POLYTECHNIC UNIVERSITY OF TURIN

Master's Degree in Microelectronics

Master's Degree Thesis

TITLE

Supervisors

Prof. Marina MONDIN

Prof. Guido MASERA

Prof. Fereydoun DANESHGARAN

Candidate

Taus ENRICO

October 2024

Summary

The work presented in this thesis focuses on the design of a 24 GHz Frequency Modulated Continuous Wave (FMCW) radar system for precise target detection and localization in short to medium range distances. Realized with the ADF5901, ADF5904, and ADF4159 chipset, the proposed system takes advantage of the merits of FMCW radar, such as robustness against challenging environmental conditions and the ability to measure target distance and velocity. The modification, a delay line between the RF output and transmit antenna, finding its place in the radar architecture introduces an artificial delay, which should create the impression of a longer distance towards the target, proportionally raising the beat frequency, which is necessary for a better resolution of the FFT-based demodulation process. This approach greatly improves the system's range precision while still allowing the computational complexity to remain reasonable. The careful placement of the delay line at the transmitter side avoids degradation in the SNR, which is often experienced whenever such delays are applied at the receiving end. The thesis also discusses strengths and weaknesses of operating at the 24 GHz frequency. While it offers compact design with adequate range resolution, this frequency range represents a good balance between the lower-frequency radars with poor spatial resolution and the higher-frequency systems that are more complicated and expensive. A very effective compromise, the 24 GHz radar remains for applications involving short range where size and cost sensitivity are important.

Most of the work involves designing and implementing the receiver architecture. The system realizes signal conditioning, filtering, decimation, and FFT processing in SystemVerilog. This focuses on architectural parallelism and pipelining for peak performance of the design. Filtering is performed using a third-order Chebyshev filter for effective noise reduction with minimum signal compromise in integrity for demodulation.

The FPGA implementation is selected for flexibility and parallel processing capabilities, which are critical in handling high-speed streams of data emanating from the radar. The architecture incorporates a decimation strategy in its design to reduce computational load without sacrificing any aspects of signal fidelity. There is also a custom-designed square root module for efficiently computing the envelope of the

signal.
These involved extensive simulations and real experiments to validate the system performance, which showed targets detectable within a few centimeters of resolution or even less. Results that meet the designed performance, in terms of precision, speed in processing, and efficiency as a whole, are demonstrated. This makes the radar suitable for applications in areas such as automotive safety and industrial automation.
This thesis, therefore, proposes a 24 GHz FMCW radar system, effectively incorporating improvements in range precision using delay line integration and efficient signal processing based on FPGA hardware. The system is able to solve such challenges as the balance between precision and complexity on one side, and cost on the other, while being easily adaptable to a wide variety of real-world sensing applications.

Acknowledgements

ACKNOWLEDGMENTS

I would especially like to thank the Erasmus project for the necessary financial support that had enabled me to engage in this research.

I am grateful to my colleagues and friends, especially Gabriele, who have never stopped encouraging me and giving me much-needed feedback which helped me get better both with work and as a person.

I am particularly grateful to my supervisors, Prof. Marina Mondin, Prof. Guido Masera and Prof. Fereydoun DaneshGaran for expert guidance, constructive criticism, and unfailing support. Their mentorship has been highly instrumental in my passing through the complexities of the research at hand, and I feel really privileged to have had them as my advisors.

I would like to express my heartiest gratitude to my family: Mom and Dad, Barbara and Oscar, and my brother, Urio, for love, patience, and understanding throughout this process.

Last but not least, my greatest gratitude goes to my girlfriend Zoe. Without her, it would have been impossible to achieve this result in the same way. Her love and support have been what gave me the strength to endure every obstacle on my way. No words are enough to compensate for what you have done for me over these years, but I hope you are as happy as I am with our success.

I am indeed grateful to everyone who, in any way possible, assisted in putting this thesis together.

Table of Contents

List of Tables

List of Figures

Acronyms

AI
: artificial intelligence

FMCW
: frequency modulated continous wave

FFT
: fast fourier transform

SV
: System Verilog

FPGA
: field programmable gate array

FSM
: finite state machine

FT
: Fourier transform

Chapter 1

Introduction and motivation

1.1 Importance of Localization

High-accuracy, low-cost localization has become a staple for many applications, from AR to robotics to navigation. Many of these require robust methods for finding the exact position of an object in any given environment, under even adverse conditions such as heavy clutter or low visibility. In the last few years, AR has witnessed great usage in various fields right from gaming to training personnel in different industrial sectors. For example, a precise location in a jet engine needs excellent localization capabilities while training a maintenance operator with AR. It is ability needed for enhanced precision in execution and minimizing chance of going wrong, hence safety, in the execution of such type of maintenance tasks.

Similarly, in robotics, precise localization is a prerequisite for performing independent tasks like autonomous navigation and manipulation of dynamic environments [1]. Precise self-localization will definitely allow robots to find their way through challenging settings, avoid collisions, and manipulate objects with high precision. That potential of this competency is particularly needed within applications like warehouse automation, where the robots should move goods efficiently without any collision, and within healthcare, where the robotic assistants have to safely navigate through hospital corridors.

The application of this precision is highly valuable to both outdoor and indoor[2] navigation systems, as it provides clear guidance while enhancing the user experience. Indoor navigation can be helpful for assisting users in big buildings such as airports, shopping malls, and hospitals. This includes GPS-type outdoor navigation systems, applicable within vehicles and for guiding pedestrians to desired destinations. Immediately after this process of localization becomes more accurate, such systems will be remarkably usable and very reliable.

1.2 Trilateration

Indeed, one of the most common ways of localization is trilateration, which is based on obtaining range information from at least three radiofrequency receivers. Trilateration basically is an algorithm for finding where something is based upon its distance from known points. The basic system of equations for performing trilateration in 2D space is achieved through the distance formula:

$$(x - x_1)^2 + (y - y_1)^2 = d_1^2 \tag{1.1}$$
$$(x - x_2)^2 + (y - y_2)^2 = d_2^2 \tag{1.2}$$
$$(x - x_3)^2 + (y - y_3)^2 = d_3^2 \tag{1.3}$$

where (x, y) is the position to be determined, (x_1, y_1), (x_2, y_2) and (x_3, y_3) are the positions of the receivers, and d_1, d_2 and d_3 are the measured distances from the unknown position to each receiver. This can be further improved using an active target that produces a distinct return signal and thus allows for unambiguous identification of the target[3].
These become considerably more complex in three-dimensional space: three-dimensional coordinates and distances. The principle is the same, though: through the solution of a system of equations derived from the measured distances, the exact position of the target is determined. Trilateration has many practical applications, starting with GPS technology, whereby the satellites are the known points and the receiver on the ground calculates its position based on its distance from these satellites.

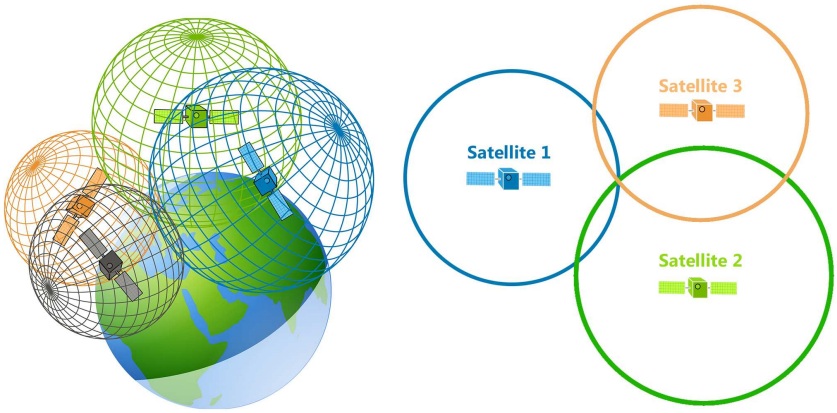


Figure 1.1: Example of trilateration with 3 or more satellites

This technique remains pertinent, as recent publications continue to refine and advance it [4].

1.3 FMCW RADAR

For deriving the range information, an FMCW RADAR is quite suitable. The FMCW RADAR sends out a signal that contains a frequency chirp, a signal whose frequency linearly increases over time. This signal is reflected from a target back to the RADAR with a time delay proportional to the distance of the target. The RADAR produces a beat frequency by mixing the transmitted signal with the received signal, this frequency is directly proportional to the distance between the RADAR and the target. The beat frequency can be defined as:

$$f_b = 2 \times \frac{R}{c} \times B \times T_c \tag{1.4}$$

where R is the range to the target, c denotes the light's speed, B is the bandwidth of the chirp and finally, T_c denotes the duration of the chirp. This linear dependence of the beat frequency on distance allows us to calculate the range to the target, which makes FMCW RADARs truly effective for localization tasks in almost all possible applications.
Particularly, FMCW RADAR has a very specific advantage with respect to areas that may render unusable optical systems, including areas that deal with bad lighting or obscurants such as fog, smoke, or dust. Whereas optical sensors rely on light, RADAR systems use radio waves, which can pass easily through obscurations and return consistent data about localization. In this respect, the application of FMCW RADAR in autonomous vehicles would be very effective, where these sensors would complement each other to maintain safe navigation in all weather conditions.

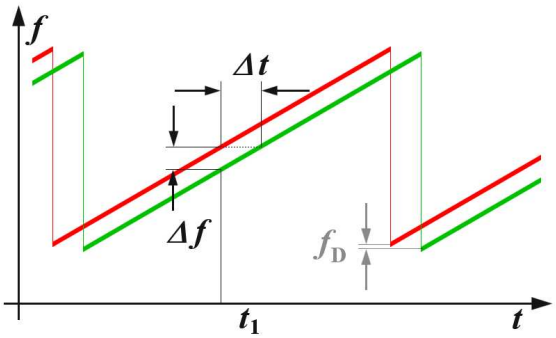


Figure 1.2: Frequency-time diagram of the relationship between a sent signal (in red) and the received one (in green) for a FMCW radar

1.4 Hardware Platforms and Receiver Architectures

The implementation of FMCW RADAR systems can be optimized by choosing the right hardware platform. Field Programmable Gate Arrays (FPGAs) provide several advantages compared to microcontrollers and Application-Specific Integrated Circuits (ASICs). FPGAs are very flexible, for instance, parallel processing is possible which gives a large speedup in computations compared to the sequential processing of microcontrollers. This issue becomes important in real-time applications such as in RADAR signal processing, where delays should be minimized. Besides, FPGAs can be reprogrammed as required to provide a level of flexibility not offered by ASICs. Whereas ASICs may offer superior performance, with lower power consumption for particular tasks, during manufacture they become set in their function and therefore are not the platform of choice when rapid prototyping is required . Yet another important decision that must be made when designing the RADAR system is whether a fully analog, fully digital, or mixed analog/digital receiver architecture is to be used. Fully analog receiver, while potentially simpler and more power-efficient, suffers from its lack of flexibility and vulnerability to component variations and noise. A fully digital receiver, on the other hand, can offer much higher flexibility and accuracy but can also be much more complex and power-hungry, in particular for high-frequency signals whose processing calls for fast ADCs and intensive digital signal processing. An architecture for the mixed analog and digital receiver combines the strengths of both approaches. In particular, the analog front-end can carry out preliminary signal conditioning, filtering and amplification. Sparing the digital components from working at too high frequency. At the same time, the digital back-end can execute more sophisticated processing tasks, like demodulation and frequency analysis, very precisely with great flexibility. Which hardware platform and which receiver architecture are optimal depends upon the particular application. A mixed analog-digital architecture, for example, is likely to be preferred for portable or battery-operated devices where power consumption is a limiting factor.

Figure 1.3: A Spartan®-7 SP701 FPGA

1.5 Thesis Focus

Within this general framework, this thesis focuses on the study, simulation, and implementation of a demodulator capable of recovering the bandpass signal generated in the downconverted RADAR signal by the active target in response to the RADAR chirp signal. The aim of the work was implementing, on an FPGA, an algorithm for data extraction from an FMCW RADAR signal in order to position targets with a high grade of precision. Looking from the perspective of target range, it can be calculated by using the demodulated received signal. The estimated beat frequency is thus obtained, and the target can then be precisely localized using the trilateration method. Further chapters will develop in detail the theory of this implementation, show simulation results, and explain how this system was implemented on an FPGA, where great advantages are enlisted for such an approach of a challenging application.

Chapter 2

The RADAR board

2.1 Introduction

The EV-RADAR-MMIC2 evaluation board chosen for this work is designed for frequency-modulated continuous-wave radar at a center frequency of 24 GHz. This evaluation board integrates an ADF5901 chip for signal generation, an ADF5904 chip for amplifying the signals, and an ADF4159 chip for further processing the signal. Every component on this board has its specific function in enabling radar functionality: generating, transmitting, and receiving high-frequency radar signals in one coherent framework. The board supports several IO connections for data communication, power supply, and external control, which makes it highly versatile in terms of experiments and development.

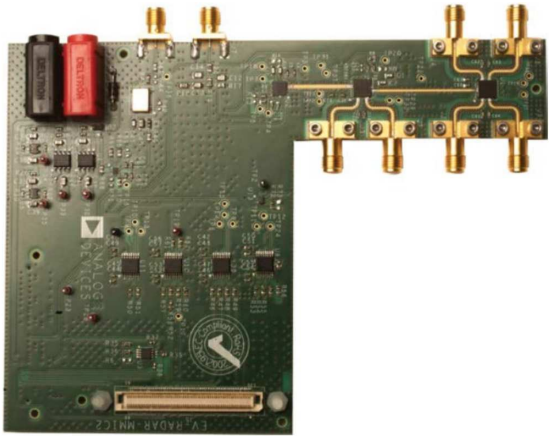


Figure 2.1: EV-RADAR-MMIC2 evaluation board

2.2 Key Components

2.2.1 ADF5901 - 24 GHz Transmitter (TX)

The AdF5901 is monolithic microwave integrated circuit able to generate high frequency signal upto 24GHz, it serves the TX functionality of the RADAR system. Its main components are a phased-locked loop, a voltage controlled oscillator and some power amplifiers.

- **PLL and VCO:** These components are responsible for generating the 24 GHz signal, which can be modulated as needed for FMCW radar operations. The PLL ensures that the signal is stable and locked to a reference frequency, while the VCO allows frequency modulation;
- **Power Amplifiers:** These amplifiers boost the generated RF signal to a level suitable for transmission, ensuring that the radar can cover the desired range;

TX IOs:

- **RF Output:** This is the primary output of the ADF5901, which delivers the modulated 24 GHz signal to the antenna;
- **Power Supply:** The ADF5901 requires a low-noise power supply, typically at 3.3 V.
- **Control IO:** These include SPI pins for configuring the PLL, VCO, and amplifiers;

2.2.2 ADF5904 - 24 GHz Receiver (RX)

The ADF5904 is a highly integrated four-channel receiver (RX) used to down-convert the incoming 24 GHz radar signals to an IF for further processing. Each channel corresponds to a different receive antenna, thus enabling MIMO radar configurations.

- **Low Noise Amplifiers (LNA):** The ADF5904 contains low-noise amplifiers to enhance the weak received signal without introducing significant noise;
- **Mixers:** These are used to down-convert the high-frequency radar signal (24 GHz) to an intermediate frequency, which is easier to process in digital form;
- **Analog Outputs:** After down-conversion, the resulting IF signals are sent to the analog output pins for further digitization and processing;

RX IOs:

- **RF Inputs:** These are connected to the radar's receiving antennas, capturing the reflected radar signals;
- **IF Outputs:** After down-conversion, the IF signals are output here, to be digitized by external ADCs;
- **Power Supply:** The ADF5904 operates on a 3.3 V power supply.
- **Control IO:** Similar to the ADF5901, the ADF5904 is configured via an SPI interface.

