Software-Defined Radio for FPGA using High Level Synthesis

Master Thesis

Infineon IPCEI program

Tarik Hamedović (Master Student of Faculty of Electrical Engineering Sarajevo)

June 2024

Table of Contents

Introduction

- 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.

Goal

- Create a comprehensive project, encompassing both theoretical foundations and clear, well-documented code.
- Ensure the design is implemented using High-Level Synthesis (HLS) tools that are widely available and compatible with any FPGA board.
- Demonstrate the differences between Hardware Description Language (HDL) and HLS designs, highlighting their outputs, benefits, and drawbacks

1-bit SDR Project

- Input RF signal through a voltage divider to the comparator operational amplifier using LVDS
- Forms SD Modulator with the integrator with minimum analog components
- 1-bit data stream directed into the Mixer that does BPSK
- Combines data with quadrature signals (sine and cosine waves).
- Sine/cosine waves synthesized by NCO.
- Dynamic frequency adjustment via UART connection.
- CIC filter used for decimation

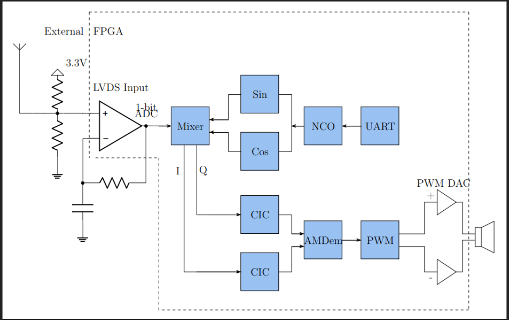


Figure 1: SDR Block Diagram

ULX3S FPGA Board

- **Board Overview**:
  - The ULX3S FPGA Board is a fully open-source development board for LATTICE ECP5 FPGAs.
- **Board Models**:
  - Available in various models differentiated by Look-Up Table (LUT) sizes: 12F, 25F, 48F, and 85F variants.
- **Tools and Programming**:
  - Boards can be synthesized, placed, and routed using Lattice Diamond or open-source tools.
- **Design and Layout**:
  - The design and layout of the board, along with its peripherals, are showcased in Figures 2 and 3 for the front and back sides, respectively.

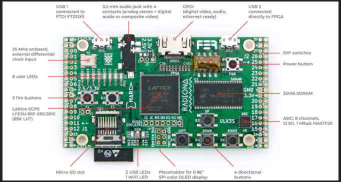

USB 1 connected to FTDI FT231XS
3.5 mm audio jack with 4 contacts (analog stereo + digital audio or composite video)
GPDI (digital video, audio, ethernet ready)
USB 2 connected directly to FPGA
25 MHz onboard, external differential clock input
8 user LEDs
2 fire buttons
Lattice ECP5 LFE5U-85F-6BG381C (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 2: Front side of ULX3S FPGA Boards

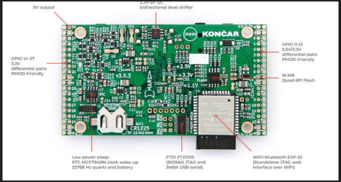

5V output
3.3V-5V I2C bidirectional level shifter
GPIO 14-27 3.3V differential pairs PMOD-friendly
GPIO 0-13 2.5V/3.3V differential pairs PMOD-friendly
16 MB Quad-SPI Flash
Low power sleep: RTC MCP7940N clock wake-up 32768 Hz quartz and battery
FTDI FT231XS (500kbit JTAG and 3Mbit USB-serial)
WiFi+bluetooth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

Figure 3: Back side of ULX3S FPGA Boards

Table of Contents

2 Software-Defined Radio

Introduction

- **Origins**:
  - Developed in the 1970s-1980s by US research groups.
  - Term ”Software Radio” coined by Joe Mitola in 1991.
- **SDR System**:
  - Converts analog signals to digital format, processes it and transmits as electromagnetic waves
- **Key Functions**:
  - Compression, power control, channel estimation, equalization, error correction. adaptive antennas and protocols.
- **FPGA Usage**:
  - High performance, low power consumption, Versatility for implementing functions.
- **Flexibility**:
  - Supports various air interfaces.
  - Adapts signal processing quickly.
  - Dynamic tool for modern telecommunications.

