOT SUPPORTED?
# PIN H17 From Bank 2 [Negative Pin 18 on Board] SigmaDelta ADC
   Feedback(1 bit DAC)
LOCATE COMP "SDFeedback"  SITE "H17";
IOBUF PORT "SDFeedback" IO_TYPE=LVDS HYSTERESIS=NA CLAMP=ON
   DIFFRESISTOR=OFF PULLMODE=NONE OPENDRAIN=OFF;
```

Listing 1.1: Constraint File for ULX3S 1-bit SDR System

The listing above shows the modified constraint file of the 1-bit SDR System for the ULX3S FPGA Board. As can be seen the signals are divided into different banks depending on their volage level aswell as IOTYPE.

1.3.2 Testing

Chapter 2

Software-Defined Radio

2.1 Introduction

The concept of Software-Defined Radio (SDR) emerged from collaborative efforts across various research groups in both private and government sectors in the United States during the 1970s and 1980s, notably including the US Department of Defense Laboratory and a team from the Garland, Texas division of E-Systems Inc. It was in 1991 that Joe Mitola[5], working with E-Systems, coined the term "Software Radio" (SR) to describe his vision of a fully software-based GSM transceiver. This idea marked a significant leap in telecommunications technology, aiming for a more flexible and software-driven approach to radio communication[4].

A Software-Defined Radio (SDR) system is a sophisticated device designed to enable efficient data transmission and reception. At its core, an SDR system transforms information—whether speech, music, or video—into a digital format, processes it, and then transmits it as an electromagnetic wave with specific characteristics like amplitude, frequency, and phase. This digital communication process involves converting analog signals into binary form through quantization for digital processing and transmission. On the receiving end, the SDR decodes the transmitted signals, navigating through noise and distortion, to reconstruct the original information.[6]

SDR performs complex signal processing tasks such as compression, power control, channel estimation, equalization, error correction, and managing adaptive antennas and protocols, particularly in technologies like WCDMA. Field Programmable Gate Arrays (FPGAs) are often favored in SDRs for their performance, low power consumption, and adaptability, offering a versatile solution for implementing various functions.[3]

Despite the implication of the name, SDR is not confined to software or processor-based platforms alone. Its defining feature is its flexibility, enabling it to support various air interfaces and adapt the signal processing chain quickly and efficiently. This adaptability makes SDR a dynamic tool in modern telecommunications, capable of meeting the evolving demands of digital communication.

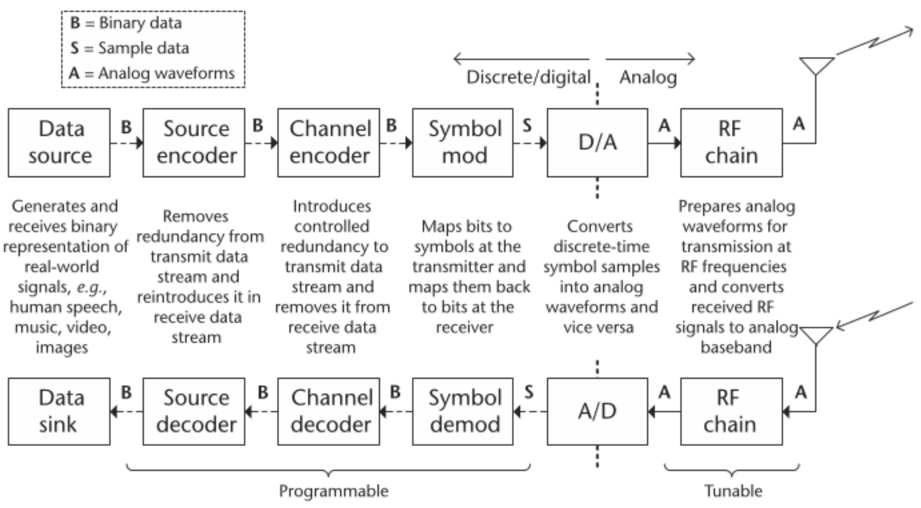

B = Binary data
S = Sample data
A = Analog waveforms
Discrete/digital
Analog
Data source
Source encoder
Channel encoder
Symbol mod
D/A
RF chain
Generates and receives binary representation of real-world signals, e.g., human speech, music, video, images
Removes redundancy from transmit data stream and reintroduces it in receive data stream
Introduces controlled redundancy to transmit data stream and removes it from receive data stream
Maps bits to symbols at the transmitter and maps them back to bits at the receiver
Converts discrete-time symbol samples into analog waveforms and vice versa
Prepares analog waveforms for transmission at RF frequencies and converts received RF signals to analog baseband
Data sink
Source decoder
Channel decoder
Symbol demod
A/D
RF chain
Programmable
Tunable

Figure 2.1: An illustration describing some of the important components that constitute a modern digital communications system.

A typical Software-Defined Radio (SDR) transceiver is composed of four principal components: Signal Processing, Digital Front End, Analog RF Front End, and an Antenna[1]. The schematic representation of a contemporary digital communication system, depicted in Figure 2.1, demonstrates the dynamic nature of SDR technology, emphasizing its reconfigurable and programmable capabilities.

The process initiates with the digital encoding of signals, aiming to enhance transmission efficiency. This is achieved by removing redundant binary patterns and introducing a controlled amount of redundancy to safeguard the data against potential errors during transmission across noisy channels.

Following this, the signal undergoes modulation, where the binary information is translated into distinct electromagnetic waveforms characterized by specific amplitudes, frequencies, and phases.

The modulated digital signals are then converted into their analog counterparts by a DAC. The Analog RF Front End takes over, upconverting the baseband analog signal to the desired Radio Frequency (RF) carrier frequency, readying it for aerial transmission through the antenna. Upon reception, the process is mirrored. The intercepted analog signal is first downconverted to a baseband frequency by the RF Front End, then digitized by an Analog-to-Digital Converter (ADC). This digital signal undergoes demodulation, converting the electromagnetic waveform back into binary data. Error correction mechanisms are employed to rectify any discrepancies caused by transmission errors, restoring the data to its original form.

A distinctive feature of SDR technology is its adaptive approach to radio frequency management, replacing conventional static filters with tunable filters. This innovation allows for real-time adjustments to the radio's frequency and bandwidth, accommodating various communication standards and optimizing the transceiver's performance across different operational conditions. This flexibility is at the heart of SDR's appeal, offering unprecedented control and efficiency in wireless communication systems.[6]

2.1.1 RF Architecture for SDR

Basic Radio Architecture

The basic radio architecture is shown on Figure 2.2. In the process of transmitting data, the initial step involves preparing the data for transmission through baseband processing. Subsequently, this processed signal undergoes modulation, wherein its frequency is shifted to align with the designated frequency band for transmission. Following modulation, the signal is amplified and transmitted via an antenna.

The channel serves as the physical link between the transmitter and receiver, which can be either wired or wireless. In the context of radio communication, the channel is typically wireless, with radio waves being emitted by an antenna at the transmitter and absorbed by another antenna at the receiver. However, it's worth noting that transmission via cable or fiber is also feasible, often employed to prevent interference with other devices utilizing the radio spectrum. Various impairments introduced in the radio channel can degrade the signal, posing challenges for the receiver in retrieving the transmitted data[2].

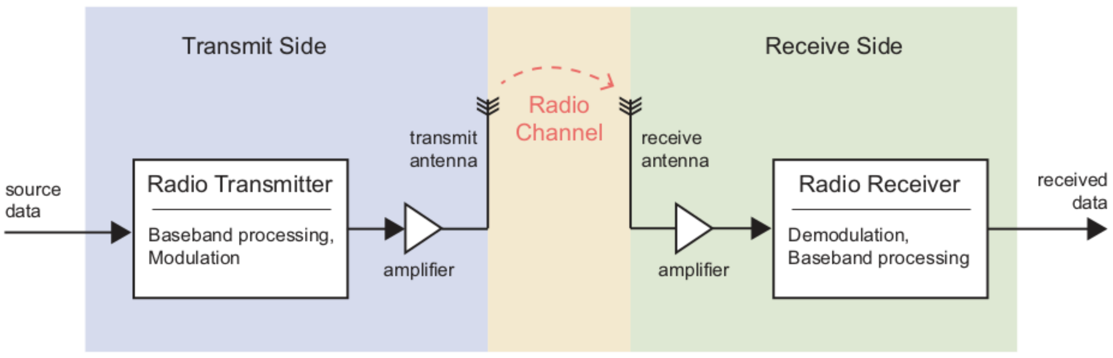


Figure 2.2: Basic Radio Architecture

Upon reception, the signal is subjected to operations reverse to those at the transmitter. Initially absorbed by a receiving antenna, the signal is then amplified, demodulated (or mixed), and further processed at baseband. Notably, there isn't a singular, definitive architecture for software-defined radio (SDR), as multiple devices and combinations thereof can be utilized to construct SDR systems.

In the realm of SDR, a diverse range of receiver architectures exists, each characterized by distinct stages and components tailored to specific operational requirements. Understanding the foundational principles behind these designs is crucial due to the absence of a universally adopted SDR architecture. Two prevalent receiver types to be elucidated further are the superheterodyne receiver and the Zero-IF receiver.

Superheterodyne Reciever

The superheterodyne receiver has been a cornerstone in radio technology, widely adopted across a variety of applications from handheld radios to unmanned aerial vehicle (UAV) data links and signal intelligence gathering. Its popularity stems from several advantages, including its ability to minimize spurious emissions through careful frequency planning, the capability to define channel bandwidth and selectivity via intermediate frequency

(IF) filters, and the flexibility to balance noise figure and linearity through strategic gain distribution across its stages.[6]

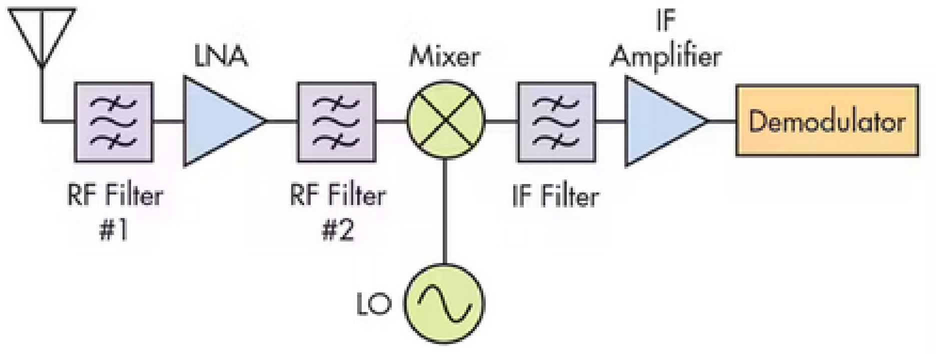


Figure 2.3: One Stage Superheterodyne Reciever Architecture

Figure 2.3 shows a traditional Superheterodyne one stage reciever architecture where initially, the received signal passes through a bandpass filter, which eliminates signals outside the desired frequency band. The signal's amplitude is then boosted by an LNA, a crucial step where the receiver's overall noise figure is significantly influenced by the LNA's own noise figure. This filter follows the LNA to discard unwanted image frequency bands, ensuring that only the desired signal proceeds to the next stage. At this juncture, the RF signal is mixed with a local oscillator (LO) signal, resulting in a lower-frequency IF signal. This IF signal is the difference between the RF and LO signal frequencies. The IF signal undergoes bandpass filtering to remove any residual unwanted signals, followed by amplification to strengthen the signal before demodulation. Finally, the amplified IF signal is demodulated to retrieve the embedded information for processing.

Superheterodyne receivers are often implemented with two frequency conversion stages. This configuration is particularly beneficial for higher-frequency applications and is shown in Figure 2.4

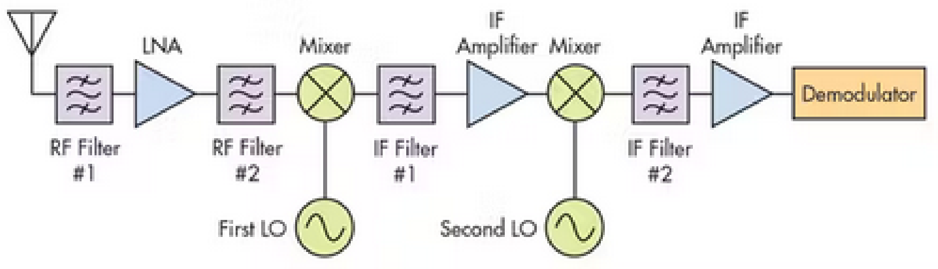


Figure 2.4: Two Stage Superheterodyne Reciever Architecture

Dual-conversion receivers, employ a two-stage frequency conversion process to enhance signal processing. Initially, a high-frequency input signal is downconverted to an IF. This IF signal is then further converted to a lower frequency for subsequent processing. This architecture includes two mixers and two local oscillators (LOs): the first mixer reduces the RF signal to the first IF frequency, and the second mixer further lowers this frequency to the second IF level. Despite adding complexity, this dual-conversion approach offers significant benefits, improving the receiver's ability to handle high-frequency signals effectively. Multi-stage superheterodyne converters, which follow a similar principle, are also widely utilized for their robust performance.

Zero-IF Recievers

The zero-IF (ZIF) or Direct-RF architecture shown on Figure **??** presents an innovative alternative to the traditional superheterodyne receiver design, streamlining the frequency conversion process and reducing component complexity. In this architecture, the signal's conversion to baseband is achieved through a singular frequency mixing stage, where the local oscillator (LO) is precisely tuned to the signal's frequency band. This approach effectively translates the received signal directly to baseband, processing it in both phase (I) and quadrature (Q) components. This method significantly reduces the need for complex and expensive RF/IF filtering, as all necessary filtering is performed at the baseband level. Here, filters are simpler, more cost-effective, and easier to design.

Furthermore, the direct-conversion strategy allows the analog-to-digital (ADC) and digital-to-analog (DAC) converters to operate directly on the I/Q data at baseband. This adjustment leads to a lower required sample rate for the converted bandwidth, thereby conserving substantial power. In essence, the receiver filters and amplifies the incoming RF signal, which then mixes with an LO signal of identical frequency. This process demodulates the signal by converting it directly to a 0-Hz baseband signal, with subsequent low-pass filtering eliminating any mixer-produced sum frequencies.

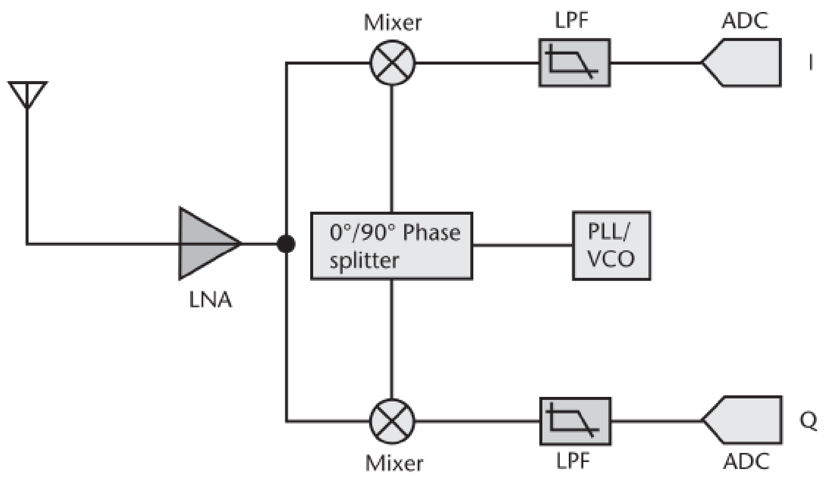


Figure 2.5: Direct-RF Reciever Architecture

However, the direct-conversion approach is not without its challenges. Real-world factors like process variation and temperature differences can impede maintaining a perfect 90-degree phase offset between the I and Q signals, which is crucial for optimal image rejection. Additionally, imperfections in LO isolation during the mixing stage can lead to carrier leakage. If these issues—image and carrier leakage—are not adequately addressed, they can diminish the receiver's sensitivity and lead to unwanted transmit spectral emissions, underscoring the importance of careful design and calibration in ZIF systems.

Bibliography

[1] Rami Akeela and Behnam Dezfouli. Software-defined radios: Architecture, state-of-the-art, and challenges. *Computer Communications*, 128:106–125, 2018.

[2] Douglas Allan, Ehinomen Atimati, Kenny W Barlee, Lewis J Brown, James Craig, Graeme Fitzpatrick, Joshua Goldsmith, Andrew Maclellan, Lewis D McLaughlin, Blair McTaggart, et al. *Software Defined Radio with Zynq Ultrascale+ RFSoC*. Number 1st. Strathclyde Academic Media, 2023.