2.2.3 ADF4159 - PLL Frequency Synthesizer

The ADF4159 is a PLL frequency synthesizer, which is basically responsible for controlling the frequency modulation of the signal to be transmitted. Being fractional-N, the synthesizer provides ultra-fine steps in frequency, hence suitable for FMCW radar. It ensures that the signal being transmitted is modulated accordingly to a linear chirp pattern necessary in FMCW range and velocity measurement systems.
ADF4159 IOs:

- **SPI Interface:** Used to configure the synthesizer's frequency, modulation parameters, and other settings;
- **Reference Input:** This is the reference clock input that determines the stability and accuracy of the PLL;
- **Control Outputs:** These control signals are sent to the ADF5901 and ADF5904 to coordinate the timing and modulation of the radar signals;

2.3 IO Interfaces and Connectivity

The onboard evaluation board offers the following IO interfaces for configuration, data output, and signal routing:

- **SPI Interface:** The primary interface for controlling the ADF5901, ADF5904, and ADF4159 chips. All configuration including PLL settings and frequency modulation parameters, is done through this serial interface:
- **Power Inputs:** The board typically requires 3.3 V and 5 V power supplies to operate the different chips and components;
- **Analog IF Outputs:** After down-conversion by the ADF5904, the IF signals are output through analog pins, which are then digitized by external ADCs for further signal processing;

- **RF Connections:** The board has SMA connectors for both the RF output of the transmitter and the RF inputs of the receiver, which are connected to the antennas;

2.4 Strengths and Limitations of 24 GHz FMCW Radar

The 24 GHz frequency range is applied to automotive and industrial radar applications because it offers a good balance in the trade-offs between resolution, range, and penetration. This operation provides several advantages of compact antenna size via a relatively short wavelength of about 12.5 mm, hence providing smaller antennas, which are favorable for compact system designs. It also does a very good compromise between range and resolution: it allows detecting targets several hundred meters away, while still getting the resolution needed to tell objects a few meters apart. Additionally, 24 GHz has less atmospheric attenuation compared with high frequency modulations like 77 GHz; this means it will perform better in weather conditions. This does not mean that 24 GHz FMCW radar systems have no disadvantages. These include lower range resolution compared to higher frequencies, which might affect the ability to differentiate between closely spaced objects. The frequency around 24 GHz is also prone to interference due to its use in the operation of Wi-Fi and industrial devices. Lower frequencies normally translate to reduced Doppler resolution, which may affect the accuracy of measurements concerning velocity.
In conclusion the 24 GHz FMCW radar system presents a good compromise among performance, complexity, and cost for short-range radar applications. It has a compactness, good range resolution, and manageable levels of interference as its strong suits, making it versatile; though limitations such as reduced resolution compared to higher frequencies are manageable through careful system design.

2.5 System Modifications: Delay Line Implementation

In this case, the radar was modified for our experimentation by adding an artificial delay line to the RF output of the transmitter. It should now introduce an artificial delay on the transmitted signal, making the object being detected appear further away. The main reason for this was to increase the beat frequency; that is, the frequency difference between the transmitted and received signals. This will turn out to improve the range resolution in our signal processing, in particular in the demodulator using an FFT-based approach, as it will be explained later.

Figure 2.2: The delay line used, an SMA 27Ghz Flexible Cable

2.5.1 Delay Line Overview

An RF system's delay line is a module that introduces programmable amount of delay in the signal passing through it. In our implementation, we have placed this line of delay between the RF output of the transmitter (ADF5901) and the transmit antenna. This ensures that the signal that is transmitted is delayed before it reaches an object to mimic increased distance. Then, this time-delayed signal interacts with the environment and the reflections are collected by the receiver.

RF Output vs. Receiver End

We carefully placed the delay line between the ADF5901 and the transmit antenna on the RF output instead of doing it at the receiving end between the receiving antenna and the ADF5904. This was based on the following reason, the degradation of the received signal: in case the delay line is on the receiver side, this would further add noise and attenuation of a signal which has undergone reflections and scattering already. The received signal is generally weaker compared to the transmitted signal. Therefore, addition of the delay line at this stage may further degrade the SNR and thus dent the overall system performance.

2.5.2 Limitations and Trade-offs

While the introduction of the delay line offered significant improvements in range precision and signal processing, there are some limitations and trade-offs associated

with this approach:

- **Noise Figure of the Delay Line:** The first major concern is about the noise figure of the delay line. Every RF element generates a certain amount of noise and attenuation in the signal, and the delay line is no exception. Because we only introduced it at the RF output, we minimized the impact from putting the delay line; however, there is some degradation in the overall quality of the signal. This can lower the effective SNR, a very critical factor for maintaining accurate target detection, especially in low-power radar systems.
- **Insertion Loss:** The power of the transmitted signal is reduced by the insertion loss originating from the delay line. This in turn reduces the effective radar range since the strength of the transmitted signal is crucial for long distance detection. However, in our system targeting short-range detection of within a few meters, this insertion loss hardly impinged on the performance.
- **Complexity of Calibration:** Introducing a delay line also requires careful calibration to ensure that the system operates correctly. The added delay changes the radar's timing and frequency characteristics, and these must be accounted for in both the hardware setup and the signal processing algorithms.

2.5.3 Conclusion on the Delay Line Implementation

In all, the introduction of a delay line in the transmitter path brought considerable gain in terms of increased beat frequency and, hence, range resolution in our FFT-based demodulation system. Although there are some disadvantages, like the noise figure and insertion loss brought in by the delay line, these were mitigated by placing the delay line at the RF output rather than the receiver end. This system remained highly effective for applications in short-range radar, reaching a good balance between complexity and cost with respect to performance.

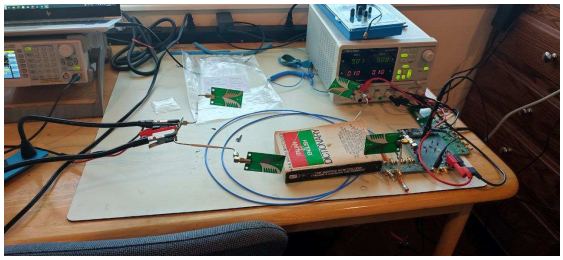


Figure 2.3: Here is a photo of the whole set-up in a very close distance test, using the evaluation board and vivaldi wideband antennas, a book keeps the delay line in place

Chapter 3

Proposed receiver

Following what has been said about the hardware and architecture choices in the previous paragraphs, an incoherent demodulation scheme has been chosen as it's simpler and more cost effective with respect to a coherent solution, in Figure 3.1 it's possible to see all the basic blocks needed to implement both the analog and digital portions of it.

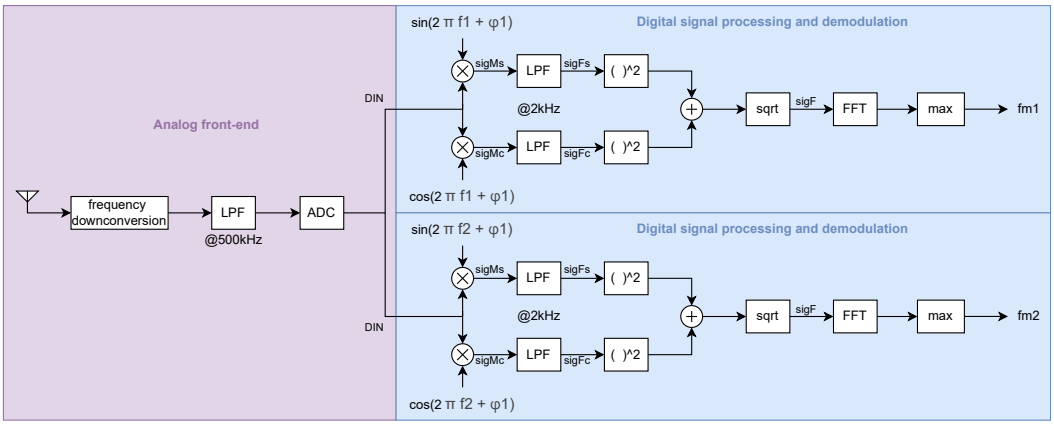


Figure 3.1: Proposed block diagram of the full receiver

The scheme presents two digital paths because the objective is to use it to detect multiple objects, each of this objects has its transponder tuned to a different frequency which in turn modulates the FMCW radar signal, essentially the role of the transponders is to up-convert the two signals by a given frequency to separate the spectrum of the two mixed signals and also to separate them from the low frequency noise clutter.
As previously explained, the signal frequency has been modulated using a triangular wave in such a way that, if the delayed returning signal is mixed with the transmitted

signal, the intermediate frequency resulting from the frequency down-conversion is proportional to the target distance. The signal is then low-pass filtered and sampled at the frequency of 500kHz, the two transponders frequencies are f1=100kHz and f2=200kHz.
The frequency down-conversion is done already inside the evaluation board but there was still the need to sample the signal before going in the digital domain, to avoid aliasing an analog low-pass filter centered at the sampling frequency must also be present. After sampling we can digitally process the signal, the signal is mixed with the center frequency, which is f1 or f2, and then low-pass filtered below 2kHz, this frequency has been chosen since it would result in a distance of more than double of what we are interested in and it's a suitable worst case.
In order to mix the signal without the need to sync to the carrier's phase an incoherent demodulator that makes use of sine and cosine mixing is used. Starting from the signal DIN, it can be demonstrated that this is in the form:

$$DIN(t) = x_1(t)cos(2\pi f_1 t + \theta_1) + x_2(t)cos(2\pi f_2 t + \theta_2) \tag{3.1}$$

where

$$x_1(t) = cos(2\pi f_{m1} t),\ x_2(t) = cos(2\pi f_{m2} t) \tag{3.2}$$

If we now multiply for sine and cosine, following for example the upper path and discarding $x_2(t)cos(2\pi f_2 t + \theta_0)$ since it will be filtered by the low-pass filter, we obtain:

$$sigMc = x_1(t)cos(2\pi f_1 t + \theta_1)cos(2\pi f_1 t + \phi_1) \tag{3.3}$$

$$sigMs = x_1(t)cos(2\pi f_1 t + \theta_1)sin(2\pi f_1 t + \phi_1) \tag{3.4}$$

Of course ϕ_1 is, in general, different from θ_1. Then the signals are low-pass filter to obtain

$$sigFc = \frac{1}{2}x_1(t)cos(\theta_1 - \phi_1) \tag{3.5}$$

$$sigFs = \frac{1}{2}x_1(t)sin(\theta_1 - \phi_1) \tag{3.6}$$

We can now extract $|x_1(t)|$ by summing the squares of this signals and taking the square root:

$$2\sqrt{sigFc^2 + sigFs^2} = |x_1(t)| \tag{3.7}$$

Once we have the modulus of the signal an FFT is used to decompose the signal into its frequency components and finally the bin with the highest absolute value is selected as modulation frequency (actually it's going to be double the frequency since it's a rectified signal).

3.1 Fast Fourier Transform (FFT)

The Fast Fourier Transform (FFT) is an efficient algorithm for computing the Discrete Fourier Transform (DFT), reducing the computational complexity from $O(N^2)$ to $O(N \log N)$. The DFT transforms a sequence of N time-domain samples $x(n)$ into the frequency domain, defined by $X(k) = \sum_{n=0}^{N-1} x(n) \cdot e^{-j\frac{2\pi kn}{N}}$, where $X(k)$ represents the frequency component at index k. The Cooley-Tukey FFT algorithm, the most commonly used, is based on a divide-and-conquer strategy that recursively splits the DFT into smaller DFTs. When N is a power of 2, the input sequence is divided into its even and odd indexed elements, which are separately processed as smaller DFTs of size $N/2$. This yields the relation $X(k) = \mathrm{DFT}_{N/2}(x(2n)) + W_N^k \cdot \mathrm{DFT}_{N/2}(x(2n+1))$, where $W_N^k = e^{-j\frac{2\pi k}{N}}$ is the twiddle factor. This recursive process continues until the DFTs are reduced to size 2, which are trivial to compute. By reusing calculations across the recursive stages, the FFT achieves a much faster runtime of $O(N \log N)$, making it fundamental in various applications such as signal processing, communications, and radar systems.

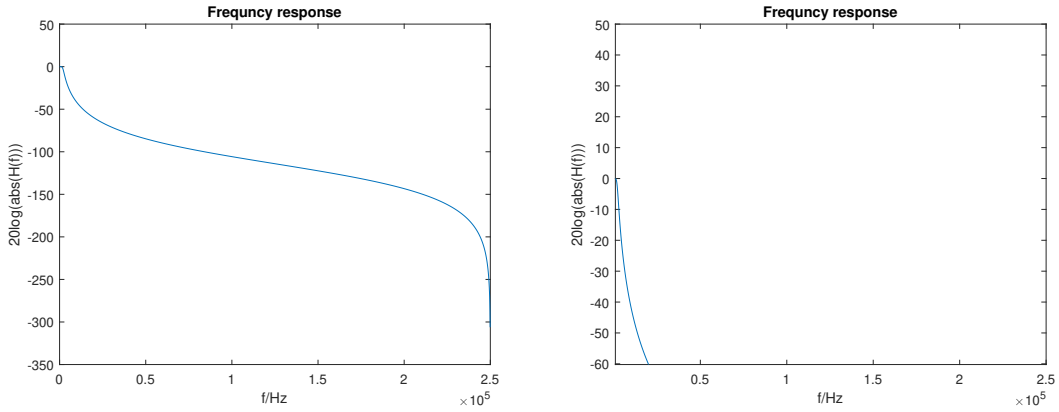
3.2 Architecture derivation

Just by looking at Figure 3.1 a lot of details are missing regarding each block both in terms of algorithm choice and parallelism. The next sections will gradually explain the whole architecture derivation starting from the main blocks and concepts but also covering every detail.

3.2.1 Filter choice

Since the filter is the block where the architectural choices most impact on performance and behaviour of the system, I started defining the architecture by choosing a filter structure, considering the cutoff frequency to sample frequency ratio is very low ($\frac{2kHz}{500kHz} = 0.004$) this means that we need a very selective filter, so the only options are either a very long FIR, in the hundreds of taps range, or an IIR. The advantages of the first choice come in the form of a linear phase and a very straightforward structure that is easy to pipeline, while the advantages of the second choice are the greatly reduced area and complexity since a few taps are necessary; this is ultimately the reason that made me pick the IIR over the FIR.
In order to simulate the entire receiving chain I started from high level MatLab scripts, first the function "signal_generation.m" in Listing A.1 creates the input signal DIN, meaning the signal after the analog portion of the receiver in Figure 3.1, it consists of two sinusoidal signals at the frequency of 1kHz and 2kHz upconverted respectively by 100kHz and 200kHz, with a sampling frequency of 500kHz observed

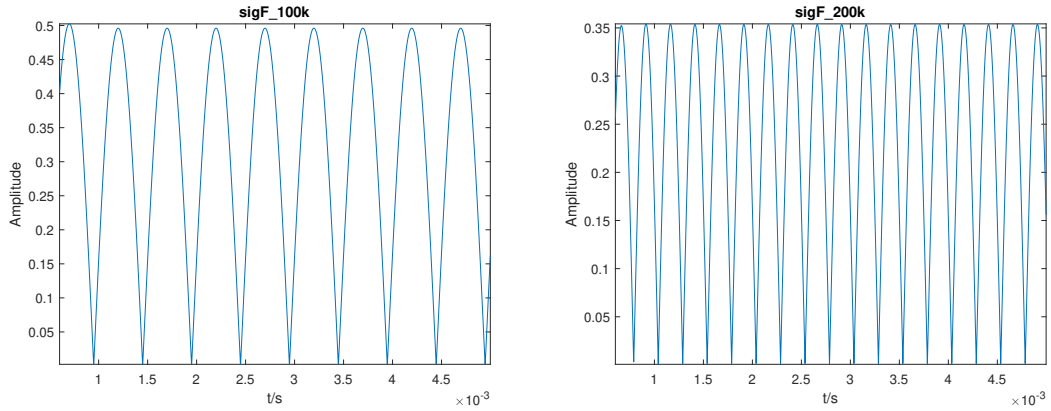
over the span of 5ms.
Once the input is defined I used the script "filter_design.m" in Listing A.2 to describe each block of the system, namely: signal generation, mixing, low-pass filtering and envelope detection through the sum of the squares as previously mentioned, throughout the script another couple of functions are called: "plot_sprectrum.m" in Listing A.3 which performs an FFT and plot the spectrum of the signal is used after each stage to ensure the correct frequency manipulations are performed and "lowpass_filtering.m" is the script in which the type of filter is selected and some of its main characteristics plotted. Through this scripts I decided for a third order Butterworth filter, which has the advantage of the flattest in-band response[5], which means that if the received signal has a modulation frequency with a bit of deviation in time the gain won't change as much as for filters with lower ripple but less flat response like a Chebishev filter for example.
Looking at the plotted figures we are interested in the steepness of the frequency response (in Figure 3.2) which is more than enough considering the two biggest components we want to separate are the two signals which are 100kHz apart more or less, the cutoff frequency and stability of the filter are taken for granted since it's been designed with an internal MatLab function.