SDR Components

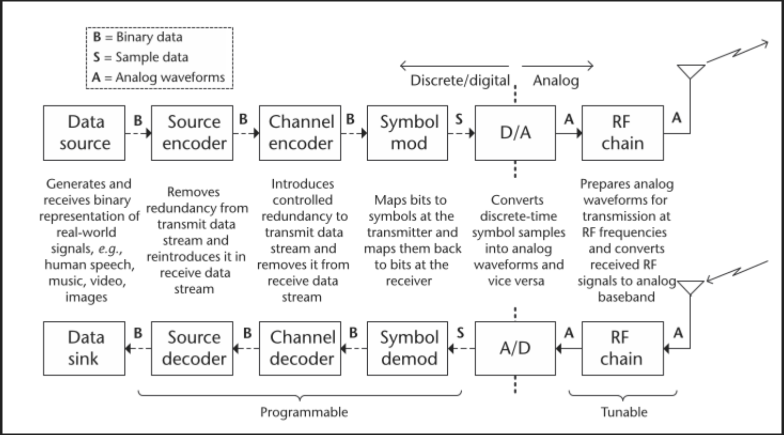

B = Binary data
S = Sample data
A = Analog waveforms
Discrete/digital
Analog
Data source
Source encoder
Channel encoder
Symbol mod
D/A
RF chain
Generates and receives binary representation of real-world signals, e.g., human speech, music, video, images
Removes redundancy from transmit data stream and 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 4: An illustration describing some of the important components that constitute a modern digital communications system.

Basic Radio Architectuure

- **Transmission Process:**
  - Baseband processing prepares data for transmission.
  - Modulation shifts signal to the designated frequency band.
  - Amplification and transmission via antenna.
  - Channel can be wireless or wired
- **Reception Process:**
  - Signal absorbed by receiving antenna.
  - Amplified, demodulated (mixed), and baseband processed.
- **SDR Architectures:**
  - No singular definitive architecture.
  - Multiple devices and combinations for SDR systems.
  - Prevalent types: Superheterodyne and Zero-IF receivers.

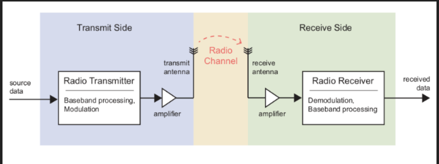


Figure 2.2: Basic Radio Architecture

Figure 5: Basic Radio Architecture

Superheterodyne Reciever Architecture

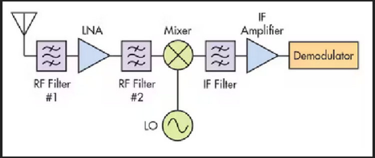


Figure 6: One Stage Superheterodyne Reciever Architecture

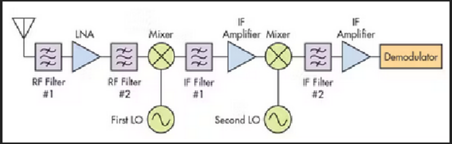


Figure 7: Two Stage Superheterodyne Reciever Architecture

Direct-RF Reciever Architecture

- The signal’s conversion to baseband is achieved through a singular frequency mixing stage, where the local oscillator is precisely tuned to the signal’s frequency band.
- Translates the received signal directly to baseband, processing it in both I and Q signals
- Reduces the need for complex and expensive RF/IF filtering, as all necessary filtering is performed at the baseband level.
- Filters are simpler, more cost-effective, and easier to design