[3] Chris H Dick and Henrik M Pedersen. Design and implementation of high-performance fpga signal processing datapaths for software defined radios. In *Embedded Systems Conference*, pages 1–16. Citeseer, 2001.

[4] Waqar Hussain, Jari Nurmi, Jouni Isoaho, and Fabio Garzia. *Computing platforms for software-defined radio*. Springer, 2016.

[5] Joseph Mitola. Software radios: Survey, critical evaluation and future directions. *IEEE Aerospace and Electronic Systems Magazine*, 8(4):25–36, 1993.

[6] Alexander M Wyglinski, Robin Getz, Travis Collins, and Di Pu. *Software-defined radio for engineers*. Artech House, 2018.

Chapter 3

High Level Synthesis

3.1 Overview

Field Programmable Gate Arrays (FPGAs) are versatile integrated circuits that can be reconfigured after manufacturing, allowing for custom circuit implementations. This reconfigurability provides a unique advantage over fixed-architecture processors like CPUs and GPUs, making FPGAs suitable for a wide range of applications, from ASIC prototyping to hardware acceleration. The evolution of FPGA technology has led to increased complexity and capabilities, including multi-die devices and system-level interconnects, encouraging both academic and industry investment. However, FPGA programming remains challenging due to its complex design process, historically reliant on Hardware Description Languages (HDLs) such as Verilog and VHDL.[9]

High-Level Synthesis (HLS) emerges as a solution to simplify FPGA design, allowing hardware functionality to be specified through software programs at a higher abstraction level. HLS tools automate the translation of behavioral descriptions into Register Transfer Level (RTL) designs, tailored to the target technology and optimizing interface and memory elements. This approach reduces time-to-market and addresses the complexity of modern systems, offering a balance between the flexibility of general-purpose processors and the efficiency of ASICs.[6]

3.2 History of HLS

The history of High-Level Synthesis (HLS) traces back to the 1970s and 1980s, marking the conceptual beginnings of this transformative technology. However, it wasn't until the turn of the century that HLS began to be recognized as a practical tool for industrial applications. Since then, the Quality of Results (QoR) provided by HLS tools has seen significant improvements with each new generation. Despite these advancements, studies reveal varying outcomes, leaving the question of whether the QoR gap has fully closed still open for debate.[6]

Research documented in papers[2][1] delves into the evolution of HLS tools over the years, highlighting the progress from early concepts to the sophisticated C/C++ based tools of today, and speculating on the future trajectory of HLS development. The journey of HLS and Accelerator-Centric Synthesis(ACS) tools visually captured in a timeline on Figure 3.1, showcasing the milestones and key developments that have shaped the HLS landscape, focusing primarily on tools based on C/C++ languages.

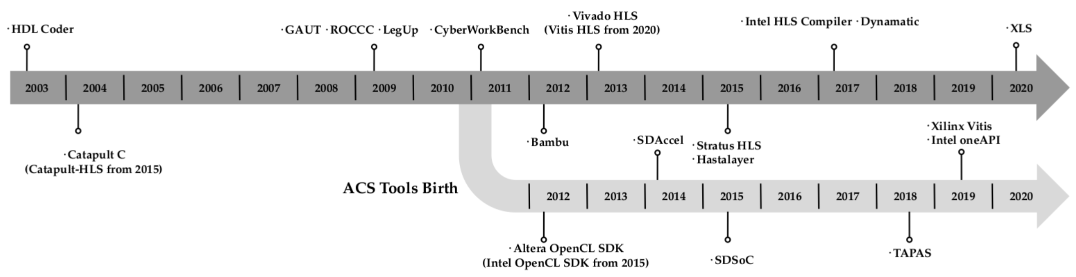


Figure 3.1: Timeline of High-Level Synthesis Tools

3.3 Analysis of Surveys

Numerous surveys[2][1][6][8][5], have provided comprehensive analyses of HLS tools, reflecting the evolving landscape of this technology. Published across different years, these surveys offer snapshots of the HLS tools available during their respective periods, highlighting the dynamic nature of HLS tool development. As the field of HLS continues to evolve, some tools discussed in earlier surveys have undergone changes, fallen out of use, or been replaced by more advanced solutions, underscoring the importance of staying updated with the latest advancements in HLS technologies.

The survey[2] presents a comprehensive overview of the advancements in High-Level Synthesis (HLS) for Field-Programmable Gate Arrays (FPGAs), highlighting the evolving landscape from early prototypes to practical deployment. It discusses the transition in the design community towards higher abstraction levels due to increasing system-on-chip complexity, underscoring the pivotal role of HLS in this shift. The paper identifies the latest generation of HLS tools, exemplified by AutoESL's AutoPilot and Xilinx's system-level platforms, as instrumental in bridging the gap between high-level language specifications and FPGA implementations. It provides an in-depth analysis of AutoPilot's HLS flow, detailing its support for a wide range of programming models, integration with FPGA-specific optimizations, and advancements in synthesis algorithms. Moreover, it showcases real-world case studies, comparing HLS-generated solutions against manually optimized designs to demonstrate HLS's viability in achieving competitive quality of results (QoR). The paper concludes by emphasizing ongoing challenges and potential future directions in HLS, particularly in enhancing simulation, verification, and domain-specific integrations, to further streamline FPGA design processes.[2]

The survey[6] provides a detailed exploration of the capabilities and performance of various HLS tools, particularly focusing on their applications and optimization possibilities. It discusses the wide range of HLS tools available, both from academia and industry, and evaluates them based on different performance metrics, such as execution time and resource utilization, using a common set of benchmarks. The paper emphasizes the significance of operation chaining, bitwidth analysis, memory space allocation, loop optimizations, and if-conversion as current research trends and potential areas for optimization in HLS tools. Furthermore, it proposes a unique methodology for comparing HLS tools, aiming to give a comprehensive view of their strengths and weaknesses in designing high-performance and energy-efficient heterogeneous systems. The survey underlines the complexity and evolving nature of HLS, highlighting the need for ongoing research to address the challenges in automatically generating hardware from high-level descriptions.

The paper[5] provides an in-depth analysis of High-Level Synthesis (HLS) in contrast to traditional manual RTL design methods, with a focus on the Quality of Results (QoR) and productivity. Through a comprehensive survey of scientific literature from 2010 onwards, covering 46 papers and 118 applications, the study evaluates the disparities in QoR and design efforts between HLS and RTL approaches. The key findings suggest that although HLS tools generally produce lower QoR compared to manual RTL designs, they significantly shorten development time, offering more than fourfold productivity improvements.

This enhanced productivity is particularly pronounced in both large-scale and small-scale projects, with HLS demonstrating a notable advantage in compactness for larger applications, despite a tendency for non-behavioral code to disproportionately affect smaller codebases.

A model case study within the paper further explores these findings by comparing HLS and RTL through practical tasks performed by individuals with varying experiences in these methodologies. This case study meticulously documents the outcomes, revealing that HLS markedly boosts productivity, thus confirming its suitability for projects requiring rapid prototyping and accelerated time-to-market.

Moreover, the paper reviews literature aimed at improving HLS tools, identifying persistent challenges and suggesting future research directions to refine HLS's effectiveness. While HLS tools currently consume more FPGA resources and may not match RTL in performance and execution time, the study observes mixed results with specific FPGA components like BRAM and DSP blocks, indicating areas where HLS may perform comparably or even more efficiently than RTL.

In summary, the paper concludes that HLS represents a valuable tool for software engineers and designers looking to undertake hardware acceleration projects, offering a significant reduction in development time and increasing productivity, despite some limitations in QoR and resource efficiency. This comprehensive analysis not only underscores the current state of HLS but also points towards its promising future with ongoing advancements and optimizations.

The paper [1] presents an extensive examination of FPGA programming abstractions, covering Hardware Description Languages (HDLs), HLS tools, and Domain-Specific Languages (DSLs). It delves into the evolution of these abstractions, highlighting their role in making FPGA programming more accessible and efficient. The survey categorizes and evaluates numerous tools and languages based on their design targets, programming models, and key features, offering insights into their benefits and limitations.

In the HLS section, it first shows the success of HLS in application areas like deep learning, video transcoding, graph processing, and genome sequencing. It then transitions into discussing the challenges and research opportunities in HLS, such as achieving high clock frequencies, addressing complex pragmas for high performance, and the transformation of legacy code to meet HLS standards. The paper also emphasizes the need for open-source HLS infrastructures, support for DSLs, and standardization efforts to overcome current limitations and enhance the technology's effectiveness.

The paper[8] proposes advancing high-level synthesis (HLS) and hardware-software co-design using Python. The paper begins by reviewing existing HLS frameworks and hig-level HDLs acknowledging the benefits they have brought to hardware design by enabling the use of more familiar programming languages to generate hardware implementations. It highlights that while most HLS frameworks have traditionally focused on C/C++ for their ease of translating into HDLs like Verilog or VHDL, there's a growing recognition of Python's potential in this space. This shift is attributed to Python's rich ecosystem, particularly its powerful libraries for scientific computation and artificial intelligence, which can significantly enhance hardware design capabilities.

The paper then explores strategies for implementing Python on FPGAs, offering three main approaches: direct execution of Python bytecode, in-FPGA implementation of "hot functions," and developing a Python to HDL transpiler. Each strategy is analyzed for its potential benefits and drawbacks, providing insights into how they might facilitate the execution of Python code on hardware platforms.

The conclusion advocates for the use of Python in higher-level synthesis and co-design, suggesting that Python's capabilities, coupled with its extensive library ecosystem, make it an excellent candidate for bridging the gap between hardware and software development. This approach aims to streamline the co-design of hybrid hardware-software solutions, ultimately making hardware acceleration more accessible to software developers and enhancing the productivity and innovation in digital hardware development.

3.4 HLS Design Flow

HLS workflows have revolutionized hardware design by significantly reducing both development and verification times and facilitating processes like power analysis. HLS tools enable the transformation of a high-level, untimed functional specification written in a High-Level Language (HLL) into a fully timed RTL implementation.

The initial phase of hardware design involves crafting a functional specification, wherein all input data is processed simultaneously, computations are executed without delay, and output data is generated concurrently. To enable realistic hardware implementation, floating-point and integer data types are converted into bit-accurate representations of specific lengths, ensuring acceptable computation accuracy. From this bit-accurate specification, an optimized hardware architecture is derived, starting with HLS tools. They translate untimed or partially timed high-level specifications into fully timed implementations. These tools automatically or semi-automatically generate custom architectures tailored to efficiently implement the specification. The resulting architecture, described at the register-transfer level (RTL), comprises a data path consisting of registers, multiplexers, functional units, and buses, along with a controller, as dictated by the specification and design constraints.

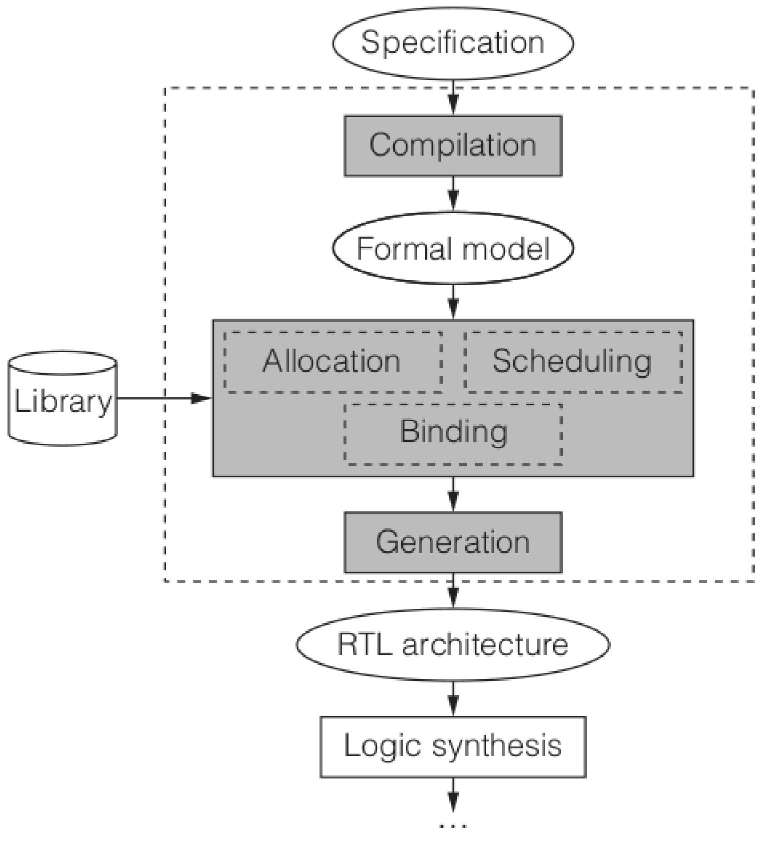


Figure 3.2: HLS Design Steps

Figure 3.2 shows the Design steps used in HLS Tools where the functional specification is transformed into a formal representation. This phase often includes optimizations such as dead-code elimination, false data dependency elimination, constant folding, and loop transformations.