(a) Whole response **(b)** Only above -60dB

Figure 3.2: Third order Chebishev filter frequency response

If we run the whole script at the end we can take a look at the two outputs (sigF of Figure 3.1) which should look like rectified sinusoidal functions with frequency equal to 1kHz and 2kHz, respectively for the 100kHz and 200kHz modulating transponders. Here are the two outputs considering that 300 samples have been discarded due to the filter delay:



(a) 100kHz transponder **(b)** 200kHz transponder

Figure 3.3: Extracted modulating functions

There's still a bit of distortion in the first peak because a trade-off between discarded samples and kept samples has been made at around 300 to ensure there's enough signal to work with when the target is very close and the frequency very low, other than that the two components are perfectly separated without the need of a bandpass filter after the analog portion of the receiver; whether the filter adequately suppresses white noise and clutter noise will be addressed later on with real data from the radar.

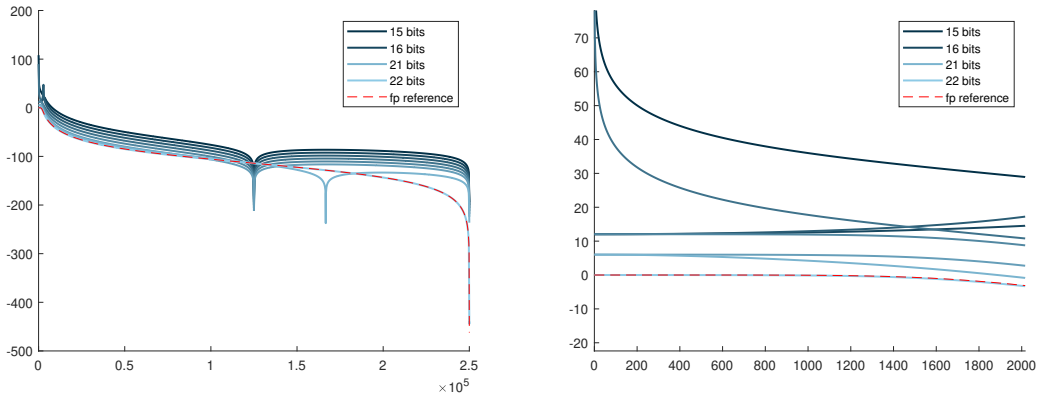
3.2.2 Architecture parallelism and discretization

Since the filter is the only block in the receiver where the design choice highly influences behaviour and performance, for the moment we can postpone designing the other blocks and instead keeping them ideal but selecting the needed parallelism and discretization for the involved signals. Using the script "discretizaztion_test.m" in Listing A.5 four are the parameters to be chosen:

- How many bits to be used for the filter coefficients
- How many samples to throw away due to the filter delay
- How much we can decimate after lowpass filtering the signal in order to save computation time
- How much zeros to be added at the end of the signal in order to increase frequency resolution (padding the FFT)

Starting with the filter coefficients parallelism, using a loop I plotted the resulting frequency response for each choice of number of bits by using a round to nearest

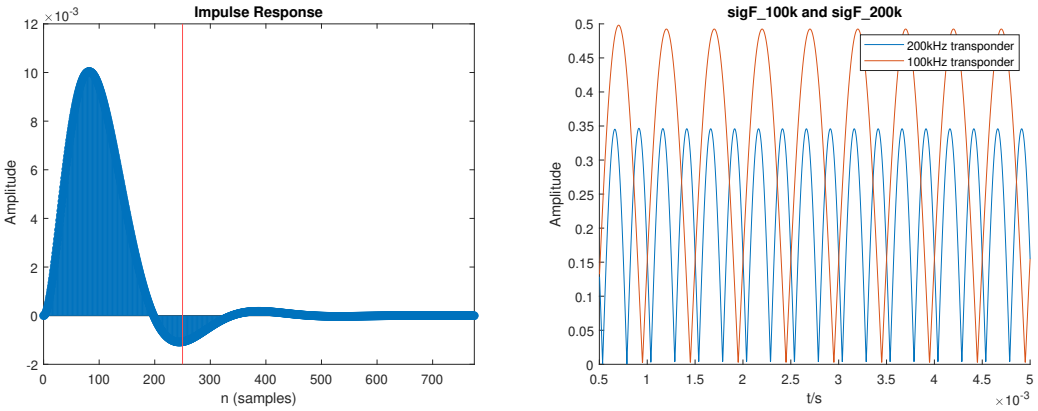
scheme and a fixed point representation with the only exception of never selecting a coefficient to be exactly zero as that resulted in a wrong behaviour of the filter. The lowest number of bits that gave an acceptable transfer function is 22 as it's possible to see in Figure 3.4.



(a) Whole response **(b)** Low-frequency in-band response

Figure 3.4: Third order quantized Chebishev filter frequency response

By looking at this figure where the frequency response for each quantized version of the filter is shown against the floating point reference design, it's clear that 22 is the minimum number of bits to be used, especially because it's the first filter with a good in-band slope and a unitary gain.
Moving onto the delay of the filter we can take a look at the filter impulse response and discard all the samples in the transient, in Figure 3.5 we can see that by choosing to discard 250 samples the transient is not over yet but it's a good compromise between avoiding it and discarding too much samples.



(a) Filter impulse response **(b)** Extracted modulating functions

Figure 3.5: Third order quantized Chebishev filter frequency response

Furthermore from the extracted modulating functions we cannot see a clear transient distorting the waves, it's interesting to notice however that even if the filter has the flattest possible response it's still attenuating 3dB (in addition to the $\frac{1}{2}$ coefficient present due to mixing as shown in Equation 3.5 and Equation 3.6) at the cutoff frequency of 2kHz as per definition and design, which is also the frequency of the modulating function for the 200kHz transponder, so the blue curve in the image has more attenuation than the orange.
To asses how much we can decimate I used an auxiliary script (Listing A.6) that uses an FFT to estimate the frequency of the functions in Figure 3.5 and measure the error as a function of the decimation factor, the result are in Figure 3.6, the relative error oscillates without raising too much until a decimation factor of 40 or even 50, which is a lot considering that $f_s = 500kHz$ imples an equivalent sampling frequency after the decimation $f_{s,eq} = 500kHz/50 = 10kHz$, and the signal we want to express has maximum frequency equal to 2kHz (4kHz when rectified) so if we want to have around 10 samples/period we can choose a much more conservative decimation factor of 15.

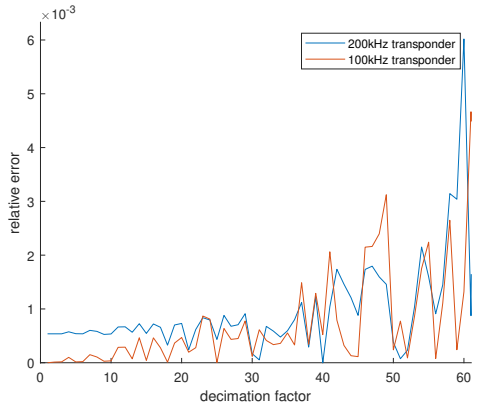


Figure 3.6: Frequency estimate degradation with respect to decimation factor

The last parameter to set now is the FFT padding, running a similar test to the one just described we obtain the results in Figure 3.7, which shows that a good choice could be to have a total of $2^{13} = 8192$ samples, considering that the signal, after being decimated, was 150 samples long this means that the signal has been padded with $8192 - 150 = 8042$ zeros; at first this might seem unreasonable since we have a signal to padding ratio of about $\frac{150}{8042} \simeq 2\%$, this raises the question whether it would be possible to just reduce both the decimation factor and the padding and keep the same 8192 total samples but with an higher signal to padding ratio. That wouldn't work because if we take a look at the formula for the FFT frequency resolution in Equation 3.8 we notice that the sample frequency contributes to the resolution as much as the number of samples and we must remember that decimating implicitly lowers the sample frequency effectively increasing also the spacing between the zeros of the padding and making the resolution higher.

$$f_{res} = \frac{1}{t_{obs}} = \frac{f_s}{\#samples} \tag{3.8}$$

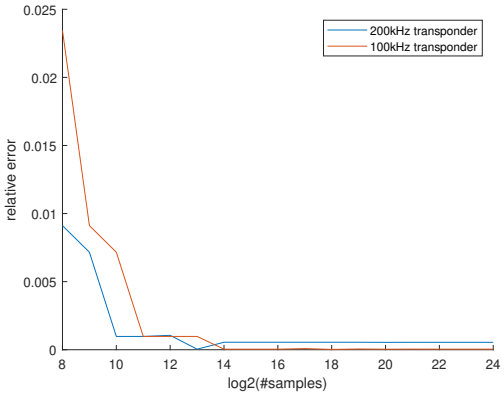


Figure 3.7: Frequency estimate degradation with respect to total number of samples

Chapter 4

Microarchitecture and System Verilog implementation

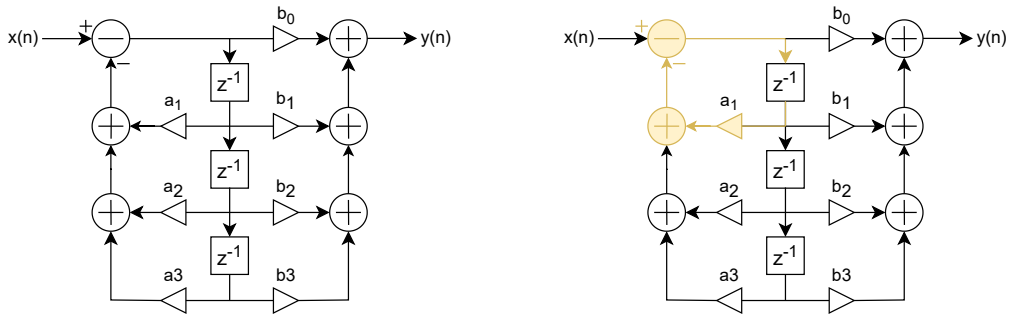
4.1 Mixing

There are several ways to realize a mixer and in particular to produce the needed sinusoidal signal, for example using a LUT or the cordic algorithm, for the specific case in Figure 3.1 we have a 500kHz sampling frequency and the mixing frequencies are $f_1 = 100kHz$ and $f_2 = 200kHz$, that means that we have a sampling period of $2\mu s$ and the mixing periods are $T_1 = 10\mu s$ and $T_2 = 5\mu s$ so that the least common multiple is $10\mu s$ for both cases (5 samples), in other words we only need to produce 5 different values of sine and cosine at f_1 and similarly at f_2. With these premises the best idea is probably a LUT since it's not worth it to implement a full cordic processor for only a few values.
Since the values coming from the ADC have a 14-bit parallelism I decided that having higher precision for the sinusoidal signals with respect to the incoming signal didn't make sense, then also the output of the mixers has a 14-bit parallelism achieved by simply discarding the LSBs from the multiplication; because of its simplicity I didn't produce a MatLab script to check the behaviour and performance of this module (but the overall design has been tested and will be discussed at the end).
Four System Verilog scripts have been produced, one for each sinusoidal signal, in Listing B.3 is reported the one for the cosine at f_1, it makes use of small LUT, a counter to address it (working as a very simple phase accumulator basically) and a multiplier to actually mix the input signal with the cosine.

In the main script (Listing B.1) that represents the whole receiver the module has been instantiated four times, one for each mixing operation.

4.2 Filter

Before implementing the filter using SV let's take a look at a possible topology and it's maximum speed with the current design choices. A common topology for an IIR filter to avoid using two buffers is the direct form II that, for a filter of third degree, looks like this:



(a) **(b)** Critical loop

Figure 4.1: Direct form II filter topology

This topology has the advantage of using only one buffer but the issue is the highlighted yellow loop, this loop sets the iteration bound of the filter at $T_\infty = \frac{Tm+2Ta}{1}$, where T_m and T_a are the multiplication and addition delays respectively, in other words no matter the universal technique that we are going to apply the critical path delay can't be less than T_∞. If we want to implement the filter on a FPGA where we exploit a pre-existing block for the multiplication, which is by far the longest operation in a filter, we ideally want to be able to isolate the delay of the multiplier in order to complete a filter operation in a single clock cycle, to reach this goal the only way is to apply a non-universal technique such as the look-ahead, the idea is to remove the smallest loop of the filter in order to reduce the iteration bound down to $T_\infty = \frac{Tm+3Ta}{2}$ which is actually less than Tm for any reasonable assumption on T_m and T_a.
If we apply this technique in its usual formulation, starting from the coefficients in Table 4.1, we obtain the coefficients in Table 4.2:

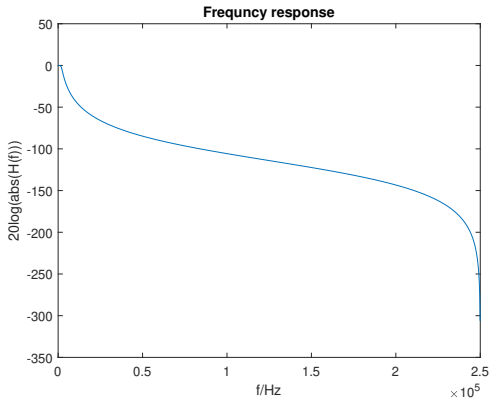
Base topology				
a coefficients	1	-2.9497	2.9007	-0.9510
b coefficients	1.9355e-06	5.8064e-06	5.8064e-06	1.9355e-06

Table 4.1: a and b coefficients for the base filter

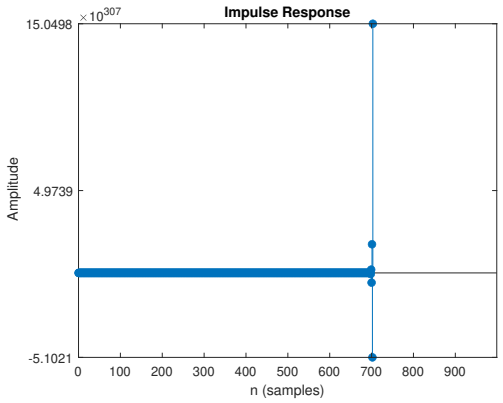
LKHD topology					
a coefficients	1	0	-5.8002	7.6054	-2.8051
b coefficients	1.9355e-06	1.1515e-0.5	2.2934e-05	1.9063e-06	5.7091e-06

Table 4.2: a and b coefficients for the LKHD filter

The objective of the algorithm is to remove the smallest loop by having the first feedback coefficient equal to 0, to achieve this a pole and a zero are added both at the numerator and the denominator so they cancel out each other and the same transfer function is achieved while having $a_1 = 0$. If we take a look at 4.2a it seems that the transfer function is indeed the same as before but if we look at the impulse response for this filter in 4.2b, it diverges.