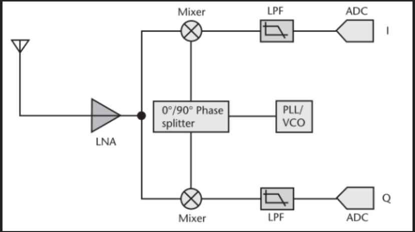


Figure 8: Direct-RF Reciever Architecture

Table of Contents

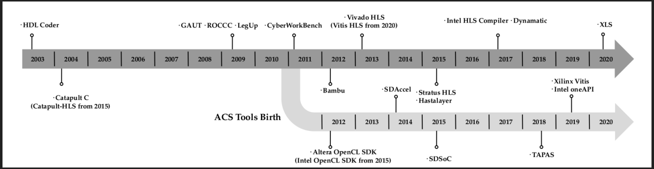
3 High Level Synthesis

FPGA and HLS overview

- **FPGAs**
  - Versatile integrated circuits reconfigurable after manufacturing.
  - Advantage over fixed-architecture processors (CPUs, GPUs).
  - Suitable for ASIC prototyping, hardware acceleration, and more.
  - Evolution: increased complexity, multi-die devices, system-level interconnects.
  - Investment from academia and industry.
  - Challenges: complex design process, reliance on HDLs (Verilog, VHDL).
- **High-Level Synthesis (HLS)**:
  - Simplifies FPGA design through higher abstraction software programs.
  - Automates translation to Register Transfer Level (RTL) designs.
  - Optimizes interface and memory elements.
  - Reduces time-to-market.
  - Balances flexibility of general-purpose processors and efficiency of ASICs.

History

- **History:**
  - Conceptual beginnings in the 1970s and 1980s.
  - Practical industrial applications recognized at the turn of the century.
- **Quality of Results (QoR):**
  - Significant improvements with each generation.
  - Studies show varying outcomes, debate on closing the QoR gap.
- **Research and Evolution:**
  - Documented progress from early concepts to sophisticated C/C++ based tools.
  - Speculation on future HLS development.
- **Timeline of HLS and ACS Tools:**
  - Visual representation of milestones and key developments in Figure below
  - Focus on tools based on C/C++ languages.



Survey Analysis

- Surveys reflect the evolving landscape of HLS technology, with tools undergoing significant changes, replacements, or improvements over time, highlighting the importance of staying updated with the latest advancements.
- HLS tools like AutoESL's AutoPilot and Xilinx's system-level platforms bridge the gap between high-level language specifications and FPGA implementations, supporting various programming models and integrating FPGA-specific optimizations.
- Evaluation of HLS tools based on execution time, resource utilization, and optimization possibilities, emphasizing the significance of operation chaining, bitwidth analysis, memory space allocation, loop optimizations, and if-conversion.
- HLS tools generally produce lower Quality of Results (QoR) compared to manual RTL designs but significantly shorten development time, offering more than fourfold productivity improvements, particularly in large-scale projects.

Survey Analysis

- Success of HLS in application areas like deep learning, video transcoding, graph processing, and genome sequencing, demonstrating its practical deployment and impact on modern computing challenges.
- Ongoing challenges include enhancing simulation, verification, domain-specific integrations, and addressing complex pragmas for high performance, with future research focusing on refining HLS effectiveness and efficiency.
- Emerging trend of using Python in HLS and hardware-software co-design, leveraging its rich ecosystem and powerful libraries for scientific computation and AI, with approaches like direct execution of Python bytecode and developing a Python to HDL transpiler.
- HLS Tools: Xilinx HLS, Mathworks HDL Coder, Siemens Catapult, Amaranth HDL, LiteX, MyHDL, PipelineC...