The next phase is resource allocation together with scheduling and binding. This phase determines the type and quantity of hardware resources required to meet design constraints. These resources encompass functional units, storage elements, buses, and connectivity components. All operations specified in the model are scheduled into clock cycles. Depending on the functional component to which an operation is mapped, it may be scheduled within a single clock cycle or spread across multiple cycles. Operations can be chained, where the output of one operation directly feeds into another. Variables that persist across cycles must be bound to storage units. Additionally, variables with non-overlapping or mutually exclusive lifetimes may share the same storage units.

Building upon decisions made in the preceding phases, the goal of this step is to apply all design decisions and generate an RTL model of the synthesized design. This model represents the final hardware architecture optimized for the given functional specification.[3]

3.5 Recent advancements using HLS

In this section, an overview will be provided of papers not previously discussed in the mentioned surveys, focusing on their key contributions without delving deeply into technical details.

The paper [7] presents CFU Playground, an open-source framework that enables rapid design and evaluation of machine learning (ML) accelerators for embedded ML systems, with a focus on Full-Stack Open-Source Framework for Tiny Machine Learning (TinyML) Acceleration on FPGAs. It emphasizes the necessity of agile design flows for hardware-software co-design and domain-specific optimizations due to the increasing adoption of specialized hardware. The framework facilitates a seamless end-to-end flow for co-design on FPGAs, offering a platform for exploring experimental architectures optimized for embedded ML. CFU Playground integrates various open-source tools across the software, RTL generation IP, and FPGA toolchains, including TensorFlow Lite Micro/TFLM, LiteX, VexRiscv, Migen, Amaranth, yosys, nextpnr, and F4PGA/SymbiFlow, promoting customization free from licensing restrictions and vendor lock-in. It demonstrates significant performance improvements in TinyML applications, showing the framework's potential for efficient, model-specific ML acceleration on resource-constrained FPGA platforms.

Maia SDR leverages High-Level Synthesis (HLS) and the Python-based hardware description language, Amaranth, to develop open-source FPGA firmware for the ADALM Pluto, enhancing its capabilities for software-defined radio applications with features like web-based interfaces and real-time signal processing.[4]

The first FPGA implementation of an approximate tensorial Support Vector Machine (SVM) classifier with algorithmic level ACTs using HLS is presented.[10]

3.6 Amaranth HLS

Bibliography

[1] Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. Fpga hls today: successes, challenges, and opportunities. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 15(4):1–42, 2022.

[2] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[3] Philippe Coussy, Daniel D Gajski, Michael Meredith, and Andres Takach. An introduction to high-level synthesis. *IEEE Design & Test of Computers*, 26(4):8–17, 2009.

[4] Daniel Estévez. Maia sdr: An open-source fpga-based sdr project focusing on the adalm pluto. 2023.

[5] Sakari Lahti, Panu Sjövall, Jarno Vanne, and Timo D Hämäläinen. Are we there yet? a study on the state of high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(5):898–911, 2018.

[6] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, et al. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2015.

[7] Shvetank Prakash, Tim Callahan, Joseph Bushagour, Colby Banbury, Alan V Green, Pete Warden, Tim Ansell, and Vijay Janapa Reddi. Cfu playground: Full-stack open-source framework for tiny machine learning (tinyml) acceleration on fpgas. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 157–167. IEEE, 2023.

[8] Alexandre Quenon and V Ramos Gomes Da Silva. Towards higher-level synthesis and co-design with python. In *Proceedings of the Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE'21)*. ACM New York, NY, USA, 2021.

[9] Emanuele Del Sozzo, Davide Conficconi, Alberto Zeni, Mirko Salaris, Donatella Sciuto, and Marco D Santambrogio. Pushing the level of abstraction of digital system design: A survey on how to program fpgas. *ACM Computing Surveys*, 55(5):1–48, 2022.

[10] Hamoud Younes, Ali Ibrahim, Mostafa Rizk, and Maurizio Valle. Algorithmic-level approximate tensorial svm using high-level synthesis on fpga. *Electronics*, 10(2):205, 2021.

Chapter 4

DSP Modules

4.1 Analog to Digital Converter

The conversion of an analog signal to a digital one is typically explained by two distinct procedures: periodic sampling and amplitude quantization. In the first step, the continuous signal is sampled at regular time intervals denoted by T_s. The discrete samples x[n] of the ongoing signal x(t)x(t) can be expressed as x[n]=x(nT_s)x[n]=x(nT_s). This sampling action has a specific repercussion in the frequency domain; it causes the original signal's frequency spectrum to replicate itself at intervals equal to the sampling frequency, defined as fs=$\frac{1}{T_s}$.

For the original continuous-time signal to be perfectly reconstructed from its samples, it's essential that these copies of the frequency spectrum do not interfere with each other. When these spectral replicas overlap, it leads to a phenomenon called aliasing, which obstructs the accurate reconstruction of the original continuous signal from its samples.[1]

4.1.1 Sigma-Delta($\Sigma - \Delta$) ADC

Overview

Sigma-Delta ($\Sigma - \Delta$) Analog-to-Digital Converters (ADCs) represent a class of high-precision and high-resolution conversion technologies pivotal in modern electronic systems. These converters distinguish themselves by employing a method that integrates oversampling, noise shaping, and digital filtering to achieve remarkable fidelity in digitizing analog signals.

The cornerstone of the ($\Sigma - \Delta$) ADC lies in its unique architecture, which, unlike traditional ADCs that sample at just above the Nyquist rate, operates at a sampling frequency significantly higher than the minimum required. This oversampling approach, coupled with a feedback loop inherent to the ($\Sigma - \Delta$) modulator, effectively pushes quantization noise to higher frequencies. As a result, the signal of interest, typically located in the lower frequency band, is left relatively unscathed by noise, thus enabling the recovery of a high-fidelity digital representation from the analog world.

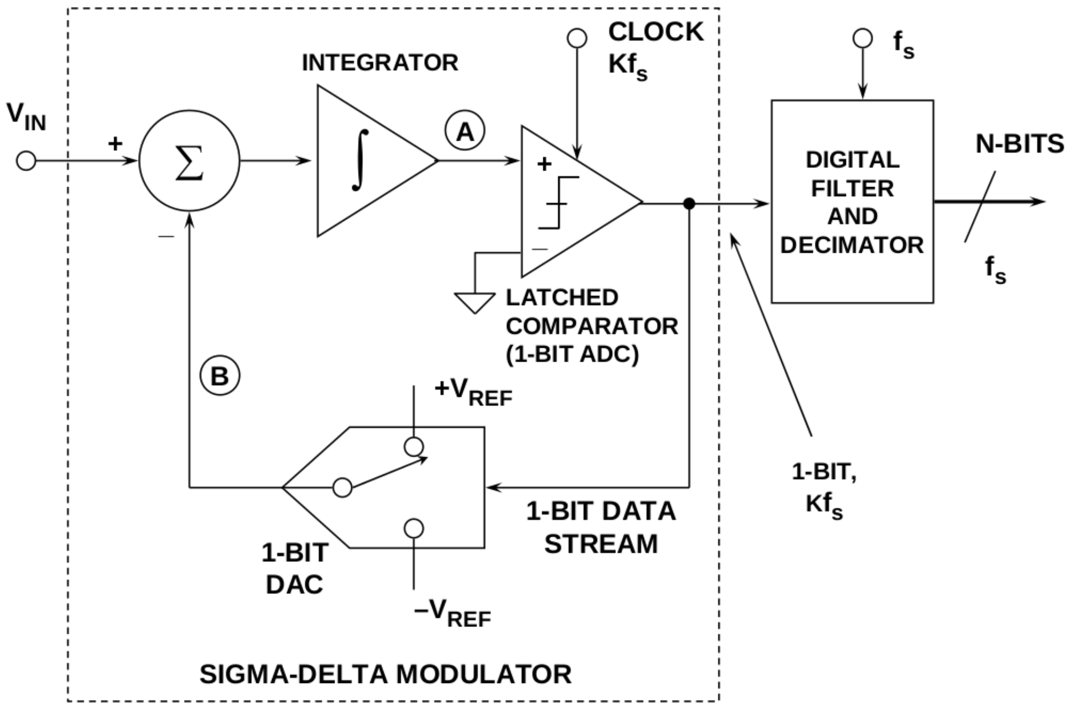


Figure 4.1: Sigma-Delta Analog to Digital Converter

The general scheme of the $\Sigma - \Delta$ converter is given in the Figure 4.1. A delta sigma ADC always consists of a delta sigma modulator which produces the bitstream and a digital filter and decimator.

One of the primary advantages of Sigma-Delta $\Sigma - \Delta$ converters lies in their predominantly digital implementation. The majority of the converter's architecture, except for a few components, uses digital circuitry, which enhances precision, reliability, and scalability in fabrication. Typically, the only analog component is the integrator, which can often be realized using a straightforward RC circuit. This simplicity in the analog domain not only facilitates easier integration into mixed-signal environments but also aligns with modern semiconductor manufacturing processes that favor digital over analog circuitry.

An additional merit of $\Sigma - \Delta$ modulation is its reliance on a 1-bit analog-to-digital conversion process, significantly simplifying the analog circuitry requirements. The analog signal processing components need not match the high resolution of the final digital output, a factor that substantially reduces the demand for precision in the analog domain. With advancements in Very-Large-Scale Integration (VLSI) technology, there has been significant progress in handling 1-bit data streams efficiently. [1].
The bitstream is a one-bit serial signal with a bit rate much higher than the data rate e.g. of the ADC. Its major property is that its average level represents the average input signal level. The serial transmission of numerically represented signal values (e.g. the serial output of a conventional ADC) is called pulse code modulation (PCM).

Sigma-Delta Modulator

Looking at the figure 4.1 it is important to understand the sigma-delta modulator circuit. The input to the integrator is a difference of the input signal and the 1-bit DAC in the feedback which is the output of the comparator that compares the integrator output with a reference voltage, commonly 0V. The output of the comparator can only have values of 1 or -1, the only instance in which it equals the input value V_{IN} is when V_{IN} is one of the two values, otherwise there will always be an error signal going in the integrator.

For example, consider a DC input for V_{IN}. and the error is negative, the negative values are accumulated by the integrator and go into the comparator. After a number of clock cylces, enough negative values will have accumulated to cause the comparator to produce -1, thereby changing the sign of the error to be positive. This process illustrates the modulator's ability to self-correct: by accumulating errors in one direction until a threshold is met, it then switches the output, which in turn, changes the direction of error accumulation. This dynamic adjustment continues, ensuring that over time, the output accurately represents the input signal, even when starting with an error in the signal representation.

If the analog signal remains at a fixed dc level for a period of time, an alternating pattern of "0s" and "1s" is obtained. The average DAC output voltage is controlled by the ones-density in the 1-bit data stream from the comparator output. As the input signal increases towards +VREF, the number of "ones" in the serial bit stream increases, and the number of "zeros" decreases. Similarly, as the signal goes negative towards –VREF, the number of "ones" in the serial bit stream decreases, and the number of "zeros" increases. For this reason, the output of the sigma-delta modulator using a 1 bit quantizer is often said to be in a pulse density(PDM) format. From a very simplistic standpoint, this analysis shows that the average value of the input voltage is contained in the serial bit stream out of the comparator. The digital filter and decimator process the serial bit stream and produce the final output data. For any given input value in a single sampling interval, the data from the 1-bit ADC is virtually meaningless. Only when a large number of samples are averaged, will a meaningful value result.[4]

Oversampling

Oversampling in the context of $\Sigma - \Delta$ ADCs is the process of sampling the input signal at a rate considerably higher than the Nyquist rate, which is twice the maximum frequency component present in the signal. The intent of oversampling is not to capture finer details of the signal itself, but rather to spread the inherent quantization noise over a wider frequency spectrum. This redistribution of noise allows $\Sigma - \Delta$ ADCs to trade off a higher sampling frequency for increased resolution within the band of interest.

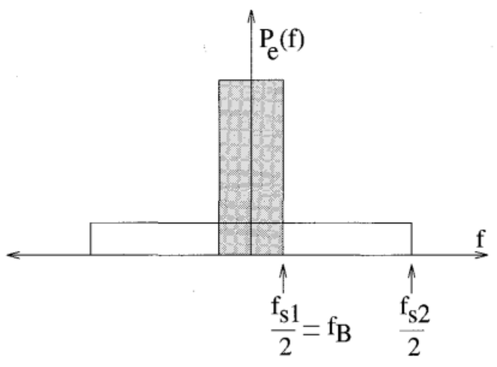


Figure 4.2: Illustration of quantization noise power spectral density in Nyquist versus Oversampled ADC

Figure 4.2 demonstrates the quantization noise power spectral density P_e(f) for standard Nyquist rate sampling at frequency f_{s1} and for an oversampling scenario at a higher frequency f_{s2}. With Nyquist sampling, where the signal bandwidth (f_B) is half the sampling rate ($\frac{f_s}{2}$), the entire quantization noise power is confined within the signal bandwidth, depicted by the area under the tall shaded rectangle. Conversely, with oversampling, the same total noise power illustrated by the area under the broader unshaded rectangle is spread over a bandwidth that exceeds the signal bandwidth (f_B) by a significant margin. Consequently, the fraction of noise power within the signal band $[-f_B,f_B]$ is considerably reduced. Subsequent digital low-pass filtering effectively eliminates noise components outside this signal band, and the signal can then be downsampled back to the Nyquist rate, preserving the enhanced SNR ratio achieved through oversampling. Note that in this scheme, we are trading speed for resolution. [1]

Another advantage, stemming directly from oversampling, is the reduced necessity for an analog anti-aliasing filter with a steep cutoff curve. This relaxation in filter requirements is due to the broader distribution of quantization noise, which lessens the likelihood of aliasing effects within the signal bandwidth. However, this benefit in the analog domain is counterbalanced by increased demands on digital processing.