(a) LKHD topology frequency response



(b) LKHD topology impulse response

This behaviour shouldn't be a surprise however, because the added pole and zero are computed in such a way to remove the need of the first feedback without imposing stability, if we want more freedom we have to keep increasing the degree of the topology until it's stable, the only problem is that the solutions are not unique as we increase the order by n so we need a more reliable way to tell which is the minimum necessary order and how to compute the solution in that case.
The literature has a lot of proposed methods to address this specific problem, for example in [6] they propose a time-variant periodic filter able to solve the

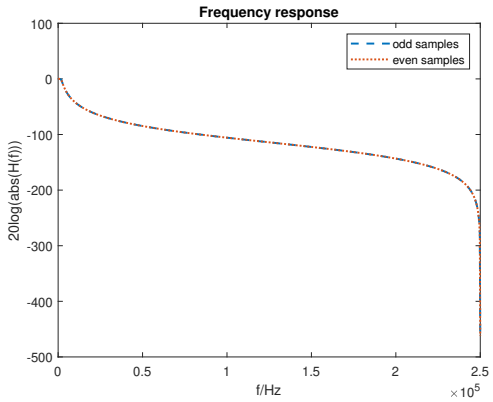
problem with a fixed increase in the filter order independently from how unstable the look-ahead topology turns out to be. Let's consider the LKHD filter with the coefficients in Table 4.2, if we compute the roots of the a vector or in other words the poles of the frequency response we obtain one pole p_0=-2.9497 whose absolute value is definitely greater than 1 and thus the filter is unstable, in the article they demonstrate that to compensate the unstable pole we need a filter of ceil(p_0)+1=4 degrees higher than the base form, making it of degree 7 while the proposed time-variant periodic solution is only 3 degrees higher making it of degree 6, as said before the method as a fixed degree increase of 3 (if we apply the look-ahead to remove only the first feedback) making it more or less efficient compared to simply increasing the complexity but keeping the filter time-invariant based on how far from 1 is the absolute value of the unstable pole, the more the pole is unstable the more convenient is the method compared to not using it.
The procedure to get the time-variant periodic filter is quite heavy from a mathematical standpoint and I believe it to be of not much interest to this thesis to go into every detail of the computation, so I will simply summarize the scripts that were used and briefly comment what they achieve without repeating the paper's content:

- Listing A.7 is the main script calling all of the others and plotting results
- Listing A.8, Listing A.9, Listing A.10, Listing A.11 and Listing A.12 are used to better highlight some of the algebraic steps to compute the intermediate variables needed for the method
- Listing A.13 makes use of the symbolic toolbox within MatLab to place the poles
- "my_impz.m" in Listing A.14 is a function very similar to the built in "impz" used to show the impulse response of a filter given its coefficients and the number of samples, with the added capability of evaluating time-variant filters

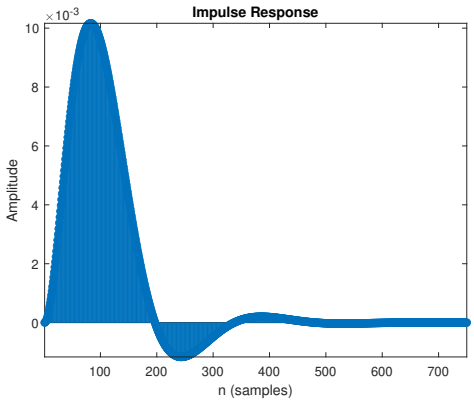
The obtained filter coefficients are in Table 4.3, this filter is a time-variant periodic filter with period=2. To characterize it it's possible to plot the frequency response for both odd and even samples as in 4.2c, the two responses are superimposed and equal to the base form; in addition I used my custom function to show that the impulse response does indeed converge (4.2d), as its supposed to do since the method comes up with filters that are stable by construction.

for odd samples	
a coefficients	b coefficients
1	1.9355e-06
0	1.1515e-05
1.1007	3.6290e-05
-12.7506	5.9132e-05
17.2126	4.5778e-05
-6.5626	-1.33562e-05
0	0
for even samples	
a coefficients	b coefficients
1	1.9355e-06
0	1.1515e-05
-5.8002	2.2934e-05
7.3308	1.8531e-05
-1.9951	4.11461e-06
-0.7965	-1.5944e-06
0.2611	-5.3148e-07

Table 4.3: Proposed time-variant look-ahead filter coefficients



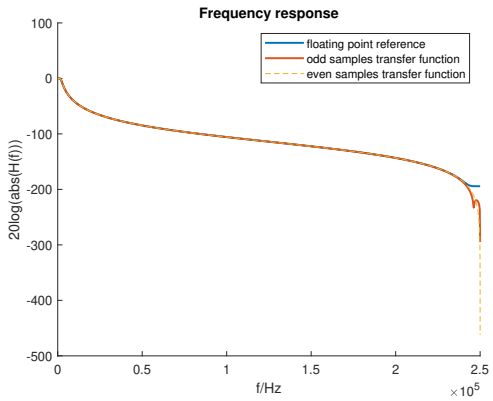
(c) Frequency response



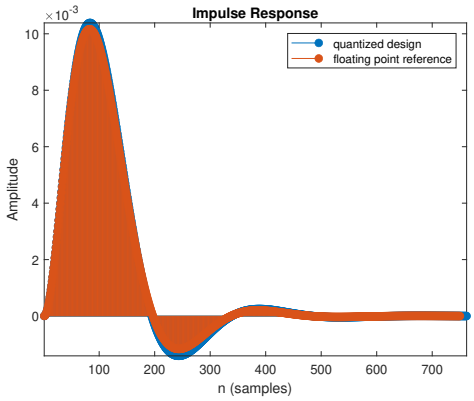
(d) Impulse response

Similarly to what has been done for the base topology we need to quantize the coefficients in order to be able to represent them using a fixed point representation, using the script in Listing A.15 I plotted one last time the frequency response (4.2e) and the impulse response (4.2f) for the filter with quantized coefficients, namely I chose to use 14 bits for the b coefficients and 32 for the a coefficients, this

is because the feedback proved to be the main source of error, moreover, to avoid wasting bits, the two set of coefficients are both integers but with a different scale factor since they have a very different range, as seen in Table 4.3 $|a_n| \in [1, 17.2126]$ while $|b_n| \in [5.3148e-07, 5.9132e-05]$ (not counting coefficients that are exactly zero and have no issues being represented no matter what quantization is chosen).



(e) Frequency response

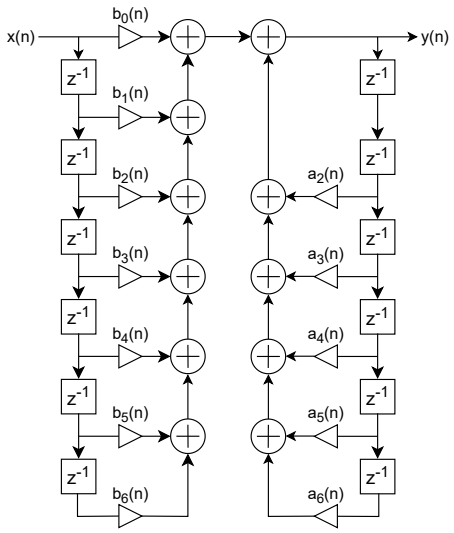


(f) Impulse response

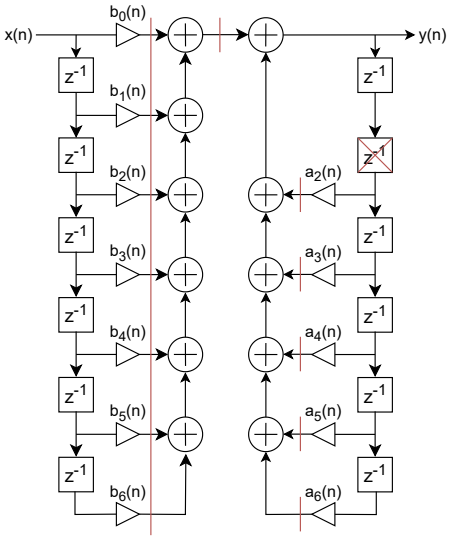
For this choice of coefficients quantization we have almost the same frequency response with the exception of some frequencies where the attenuation was so huge that a small loss does not matter and at the same time the impulse response is very similar, in particular it's stable and converges in more or less the same number of samples.

4.2.1 Final topology and pipelining

In order to reach the maximum speed that the change in filter design enabled we need to make use of pipelining and retiming, before doing that one important note is that we can't exploit the direct form II for this filter because linearity and time-invariance are both prerequisites for that topology transformation, so we are going to use the direct form I in 4.2g and after some pipeling and retiming we can obtain the topology in 4.2h.



(g) Before pipelining and retiming



(h) After pipelining and retiming

Of course the coefficients are a function of n since the filter is time-variant, specifically they are a function of $mod_2(n)$ since the period is equal to 2. After some retiming and pipelining we obtain $T_{cp} = max\{5T_a, T_m\}$ which, based on my experience, should be about the same, maybe even slightly larger for the multiplication. If this is found to be unbalanced after synthesis another possible solution is to simply put a number of registers at the end of the block and exploit the retiming command of the tool itself, either way we obtained a critical path delay that is less than the initial $T_{inf} = T_m + 3T_a$ and so better than any result we could have obtained without transforming the topology.

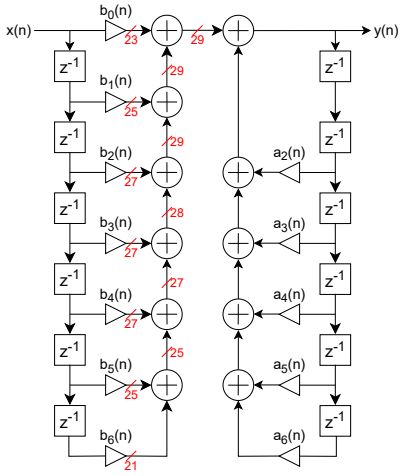
4.2.2 Detailed implementation and parallelism

Before writing any HDL 3 steps are still necessary:

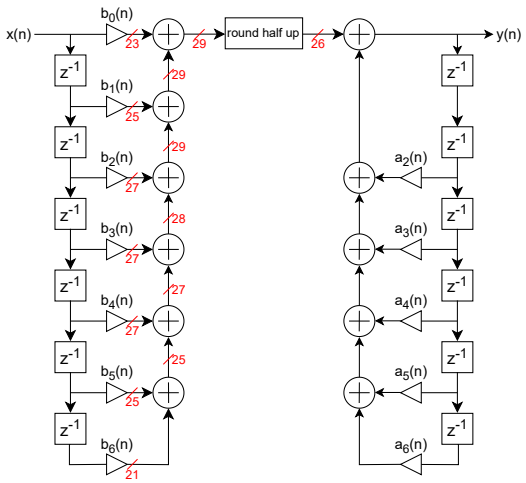
- Define the parallelism for every signal in the circuit
- Round the excess bits coming from the multipliers
- Put at least one saturation block to avoid random spikes in the signal to deteriorate too much the filtering process result

Since I treated every number throughout the design as an integer whose scale factor changes accordingly to it's maximum worst case value, starting from the

feed-forward path we have a 14 bits input coming from the mixing stage that gets multiplied by 14 bits coefficients making the result of the multiplications 27 bits or less because some of the b coefficients are below the maximum and will produce numbers with a foreseeable sign extension that can be simplified. In Listing A.16, from line 27 to 31, I used cumulative sums to asses the parallelism of all the signals after the feed-forward multiplications (vector B_bits) and after the sums (vector Bsum_bits) without removing LSBs for the moment, which makes sense since every coefficient is already quantized down to only have useful LSBs. The resulting parallelism from this calculations is in 4.2i, omitting the sign extension needed when an operator has two inputs with a different parallelism.
Once the feed-forward signal has been computed I decided that after this seven sums I could throw away some LSBs and since there's always a FFCS possible between the two sub-graphs that are connected with a single arrow I went for a somewhat sophisticated rounding scheme at the expense of complexity and delay, the latter is not a problem if the design is further pipelined as briefly mentioned, the rounding scheme is in Listing A.17 and simply implements a round half-up. With this module added I was able to remove 3 LSBs based on the final filter performance that I will show later when also the feed-back sub-graph is commented, in 4.2j it can be seen the filter including the said rounding.



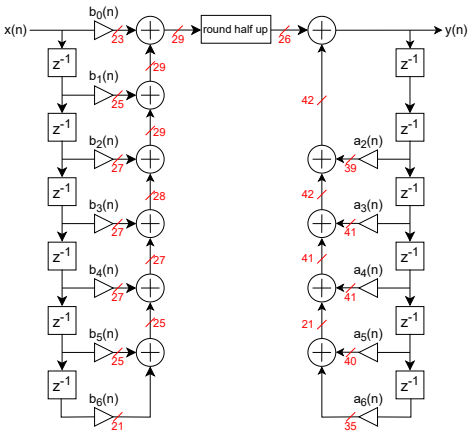
(i) Feed-forward path parallelism



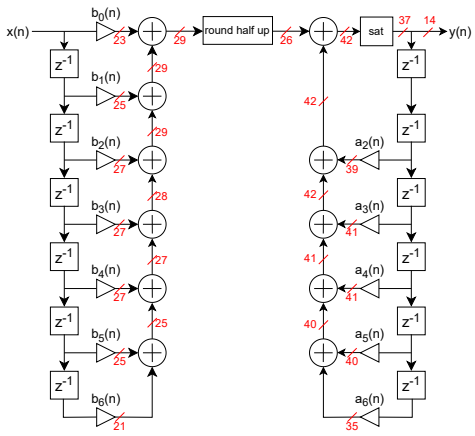
(j) Feed-forward path rounding

To asses the parallelism of the feed-back I tried to avoid sophisticated rounding schemes and simply opted to truncate the outputs of the multipliers, but before even doing that one problem must be addressed: the maximum value of the output is not known a priory, so I went with a pragmatic solution which consist in observing the

amplitude of the extracted modulating functions when the receiver was really close to the transponder, in this way I can use cumulative sums as before to compute each node parallelism, from line 35 to 41 in Listing A.16, the result of this computations decides the number of MSBs while the LSBs are truncated based on the achieved performance of the filter as said before for the rounding. In 4.2k every parallelism is shown, after that in 4.2l I inserted a saturation block to ensure that, even during spurious transients, the filter never has an output with a bigger absolute value than the considered worst case for the output sinusoid.



(k) Feed-forward path parallelism



(l) Feed-forward path rounding

The output is finally truncated back to 14 bits since the precision needed for the feedback loop is not needed anymore.
A function has been written on MatLab (Listing A.18) to replicate this exact RTL design down to every bit handling for validation purposes.

4.2.3 Results and performances

With all this considerations in mind the said script in A.16 finally plots the impulse response to check that it is as close as possible to the reference one, the result is in Figure 4.2.

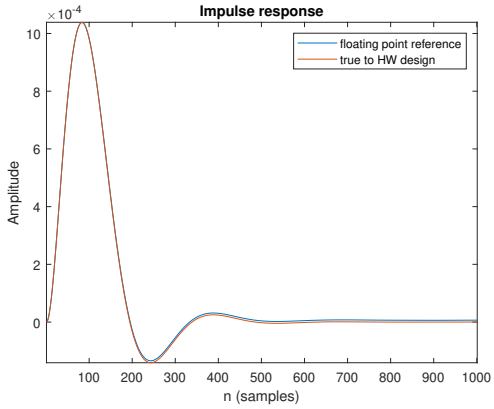


Figure 4.2: Impulse response comparison

The main difference is in the small offset apparent when the reference completely goes to zero and the quantized design doesn't. The main culprit for this behaviour is the truncation after the feed-back multiplications, if this offset is a problem one could truncate less at the expense of complexity or deploy a more robust rounding also in the feedbcak at the expense of complexity and potentially speed, for my intents and purposes this behaviour doesn't influence much the "quality" of the filtered signal as it will later be demonstrated.

4.2.4 System verilog implementation

The main script implementing the filter is the module in Listing B.2, it takes as inputs a signal "DIN" along with its valid signal "VIN", the clock "clk", an asynchronous active low reset "rst_n" and the filter coefficients "a_coefficients" and "b_coefficients", since the filter has been tailored to a specific case it could be argued that it works only with the designed coefficients but I kept them as inputs for flexibility and generality. The filtered output is "VOUT" along its valid signal "VOUT", using these handshake signals makes it easier to orchestrate the whole architecture with an higher in hierarchy module that doesn't need to give and receive start signals itself but just connect each sub-module.
Here is a brief description of how the filter module is implemented without going into every small module instantiated:

- (lines 22 through 37) - Two registers are instantiated for the output signal and valid

- (lines 44 through 56) - Two shift registers are instantiade for the feed-back and feed-forward buffers

- (lines 58 through 64) - A small 1-bit counter is instantiated to keep track of the periodicity of the filter, a "0" means even sample, a "1" means odd sample and so the right coefficients to use in the multiplications are selected accordingly
- (lines 66 through 93) - 12 multipliers are instantiated using also a couple of generate to handle all the multiplications
- (lines 96 through 260) - All of the additions and subtractions are made carefully handling the excess MSBs and LSBs
- (lines 262 through 269) - A saturator is instantiated to avoid spurious temporary out of the dynamic behaviours

The filter module is then instantiated four times (to filter the four mixed signals) in the main system verilog module (Listing B.1) from line 16 to line 53.

Validation

Using a simple testbench it's possible to test the functionality of this core component: as it's possible to see in Figure 4.3, DIN changes after a combinational delay following the rising edge of the clock so then the output y also switches, DOUT is y sampled. There's only a one clock delay between VIN and VOUT as expected since the system has no pipeline included in this testing phase.
The reset behviour looks correct as weel as the simple periodicity counter odd_even. The values coming out of each of the four filters has been checked against the Matlab reference and proved to be correct.

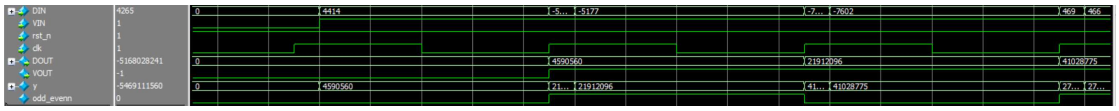


Figure 4.3: Screenshot of the main signals evolution in the filter after the cosine mixing at f_1

4.3 Decimation

As explained in detail in subsection 3.2.2, we can exploit the fact that the signal has been low-pass filtered by decimating it by a factor of 15, from line 131 through 193 of the main receiver module (Listing B.1), first a counter is initialized in order to send only one validation signal to the next block every 15 validation signals coming from the filter, then 5 registers are used to store the 4 mixed signal and the

slower validation signal itself, this design choice allows for reduced complexity of the whole design both by increasing the FFT resolution as shown in Equation 3.8 and by making possible a serial implementation for the square root in which every signal takes 15 clock cycles to be square rooted.