HLS Design Flow

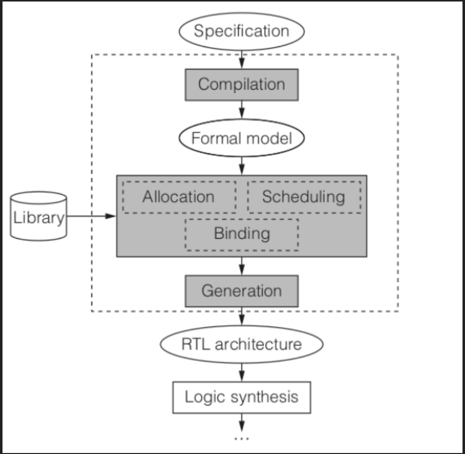


Figure 10: HLS Design Flow

Amaranth HLS

- Amaranth (formerly known as nMigen) is a Python-based hardware description language (HDL) designed to facilitate the design of digital hardware, particularly for FPGA and ASIC development. It provides a modern and flexible approach to hardware design, leveraging the power and readability of Python. Here’s a detailed explanation of Amaranth HLS:
- Supports multiple FPGA platforms and can be extended to support new platforms.
- Offers built-in simulation tools for verifying designs before synthesis.
- Designs in Amaranth are highly modular and composable, promoting reusable and maintainable code.

Example Code

```
class Blinky(Elaboratable):

    def __init__(self, num_leds=8, clock_divider=21):
        self.num_leds = num_leds
        self.clock_divider = clock_divider
        self.leds = Signal(num_leds)

    def elaborate(self, platform):
        # Create a new Amaranth module
        m = Module()

        # This is a local signal, which will not be accessible from outside.
        count = Signal(self.clock_divider)

        # If the platform is not defined then it is simulation
        if platform is not None:
            leds = [platform.request("led", i) for i in range(self.num_leds)]
            m.d.comb += [led.o.eq(self.leds[i]) for i, led in enumerate(leds)]

        # In the sync domain all logic is clocked at the positive edge of
        # the implicit clk signal.
        m.d.sync += count.eq(count + 1)
        with m.If(count == (2**self.clock_divider - 1)):
            m.d.sync += [
                self.leds.eq(~self.leds),
                count.eq(0)
            ]

        return m
```

Simulation Code

```python
def testbench():
    for _ in range(runtime):
        yield Tick()

# Instantiate the Blinky module
dut = Blinky(num_leds, clock_divider)

# Create a simulator
sim = Simulator(dut)
sim.add_clock(1e-6 / clock_frequency)
sim.add_process(testbench)
with sim.write_vcd(f"{top_name}.vcd", f"{top_name}.gtkw", traces=[dut.leds]):
    sim.run()

# Open GTKWave with the generated VCD file if --gtkwave is set
if args.gtkwave:
    subprocess.run(["gtkwave", f"{top_name}.vcd"])
```

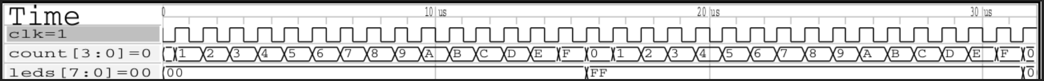

Time
clk=1
count[3:0]=0
leds[7:0]=00
0
10 us
20 us
30 us
1 2 3 4 5 6 7 8 9 A B C D E F 0 1 2 3 4 5 6 7 8 9 A B C D E F 0
00
FF
0

Figure 11: Simulation Output

Verilog Code Example

```verilog
/* Generated by Yosys 0.42+10 (git sha1 ef9045882, g++ 11.4.0-1ubuntu1~22.04 -fPIC -Os) */

(* top =  1  *)
(* src = "/home/user/SDR-HLS/HLSImplementation/Examples/1.Blinky/Blinky.py:24" *)
(* generator = "Amaranth" *)
module top(rst, leds, clk);
  reg \$auto$verilog_backend.cc:2352:dump_module$1  = 0;
  wire [21:0] \$1 ;
  wire \$2 ;
  wire [7:0] \$3 ;
  reg [20:0] \$4 ;
  reg [7:0] \$5 ;
  (* src = "/home/user/FPGA/amaranth/amaranth/hdl/_ir.py:283" *)
  input clk;
  wire clk;
  (* src = "/home/user/SDR-HLS/HLSImplementation/Examples/1.Blinky/Blinky.py:27" *)
  reg [20:0] count = 21'h000000;
  (* src = "/home/user/SDR-HLS/HLSImplementation/Examples/1.Blinky/Blinky.py:20" *)
  output [7:0] leds;
  reg [7:0] leds = 8'h00;
  (* src = "/home/user/FPGA/amaranth/amaranth/hdl/_ir.py:283" *)
  input rst;
  wire rst;
  (* src = "/home/user/SDR-HLS/HLSImplementation/Examples/1.Blinky/Blinky.py:27" *)
  always @(posedge clk)
    count <= \$4 ;
  (* src = "/home/user/SDR-HLS/HLSImplementation/Examples/1.Blinky/Blinky.py:20" *)
```