It's important to underline that achieving high-resolution digital conversion mandates a significant oversampling rate, with f_s. The oversampling rate needs to be signifantly higher than the signal bandwith while still being practically possible to implement given the circuits limitations.

4.1.2 Sigma-Delta ADCs in FPGA

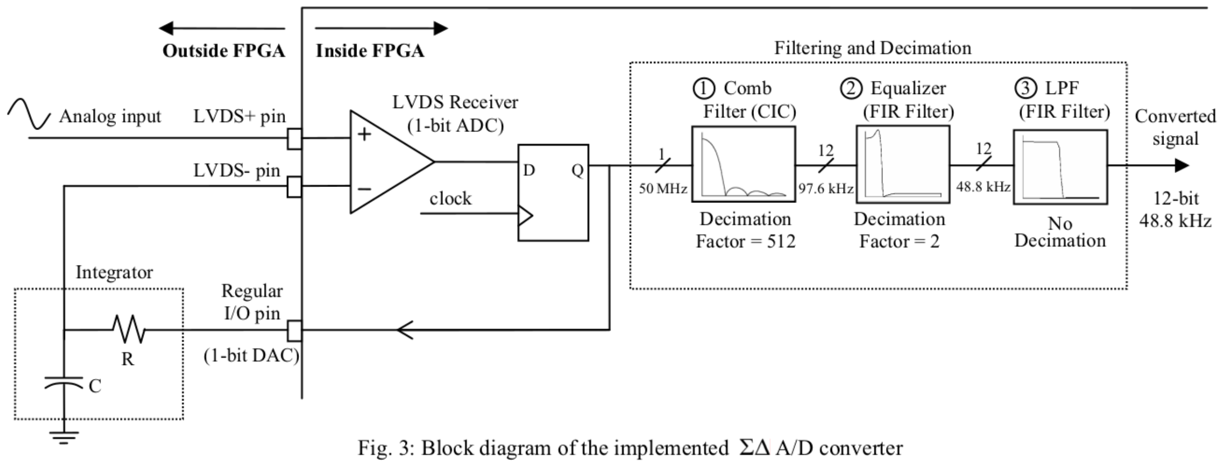


Fig. 3: Block diagram of the implemented ΣΔ A/D converter

Figure 4.3: Sigma-Delta ADC on FPGA

This section briefly explains how the Sigma-Delta ADC can be used on a FPGA using an LVDS input as an comparator so that it minimizes the use of external analog components. LVDS is a low swing, differential signaling technology which allows single channel data transmission at up to hundreds of megabits per second and it will be explained in more detail in a later section.[7]

The so far explained $\Sigma - \Delta$ ADC is shown on figure 4.3 where the only difference is the realization of the comparator.
As can be seen, it uses 3 FPGA pins: 2 LVDS pins for the analog signals, and one conventional I/O pin which feeds the integrator. By applying the analog input to the comparator's positive side, it is continuously compared against the integrated signal, limiting external components to just a resistor and capacitor that construct the integrator.

In the simplest version of this design, the integrator features a single resistor and capacitor, forming a basic low-pass filter for the Pulse Width Modulation (PWM) feedback signal. It has the advantage of low parts count. Its chief disadvantage is that the analog signal is limited to the input voltage range of the comparator. The time-constant, $\tau =$ RC, should be made large enough to adequately filter the PWM stream, but not so large to dampen response time. Given the over-sampling clock frequency, fCLK, then τ x fCLK = 200 to 1000 is recommended. An optional resistor can be placed in line with the analog input to protect the high-impedance input of the comparator.[6]

This innovative approach facilitates the incorporation of cost-effective A/D conversion capabilities into FPGAs, which traditionally lack analog interfaces.[5] The filtering and decimation stage, depicted in Figure 4.3 consists of a Cascaded Integrator-Comb (CIC) filter for sampling rate reduction, an equalizer to offset magnitude droop while adjusting the rate, and a low-pass filter that ensures the output remains within the desired frequency band.

4.1.3 Low Voltage Differential Signaling

The unprecedented surge in digital communications has become the primary catalyst for advancements in high-speed interconnects across chips, functional boards, and systems. In this digital epoch, the data, fundamentally digital, is predominantly transmitted through Analog Low-Voltage Differential Signaling (LVDS) technology. LVDS has been the preferred choice for designers due to its established capabilities in facilitating high-speed data transmission. It offers significant benefits such as reduced power consumption, superior noise immunity, and cost-effectiveness. These advantages have rendered LVDS particularly attractive for point-to-point applications within the realms of telecommunications, data communications, and display technologies. The adoption of LVDS underscores the industry's commitment to leveraging proven technologies that enhance performance while optimizing resource utilization.

The physical layer of the Low-Voltage Differential Signaling (LVDS) system features an equivalent circuit structure as depicted in Figure 4.4. Within this structure, the driver is equipped with a current source designed to limit the output to approximately 3 mA. Furthermore, a switch box is utilized to direct the current through the termination resistor, thereby enabling the differential driver to facilitate odd-mode transmission. This mode of transmission is characterized by equal yet opposite currents coursing through the transmission lines, with the currents returning within the wire pair. This configuration minimizes the loop area through which the current flows, significantly reducing the generation of electromagnetic interference (EMI).[3]

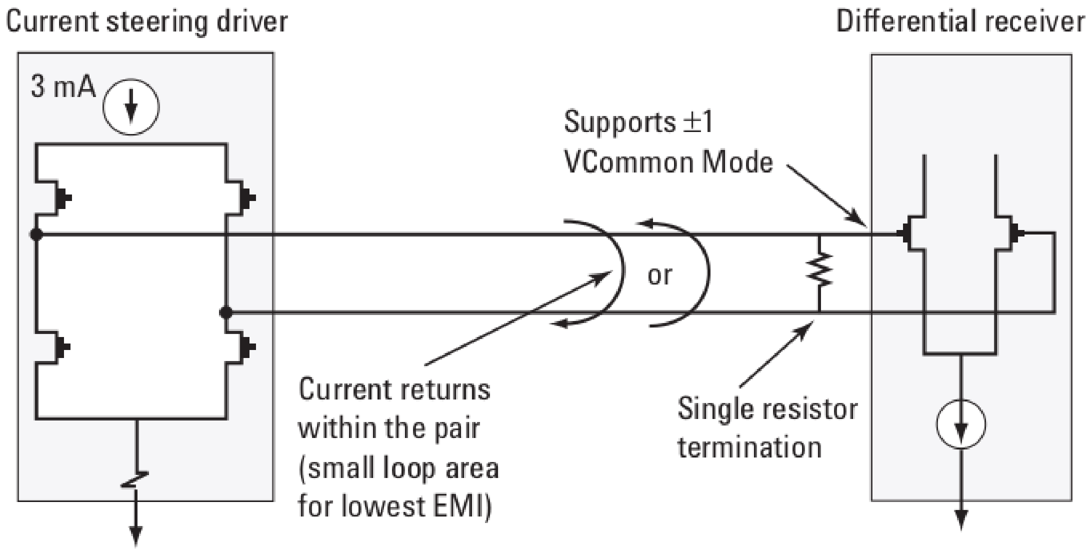

Current steering driver
3 mA
Differential receiver
Supports ±1 VCommon Mode
or
Current returns within the pair (small loop area for lowest EMI)
Single resistor termination

Figure 4.4: The equivalent circuit structure of the LVDS physical layer [3]

The inclusion of a current source plays a pivotal role in mitigating any potential spike currents that may arise during signal transitions. Since the input impedance of the receiver is high, the entire current effectively flows through the 100 Ω termination resulting in a (nominal) 350 mV voltage across the receiver inputs. The receiver threshold is guaranteed to be 100 mV or less, and this sensitivity is maintained over a wide common mode from 0V to 2.4V.[2]

The differential receiver is a high-impedance device that detects differential signals as low as 20 mV and then amplifies them into standard logic levels. The signal has a typical driver offset of 1.2 V, and the receiver accepts an input range of ground to 2.4 V. This allows rejection of common mode noise picked up along the interconnect of up to ± 1 V.[3]

In the process of determining the optimal signal-level voltages for drivers and receivers, the standards committee meticulously evaluated the integration of Low-Voltage Differential Signaling (LVDS) across a diverse spectrum of technologies, including Bipolar, BiCMOS, CMOS, and GaAs. Furthermore, the working group exhibited a forward-thinking approach by considering a broad array of power supply specifications, encompassing 5 V, 3.3 V, and 2.5 V. [3]

To increase noise immunity and noise margins even further, LVDS uses differential data transmission. Differential signals are immune to common-mode noise, the primary source of system noise. Because its voltage change between logic states is only 300 mV, LVDS can change states very fast. Low voltage swing reduces power consumption because it lowers the voltage across the termination resistors and lowers the overall power dissipation.

Bibliography

[1] Pervez M Aziz, Henrik V Sorensen, and J Vn der Spiegel. An overview of sigma-delta converters. *IEEE signal processing magazine*, 13(1):61–84, 1996.

[2] Texas Instruments. Lvds owner's manual. *Jan*, 2008.

[3] Stephen Kempainen. Low-voltage differential signaling (lvds). *Altera Co-operation*, 2002.

[4] Walt Kester. Adc architectures iii: Sigma-delta adc basics. *Analog Devices, MT022*, 2008.

[5] Harsha Vardhini Palagiri, Madhavi Latha Makkena, and Krishna Reddy Chantigari. Performance analysis of first order digital sigma delta adc. In *2012 Fourth international conference on computational intelligence, communication systems and networks*, pages 435–440. IEEE, 2012.

[6] Lattice Semiconductor. Simple sigma-delta adc, 2019.

[7] Fabio Sousa, Volker Mauer, Neimar Duarte, Ricardo P Jasinski, and Volnei A Pedroni. Taking advantage of lvds input buffers to implement sigma-delta a/d converters in fpgas. In *2004 IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 1, pages I–1088. IEEE, 2004.

4.2 Cascaded Integrator-Comb Filters

4.2.1 Introduction

Cascaded integrator-comb (CIC) digital filters are computationally-efficient implementations of narrowband lowpass filters, and are often embedded in hardware implementations of decimation, interpolation, and delta-sigma converter filtering.[7]
Large rate changes require very narrow band filters,fast multipliers and very long filters. This can end up being the largest bottleneck in a DSP system.[3]

CIC filters are well-suited for anti-aliasing filtering prior to decimation (sample rate reduction) and for anti-imaging filtering for interpolated signals (sample rate increase). Both applications are associated with very high-data rate filtering such as hardware quadrature modulation and demodulation in modern wireless systems, and delta-sigma A/D and D/A converters. Implementing an FIR filter can consume quite a bit of FPGA resources that are often scarce. An important benefit of CIC filters is that it's implementation does not use any multipliers.

To lay a solid foundation for comprehending the intricacies of CIC Filters, it is essential to first delve into the concept of Moving Average Filters. This initial step is crucial as CIC filters can be viewed as a sophisticated variation of the moving average filter, employing a unique method of implementation.

4.2.2 Moving Average Filter

As the name implies, the moving average filter operates by averaging a number of points from the input signal to produce each point in the output signal. Because it is so very simple, the moving average filter is often the first thing tried when faced with a problem. Even if the problem is completely solved, there is still the feeling that something more should be done. Not only is the moving average filter very good for many applications, it is optimal for a common problem, reducing random white noise while keeping the sharpest step response.[9]

However, moving average filters exhibit significant drawbacks, including pronounced attenuation within the pass band, a gradual transition from pass to stop band, undesirable time domain responses such as ringing, and insufficient stop band attenuation.

The conventional D-point moving average process necessitates D-1 summations and a single division operation for each output and is shown in figure 4.5.

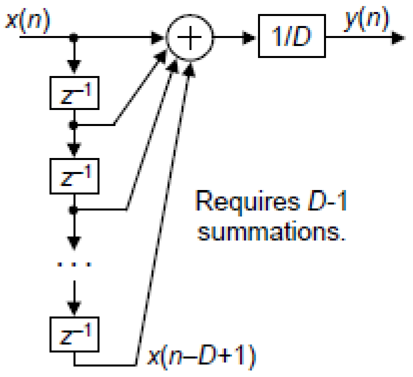


Figure 4.5: D-point Moving Average filter[7]

The D-point moving-average filter's time-domain output is expressed as

$$y(n) = \frac{1}{D}[x(n) + x(n-1) + x(n-2) + x(n-3) + ... + x(n-D+1)] \tag{4.1}$$

where n is the time-domain index. The z-domain expression for this moving averager's output is

$$Y(z) = \frac{1}{D}[X(z) + X(z)z^{-1} + X(z)z^{-2} + ... + X(z)z^{-D+1}] \tag{4.2}$$

while its z-domain HMA(z) transfer function is

$$H_{MA}(z) = \frac{Y(z)}{X(z)} = \frac{1}{D}[1 + z^{-1} + z^{-2} + ... + z^{-D+1} = \frac{1}{D}\sum_{n=0}^{D-1} z^{-n} \tag{4.3}$$

Using the finite sum formula we get the following equation

$$H_{MA}(z) = \frac{1}{D}\sum_{n=0}^{D-1} z^{-n} = \frac{1}{D} \cdot \frac{1 - z^{-D}}{1 - z^{-1}} \tag{4.4}$$

Figure 4.6 illustrates that the moving average filter functions as a lowpass filter, characterized by a frequency response reminiscent of a sinc function. This figure displays the frequency responses for moving average filters of 1st Order, 2nd Order, and 4th Order. The position of the initial null in the frequency response is determined by the span, D, over which the moving average is computed. Typically, nulls occur at frequency intervals of $\frac{fs}{D}$, with fs representing the sampling rate and D the number of points in the moving average.
Given that the filter coefficients are constant, the customization of the filter hinges on just two adjustable parameters: the quantity of samples averaged together and the count of filters linked in series.

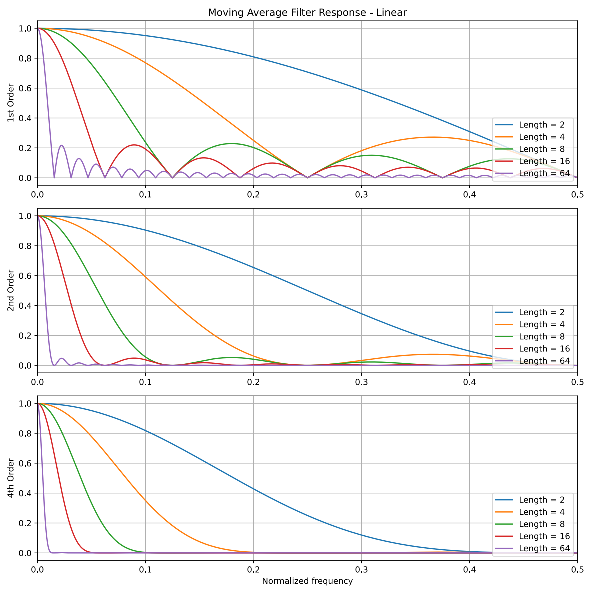


Figure 4.6: Plot of Frequency Response of Moving Average filter for different Lengths and Orders

Usually, the filters frequency behavior is observed using the log Y axis shown in Figure 4.7

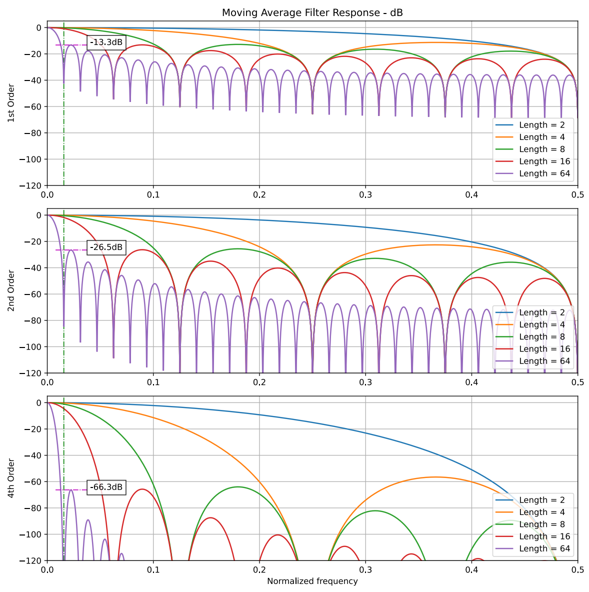


Figure 4.7: Plot of Frequency Response of Moving Average filter for different Lengths and Order with Log Y Axis

The moving average filter lacks a precisely defined pass band; it begins with a flat response near the zero frequency and gradually steepens as the frequency increases.
For a first-order filter, the attenuation in the stop band is distinctly defined at -13.3dB, regardless of the sample count being averaged.
Increasing the filter's order boosts the stop band attenuation proportionately. The attenuation can be calculated by multiplying the first order filter's stop band attenuation by the number of stages, resulting in -26.5dB for two stages and -66.3dB for four stages.

4.2.3 Recursive Running Sum Filter

The basic structure of the Recursive Running Sum Filter is given in the Figure 4.8.

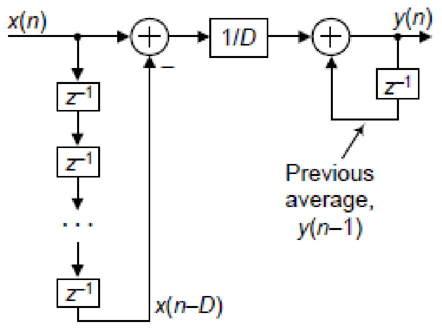


Figure 4.8: D-point Recursive Running Sum filter[7]

Ignoring the 1/D scaling, there we see that the current input sample x(n) is added to, and the oldest input sample x(n–D) is subtracted from, the previous output average y(n–1). It's called "recursive" because it has feedback. [7]
Each filter output sample is retained and used to compute the next output value. The recursive running sum filter's difference equation is

$$y(n) = \frac{1}{D}[x(n) - x(n - D)] + y(n - 1) \tag{4.5}$$

having a z-domain H_{RRS}(z) transfer function of

$$H_{RRS}(z) = \frac{Y(z)}{X(z)} = \frac{1}{D} \cdot \frac{1 - z^{-D}}{1 - z^{-1}} \tag{4.6}$$

Equation 4.2.2 is the nonrecursive expression, and Equation 4.2.3 is the recursive expression, for a D-point averager.