4.4 Unsigned conversion, square and square root

Following the block diagram in Figure 3.1 after filtering we need to square the numbers, as I briefly mentioned before the output of the filter is 37-bit wide and gets cut down to 14 again by removing LSBs, I decided to use an unsigned representation from now on since after squaring it wouldn't make a difference anyway and we are able to save one more LSB with the same signal dynamic, in the main receiver script (Listing B.1) from line 197 through 219 four instances of a module called "SignedToUnsigned" are basically doing a modulus operation to convert from a 15-bits signed to a 14-bit unsigned number without losing precision, this modulus operation does not affect the overall behaviour since we are gonna square the numbers with simple multipliers from line 221 through 248, the results are kept at the full 28 bits to avoid losing precision.

4.4.1 Square root module

Since we now have a 28-bits wide signal and a decimation factor of 15 that means that we can use a fully serial algorithm to save area and consumption.
One such algorithm is for example present in [7], this article presents a way to calculate the square root of an N-bit number in N/2+1 clock cycles which is exactly what we need since we have a 28 bit signal and 15 clock cycles for every sample, the big advantage is definitely in terms of complexity: there are only 5 registers, 1 adder, 1 subtractor, 3 shifters, 5 logical OR modules, 1 comparator, and 1 multiplexer.
Here is a pseudo-code of the algorithm:

- **Start**
- Prepare input data D (radicand), remainder R, square root Q (quotient), partial factor F, and bit-index i.
  - Initialize radicand with input data value
  - Set $R = 0$, $Q = 0$, $F = 0$, $i = n$, where n is the MSB bit-index of D.
- If the radicand has an odd number of digits:
  - Expand the radicand by adding a bit of "0" as MSB.
  - Then proceed to the next step.

Otherwise:

  - Proceed to the next step.

- Divide the radicand into sub-groups, each consisting of 2 digits starting from the integer LSB.
- Begin calculations from the MSB sub-group to the LSB sub-group.
  - Treat the current sub-group as the current partial remainder.
  - $Rt = D[i : i-1]$, where t is the time index indicator.
- Compare the current partial remainder to the current partial factor $(Ft \ll 1)|1$.
  - If the current partial remainder is greater than or equal to the current partial factor:
    - Update Q; $Q_{t+1} = (Q_t \ll 1)|1$
    - Update F; $F_{t+1} = ((F_t + F_t[0]) \ll 1)|1$
  - Otherwise:
    - Update Q; $Q_{t+1} = (Q_t \ll 1)|0$
    - Update F; $F_{t+1} = ((F_t + F_t[0]) \ll 1)|0$
- Subtract the partial remainder by the result of the factor multiplication:
  - Append the subtraction result with the next sub-group data of D in the LSB position of the partial remainder to update R.
  - $R_{t+1} = ((R_t - (F_t \times F_t[0])) \ll 2)|D[i-2 : i-3]$
- Update the current indexes for the next iteration:
  - $t + 1$ is updated to t
  - $i - 2$ is updated to i
  - $i - 3$ is updated to $i - 1$
- If the process is not complete:
  - Return to step 7 and repeat the process.

  Otherwise:

  - The latest Q value is the final square root.
  - The latest R value is the final remainder.
- **End**

The method considers two bits at a time to achieve the N/2+1 speed mentioned before and computes quotient and remainder simultaneously, the extra clock cycle is to adjust the result similarly to a non restoring division.

4.4.2 System verilog implementation

Here is a possible RTL representation of the algorithm:

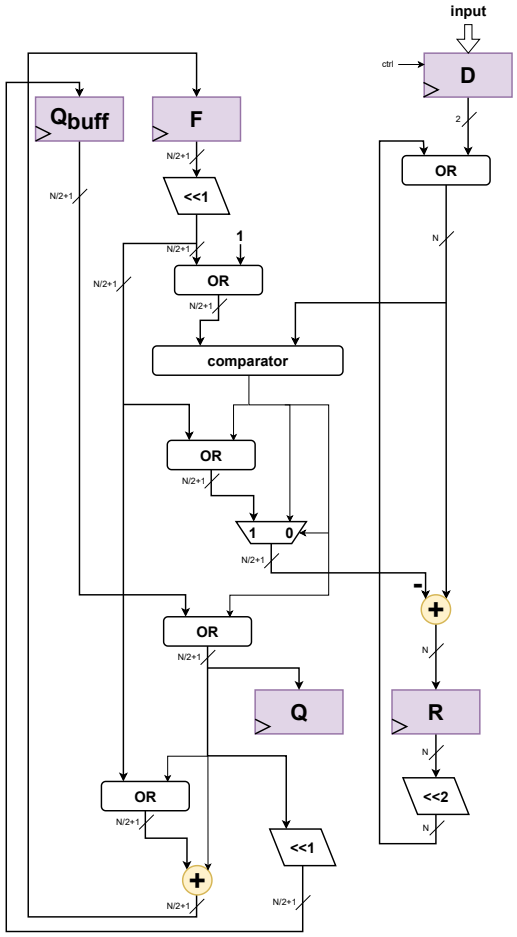


Figure 4.4: Square root module RTL representation

In Listing B.4 5 registers are instantiated according to the scheme in Figure 4.4, while for the combinational logic like the bit-wise OR operations and the mux an always comb block is used to manage everything at high level.

Validation

Using a simple testbench it's possible to test the functionality of this core component: in Figure 4.5 it's possible to see that only one VOUT is produce every 15 clock

periods as expected from the serial implementation and decimation previously discusssed, the same applies to the start signal; If we take the last value of Q before the second start we get 941 so that $Q^2 = 885481$, very close to the radicand which is D_reg after the start signal gets sampled and it's equal to 886986.
The difference between these two numbers, equal to $D\_reg - Q^2 = 886986 - 885481 =$ is cointained in the remainder register R at the end of the computation.

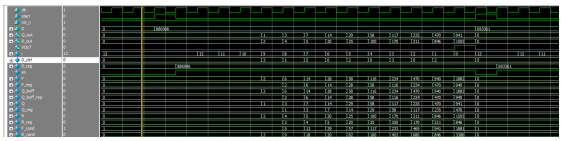


Figure 4.5: Evolution of the signal in the module working with the f_1 transponder signals

4.5 FFT

4.5.1 Algorithm and general architecture

In subsection 3.2.2 we decided for a 8192 samples FFT but it's possible to exploit the fact the we are dealing with a real-valued input signal to halve the number of samples to 4096 at expense of a final conversion layer.
In [8], section 3.2, we can find the formulas to achieve the simplification.
First we need to halve the signal length by creating a new complex signal that stores half the samples in the imaginary portion of the signal:

$$\begin{aligned} x_1(n) &= g(2n) \\ x_2(n) &= g(2n+1) \\ x(n) &= x_1(n) + jx_2(n) \end{aligned} \tag{4.1}$$

Where g(n) is the original real-valued input signal and x(n) is the complex signal half the size of the original.
Then we can get $G(k) = FT\{g(n)\}$ as:

$$\begin{aligned} G(k) &= X(k)A(k) + X^*(N-k)B(k) \\ &\text{where N is half the length of g and} \\ A(k) = \frac{1}{2}(1 - jW_{2N}^k) &\text{ and } B(k) = \frac{1}{2}(1 + jW_{2N}^k) \end{aligned} \tag{4.2}$$

As previously mentioned the problem now is to evaluate $X(k)$ which has half the samples so requires half the butterflies at each stage and one less stage, but there's

one more "stage" required to get back to G through Equation 4.2, so overall we exactly halved the complexity of the operation.
Since we have to process "only" 4096 samples, with modern FPGAs reaching clock speeds of hundred of MHz and an observation time of the signal of 5ms we can conclude that the best approach is probably to do a fully serial architecture since the time constraints aren't that aggressive and we would still achieve the full FFT operation in under the time required to capture a frame of the signal.

4.5.2 Finite state machine and behaviour

Using the MatLab script in Listing A.19 I implemented every block with a bit to bit correspondence to the real architecture to check performance and behaviour of the whole architecture, in particular I evaluated a minimum of 20 bits for the FFT internal signals to have high accuracy and recover the deterioration due to the right-shift of 2 positions for the first FFT stage and 1 position for the consecutive ones needed to keep the same parallelism for each stage even if the range is growing. Additionally I also observed the need for a signal conditioning block that amplifies the signal and makes it signed again to make sure that no matter the power of the input signal, so no matter the distance of the object, the FFT dynamic is fully utilised and there's no unnecessary discretization error.
The resulting FSM is the following:

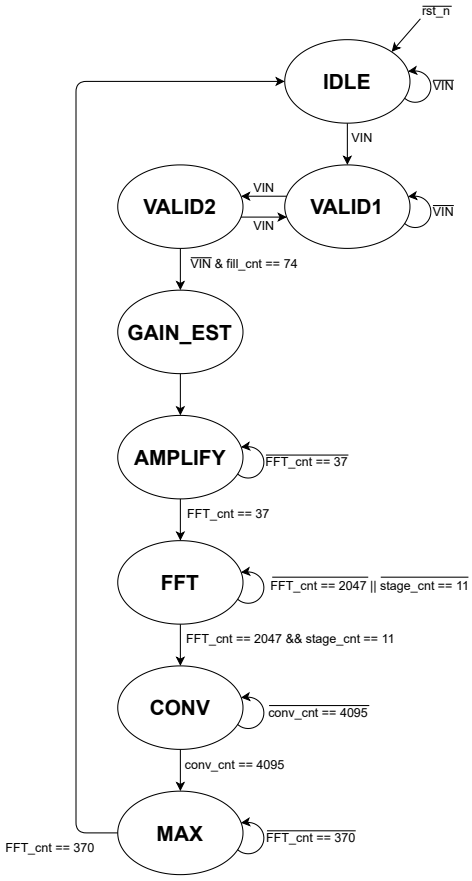


Figure 4.6: FSM diagram

Here's a summary of each state role:

- IDLE is the state the FSM is in while waiting for the first filtered sample, it's entered with the first global reset and exited once the first valid from the last stage arrives
- VALID1 and VALID2 are used to group two samples together as the real and imaginary part of a single sample as shown in Equation 4.1, in the main module of Listing B.1, from line 273 a shift register used to store the two samples is instantiated, then a counter called fill counter is used to count to 74 (75 including 0) to determine when all the 150 samples present after the decimation have all been put inside the RAM instantiated at line 370, the FSM itself is always controlling the RAM accesses and is instantiated at line

324, it also enables the counter when in VALID2 state.
The module for the RAM in Listing B.5 describes a memory with 2^{12} 40-bit locations to accommodate both the real and imaginary part of every sample in each location, additionally it has two ports both for writing and reading since the butterfly module operates on two inputs and gives two outputs.

- While the RAM was being filled a register and a comparator, instantiated from line 385 of Listing B.1 where used to find the biggest sample in order to decide the gain of the amplification, GAIN_EST is a one-cycle state that simply counts the number of leading zeros of the biggest sample to later shift all the samples in the RAM by that value using a module instantiated at line 413 and a register to contain the gain value

- the AMPLIFY state then reads from the RAM all the samples, amplifies them, and puts the back into the RAM while also converting back to signed numbers by simply subtracting half of the dynamic to the unsigned samples coming from the square root, this state is basically the signal conditioning state

- the FFT state is where the butterfly module in Listing B.6 is used to realize a frequency decimated cooley-tukey algorithm, this module take as input two complex samples A and B from the RAM and one twiddle W factor from a ROM, the two outputs are the complex-valued A' and B':

$$\begin{aligned} A' &= A + BW \\ B' &= A - BW \end{aligned} \tag{4.3}$$

  The result is carried out in fixed point arithmetic and then rounded using the same half-up rounding module described previously. Then a right shift of one or two position is used to control the growing dynamic of numbers at each stage of the FFT, the number of right shifts is determined using the signal first_stage which is equal to 1 only for the first stage.
  An overall scheme of the whole FFT process (in the case of 16 samples) is shown in Figure 4.7, so there's still a problem to be faced, how to generate the addresses of the samples to feed the butterfly (and the twiddle factor address as well).
  It's not hard to derive this three addresses using two counters, one for the 2048 butterfly operations (half of the sample number) and one for the 12 stages: in the main script Listing B.1 this two counters are FFT_count (line 442) and FFT_stage_count (line 453), then an additional module called ing_addr_generator (line 463, Listing B.7) performs some simple masking and bitwise or operations on the two counters outputs to achieve the wanted behaviour of producing the three addresses.

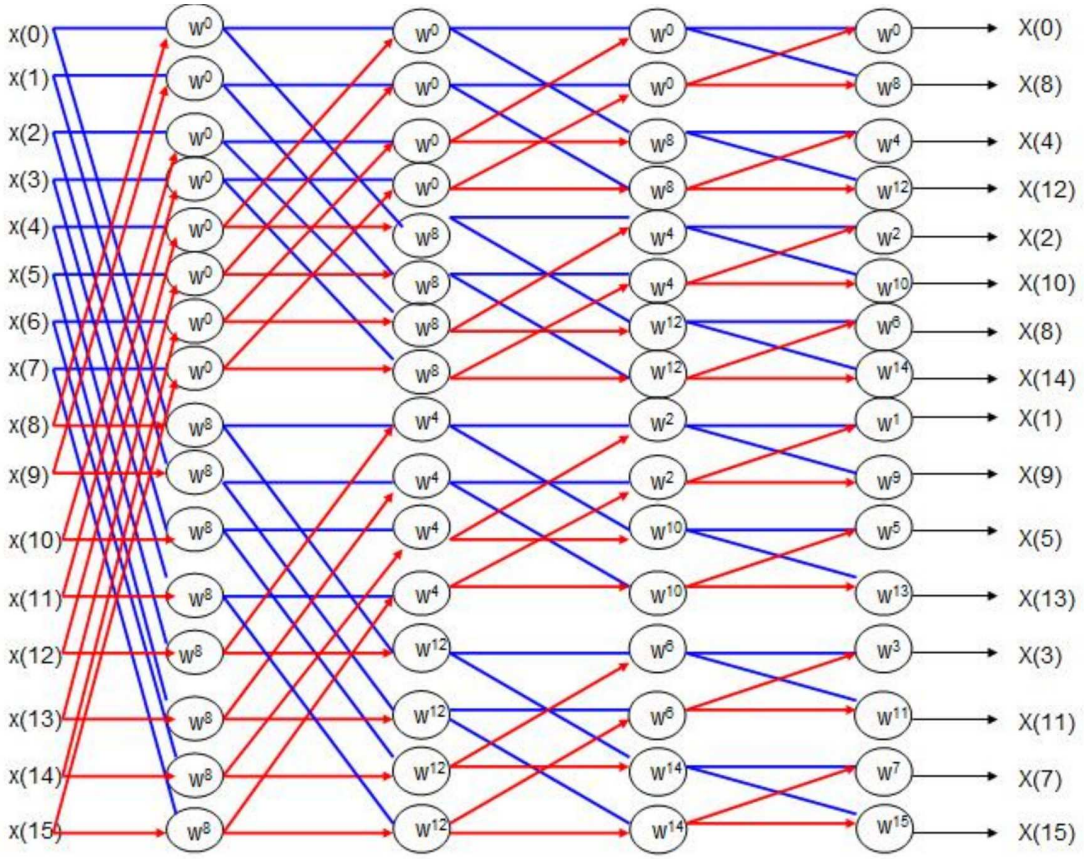


Figure 4.7: FFT decimation in frequency scheme (for 16 samples)

Note that the last stage leaves the results out of order so in my architecture I directly save the results of the last stage in a different memory (to avoid overwriting locations with useful information) called auxiliary RAM, instantiated at line 552, all of this is handled directly by the FSM from line 174 to 214.

- the CONV state simply does what I already explained in detail, namely the operations in Equation 4.2 similarly managing the RAM and using two ROMs for the needed constants.
- the last state MAX searches for the biggest absolute value of every frequency sample, by simply finding the maximum between the sum of the squares of the real and imaginary parts of every bin, there's no need to actually implement an absolute value module (that would then require a square root again), the value of the bin and its number are both saved so that we have our answer on the biggest power frequency bin, which is gonna be the one determining the distance of the object.

Chapter 5

Synthesis and implementation

Synthesis

The last step in the design is to chose an FPGA board to synthesize and implement the design.
Using Xilinx Vivado it's easy to create a project, add all the HDL files and hit "run Synthesis" but I had an issue with the RAM memory since I designed it with an asynchronous read mode that it's possible only if distributed RAM is used on the board, but this kind of RAM realized with LUTs even if very fast and flexible has a big limitation in terms of how much of it is on the board so I decided to use the more standard block RAM which there is plenty of in any modern FPGA and run it at double the frequency of the rest of the circuit in a way to achieve two reads in the first half of a clock cycle and two writes in the second half.
If we add a memory wrapper between this RAM and the rest of the circuit it can be used as if the read was asynchronous and that allowed me to avoid modifying the entire design. I chose as target board a Spartan 7 SP701 (Figure 1.3) since it's a the lower end of performance, price and area and it's more suitable to my application.