Verilog Code Example

3 High Level Synthesis

```
always @(posedge clk)
   leds <= \$5 ;
  assign \$1  = count + (* src = "/home/user/SDR-HLS/HLSImplementation/Examples/1.Blinky/Blinky.py:36"
       *) 1'h1;
  assign \$2  = count == (* src = "/home/user/SDR-HLS/HLSImplementation/Examples/1.Blinky/Blinky.py:37"
       *) 21'h1fffff;
  assign \$3  = ~ (* src = "/home/user/SDR-HLS/HLSImplementation/Examples/1.Blinky/Blinky.py:39" *) leds
       ;
  always @* begin
    if (\$auto$verilog_backend.cc:2352:dump_module$1 ) begin end
    \$5  = leds;
    if (\$2 ) begin
      \$5  = \$3 ;
    end
    if (rst) begin
      \$5  = 8'h00;
    end
  end
  always @* begin
    if (\$auto$verilog_backend.cc:2352:dump_module$1 ) begin end
    \$4  = \$1 [20:0];
    if (\$2 ) begin
      \$4  = 21'h000000;
    end
    if (rst) begin
      \$4  = 21'h000000;
    end
  end
endmodule
```

Table of Contents

4 Simulation

cocoTB

CocoTB (*Coroutine-based Co-simulation TestBench*) is a framework for verifying digital designs. It leverages Python as the primary language to create testbenches, allowing for a highly flexible and powerful verification environment.

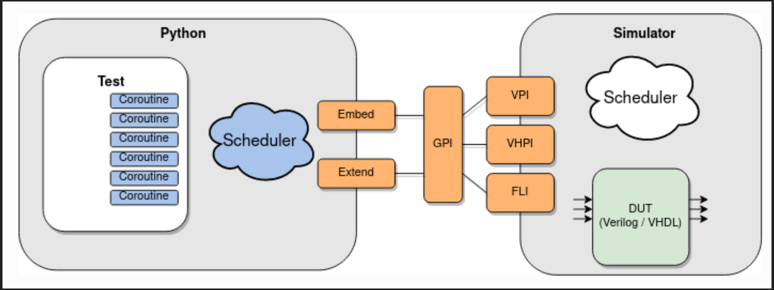


Figure 12: cocoTB Working principle

Table of Contents

5 Digital Signal Processing Modules

Sigma-Delta Analog to Digital Converters

- Class of high- precision and high-resolution conversion technologies
- Method that integrates oversampling, noise shaping, and digital filtering
- Predominantly digital implementation
- To gain one additional bit (increase the SNR by 6 dB), oversampling by a factor-of-4 is necessary,