The standard moving averager in Figure 4.5 must perform D–1 additions per output sample. The recursive running sum filter has the advantage that only one addition and one subtraction are required per output sample, regardless of the delay length D. This computational efficiency makes the recursive running sum filter attractive in many applications seeking noise reduction through averaging.[7]

4.2.4 CIC Filters

If the delay line representation is condensed and 1/D scaling is ignored in Figure 4.8 the classic form of a 1st-order CIC filter is obtained, whose cascade structure is shown in Figure 4.9.[7]

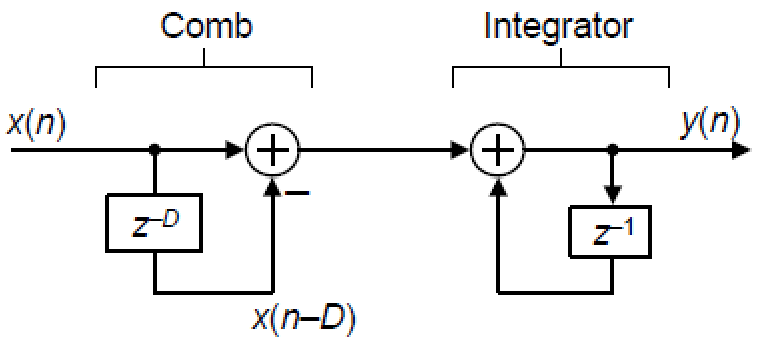


Figure 4.9: D-point recursive running sum filter in a cascaded integrator-comb implementation[7]

The integrator section of CIC filters is implemented as a single-pole IIR filter with a unity feedback coefficient.[3] The integrator is also know as an accumulator.

$$y[n] = y[n-1] + x[n] \tag{4.7}$$

The system fuction for a single integrator is

$$H_I(z) = \frac{1}{1 - z^-1} \tag{4.8}$$

The power response is basically a low-pass filter with a -20 dB per decade (-6 dB per octave) rolloff, but with infinite gain at DC. This is due to the single pole at z = 1; the output can grow without bound for a bounded input. In other words, a single integrator by itself is unstable.[3]

The feedforward portion of the CIC filter is called the comb section. D is a design parameter and is called the differential delay. It can be any positive integer, but it is usually limited to 1 or 2. The comb stage subtracts a delayed input sample from the current input sample and its time-domain equation is

$$y[n] = x[n] - x[n-D] \tag{4.9}$$

While the corresponding transfer function is given by

$$H_C(z) = 1 - z^D \tag{4.10}$$

When D = 1, the power response is a high-pass function with 20 dB per decade (6 dB per octave) gain (after all, it is the inverse of an integrator).

The CIC filter's time-domain difference equation is[7]

$$y(n) = x(n) - x(n - D) + y(n - 1) \tag{4.11}$$

and its z-domain transfer function is

$$H_{CIC}(z) = \frac{Y(z)}{X(z)} = \frac{1 - z^{-D}}{1 - z^{-1}} \tag{4.12}$$

They have the same z-domain pole/zero locations, their frequency magnitude responses have identical shapes, their phase responses are identical, and their transfer functions differ only by a constant scale factor.

The CIC filter's frequency response is:

$$H_{CIC}(e^{j2\pi f}) = e^{(-j2\pi f(D-1)/2)} \cdot \frac{sin(2\pi f D/2)}{sin(2\pi f/2)} \tag{4.13}$$

While the CIC filter's magnitude response is, where 2πf is equal to ω:

$$|H_{CIC}(e^{j\omega})| = |\frac{sin(\omega D/2)}{sin(\omega/2)}| \tag{4.14}$$

If we ignore the phase factor in Equation 4.2.4, that ratio of sin() terms can be approximated by a $\frac{sin(x)}{x}$ function.

The potential issue of a filter pole on the unit circle of the z-plane is not a concern for the H_{CIC}(z) transfer function, due to the CIC filter's coefficients being ones, which fixed-point formats can represent perfectly. This ensures the filter pole stays within the unit circle. Despite being recursive, CIC filters are stable, have linear-phase, and possess finite impulse responses, eliminating the usual risks.

Due to their linear properties, it is advisable to interchange the positions of the comb and integrator, while also embedding decimation by an integer factor R, indicating a change in sample rate, as depicted in Figure 4.10. Demonstrating this arrangement involves simply formulating the corresponding time-domain equations for y(n)y(n) for any given sample size, confirming that swapping the comb and integrator sections does not alter the fundamental equation.

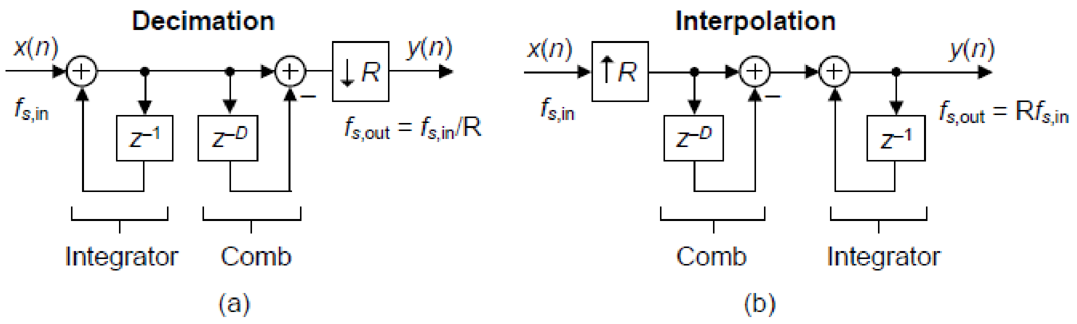


Figure 4.10: Single-stage CIC filters used in (a) decimation and (b) interpolation. (Sample rates fs,in and fs,out are the sample rates of the x(n) and y(n) sequences respectively.)[7]

The decimation (also called "down-sampling") operation ↓R means discard all but every Rth sample, resulting in an output sample rate of $f_{s,out} = \frac{f_{s,in}}{R}$. This will be explained in detail in a later subsection.

The most common method to improve CIC filter anti-aliasing and image-reject attenuation is by cascading multiple CIC filters. The transfer function of the M-th order CIC filter is given below

$$H_{CIC,M-thorder}(z) = \frac{W(z)}{X(z)} = [\frac{1-z^{-D}}{1-z^{-1}}]^M \tag{4.15}$$

While the magnitude response of the M-th order CIC filter is

$$|H_{CIC,M-thorder}(e^{j\omega})| = |\frac{sin(\omega D/2)}{Dsin(\omega/2)}|^M \tag{4.16}$$

The above equation has a high droop in the desired passband that is dependent upon the decimation factor D and the cascade size M.[2] The trade-off for enhanced anti-aliasing performance includes the need for more hardware adders and greater passband droop in the CIC filter.

The integrator section consistently utilizes a single delay register, whereas the comb section incorporates a number of delay registers equal to the moving average filter's defined sample count. Despite the comb section's substantial use of delay registers, these filters are predominantly employed for decimation and interpolation purposes. This discussion will concentrate on decimation, especially in the context of examining the Receiver component of a Software Defined Radio (SDR) application on an FPGA. Observing Figure 4.10(a) reveals that after processing through the comb section, R computed values are systematically omitted.

In CIC filters, the comb section can precede, or follow, the integrator section. However it's sensible to put the comb section on the side of the filter operating at the lower sample rate. Swapping the Figure 4.10 comb filters with the down-sampling operations results in the most common implementation of CIC filters as shown in Figure 4.11. This was one of the key features of CIC filters introduced to the world by Hogenauer in 1981[5].

In Figure 4.11 the decimation filter's comb section now has a reduced delay length (differential delay) of N = D/R. That's because an N-sample comb delay after down-sampling by R is equivalent to a D-sample comb delay before down-sampling by R. Likewise for the interpolation filter; an N-sample comb delay before up-sampling by R is equivalent to a D-sample comb delay after up-sampling by R.[7]

Those Figure 4.11 configurations yield two major benefits: first, the comb sections' new differential delay lines are decreased in length to N = D/R reducing data storage requirements; second, the comb section now operates at a reduced clock rate. Both of these effects reduce hardware power consumption. [7]

When used as part of a decimator, a moving average filter that started out as a design with (n-1) delay stages and (n-1) adders running at the incoming sample rate, has been reduced to 2 delay stages, 1 adder, and 1 subtractor, and half of the logic is running at a much slower rate.

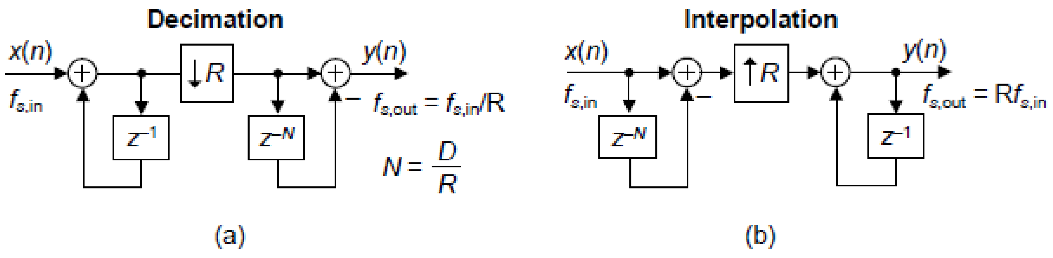


Figure 4.11: Reduced comb delay, single-stage, CIC filter implementations: (a) for decimation; (b) for interpolation. [7]

4.2.5 CIC Filter Gain

The gain at DC (0 Hz) for a 1st-order CIC decimation filter is given by the delay D of the comb filter, which equals NR. For CIC decimation filters of higher orders, the total gain is calculated as $(NR)^M$. To normalize the output of an Mth-order CIC decimation filter to a gain of one, one can simply perform a binary right shift by $log_2((NR)^M)$ bits, provided that $(NR)^M$ is an exact power of two.

4.2.6 Decimation

Decimation is the two-step process of lowpass filtering followed by an operation known as downsampling. A sequence can be dowsampled by a factor of M by retaining every Mth sample and discarding all the remaining samples. Relative to the original sample rate, fs,old, the sample rate of the downsampled sequence is[6]

$$f_{s,new} = \frac{f_{s,old}}{M} \tag{4.17}$$

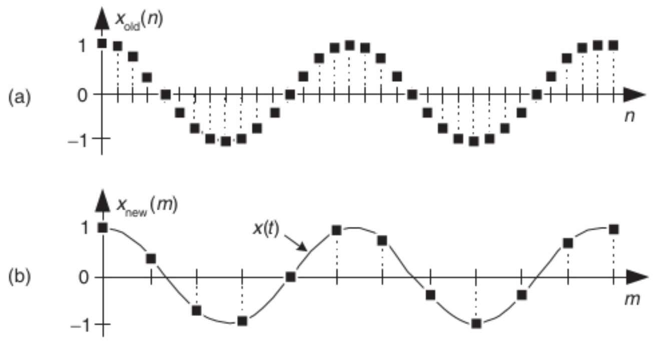


Figure 4.12: Sample rate conversion: (a) original sequence; (b) downsampled by M = 3 sequence[6]

Figure 4.12 shows an example of sample rate conversion with a factor of M=3, while the Figure 4.13 shows the spectral implications of downsampling.

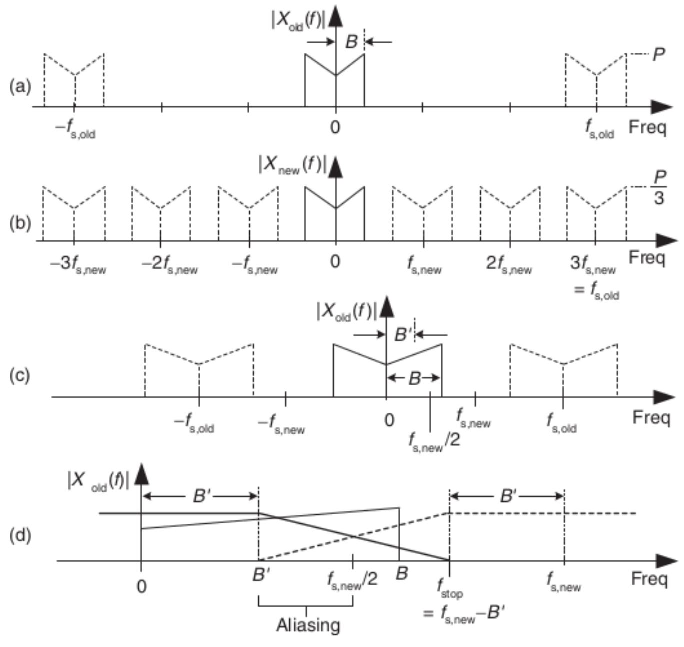


Figure 4.13: Decimation by a factor of three: (a) spectrum of original $x_{old}(n)$ sig- nal; (b) spectrum after downsampling by three; (c) bandwidth B' is to be retained; (d) lowpass filter's frequency response relative to bandwidth B'.[6]

For a decimation CIC filter, the moving average filter's length needs to be a whole multiple of the decimation factor, which often is 1. As a result, the only method to adjust the stopband attenuation is by adding more stages of integrator and comb sections. However, this approach simultaneously increases the attenuation in the passband.

Consider a 5th order CIC filter with a filter length of 4 and a sampling frequency of 80kHz. This setup allows for the processing of frequency components in the incoming signal up to 40kHz.

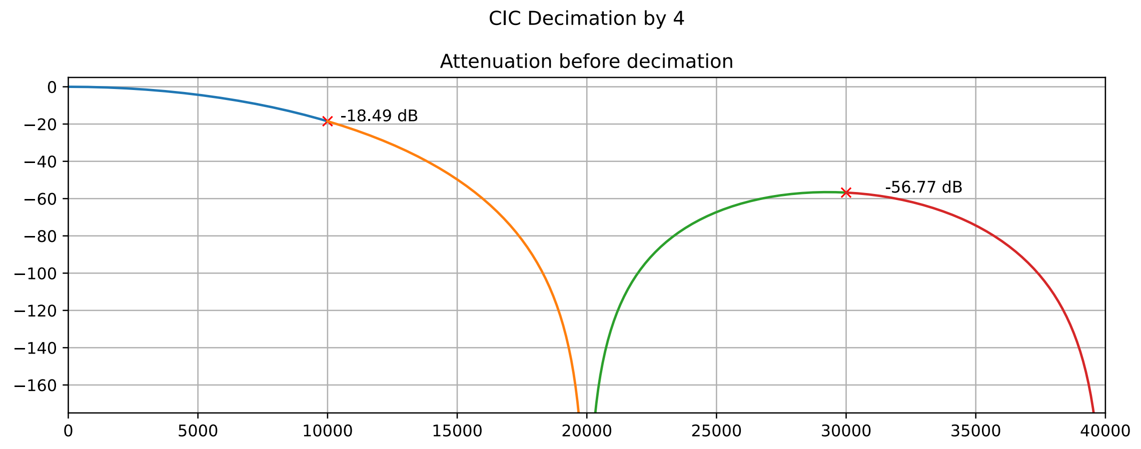


Figure 4.14: CIC Filter Attenuation before Decimation

The filter's length of 4 results in a dual-lobe structure. Typically, one might assert that this filter exhibits a stopband attenuation of 56.77dB. Nonetheless, given the intrinsic relationship between the CIC filter's decimation ratio and its filter length, the decimation ratio must also be set to 4.

Consequently, the decimating filter reduces the output sample rate to 20kHz, down from the original 80kHz, which halves the signal bandwidth, limiting it to 0 to 10kHz.

Following the decimation process, frequency components of the original signal that exceed the bandwidth of the filtered signal will manifest as aliasing within the 0 to 10kHz range.

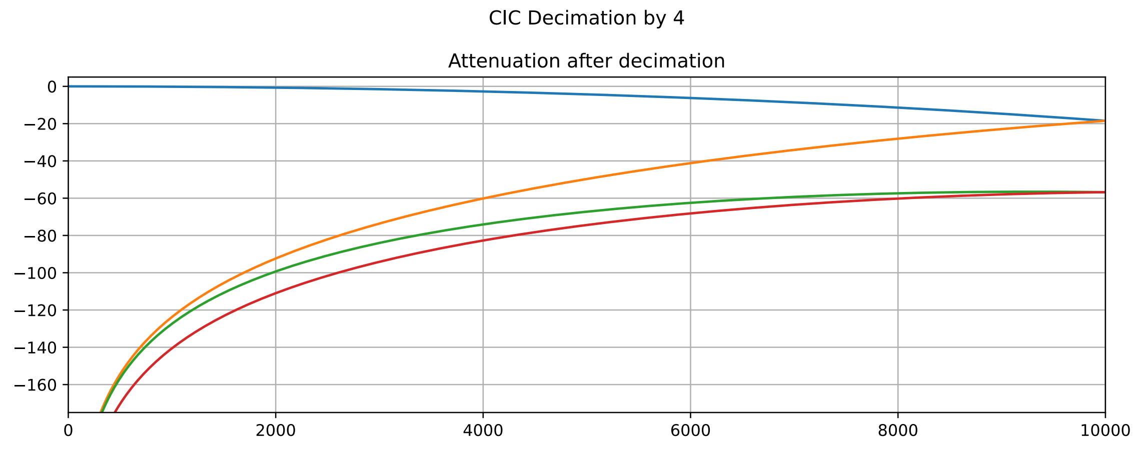


Figure 4.15: CIC Filter Attenuation after Decimation

The Frequency response of CIC filter after decimation is given in Figure 4.15. The blue curve represents the genuine signal, confined to a 10kHz bandwidth, while the other curves introduce distortion.

The anticipated stopband attenuation of 56.77dB does not hold; the actual stopband attenuation is identified at 18.49dB, where the blue curve concludes and the orange curve initiates. Furthermore, the passband attenuation is not uniform; it decreases from 0 to -18.49dB.

Employing a decimating CIC filter typically forms part of a layered decimation strategy.

Utilizing a 4x decimation CIC filter to distill a 10kHz bandwidth signal from an input with a 40kHz bandwidth is unconventional. The CIC filter merely serves as an initial phase to downsample from a higher to a more manageable intermediate rate. Subsequent stages often involve traditional FIR filters to achieve the final desired sampling rate.

In the given scenario, if the signal of interest spans from 0 to 2000Hz, the CIC filter attenuates signal components that alias into this range by over 92dB, with a minimal passband attenuation of merely 0.7dB which can be seen on Figure 4.16

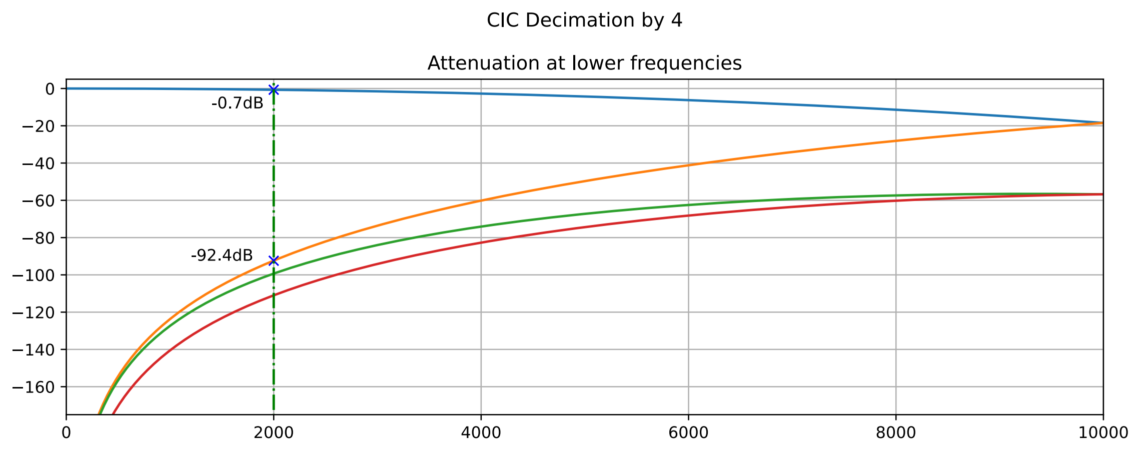


Figure 4.16: CIC Filter Attenuation at lower frequencies

Now, the only requirement is to implement one or two filters that exhibit a precise passband response from 0 to 2kHz and a stopband from approximately 3kHz to 10kHz. This approach significantly reduces computational demands compared to a filter maintaining the same passband while extending its stopband from 3kHz to 40kHz.

4.2.7 Register Word Widths and Arithmetic Overflow

CIC filters suffer from accumulator arithmetic register overflow because of the unity feedback at each integrator stage. This overflow is of no consequence as long as the following two conditions are met[7]:

- each stage is implemented with two's complement (non-saturating) arithmetic
- the range of a stage's number system is greater than or equal to the maximum value expected at the stage's output.

When two's complement fixed-point arithmetic is used, the number of bits in an Mth-order CIC decimation filter's integrator and comb registers must accommodate the filter's input signal's maximum amplitude times the filter's total gain of (NR)M. To be specific, overflow errors are avoided if the number of integrator and comb register bit widths is at least

$$registerbitwidths = x(n)bits + \lceil M log_2(NR) \rceil \tag{4.18}$$

where x(n) is the input to the CIC filter, and $\lceil k \rceil$ means if k is not an integer, round it up to the next larger integer. [7]

4.2.8 Compensation/Preconditioning FIR Filters

Addressing this subject is crucial for enhancing CIC filter attenuation efficiently, utilizing minimal resources by employing a FIR filter compensator. This will be elaborated upon in subsequent discussions, as explored in forthcoming papers.[8][4][2][1]

4.2.9 Implementation in Verilog

The Verilog implementation is taken from here which is the official github repository of the 1bit SDR.