Implementation

I chose as strategy "performance with retiming" for the reasons explained in the filter section and after some simple lines of code to set the clock constraint I got this result:

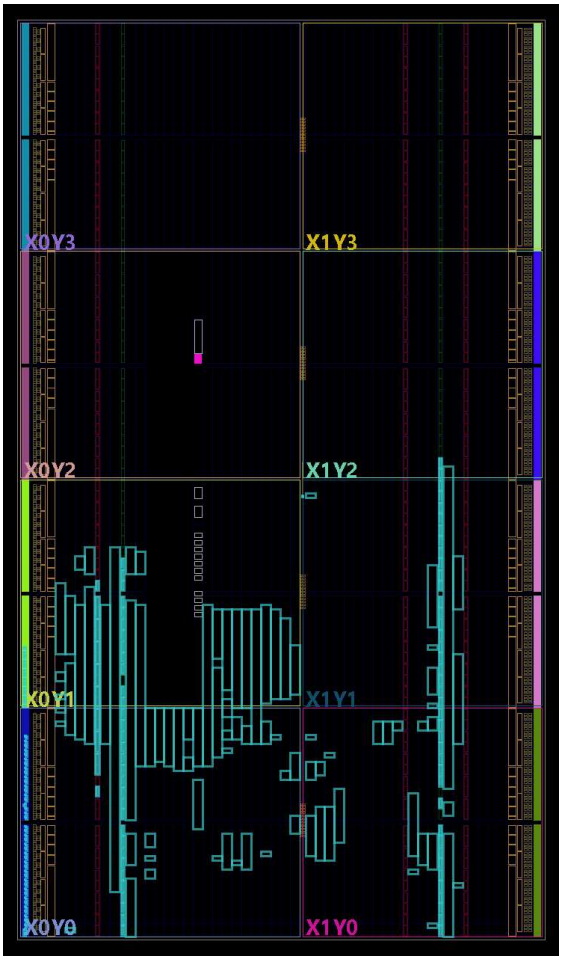


Figure 5.1: Implemented design

basically I let the tool do the technology mapping and routing and got the result in Figure 5.1, checking hardware utilization in Figure 5.2: I am using only 6% of clock buffers, 15% of IO, 8% of block RAM, less than 1% of flip flop and LUT base RAM and 8% of LUT. The only hardware component that approaches an high utilization is the DSP block, utilized for the multipliers and adders.

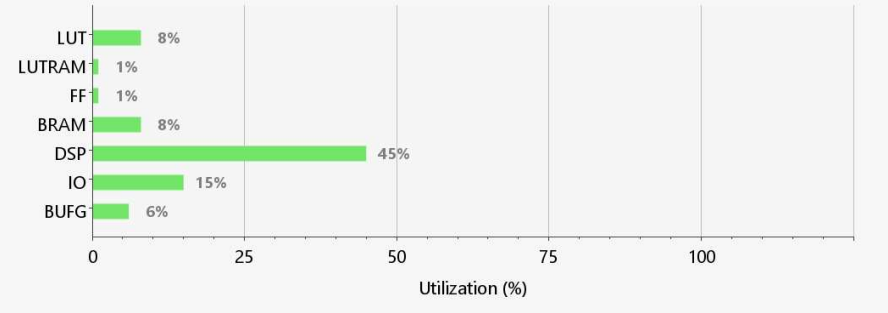


Figure 5.2: Enter Caption

Since the project mainly elaborates signals it's perfectly reasonable to achieve high utilization on the block that can add and multiply, so I'd say the choice of the board is right and gets utilized to its potential.
The speed achieved with this design is of about 50MHz for the memories and 25MHz for the remaining circuits, this is definitely possible to improve by a factor of 2/3 but the main concern for me is the delay, looking at previous simulations the computation take about 30000 clock cycles, so the delay comes out to be $D = 30000/25MHz = 1.2ms$ that is perfectly accetable with an observation time of $5ms$, we even have time to handle multiple radars or different kind of operation in the remaining 4ms. The total on-chip power comes out to be about 0.2W which is indeed a low power consumption.

Chapter 6

Results and conclusions

The script Listing A.20 will be used as a reference and basically follows the block diagram in Figure 3.1 implementing everything as in the SV but without truncations, saturations, roundings and with a floating point representation, in other words it follows the architecture limitations in terms of decimation and padding selection but with machine precision calculations; the output of this script will be compared to the output of the aforementioned script in Listing A.19 which follows the architecture bit by bit.
To asses the precision of the system we can keep the target still and measure multiple times its distance, ideally the system should return the same value of the distance every time, that means that the FFT shows the main peak always at the same frequency, in Table 6.1 we can see that that's exactly the case for the machine precision script, the output is always 1416.02Hz, while the quantized structure shows a small fluctuation around 1370Hz, namely we have a standard deviation of 0Hz in the machine precision case and 7.05Hz in the other case, this deviation is less than 1% of the mean of the values; the difference in values between 1416.02Hz and the mean of the second column which is 1371.3Hz is not necessarily concerning if we see it as a systematic error that can be compensated with a proper calibration.

Frame number	Machine precision estimate	Architecture estimate
1	1416.02Hz	1375.33Hz
2	1416.02Hz	1375.33Hz
3	1416.02Hz	1375.33Hz
4	1416.02Hz	1359.05Hz
5	1416.02Hz	1371.26Hz

Table 6.1: Frequency estimate in the case of machine precision and the quantized architecture

Here is another measurement with the object at a smaller distance:

Frame number	Machine precision estimate	Architecture estimate
1	1106.77Hz	1070.15Hz
2	1106.77Hz	1070.15Hz
3	1106.77Hz	1070.15Hz
4	1106.77Hz	1057.94Hz
5	1106.77Hz	1070.15Hz

Table 6.2: Frequency estimate in the case of machine precision and the quantized architecture

Again we can notice there's a different mean value between the two measurements and the standard deviation is 0Hz for the machine precision case while it's a few Hz (5.46Hz) in the quantized architecture.
Since both estimate appear to be stable but the fixed point implementation has an offset and possibly also a gain error, it's important to wonder if it can predict the distance correctly by compensating the two systematic errors.
In Figure 6.1 the result with the fixed point arithmetic are plotted on the x-axis against the results of the floating point arithmetic in order to use a linear fitting to to compensate gain and offset errors by finding a linear relationship between the two estimates: the fitting function is

$$y = 1.017x + 20.73 \tag{6.1}$$

Finally we can check if the model correctly compensate the errors and can reach high precision for future measurements that haven't been considered in the fitting.

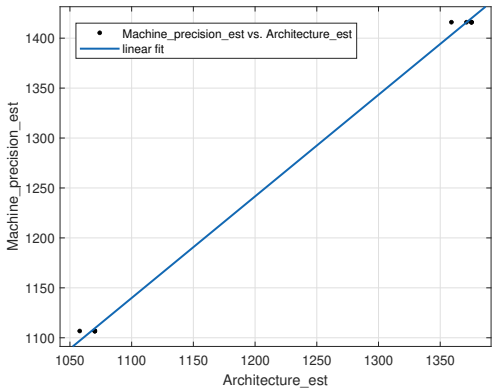


Figure 6.1: Fitting curve

For example, putting the target a bit closer, we obtain $878.906Hz$ with the floating point reference architecture and $842.285Hz$ in the fixed point case, using Equation 6.1 the compensated value is $compensated = uncompensated \cdot 1.017 + 20.73 = 877.3338Hz$ which is within 2% of the reference value and it's in the same order of magnitude of the standard deviation itself, doing the same calculations with other distances gives similar results with errors always below 2%. Alternatively, the compensation step could be skipped altogether if we realize that the system still needs a calibration either way, meaning that once implemented on the real radar board, we should use a look-up table or fitting between the output frequency and the distance itself rather than a reference output frequency, that's because even if they are mathematically linked with a deterministic and known formula the real world non-linearities can't be enclosed in a formula. In conclusion, it has been developed a system

- able to reject clutter noise very well to the point of having 0Hz of standard deviation between frames (with floating point arithmetic at least)
- with a standard deviation smaller than 1% of the mean value even when highly optimized using a cheaper fixed point arithmetic and an error below 2%
- able to detect two (or more) objects at the same time

.

Appendix A

Matlab scripts

Listing A.1: signal_generation.m

```
function [signal]=signal_generation(fs, t_obs)

rng(1);

t=0:1/fs:t_obs;

f_100k=1000;
signal_100k=sin(2*pi*100*10^3*t+2*pi*rand).*sin(2*pi*f_100k*t+2*pi*
    rand);

f_200k=2000;
signal_200k=sin(2*pi*200*10^3*t+2*pi*rand).*sin(2*pi*f_200k*t+2*pi*
    rand);

signal=signal_100k+signal_200k;

end
```

Listing A.2: filter_design.m

```
clc
clear
close all

set(0, 'DefaultFigureWindowStyle', 'docked');

% Generate ideal signal
fs=500*10^3;
t_obs=5*10^-3;
signal=signal_generation(fs, t_obs);

```

```
% Plot input signal spectrum
plot_spectrum(signal, fs)

%% Mixing
% Mixing @100kHz
f0=100*10^3;
t=0:1/fs:t_obs;
cosine_100k=cos(2*pi*f0*t); % cosine carrier
sine_100k=sin(2*pi*f0*t); % sine carrier
sigMc_100k=signal.*(cosine_100k); % cosine mixing
sigMs_100k=signal.*(sine_100k); % sine mixing
plot_spectrum(sigMc_100k,fs)
plot_spectrum(sigMs_100k,fs)

% Mixing @200kHz
f0=200*10^3;
cosine_200k=cos(2*pi*f0*t); % cosine carrier
sine_200k=sin(2*pi*f0*t); % sine carrier
sigMc_200k=signal.*(cosine_200k); % cosine mixing
sigMs_200k=signal.*(sine_200k); % sine mixing
plot_spectrum(sigMc_200k,fs)
plot_spectrum(sigMs_200k,fs)

%% Low-pass filtering
% Low-pass filtering the signal @100kHz
f_cut=2*10^3;
order=3;
group_delay=300;
sigFc_100k=lowpass_filtering(f_cut,fs,order,sigMc_100k);
sigFc_100k=sigFc_100k(group_delay:end);

sigFs_100k=lowpass_filtering(f_cut,fs,order,sigMs_100k);
sigFs_100k=sigFs_100k(group_delay:end);

% Low-pass filtering the signal @200kHz
f_cut=2*10^3;
order=3;
sigFc_200k=lowpass_filtering(f_cut,fs,order,sigMc_200k);
sigFc_200k=sigFc_200k(group_delay:end);

sigFs_200k=lowpass_filtering(f_cut,fs,order,sigMs_200k);
sigFs_200k=sigFs_200k(group_delay:end);

%% Envelope detection
% Envelope @100kHz
sigF_100k=sqrt(sigFc_100k.^2+sigFs_100k.^2);
figure
plot(sigF_100k)

```

```
61 plot_spectrum(sigF_100k, fs)
62
63 % Envelope @200kHz
64 sigF_200k=sqrt(sigFc_200k.^2+sigFs_200k.^2);
65 figure
66 plot(sigF_200k)
67
68 plot_spectrum(sigF_200k, fs)
```

Listing A.3: plot_spectrum.m

```
1  function plot_spectrum(signal, fs)
2
3  Ns=length(signal);
4
5  if mod(Ns, 2) == 0
6      f_range = linspace(-0.5*fs, 0.5*fs-fs/Ns, Ns)';
7  else
8      f_range = linspace(-0.5*fs, 0.5*fs, Ns)';
9  end
10
11 Signal=fftshift(fft(signal));
12 figure
13 plot(f_range,20*log(abs(Signal)));
14 title(inputname(1),'Interpreter','none');
15
16 end
```

Listing A.4: lowpass_filtering.m

```
1  function [filtered_signal]=lowpass_filtering(f_cut,fs,order,signal)
2
3  %filter definition
4  [b,a]=butter(order,f_cut/(fs/2));
5
6  %[b,a]=cheby1(order,1,f_cut/(fs/2));
7
8  %plotting filter characteristics
9  figure;
10 [H,f]=freqz(b,a,length(signal),fs);
11 plot(f, 20*log(abs(H)))
12 figure
13 plot(f, angle(H))
14 figure
15 grpdelay(b, a, 10^4, fs)
16
17 filtered_signal=filter(b,a,signal);
18 save("h","a","b")
19 end
```

Listing A.5: discretization_test.m

```
clc
clear
close all

set(0, 'DefaultFigureWindowStyle', 'docked');

%% Filter bits selection
n_bit_filter=15:22; % candidate number of bits


N=length(n_bit_filter);
fs=500*10^3;

transfer_function=figure;
hold on
colororder(create_color_gradient(N));

for i=1:N

    figure(transfer_function)
    evaluate_filter(n_bit_filter(i)); % quantize filter with the
    given number of bits
    load("qh.mat")
    [H,f]=freqz(b,a,10^6,fs);
    plot(f, 20*log10(abs(H)), LineWidth=1.5)

end

load("h.mat") % load floating point filter design

figure(transfer_function)
[H,f]=freqz(b,a,10^6,fs);
plot(f, 20*log10(abs(H)), '--', "Color", "red") % plot floating point
    transfer function as reference
legend("15 bits", "16 bits", "", "", "", "", "21 bits", "22 bits", "
    fp reference")

n_bit_filter=22; % chose by looking at the transfer functions
evaluate_filter(n_bit_filter); % quantize one more time with the
    selected number of bits
load("qh.mat")
figure;
[H,f]=freqz(b,a,10^6,fs);
plot(f, 20*log10(abs(H))) % plot transfer function of the quantized
    filter
figure
plot(f, angle(H)) % plot phase w.r.t frequency
figure
```

```
44 grpdelay(b,a,10^4,fs) %plot group delay w.r.t frequency
45
46 %% Group delay selection
47
48 figure
49 impz(b,a)
50 xline(250, "r")
51
52 gd=250;
53 [sigF_100k, sigF_200k]=evaluate_gd(gd);
54 figure
55 hold on
56 plot((251:2251+250)/(2251+250)*(5*10^-3),sigF_200k)
57 plot((251:2251+250)/(2251+250)*(5*10^-3),sigF_100k)
58 xlabel("t/s")
59 ylabel("Amplitude")
60 title("sigF\_100k and sigF\_200k")
61 legend("200kHz transponder", "100kHz transponder")
62 %% Decimation factor selection
63 decimation_factor=1:2:30;
64
65 N=length(decimation_factor);
66
67 fs0=fs;
68 sigF_200k_0=sigF_200k;
69 sigF_100k_0=sigF_100k;
70
71 for i=1:N
72     sigF_200k=sigF_200k_0(1:decimation_factor(i):end); % decimation
73     sigF_100k=sigF_100k_0(1:decimation_factor(i):end); % decimation
74
75     fs=fs0/decimation_factor(i); % updating the sample frequency
       accordingly
76
77     est_freq_FFT=freq_estimate(fs,10^7,(sigF_200k-mean(sigF_200k)).*
       hann(1, length(sigF_200k))); % estimate frequency with fft
78     err_decimation_200k(i)=abs((est_freq_FFT+2000)/2000);
79
80     est_freq_FFT=freq_estimate(fs,10^7,(sigF_100k-mean(sigF_100k)).*
       hann(1, length(sigF_100k))); % estimate frequency with fft
81     err_decimation_100k(i)=abs((est_freq_FFT+1000)/1000);
82 end
83
84 figure
85 hold on
86 plot(decimation_factor,err_decimation_200k)
87 plot(decimation_factor,err_decimation_100k)
88
89 decimation_factor=15;
```

```
sigF_200k=sigF_200k_0(1:decimation_factor:end);
sigF_100k=sigF_100k_0(1:decimation_factor:end);
fs=fs0/decimation_factor;

%% Padding selection

pad_start=ceil(log2(length(sigF_200k)));
pad_stop=ceil(log2(10^7));

padding=pad_start:pad_stop;
N=length(padding);

for i=1:N
    est_freq_FFT=freq_estimate(fs,2^padding(i),(sigF_200k-mean(
    sigF_200k)).*hann(1, length(sigF_200k)));
    err_padding_200k(i)=abs((est_freq_FFT+2000)/2000);

    est_freq_FFT=freq_estimate(fs,2^padding(i),(sigF_100k-mean(
    sigF_100k)).*hann(1, length(sigF_100k)));
    err_padding_100k(i)=abs((est_freq_FFT+1000)/1000);
end

figure
hold on
plot(padding,err_padding_200k)
plot(padding,err_padding_100k)

padding=13;
```