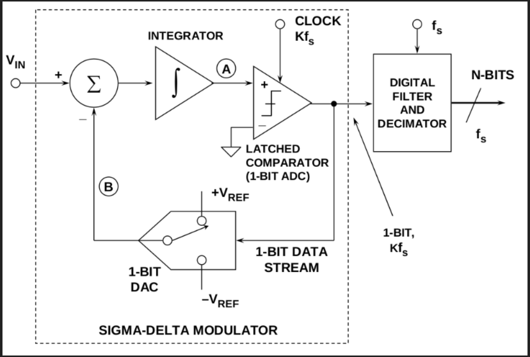


Figure 13: Sigma-Delta ADC

Sigma-Delta Analog to Digital Converters

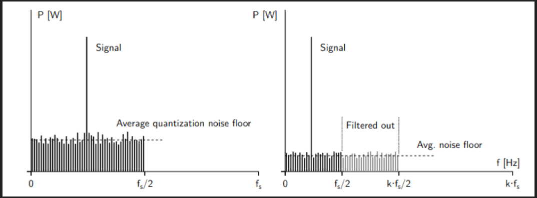


Figure 14: Quantization Noise

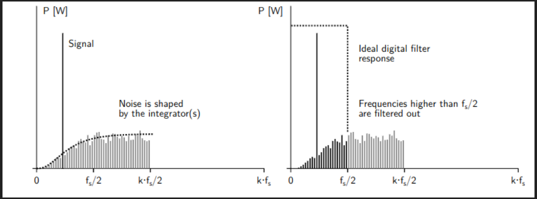


Figure 15: Noise Shaping

Sigma-Delta Analog to Digital Converters

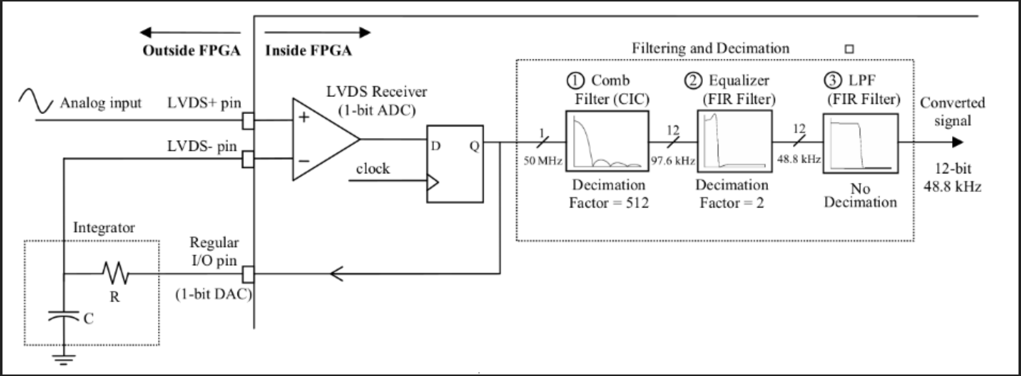


Figure 16: Sigma Delta ADCs on FPGA

Sigma-Delta Analog to Digital Converters

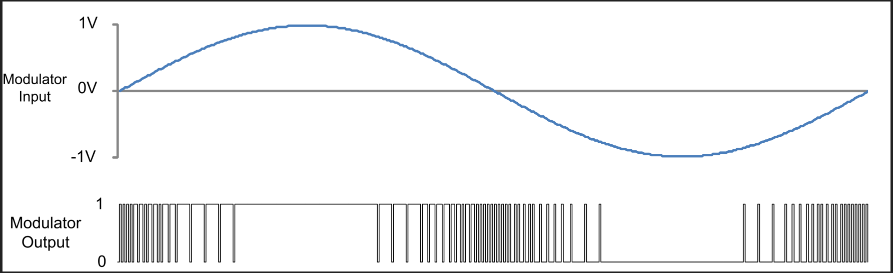


Figure 17: Sigma Delta ADC Input and Output

Cascaded Integrator-Comb Filters

- Computationally-efficient implementation of narrowband lowpass filters
- Does not use any multipliers
- 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)

Cascaded Integrator-Comb Filters

- Two ajustable parameters: the quantity of samples averaged together and the count of filters linked in series.
- The standard MA must perform D−1 additions per output sample. The RRS filter has the performs one addition and one subtraction per output sample, regardless of the delay length D.