```
module CIC
  (input wire                clk,
   input wire [7:0]   Gain,
   input wire signed [11:0]  d_in,
   output reg signed [11:0]  d_out,
   output reg          d_clk);

  parameter width = 64;
  parameter decimation_ratio = 16;

  reg signed [width-1:0] d_tmp, d_d_tmp;


  // Integrator stage registers

  reg signed [width-1:0] d1;
  reg signed [width-1:0] d2;
  reg signed [width-1:0] d3;
  reg signed [width-1:0] d4;
  reg signed [width-1:0] d5;

  // Comb stage registers

  reg signed [width-1:0] d6, d_d6;
  reg signed [width-1:0] d7, d_d7;
  reg signed [width-1:0] d8, d_d8;
  reg signed [width-1:0] d9, d_d9;
  reg signed [width-1:0] d10;

  reg signed [width-1:0] d_scaled;
  reg [15:0] count;

  reg v_comb;  // Valid signal for comb section running at output rate

  reg d_clk_tmp;


  always @(posedge clk)
    begin


      // Integrator section
      d1 <= d_in + d1;

      d2 <= d1 + d2;

      d3 <= d2 + d3;

      d4 <= d3 + d4;

      d5 <= d4 + d5;

      // Decimation

      if (count == decimation_ratio - 1)
        begin
          count <= 16'b0;
          d_tmp <= d5;
          d_clk_tmp <= 1'b1;
          v_comb <= 1'b1;
        end else if (count == decimation_ratio >> 1)
          begin
            d_clk_tmp <= 1'b0;
            count <= count + 16'd1;
            v_comb <= 1'b0;
          end else
            begin
              count <= count + 16'd1;
              v_comb <= 1'b0;
            end

    end

  always @(posedge clk)  // Comb section running at output rate
    begin
      d_clk <= d_clk_tmp;


      if (v_comb)
        begin
          // Comb section
          d_d_tmp <= d_tmp;

          d6 <= d_tmp - d_d_tmp;
          d_d6 <= d6;

          d7 <= d6 - d_d6;
          d_d7 <= d7;

          d8 <= d7 - d_d7;
          d_d8 <= d8;

          d9 <= d8 - d_d8;
          d_d9 <= d9;

          d10 <= d9 - d_d9;

          d_scaled <= d10;