Listing A.6: freq_estimate.m

```
function [fm_est]=freq_estimate(fs,N,signal)
% This function plots the FFT of the signal "input" calculated on N
% points from the frequency fmin to the frequency fmax

Xf=fftshift(fft(signal,N));
Xf_abs=abs(Xf);
[~,Imax]=max(Xf_abs(1:N/2));

Ns=length(Xf_abs);
if mod(Ns, 2) == 0
    f_range = linspace(-0.5*fs, 0.5*fs-fs/Ns, Ns)';
else
    f_range = linspace(-0.5*fs, 0.5*fs, Ns)';
end


fm2=f_range(Imax);
fm_est=fm2/2;
```

```
end
```

Listing A.7: lkhd_stabilization.m

```
clc
clear
close all

set(0, 'DefaultFigureWindowStyle', 'docked');

load("..\..\h.mat")
A=a;
B=b;

d=2;

F=find_F(A,d);

PHI=construct_PHI(F,d);
PSI=PHI^d;
GAMMA=[zeros(d-1,1); 1];
C=[1 zeros(1,d-1)];
G_bar=construct_G_bar(PHI,GAMMA,d);

lambda=[0.9,0.9];
K=find_K(PSI,C,lambda);

H_bar=G_bar^-1*K;

F_hat=construct_F_hat(F,H_bar,d);
A_hat=construct_A_hat(F_hat,A);
B_hat=construct_B_hat(F_hat,B);

fs=500000;
[H,f]=freqz(B_hat(1,:),A_hat(1,:),10^6,fs);
plot(f, 20*log10(abs(H)), "--", LineWidth=1.5)
hold on
[H,f]=freqz(B_hat(2,:),A_hat(2,:),10^6,fs);
plot(f, 20*log10(abs(H)), ":", LineWidth=1.5)

figure
my_impz(B_hat,A_hat)

load("..\..\decimated_DL.mat")
signal=decimated_DL;
figure
plot(my_filter(B_hat,A_hat,signal))

save("lkhdh","A_hat","B_hat")
```

```
46
47 [a_prime, b_prime]=compute_lkhd(a,b,1)
```

Listing A.8: construct_PHI.m

```
1 function PHI=construct_PHI(F,d)
2
3 PHI=[zeros(d-1,1) eye(d-1,d-1)];
4 PHI=[PHI; [zeros(1,d-(length(F)-1)) flip(-F(2:end))]];
5
6 end
```

Listing A.9: construct_G_bar.m

```
1 function G_bar=construct_G_bar(PHI,GAMMA,d)
2
3 G_bar=[];
4 for i=1:d
5 G_bar=[G_bar PHI^(d-i)*GAMMA];
6 end
7
8 end
```

Listing A.10: construct_F_hat.m

```
1 unction F_hat=construct_F_hat(F,H_bar,d)
2
3 F_hat=zeros(d,length(F)+length(H_bar));
4 for i=1:d
5 F_hat(i,f1:length(F))=F;
6 F_hat(i,length(F)+i)=H_bar(i);
7 end
8
9 end
```

Listing A.11: construct_A_hat.m

```
1 function A_hat=construct_A_hat(F_hat,A)
2
3 A_hat=zeros(size(F_hat,1),size(F_hat,2)+size(A,2)-1);
4 for i=1:size(F_hat,1)
5 A_hat(i,:)=conv(F_hat(i,:),A);
6 end
7
8 end
```

Listing A.12: construct_B_hat.m

```
1 function B_hat=construct_B_hat(F_hat,B)
```

```
2
3 B_hat=zeros(size(F_hat,1),size(F_hat,2)+size(B,2)-1);
4 for i=1:size(F_hat,1)
5 B_hat(i,:)=conv(F_hat(i,:),B);
6 end
7
8 end
```

Listing A.13: find_K.m

```
1 function K=find_K(PSI,C,lambda)
2
3 % Ensure that Psi and C have compatible dimensions
4 n = size(PSI, 1);
5
6 % Define the symbolic variables for K
7 syms K [n 1]
8
9 % Compute the characteristic polynomial of (Psi - KC)
10 char_poly=charpoly(PSI-K*C);
11
12 % Define the desired characteristic polynomial
13 desired_char_poly=poly(lambda);
14
15 % Create equations by matching the coefficients
16 equations=[];
17 for i=1:n+1
18     equations=[equations;char_poly(i)==desired_char_poly(i)];
19 end
20
21 % Solve the equations for K
22 solution=solve(equations, K);
23
24 % Extract the numeric values of K
25 K=[double(struct2array(solution))]';
26
27 end
```

Listing A.14: my_impz.m

```
1 function my_impz(B_hat,A_hat)
2
3 d=size(A_hat,1);
4 ff_buffer=zeros(1,size(B_hat,2)-1);
5 fb_buffer=zeros(1,size(B_hat,2)-1);
6
7 for k = 1:10^6
8     if k == 1
9         b=B_hat(1,:);
```

```
10          a=A_hat(1,:);
11          ff=dot([1  ff_buffer],b);
12          fb=dot(fb_buffer,a(2:end));
13          imp_resp(k)=ff-fb;
14      else
15          b=B_hat(mod(k+1,d)+1,:);
16          a=A_hat(mod(k+1,d)+1,:);
17          ff=dot([0  ff_buffer],b);
18          fb=dot(fb_buffer,a(2:end));
19          imp_resp(k)=ff-fb;
20      end
21
22      ff_buffer(2:end)=ff_buffer(1:end-1);
23      if(k==1)
24          ff_buffer(1)=1;
25      else
26          ff_buffer(1)=0;
27      end
28      fb_buffer(2:end)=fb_buffer(1:end-1);
29      fb_buffer(1)=imp_resp(k);
30  end
31
32  I=find((imp_resp/max(imp_resp))>10^-4, 1, 'last');
33  stem(imp_resp(1:I),"filled")
34  axis("tight")
35
36  end
```

Listing A.15: lkhd_discretization.m

```
1   clc
2   clear
3   close all
4
5   addpath("D:\tesi\tesi3.0")
6   load("lkhdh.mat")
7   fs=500*10^3;
8
9   set(0, 'DefaultFigureWindowStyle', 'docked');
10
11  n_bit_filter_a=32; % candidate number of bits
12  n_bit_filter_b=14;
13
14  largest_coeff=max(abs(A_hat),[],"all");
15  integer_bits=ceil(log2(largest_coeff))+1;
16  A_hat=quantize(A_hat,n_bit_filter_a,integer_bits,true);
17
18  B_hat=B_hat./sum(B_hat,2).*sum(A_hat,2);
19
```

```
largest_coeff=max(abs(B_hat),[],"all");
integer_bits=ceil(log2(largest_coeff))+1;
B_hat=quantize(B_hat,n_bit_filter_b,integer_bits,true);

save("qlkhdh.mat","A_hat","B_hat")

figure
my_impz(B_hat,A_hat)
hold on
load("lkhdh.mat")
my_impz(B_hat,A_hat)
legend("quantized design", "floating point reference")

figure
hold on

for j=1:2

    load("qlkhdh.mat")
    [H,f]=freqz(B_hat(j,:),A_hat(j,:),10^6,fs);
    plot(f, 20*log10(abs(H)), LineWidth=1.5)

end
load("lkhdh.mat")
[H,f]=freqz(B_hat(1,:),A_hat(1,:),10^6,fs);
plot(f, 20*log10(abs(H)), '--') % plot floating point transfer
    function as reference


title("Frequency response")
xlabel("f/Hz")
ylabel("20log(abs(H(f)))")
legend("floating point reference", "odd samples transfer function", "
    even samples transfer function")
```

Listing A.16: main_physical_filter.m

```
clc
clear
close all

addpath("D:\tesi\tesi3.0")
set(0, 'DefaultFigureWindowStyle', 'docked');

t_obs=0.5*10^-3;
fs=500000;
t=0:1/fs:t_obs;

A=10^-3;
```

```
13 f=0.5*10^3;
14 load("decimated_DL.mat")
15 signal=decimated_DL;
16
17 signal=quantize(signal, 14, 1, false);
18
19 save_as_DIN(signal);
20
21 load("qlkhdh.mat")
22
23 fb_fractional_bits=42;
24 ff_fractional_bits=37;
25
26 %%%%%%%% feed-forward parallelism
27 max_Bsum_out=flip(max(cumsum(flip(abs(B_hat(1,:)))),cumsum(flip(abs(
       B_hat(2,:))))));
28 Bsum_bits=40+ceil(log2(max_Bsum_out))+1;
29
30 max_B=max(abs(B_hat(1,:)),abs(B_hat(2,:)));
31 B_bits=40+ceil(log2(max_B))+1;
32
33 %%%%%%%% feed-back parallelism
34
35 worst_case_amplitude=0.01;
36
37 max_Asum_out=worst_case_amplitude*flip(max(cumsum(flip(abs(A_hat(1,:)
       ))),cumsum(flip(abs(A_hat(2,:))))));
38 Asum_bits=fb_fractional_bits+ceil(log2(max_Asum_out))+1;
39
40 max_A=worst_case_amplitude*max(abs(A_hat(1,:)),abs(A_hat(2,:)));
41 A_bits=fb_fractional_bits+ceil(log2(max_A))+1;
42
43
44 figure
45 plot(physical_filter(B_hat,A_hat,[0.1 zeros(1,1000)],
       ff_fractional_bits,fb_fractional_bits))
46 hold on
47 fb_fractional_bits=100;
48 plot(physical_filter(B_hat,A_hat,[0.1 zeros(1,1000)],
       ff_fractional_bits,fb_fractional_bits))
49 title("Impulse response")
50 xlabel("n (samples)")
51 ylabel("Amplitude")
52 legend("floating point reference", "true to HW design")
53 axis("tight")
```

Listing A.17: round_half_up.m

```
1 function rounded = round_half_up(to_be_rounded, fractional_bits)
```

```
    % Determine the quantization step
    quantization_step = 1 / (2 ^ fractional_bits);

    % Check if the input is complex
    if ~isreal(to_be_rounded)
        % Separate real and imaginary parts
        real_part = real(to_be_rounded);
        imag_part = imag(to_be_rounded);

        % Apply rounding to both parts recursively
        rounded_real_part = round_half_up(real_part, fractional_bits);
        rounded_imag_part = round_half_up(imag_part, fractional_bits);

        % Combine the rounded real and imaginary parts
        rounded = rounded_real_part + 1i * rounded_imag_part;
    else
        % Adjust the value for half-up rounding
        adjusted_value = to_be_rounded + quantization_step / 2;

        % Truncate the values by rounding down (floor)
        rounded = floor(adjusted_value / quantization_step) *
    quantization_step;
    end
end
```

Listing A.18: physical_filter.m

```
function filtered_signal=physical_filter(B_hat,A_hat,signal,
    ff_fractional_bits,fb_fractional_bits)

d=size(A_hat,1);
ff_buffer=zeros(1,size(B_hat,2)-1);
fb_buffer=zeros(1,size(B_hat,2)-1);

for k = 1:length(signal)

    b=B_hat(mod(k+1,d)+1,:);
    a=A_hat(mod(k+1,d)+1,:);

    ff=[signal(k) ff_buffer].*b;
    %ff*2^40 %compare to mult_results_b
    ff=sum(ff);
    %ff*2^40 %compare to sum6
    ff=round_half_up(ff,ff_fractional_bits);
    %ff*2^37 %compare to ff


```

```
20      fb=fb_buffer.*a(2:end);
21      fb=truncate(fb,fb_fractional_bits);
22      %fb*2^42 %compare to mult_results_a
23      fb=sum(fb);
24      %fb*2^42 %compare to sum10
25
26      filtered_signal(k)=ff-fb;
27      %filtered_signal*2^42 %compare to diff11
28
29      if(filtered_signal(k)>2^-6-2^-(6+fb_fractional_bits))
30          filtered_signal(k)=2^-6;
31      end
32
33      if(filtered_signal(k)<-2^-6)
34          filtered_signal(k)=-2^-6;
35      end
36
37      %filtered_signal*2^42 %compare to y
38
39      ff_buffer(2:end)=ff_buffer(1:end-1);
40      ff_buffer(1)=signal(k);
41      fb_buffer(2:end)=fb_buffer(1:end-1);
42      fb_buffer(1)=filtered_signal(k);
43  end
44
45  % Plot the impulse response
46  % figure
47  % plot(filtered_signal)
48
49  end
```

Listing A.19: main_physical.m

```
1   clc
2   clear
3   close all
4
5   set(0, 'DefaultFigureWindowStyle', 'docked');
6
7   dout=readtable('D:\tesi\implementation\data\dout_values.txt');
8
9   %data([dout.Var1+1;dout.Var2+1])=[dout.Var3+1i*dout.Var4;dout.Var5+1i
        *dout.Var6];
10  data=dout.Var1;
11
12  save("data", "data")
13
14  addpath("D:\tesi\tesi3.0")
15  load("decimated_DL.mat")
```

```
16 signal=decimated_DL;
17
18 signal=signal_conditioning(signal,1-2^-13,-1);
19
20 signal=quantize(signal, 14, 1, false);
21 save_as_DIN(signal);
22
23 % Initial sample frequency
24 fs=500*10^3;
25
26 % Plot input signal spectrum
27 plot_spectrum(signal, fs)
28
29 %% Mixing
30 % Mixing @100kHz
31 f0=100*10^3;
32 Ns=length(signal);
33 t=0:1/fs:(Ns-1)/fs;
34
35 cosine_100k=cos(2*pi*f0*t+1); % cosine carrier
36 cosine_100k=quantize(cosine_100k, 14, 1, false);
37
38 sine_100k=sin(2*pi*f0*t+1); % sine carrier
39 sine_100k=quantize(sine_100k, 14, 1, false);
40
41 sigMc_100k=signal.*(cosine_100k); % cosine mixing
42 sigMc_100k=truncate(sigMc_100k, 13);
43
44 sigMs_100k=signal.*(sine_100k); % sine mixing
45 sigMs_100k=truncate(sigMs_100k, 13);
46
47 plot_spectrum(sigMc_100k,fs)
48 plot_spectrum(sigMs_100k,fs)
49
50 % Mixing @200kHz
51 f0=200*10^3;
52 Ns=length(signal);
53 t=0:1/fs:(Ns-1)/fs;
54
55 cosine_200k=cos(2*pi*f0*t+1); % cosine carrier
56 cosine_200k=quantize(cosine_200k, 14, 1, false);
57
58 sine_200k=sin(2*pi*f0*t+1); % sine carrier
59 sine_200k=quantize(sine_200k, 14, 1, false);
60
61 sigMc_200k=signal.*(cosine_200k); % cosine mixing
62 sigMc_200k=truncate(sigMc_200k, 13);
63
64 sigMs_200k=signal.*(sine_200k); % sine mixing
```

```
65 sigMs_200k=truncate(sigMs_200k, 13);
66
67 plot_spectrum(sigMc_200k,fs)
68 plot_spectrum(sigMs_200k,fs)
69
70 %% Low-pass filtering
71 % Low-pass filtering the signal @100kHz
72 load("qlkhdh.mat")
73
74 bits_post_filter=15;
75 fb_fractional_bits=42;
76 ff_fractional_bits=37;
77
78 group_delay=250;
79 sigFc_100k=physical_filter(B_hat,A_hat,sigMc_100k,ff_fractional_bits,
       fb_fractional_bits);
80 sigFc_100k=sigFc_100k(group_delay+1:end);
81 sigFc_100k=truncate(sigFc_100k, 5+bits_post_filter);
82
83 sigFs_100k=physical_filter(B_hat,A_hat,sigMs_100k,ff_fractional_bits,
       fb_fractional_bits);
84 sigFs_100k=sigFs_100k(group_delay+1:end);
85 sigFs_100k=truncate(sigFs_100k, 5+bits_post_filter);
86
87 % Low-pass filtering the signal @200kHz
88 sigFc_200k=physical_filter(B_hat,A_hat,sigMc_200k,ff_fractional_bits,
       fb_fractional_bits);
89 sigFc_200k=sigFc_200k(group_delay+1:end);
90 sigFc_200k=truncate(sigFc_200k, 5+bits_post_filter);
91
92 sigFs_200k=physical_filter(B_hat,A_hat,sigMs_200k,ff_fractional_bits,
       fb_fractional_bits);
93 sigFs_200k=sigFs_200k(group_delay+1:end);
94 sigFs_200k=truncate(sigFs_200k, 5+bits_post_filter);
95
96 %% Envelope detection
97 % Envelope @100kHz
98 sigF_100k=physical_sqrt(sigFc_100k.^2+sigFs_100k.^2, 40);
99 sigF_100k=sigF_100k(1:15:end)*2^20;
100 fs=fs/15;
101 gain=2^floor(log2(2^14/max(sigF_100k)));
102 sigF_100k=sigF_100k*2^6;
103 sigF_100k=sigF_100k*gain;
104
105 sigF_100k=sigF_100k-524288; % need to be signed to work with the FFT
106
107 figure
108 plot(sigF_100k)
109
```

```
110 sigF_100k=sigF_100k(1:2:end)+sigF_100k(2:2:end)*1i;
111
112 X=physical_FFT(sigF_100k);
113
114 N=length(X);
115
116 X=[X,X(1)];
117
118 G=physical_conversion(X, N);
119
120 G=[G, G(end-1:-1:2)];
121
122 Ns=2*N;
123
124 if mod(Ns, 2) == 0
125     f_range = linspace(-0.5*fs, 0.5*fs-fs/Ns, Ns)';
126 else
127     f_range = linspace(-0.5*fs, 0.5*fs, Ns)';
128 end
129
130 figure
131 plot(f_range,20*log(abs(fftshift(G))));
132 title("G",'Interpreter','none');
```