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

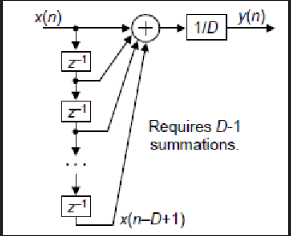


Figure 18: D-point Moving Average filter

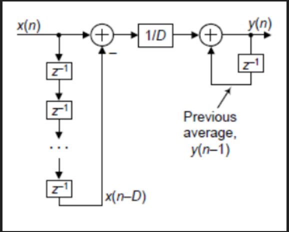


Figure 19: D-point Recursive Running Sum filter

Cascaded Integrator-Comb Filters

- Integrator Section

$$y[n] = y[n-1] + x[n] \qquad (2)$$

$$H_I(z) = \frac{1}{1 - z^-1} \qquad (3)$$

- Comb Section

$$y[n] = x[n] - x[n-D] \qquad (4)$$

$$H_C(z) = 1 - z^D \qquad (5)$$

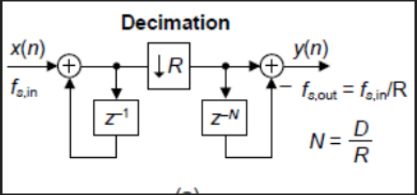


Figure 20: 1st order CIC Filter

- CIC Filter

$$y(n) = x(n) - x(n-D) + y(n-1) \qquad (6)$$

$$H_{CIC}(z) = \frac{Y(z)}{X(z)} = \frac{1 - z^{-D}}{1 - z^{-1}} \qquad (7)$$

Cascaded Integrator-Comb Filters

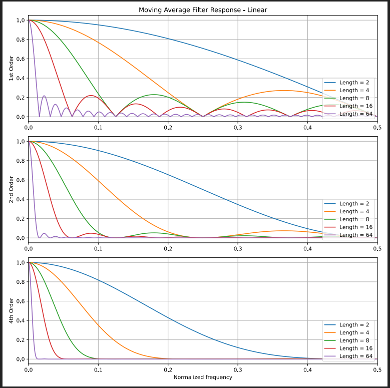


Figure 21: Plot of Frequency Response of CIC filter for different Lengths and Orders

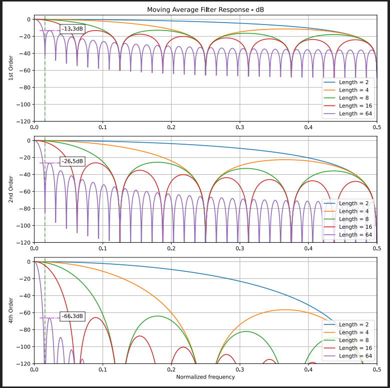


Figure 22: Plot of Frequency Response of CIC filter for different Lengths and Order with Log Y Axis

Cascaded Integrator-Comb Filters

- *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.*

Numerically Controlled Oscillator

- Digital circuits generating precise and stable frequencies
- The most prevalent techniques include the use of the Coordinate Rotation Digital Computer (CORDIC) algorithm and Sine-Lookup Tables

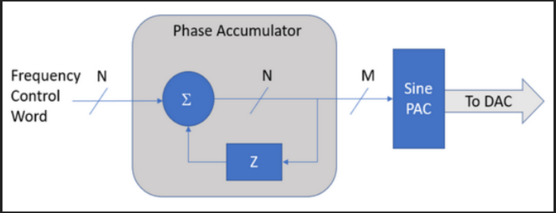


Figure 23: Numerically Controlled Oscillator Diagram

Numerically COntrolled Oscillator

$$x[n] = sin(2\pi n\frac{f}{fs}) \tag{8}$$

$$x[n] = sin(2\ pi\phi[n]) \tag{9}$$

$$\phi[n] = \frac{f}{f_s}n \tag{10}$$

$$\phi[n] = \phi[n-1] + \frac{f}{f_s} \tag{11}$$

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.