          d_out <= d10 >>> (width - 12 - Gain);
        end
    end
endmodule
```

Listing 4.1: Verilog Code for NCO

4.2.10 Implementation in HLS

Bibliography

[1] G Jovanovic Dolecek. Simple wideband cic compensator. *Electronics Letters*, 45(24):1270–1272, 2009.

[2] G Jovanovic Dolecek and Sanjit K Mitra. Simple method for compensation of cic decimation filter. *Electronics Letters*, 44(19):1, 2008.

[3] Matthew P Donadio. Cic filter introduction. In *IEEE International Symposium on Communications*, pages 1–6, 2000.

[4] Chenghui Gan and Xuemei Li. Improved cic decimation filter on software defined radio. In *Proceedings of the 2021 9th International Conference on Communications and Broadband Networking*, pages 232–238, 2021.

[5] Eugene Hogenauer. An economical class of digital filters for decimation and interpolation. *IEEE transactions on acoustics, speech, and signal processing*, 29(2):155–162, 1981.

[6] Richard G Lyons. *Understanding digital signal processing, 3/E*. Pearson Education India, 1997.

[7] Rick Lyons. A beginner's guide to cascaded integrator-comb (cic) filters. *DSP related*, 2020.

[8] Richa RK Singh. Polynomial based design of cic compensation filter used in software defined radio for multirate signal processing. *International Journal of Computer Applications*, 975:8887, 2012.

[9] Steven W Smith et al. The scientist and engineer's guide to digital signal processing, 1997.

4.3 Numerically Controlled Oscillator

4.3.1 Overview

Numerically Controlled Oscillators (NCOs), also incorrectly known as Direct Digital Synthesizers (DDS), are integral components in the domain of digital signal processing and communication systems. These digital circuits generate a wide range of frequencies with high precision and stability, leveraging digital computational power to produce analog waveforms. Unlike traditional analog oscillators, which rely on the physical properties of components like resistors, capacitors, and inductors to determine frequency, NCOs achieve frequency generation through digital means, offering superior accuracy, flexibility, reliablity and control[5].
A crucial distinction to note is between NCOs and DDS. While both are integral to frequency synthesis, the defining characteristic of DDS includes presenting the output of a lookup table through a digital-to-analog converter (DAC) followed by a low-pass filter to produce an analog waveform. [3]

Implementing an NCO can be achieved through various methodologies, each with its unique advantages. The most prevalent techniques include the use of the Coordinate Rotation Digital Computer (CORDIC) algorithm and Sine-Lookup Tables.[4]
The CORDIC algorithm, renowned for its precision, operates through swift rotations in digital coordinates, employing only shift and add operations—thus, eliminating the need for multipliers. However, its precision comes at the cost of speed, as generating n-bit output requires n clock cycles. Consequently, CORDIC is most suited for applications demanding high precision at lower frequencies. On the other hand, Sine-Lookup Tables offer a straightforward and rapid solution at the expense of increased resource consumption and potential for higher spurious frequencies. This method is favored in scenarios requiring high-speed operation with ample hardware resources available.[1]

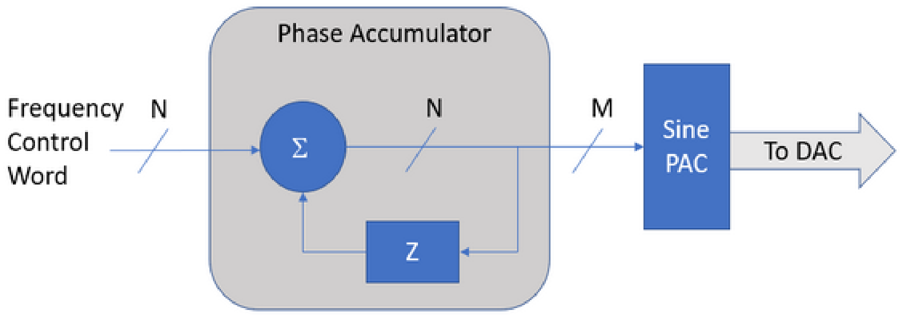


Figure 4.17: Numerically Controlled Oscillator Diagram

An NCO (Numerically Controlled Oscillator) is typically implemented utilizing a reference clock, a phase accumulator, and a lookup table (LUT) as can be seen in Figure 4.17. The phase accumulator systematically progresses, addressing each entry within the LUT, which in turn outputs the corresponding phase value over a complete 0 to 2π cycle.

4.3.2 Working principle

NCO is a digital implementation of a Voltage Controlled Oscillator(VCO) so it can only work on signals that are sampled. If a sine wave, without the loss of generality, is sampled every T_s seconds or the sampling frequency is $f_s = \frac{1}{T_s}$ the following equation holds true

Since digital implementations can only work on sampled signals, we'll need to sample this sine wave every Ts seconds. To keep our notation straight, we'll now index this sine wave output by sample number, n, rather than by time, t.

$$x[n] = sin(2\pi n \frac{f}{fs}) \tag{4.19}$$

The crux of the algorithm lies in the sine wave expression, which, when the 2π factor is omitted for simplicity, shows that the changing phase value can be represented as n multiplied by the ratio of the desired frequency f to the sampling frequency f_s, symbolized as ϕ[n]. Rewritting the equation 4.3.2 using the above declared changing phase value

$$x[n] = sin(2\ pi\phi[n]) \tag{4.20}$$

Specifically, the phase value is defined by,

$$\phi[n] = \frac{f}{f_s} n \tag{4.21}$$

The original formula used in the NCO, indicated as Equation 4.3.2, determines the phase by multiplying the sequence number n by the ratio of the frequency f to the sampling rate f_s. This method is hardware-intensive because it relies on multiplication. To improve on this, we can update the equation to add the phase step in each cycle instead of multiplying. This change is not just more efficient for hardware; it also allows the NCO to keep track of any phase changes that have built up over time, rather than resetting the phase at the start. This means phase shifts are handled correctly no matter when they occur

$$\phi[n] = \phi[n-1] + \frac{f}{f_s} \tag{4.22}$$

The variable $\phi[n]$ symbolizes the cumulative rotations made around the unit circle, ranging from 0 to 2π. Within this representation, the integer component specifies the total number of complete rotations undertaken, and the fractional part shows the current position of the signal as a proportion of a full circle.

4.3.3 Phase accumulator bits

Typically, the focus lies on the angular fraction of the signal, leading to the integer part often being disregarded. Given that this fractional part possesses a finite resolution, when it is represented with W bits, this configuration of the phase accumulator is illustrated as depicted in Figure 4.18[6].

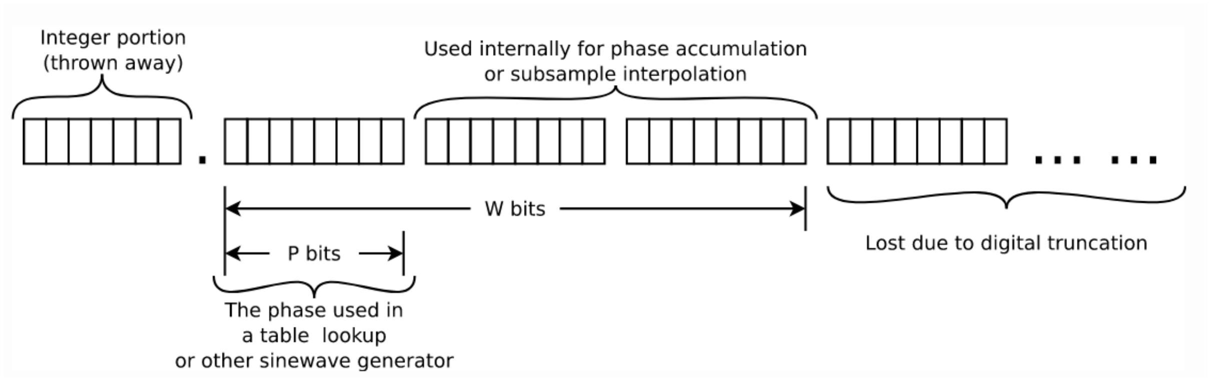

Integer portion (thrown away)
Used internally for phase accumulation or subsample interpolation
W bits
P bits
Lost due to digital truncation
The phase used in a table lookup or other sinewave generator

Figure 4.18: Phase Accumulator Bits of an NCO

The fixed-point phase representation is given by discarding the integer part of $\phi[n]$ and scaling the fractional part by 2^W, effectively representing it by W bits.

Often, the value in the phase accumulator can be quite large, varying between 16 and 64 bits. If all these bits were used to index a sine Lookup Table (LUT), it would mean having a massive LUT, which isn't practical. A more efficient method involves using only the top P bits of the accumulator to index the sine LUT. This approach maintains the desired precision while significantly reducing the LUT's size, making it a smarter choice for hardware implementation.

The core component of a Numerically Controlled Oscillator (NCO) is the phase accumulator, characterized by its bit width. The secondary element in generating sine wave outputs is the utilization of a sine Lookup Table (LUT) or, alternatively, a CORDIC algorithm; however, for the sake of simplicity, the focus here will be on the sine LUT, which is more commonly employed.

In an NCO, to achieve different output frequencies using the same phase accumulator, the variable aspect is the frequency step, calculated by the formula

$$f_d = \frac{phase_{step} \cdot fclk}{2^W} \tag{4.23}$$

where f_d is the desired frequency of the output signal from the NCO; $phase_{step}$ represents the step that increments the phase accumulator every clock cycle; $fclk$ is the frequency of the clock and W is the number of bits in the accumulator.
Given a fixed W (bit width of the accumulator), and f_{clk} (clock frequency), adjusting f_d necessitates altering the $phase_{step}$ value.

Consider a scenario where the phase step is fixed at 1 and the phase accumulator, along with the sine LUT, both have a bit width of 11 bits. For an 80MHz clock, this method would yield a frequency of approximately 39kHz. To double the frequency of the output signal, one can simply double the phase step. This principle applies similarly for frequencies 4 times, 8 times higher, and so forth, up to the maximum frequency of $\frac{fclk}{2}$ as constrained by the Nyquist theorem.

However, generating a frequency half that value, say 19.5kHz, poses a challenge as it's not feasible to halve the phase step directly. This limitation underscores the necessity for a larger phase accumulator that provides finer granularity between phase values, enabling the precise generation of lower frequencies.

By expanding the phase accumulator to 15 bits and adjusting the phase step to 16 (instead of 1), it becomes possible to maintain the sine LUT at 11 bits while enhancing precision and enabling the generation of lower frequencies. For instance, incrementing the phase accumulator by 8 rather than 16 allows for finer frequency control.[2]

This approach illustrates that achieving lower frequencies requires a larger phase accumulator, as smaller phase steps between values are crucial for accurate signal generation in an NCO. Conversely, generating higher frequencies is less demanding on the hardware. Through the strategic adjustment of the phase accumulator's bit width and phase step, an NCO can flexibly generate a wide range of frequencies with improved precision, particularly at the lower end of the spectrum. However, practical design considerations often impose limitations on available resources and budget, which introduces an inherent quantization error in the NCO's operation. The bit depth W of the phase accumulator is responsible for phase quantization error, while the address bit width P of the LUT leads to amplitude quantization error. Together, these sources of quantization error result in a phenomenon known as quantization distortion, which can impact the NCO's signal fidelity and overall performance.

To achieve the theoretical ideal in Numerically Controlled Oscillator (NCO) design, certain conditions must be met, including:

- Complete utilization of all W bits from the phase accumulator for lookup table addressing, thereby eliminating any phase truncation errors. This ensures that the phase resolution is maximized, and the phase of the output signal is as precise as the design allows, with no detail lost due to the discretization of the phase accumulator.
- An infinitely wide lookup table, which removes any amplitude quantization errors.

4.3.4 Possible improvements on the design

In the design of Numerically Controlled Oscillators (NCOs), the phase accumulator's upper bits are often utilized to index a sine wave Lookup Table (LUT), efficiently generating sine wave outputs. However, the lower bits, beyond those directly indexing the LUT, hold potential for enhancing frequency resolution and reducing output distortion.

These lower bits account for the fractional part of the phase increment, enabling the NCO to access frequencies that are not aligned with integer multiples of the LUT's base frequency. This capability is critical for achieving fine frequency granularity, allowing for the generation of sine waves at intermediate frequencies. This is achieved by occasionally causing the LUT index to advance by more than one position or even skip positions, facilitating the creation of non-integer frequency steps.

Furthermore, these additional bits can be instrumental in implementing interpolation schemes. Interpolation between LUT entries can significantly mitigate phase noise and output distortion, presenting a smoother and more accurate waveform. While this requires more complex hardware, the payoff in improved signal quality is often worth the investment.

In efforts to further minimize distortion, techniques such as dithering and multiple-entry interpolation are employed. Dithering introduces a controlled amount of random noise to the phase accumulator output, making any quantization errors less noticeable by dispersing them across the signal spectrum. Meanwhile, accessing multiple LUT entries and interpolating between them can reduce the reliance on large LUTs for high fidelity outputs, striking a balance between hardware complexity and signal purity.

4.3.5 Example signals

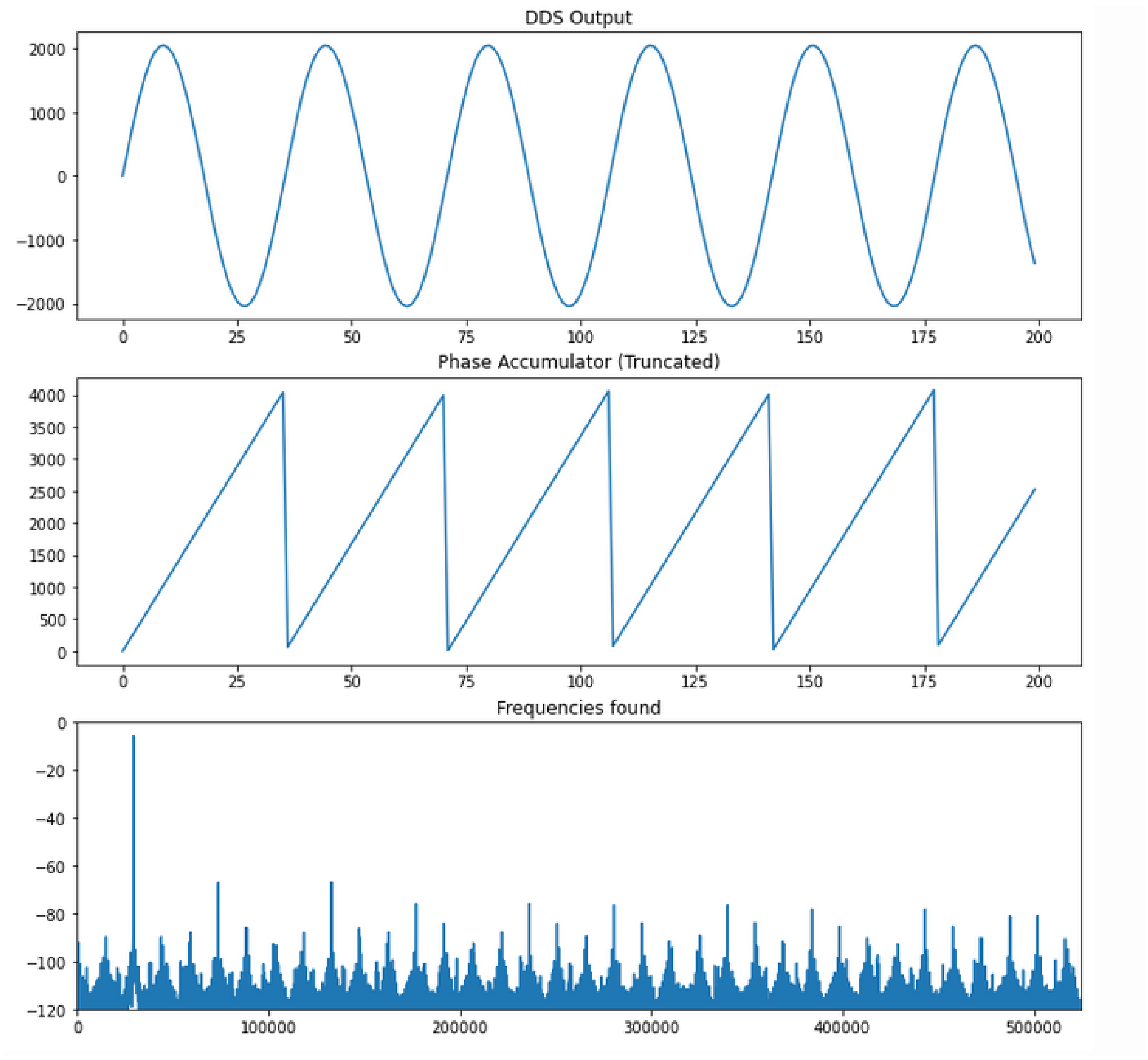


Figure 4.19: Output signal of NCO with its phase accumulator

Figure 4.19 depicts the output of the NCO along with its phase accumulator value and Fast Fourier Transform(FFT). On the first graph it can be seen that the given sinusoidal output is pretty precise. While the interesting part is happening on the second graph where it can be clearly seen that when the sinusoid reaches 2π or a full cylce the phase accumulator overflows and starts accumulating from the beginning so that it is effectivlely given with a modulo 2π. The lower graph shows that the given frequency is 440Hz which was the one that was desiren given the unique phase step.

4.3.6 Implementation in Verilog

The Verilog implementation is taken from here which is the official github repository of the 1bit SDR.