Numerically Controlled Oscillator

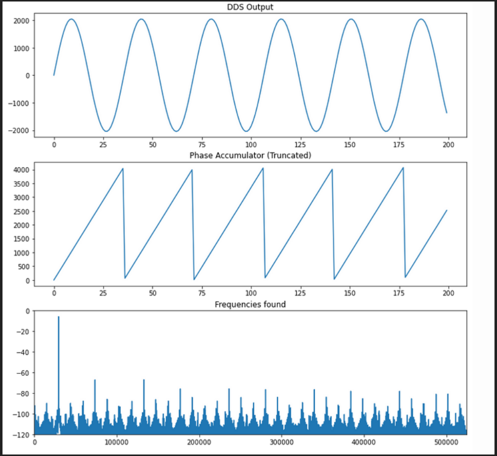


Figure 24: Output signal of NCO with its phase accumulator

Pulse Width Modulated Digital to Analog Conversion

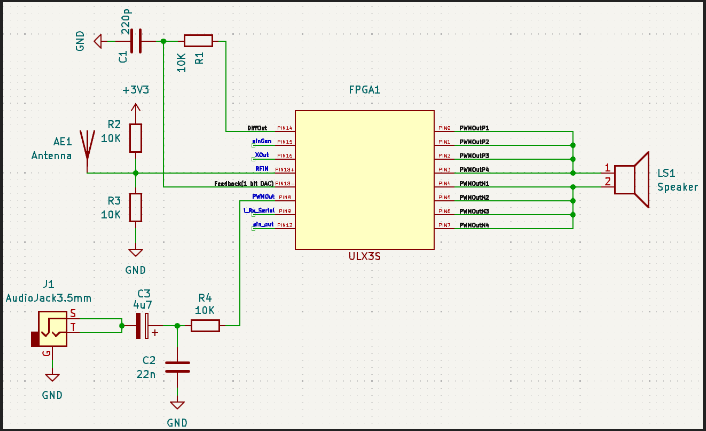


Figure 25: ULX3S Schematic for 1-bit SDR

Pulse Width Modulated Digital to Analog Conversion

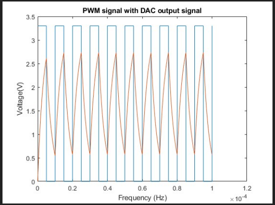


Figure 26: DAC Output for 1st order RC filter where R = 318 Ω, C = 10nF

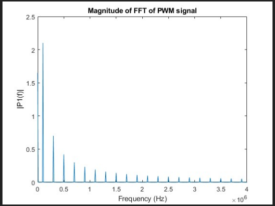


Figure 27: FFT of PWM signal

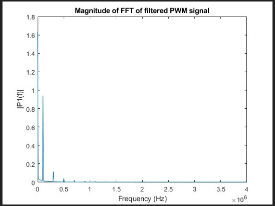


Figure 28: FFT of DAC signal

Pulse Width Modulated Digital to Analog Conversion

5 Digital Signal Processing Modules

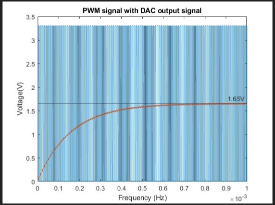


Figure 29: DAC Output for 1st order RC filter where R = 15924 Ω, C = 10nF

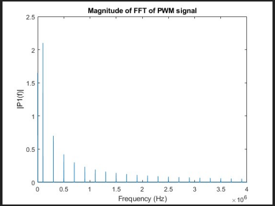


Figure 30: FFT of PWM signal

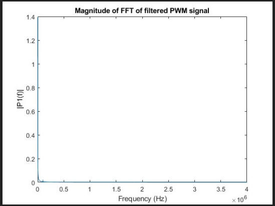


Figure 31: FFT of DAC signal

Table of Contents

6 Conclusion

Current Progress and Next Steps

- **Thesis Status**: The thesis is still in progress, but the foundational building blocks are established.
- **Verilog Design Testing**:
  - Conduct thorough tests on the Verilog components of the design.
  - Ensure flawless operation by debugging and documenting each DSP block in detail.
- **HLS Design Testing**:
  - Perform equivalent tests on the HLS components of the design.
  - Document the outputs, benefits, and drawbacks of the HLS approach.

Software-Defined Radio for FPGA using High Level Synthesis

Thank you