```verilog
module nco_sig
(clk,
phase_inc_carr,
phase_accum,
sin_out,
cos_out
);

input clk;
input [63:0] phase_inc_carr;

output  sin_out;
output  cos_out;
parameter IDLE_nco = 0, START_nco = 1;
reg state_nco_carr = IDLE_nco ;

output reg [63:0] phase_accum;


assign sin_out = (phase_accum[63] == 1'b1)? 1'b0 : 1'b1 ;
assign cos_out = ((phase_accum[63] ^ phase_accum[62]) == 1'b1)? 1'b0 :  1'
    b1;


always@(posedge clk) begin
  phase_accum <= phase_accum + phase_inc_carr;
end

endmodule
```

Listing 4.2: Verilog Code for NCO

4.3.7 Implementation in HLS

The HLS implementation was followed from here.

```python
#!/usr/bin/env python3

from amaranth import *
from amaranth.sim import *
import matplotlib.pyplot as plt
import numpy as np
from math import cos, pi, ceil, log2

def cos255(x, x_max_val):
    res = (cos(x*2*pi/x_max_val)+1)*255/2
    res = int(res)
    return res

def identity127(x, x_max_val):
    return 127*x/x_max_val

class NCO(Elaboratable):
    def __init__(self, fpga_clk, mem_size, func, min_input_freq, max_input_freq, max_output_val):

        # accumulator definition
        self.gain = 10
        self.acc_size = ceil(log2(fpga_clk) - log2(min_input_freq) + self.gain)
        self.acc = Signal(self.acc_size)

        # correction of the accumulator imprecision
        self.actual_min_freq = fpga_clk/pow(2, self.acc_size)
        self.corr = 1
        if self.actual_min_freq < 1:
            self.corr = round(1/self.actual_min_freq)

        # defining input and output
        self.output_size = ceil(log2(max_output_val))
        self.output = Signal(self.output_size)
        self.input = Signal(ceil(log2(max_input_freq - min_input_freq)))

        # defining the shape of the output signal in memory
        x = [i for i in range(mem_size)]
        y = []
        for i in x :
            y.append(func(i, max_output_val))

        self.memory = Memory(width = self.output_size, depth = mem_size, init = y)
        self.addr_size = ceil(log2(mem_size))

    def elaborate(self, platform):
        m = Module()

        step = Signal(self.acc_size)

        m.d.comb += [
            step.eq(self.input + self.corr),
            self.output.eq(self.memory[self.acc[-self.addr_size:]])
        ]

        m.d.sync += [
            self.acc.eq(self.acc + step)
        ]

        return m


if __name__=="__main__":
    fpga_clk = int(48e3)
    mem_size = 1023
    func = cos255
    min_input_freq = 1
    max_input_freq = 600
    output_max_val = 255

    dut = NCO(fpga_clk, mem_size, func, min_input_freq, max_input_freq,
   output_max_val)

    sim = Simulator(dut)

    def proc():
        smalltime = fpga_clk/16
        xs = [i for i in range(mem_size)]
        ys = []

        for x in xs:
            val = yield dut.memory[x]
            ys.append(val)
        fig, ax = plt.subplots()
        ax.plot(xs, ys)
        plt.show()

        yield dut.input.eq(440)
        for _ in range(smalltime):
            yield Tick()
        yield dut.input.eq(493)
        for _ in range(smalltime):
            yield Tick()
        yield dut.input.eq(524)
        for _ in range(smalltime):
            yield Tick()

    sim.add_clock(1/fpga_clk)
    sim.add_process(proc)

    with sim.write_vcd("test.vcd"):
        sim.run()
```

Listing 4.3: HLS Code for NCO

Bibliography

[1] Gai Peng Ao. The basic principle and fpga implementation of nco. In *2012 Second International Conference on Instrumentation, Measurement, Computer, Communication and Control*, pages 90–94. IEEE, 2012.

[2] fpga4fun. Direct digital synthesis 3 - phase accumulator. Accessed: 2024-03-17.

[3] Gopal D Ghiwala, Pinakin P Thaker, and Gireeja D Amin. Realization of fpga based numerically controlled oscillator. *IOSR Journal of VLSI and Signal Processing (IOSR-JVSP)*, 1(5):7–11, 2013.

[4] Sameer Kadam, Dhinesh Sasidaran, Amjad Awawdeh, Louis Johnson, and Michael Soderstrand. Comparison of various numerically controlled oscillators. In *The 2002 45th Midwest Symposium on Circuits and Systems, 2002. MWSCAS-2002.*, volume 3, pages III–III. IEEE, 2002.

[5] Nehal A Ranabhatt, Sudhir Agarwal, Raghunadh K Bhattar, and Priyesh P Gandhi. Design and implementation of numerical controlled oscillator on fpga. In *2013 Tenth International Conference on Wireless and Optical Communications Networks (WOCN)*, pages 1–4. IEEE, 2013.

[6] zipcpu. Building a numerically controlled oscillator, 2017. Accessed: 2024-03-17.

4.4 Pulse Width Modulation(PWM) Digitial to Analog Conversion

4.4.1 Overview

In this design, the digital signal processed for audio output is directly transmitted to the speaker without an intervening amplifier—this is noteworthy because it diverges from the common practice of amplifying a Pulse Width Modulated (PWM) signal with a low-power Class D amplifier. Instead, the scheme implemented here employs eight distinct PWM outputs. This a primitive version of the explained Sigma-Delta Converter. Half of these outputs are standard, non-inverted PWM signals, while the remaining four are their inverted counterparts. To connect to the speaker, which requires only a positive and a negative connection, the four non-inverted signals converge into a single line that leads to the speaker's positive terminal. Similarly, the four inverted signals merge into a single line as well, feeding into the speaker's negative terminal.

In the configuration described, signal strength is effectively doubled, a direct consequence of Kirchhoff's first law, which states that currents at a junction sum up. By combining four non-inverted PWM signals into one and four inverted PWM signals into another, the resulting currents from each set are combined when they converge at the speaker's terminals. Moreover, since the signals are phase-inverted relative to each other, they create a differential voltage input at the speaker. This differential mode of operation not only enhances the peak-to-peak voltage across the speaker, leading to a higher voltage swing but also boosts the power output due to the concurrent increase in voltage and current.

In scenarios where FPGAs or microcontrollers do not incorporate a Digital-to-Analog Converter (DAC), it's a typical practice to employ a standard PWM output combined with a low-pass filter to effectively create a DAC. This approach is also adopted in the current setup, albeit with the notable absence of an explicit low-pass filter. The reasoning behind this omission lies in the speaker's inherent characteristics; as an inductive load, it naturally functions as a low-pass filter. While it is feasible to integrate a low-pass filter into this configuration, optimizing the audio output quality would necessitate precise adjustments to the filter's parameters.

The output voltage at the low-pass filter, serving as a nominal DAC voltage, is influenced by two factors: the duty cycle of the PWM signal and its logic-high voltage. When a lowpass filter is applied in the frequency domain it eliminates the higher-frequency parts of an input signal. The equivalent effect in the time domain is a form of averaging. Hence, when a PWM signal undergoes low-pass filtering, what is essentially derived is its average value.

Consider a scenario where the PWM signal has a 50% duty cycle, meaning the signal is high for half the time and low for the other half, and operates at a logic level of 3.3 V. The nominal DAC voltage in this case would be 1.65 V.

The resolution of the DAC, when implemented through PWM, hinges on the variety of duty cycles the PWM can generate. This variability, in turn, is determined by the bit depth of the counter register responsible for producing the PWM signal.

4.4.2 Example

In this example, consider a scenario where a PWM signal with a 100kHz frequency and a 50% duty cycle is utilized. Assuming the signal operates within a voltage range of 0V to 3.3V, the expected Mean Value at the DAC output would be approximately 1.65V. The critical factor in achieving the accurate analog output is the selection of a suitable low-pass filter that effectively converts the PWM signal to its corresponding analog value.

Employing a simple RC first-order low-pass filter, characterized by a one-pole transfer function with a pole located at $\frac{1}{RC}$, and choosing a capacitance value of C = 10nF and a resistance of R = 318 **Ω**, the DAC's output response is illustrated in Figure 4.20. Additionally, the Fast Fourier Transform (FFT) analysis of the PWM signal prior to filtering is presented in Figure 4.21, while Figure 4.22 showcases the FFT results following the filtering process.

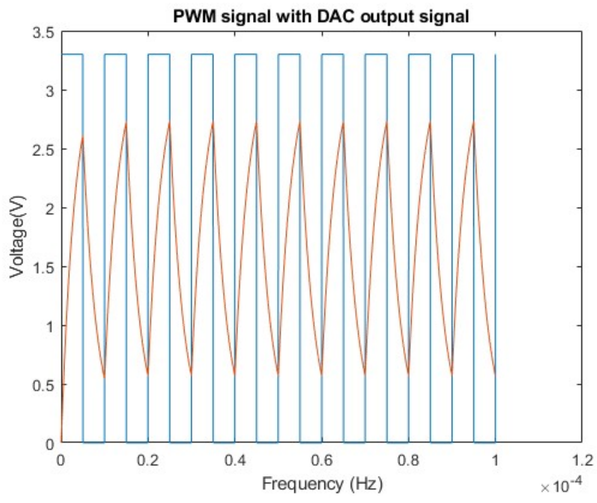


Figure 4.20: DAC Output for 1st order RC filter where R = 318 **Ω** C = 10nF

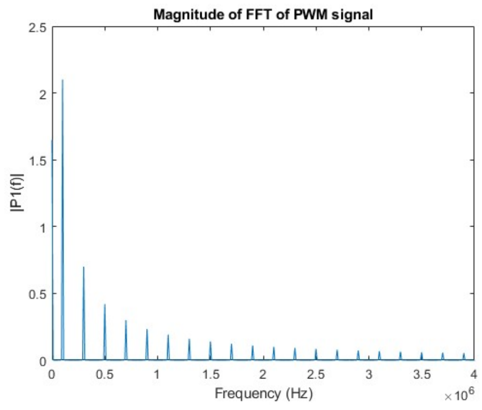


Figure 4.21: FFT of PWM signal

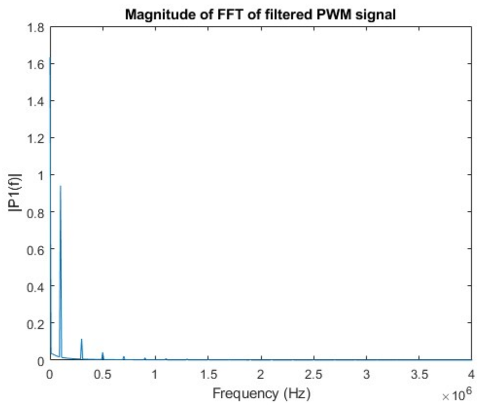


Figure 4.22: FFT of DAC signal

Observing Figure 4.20, it becomes apparent that the DAC's output does not align with the expected outcome, where the role of the RC filter is to eliminate all harmonic components, retaining only the DC part to derive the analog signal. The problem in achieving this stems from the chosen RC values being too low.

The FFT analysis, as visualized, confirms the characteristic spectral pattern of a square wave in the PWM signal. This pattern is marked by a prominent peak at the carrier frequency, followed by diminishing harmonics at odd multiples of the carrier frequency, specifically at three times, five times the carrier frequency, and so on. This spectral composition is largely unaffected by variations in the duty cycle, maintaining a consistent frequency spectrum across different duty cycles.

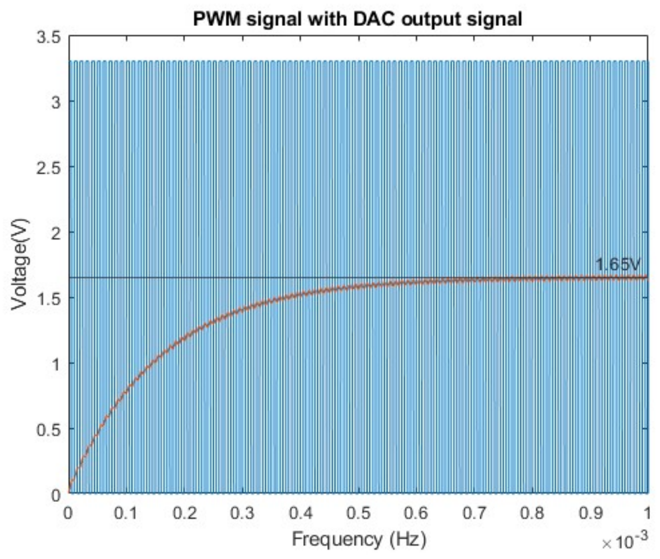


Figure 4.23: DAC Output for 1st order RC filter where R = 15924 Ω C = 10nF

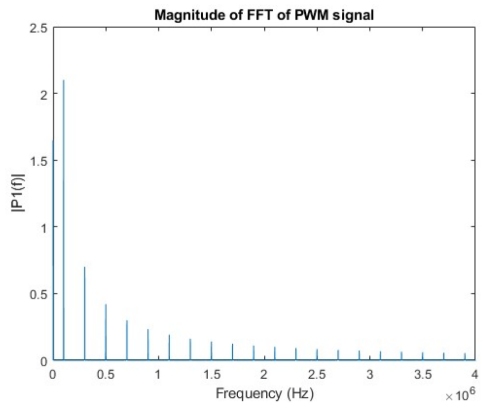


Figure 4.24: FFT of PWM signal

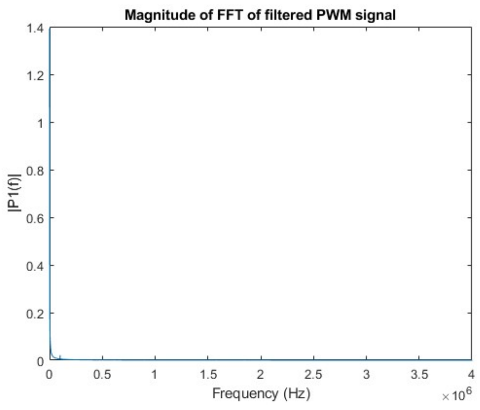


Figure 4.25: FFT of DAC signal

The improvement in ripple is significant on Figure 4.23, but the prolonged duration required for the output to stabilize at the intended DAC voltage presents a problem. This effect stems from the higher resistance in the RC filter, which, while reducing the cutoff frequency, at the same time extends the time constant. A higher resistance value diminishes the current flow towards the capacitor, resulting in a slower charging rate for the capacitor.
Looking at the Figure 4.25 the higher harmonics are properly attenuated and mostly the DC component is present, although a bit reduced at aproximately 1.4V.
Enhancements to this design could involve adopting a higher-order low-pass filter.

4.4.3 Possible improvements

Although this adjustment elevates the design's complexity, it significantly refines the output quality. Additionally, increasing the PWM frequency stands as another viable strategy for improvement, offering further optimization of the DAC's performance.