Listing A.20: main_with_data.m

```
1  clc
2  clear
3  close all
4
5  set(0, 'DefaultFigureWindowStyle', 'docked');
6  addpath("D:\tesi\tesi3.0\Masera\lkhd_stabilization")
7  % Load measurement
8  load('decimated_DL.mat')
9  signal=decimated_DL;
10
11 signal=signal_conditioning(signal,1-2^-13,-1);
12
13 % Initial sample frequency
14 fs=500*10^3;
15
16 % Plot input signal spectrum
17 plot_spectrum(signal, fs)
18
19 %% Mixing
20 % Mixing @100kHz
21 f0=100*10^3;
22 Ns=length(signal);
23 t=0:1/fs:(Ns-1)/fs;
```

```
cosine_100k=cos(2*pi*f0*t); % cosine carrier
sine_100k=sin(2*pi*f0*t); % sine carrier
sigMc_100k=signal.*(cosine_100k); % cosine mixing
sigMs_100k=signal.*(sine_100k); % sine mixing
plot_spectrum(sigMc_100k,fs)
plot_spectrum(sigMs_100k,fs)

% Mixing @200kHz
f0=200*10^3;
Ns=length(signal);
t=0:1/fs:(Ns-1)/fs;
cosine_200k=cos(2*pi*f0*t); % cosine carrier
sine_200k=sin(2*pi*f0*t); % sine carrier
sigMc_200k=signal.*(cosine_200k); % cosine mixing
sigMs_200k=signal.*(sine_200k); % sine mixing
plot_spectrum(sigMc_200k,fs)
plot_spectrum(sigMs_200k,fs)

%% Low-pass filtering
% Low-pass filtering the signal @100kHz
load("h.mat")

group_delay=250;
sigFc_100k=filter(b,a,sigMc_100k);
sigFc_100k=sigFc_100k(group_delay+1:end);

sigFs_100k=filter(b,a,sigMs_100k);
sigFs_100k=sigFs_100k(group_delay+1:end);

% Low-pass filtering the signal @200kHz
f_cut=2*10^3;
order=3;
sigFc_200k=filter(b,a,sigMc_200k);
sigFc_200k=sigFc_200k(group_delay+1:end);

sigFs_200k=filter(b,a,sigMs_200k);
sigFs_200k=sigFs_200k(group_delay+1:end);

%% Envelope detection
% Envelope @100kHz
sigF_100k=sqrt(sigFc_100k.^2+sigFs_100k.^2);
figure
plot(sigF_100k)

plot_spectrum(sigF_100k, fs)

% Envelope @100kHz
sigF_200k=sqrt(sigFc_200k.^2+sigFs_200k.^2);
figure
```

```
73 plot(sigF_200k)
74
75 plot_spectrum(sigF_200k, fs)
76
77 % Decimation
78 sigF_100k=sigF_100k(1:15:end);
79 fs=fs/15;
80
81 figure
82 plot(sigF_100k)
83
84 plot_spectrum([sigF_100k, zeros(1,2^12-150)], fs)
```

Appendix B

HDL

Listing B.1: rx.sv

```
module rx (

    // Inputs
    input logic signed [13:0] DIN,
    input logic VIN,
    input logic rst_n,
    input logic clk,
    input logic signed [13:0] b_coefficients [1:0][6:0],
    input logic signed [31:0] a_coefficients [1:0][6:2],

    // Outputs
    output logic unsigned [39:0] DOUT,
    output logic VOUT
);

    ////////////////////////////////////// Mixing
    //////////////////////////////////////

    logic signed [13:0] sigMc_100k;
    mixer_cosine_100k mixer_cosine_100k_inst(
    .clk(clk),
    .VIN(VIN),
    .rst_n(rst_n),
    .input_signal(DIN),
    .mixed_signal(sigMc_100k)
    );

    logic signed [13:0] sigMs_100k;
    mixer_sine_100k mixer_sine_100k_inst (
    .clk(clk),
    .VIN(VIN),
```

```
31      .rst_n(rst_n),
32      .input_signal(DIN),
33      .mixed_signal(sigMs_100k)
34      );
35
36      logic signed [13:0] sigMc_200k;
37      mixer_cosine_200k mixer_cosine_200k_inst(
38      .clk(clk),
39      .VIN(VIN),
40      .rst_n(rst_n),
41      .input_signal(DIN),
42      .mixed_signal(sigMc_200k)
43      );
44
45      logic signed [13:0] sigMs_200k;
46      mixer_sine_200k mixer_sine_200k_inst (
47      .clk(clk),
48      .VIN(VIN),
49      .rst_n(rst_n),
50      .input_signal(DIN),
51      .mixed_signal(sigMs_200k)
52      );
53
54      ////////////////////////////////////// Low-pass filtering
        //////////////////////////////////////
55
56      logic VOUT_filters;
57      logic signed [36:0] sigFc_100k;
58      cheb_lpf3_lkhd filter_cosine_100k (
59          .DIN(sigMc_100k),
60          .VIN(VIN),
61          .rst_n(rst_n),
62          .clk(clk),
63          .b_coefficients(b_coefficients),
64          .a_coefficients(a_coefficients),
65          .DOUT(sigFc_100k),
66          .VOUT(VOUT_filters)
67      );
68
69      logic signed [36:0] sigFs_100k;
70      cheb_lpf3_lkhd filter_sine_100k (
71          .DIN(sigMs_100k),
72          .VIN(VIN),
73          .rst_n(rst_n),
74          .clk(clk),
75          .b_coefficients(b_coefficients),
76          .a_coefficients(a_coefficients),
77          .DOUT(sigFs_100k),
78          .VOUT()
```

```
79      );
80
81      logic signed [36:0] sigFc_200k;
82      cheb_lpf3_lkhd filter_cosine_200k (
83          .DIN(sigMc_200k),
84          .VIN(VIN),
85          .rst_n(rst_n),
86          .clk(clk),
87          .b_coefficients(b_coefficients),
88          .a_coefficients(a_coefficients),
89          .DOUT(sigFc_200k),
90          .VOUT()
91      );
92
93      logic signed [36:0] sigFs_200k;
94      cheb_lpf3_lkhd filter_sine_200k (
95          .DIN(sigMs_200k),
96          .VIN(VIN),
97          .rst_n(rst_n),
98          .clk(clk),
99          .b_coefficients(b_coefficients),
100         .a_coefficients(a_coefficients),
101         .DOUT(sigFs_200k),
102         .VOUT()
103     );
104
105     // Ignore the first 250 results of the filtering
106     logic del_en;
107     logic [7:0] del_cnt;
108
109     SR_FF del_FF (
110     .set(del_cnt==8'd249 && VOUT_filters),
111     .reset(1'b0),
112     .clk(clk),
113     .rst_n(rst_n),
114     .reg_en(1'b1),
115     .Q(del_en)
116     );
117
118
119     counter #(8, 250) del_cnt_inst (
120     .clk(clk),          // Clock signal
121     .rst_n(rst_n),        // Asynchronous active low reset
122     .srst(1'b0),          // Asynchronous reset
123     .en(VOUT_filters && !del_en),       // Enable signal
124     .count(del_cnt) // Counter output
125     );
126
127     logic VOUT_filters_del;
```

```
128     assign VOUT_filters_del = VOUT_filters && del_en;
129
130
131     ////////////////////////////// decimation
        ////////////////////////////////////////////////////////////
132
133     logic [3:0] dec_cnt;
134     counter #(4, 15) dec_cnt_inst (
135     .clk(clk),          // Clock signal
136     .rst_n(rst_n),        // Asynchronous active low reset
137     .srst(1'b0),          // Asynchronous reset
138     .en(VOUT_filters_del),          // Enable signal
139     .count(dec_cnt) // Counter output
140     );
141
142     logic VIN_square;
143     assign VIN_square = (dec_cnt==4'd0) && VOUT_filters_del;
144
145     logic [14:0] dec_sigFc_100k;
146     register #(15) dec_sigFc_100k_reg (
147     .data_in(sigFc_100k[36:22]),
148     .clk(clk),
149     .rst_n(rst_n),
150     .srst(1'b0),
151     .reg_en(VIN_square),
152     .data_out(dec_sigFc_100k)
153     );
154
155     logic [14:0] dec_sigFs_100k;
156     register #(15) dec_sigFs_100k_reg (
157     .data_in(sigFs_100k[36:22]),
158     .clk(clk),
159     .rst_n(rst_n),
160     .srst(1'b0),
161     .reg_en(VIN_square),
162     .data_out(dec_sigFs_100k)
163     );
164
165     logic [14:0] dec_sigFc_200k;
166     register #(15) dec_sigFc_200k_reg (
167     .data_in(sigFc_200k[36:22]),
168     .clk(clk),
169     .rst_n(rst_n),
170     .srst(1'b0),
171     .reg_en(VIN_square),
172     .data_out(dec_sigFc_200k)
173     );
174
175     logic [14:0] dec_sigFs_200k;
```

```
176     register #(15) dec_sigFs_200k_reg (
177     .data_in(sigFs_200k[36:22]),
178     .clk(clk),
179     .rst_n(rst_n),
180     .srst(1'b0),
181     .reg_en(VIN_square),
182     .data_out(dec_sigFs_200k)
183     );
184
185     logic VOUT_square;
186     register #(1) VOUT_square_reg (
187     .data_in(VIN_square),
188     .clk(clk),
189     .rst_n(rst_n),
190     .srst(1'b0),
191     .reg_en(1'b1),
192     .data_out(VOUT_square)
193     );
194
195     ////////////////////////////// unsigned conversion, square and
        square root ////////////////////////
196     // unsigned conversion
197     logic signed [13:0] u_sigFc_100k;
198     SignedToUnsigned #(15) sign_cosine_100k (
199     .signed_in(dec_sigFc_100k),
200     .unsigned_out(u_sigFc_100k)
201     );
202
203     logic signed [13:0] u_sigFs_100k;
204     SignedToUnsigned #(15) sign_sine_100k (
205     .signed_in(dec_sigFs_100k),
206     .unsigned_out(u_sigFs_100k)
207     );
208
209     logic signed [13:0] u_sigFc_200k;
210     SignedToUnsigned #(15) sign_cosine_200k (
211     .signed_in(dec_sigFc_200k),
212     .unsigned_out(u_sigFc_200k)
213     );
214
215     logic signed [13:0] u_sigFs_200k;
216     SignedToUnsigned #(15) sign_sine_200k (
217     .signed_in(dec_sigFs_200k),
218     .unsigned_out(u_sigFs_200k)
219     );
220
221     // squaring
222     logic signed [27:0] s_sigFc_100k;
223     u_multiplier #(14, 14, 28) multiplier_cosine_100k (
```

```
224             .multiplicand(u_sigFc_100k),
225             .multiplier(u_sigFc_100k),
226             .product(s_sigFc_100k)
227     );
228
229     logic signed [27:0] s_sigFs_100k;
230     u_multiplier #(14, 14, 28) multiplier_sine_100k (
231             .multiplicand(u_sigFs_100k),
232             .multiplier(u_sigFs_100k),
233             .product(s_sigFs_100k)
234     );
235
236     logic signed [27:0] s_sigFc_200k;
237     u_multiplier #(14, 14, 28) multiplier_cosine_200k (
238             .multiplicand(u_sigFc_200k),
239             .multiplier(u_sigFc_200k),
240             .product(s_sigFc_200k)
241     );
242
243     logic signed [27:0] s_sigFs_200k;
244     u_multiplier #(14, 14, 28) multiplier_sine_200k (
245             .multiplicand(u_sigFs_200k),
246             .multiplier(u_sigFs_200k),
247             .product(s_sigFs_200k)
248     );
249
250     // sqrt
251     logic [13:0] sigF_100k;
252     sqrt #(28) sqrt_100k (
253     .clk(clk),
254     .start(VOUT_square),
255     .rst_n(rst_n),          // Active low reset
256     .D(s_sigFc_100k+s_sigFs_100k),      // Input number
257     .Q_out(sigF_100k),  // Integer square root
258     .R_out(),    // Remainder
259     .VOUT(VOUT_sqrt)              // Output validation signal
260     );
261
262     logic [13:0] sigF_200k;
263     sqrt #(28) sqrt_200k (
264     .clk(clk),
265     .start(VOUT_square),
266     .rst_n(rst_n),          // Active low reset
267     .D(s_sigFc_200k+s_sigFs_200k),      // Input number
268     .Q_out(sigF_200k),  // Integer square root
269     .R_out(),    // Remainder
270     .VOUT()              // Output validation signal
271     );
272
```

```
273     ////////////////////////////////////////// RAM interface
        between filtering and FFT ////////////////////////////////
274     logic [13:0] fill_SR [1:0];
275
276     shift_register #(14, 2) shift_reg_inst (
277     .data_in(sigF_100k),        // N-bit input data to be shifted in
278     .clk(clk),              // Clock signal
279     .rst_n(rst_n),          // Asynchronous active low reset
280     .shift_en(VOUT_sqrt),       // Shift enable signal
281     .data_out(),       // N-bit shift register output
282     .parallel_out(fill_SR) // Parallel output of all register
        contents
283     );
284
285     logic [6:0] fill_cnt; // must fill first 150/2=75 positions
286     logic fill_cnt_en;
287     counter #(7, 75) filling_cnt_inst (
288     .clk(clk),          // Clock signal
289     .rst_n(rst_n),          // Asynchronous active low reset
290     .srst(1'b0),            // Asynchronous reset
291     .en(fill_cnt_en),           // Enable signal
292     .count(fill_cnt) // Counter output
293     );
294
295     logic w_en_1, w_en_2;
296     logic aw_en_1, aw_en_2;
297     logic gain_reg_en;
298
299     logic [11:0] r_addr_1, r_addr_2, w_addr_1, w_addr_2;
300     logic [39:0] r_data_1, r_data_2, w_data_1, w_data_2;
301     logic [11:0] ar_addr_1, ar_addr_2, aw_addr_1, aw_addr_2;
302     logic [39:0] ar_data_1, ar_data_2, aw_data_1, aw_data_2;
303
304     logic [10:0] FFT_cnt;
305     logic FFT_cnt_en;
306     logic VOUT_FFT;
307
308     logic [39:0] amplified_1;
309     logic [39:0] amplified_2;
310     logic signed [39:0] s_amplified_1;
311     logic signed [39:0] s_amplified_2;
312
313     logic [39:0] A_prime, B_prime;
314     logic FFT_stage_en;
315     logic [11:0] ing1_addr, ing2_addr;
316     logic [3:0] stage_cnt;
317     logic conv_cnt_en;
318
319     logic [19:0] Gr, Gi;
```

```
320     logic [39:0] G;
321     assign G = {Gr, Gi};
322 
323     logic [11:0] conv_cnt;
324     FFT_FSM FFT_FSM_inst(
325     .clk(clk),          // Clock signal
326     .rst_n(rst_n),      // Reset signal
327     .VIN(VOUT_sqrt),          // Input signal to control state
        transitions
328     .fill_cnt(fill_cnt),
329     .fill_SR({fill_SR[1],fill_SR[0]}),
330 
331 
332     .w_addr_1(w_addr_1),
333     .w_data_1(w_data_1),
334     .w_addr_2(w_addr_2),
335     .w_data_2(w_data_2),
336     .r_addr_1(r_addr_1),
337     .r_addr_2(r_addr_2),
338 
339     .aw_addr_1(aw_addr_1),
340     .aw_data_1(aw_data_1),
341     .aw_addr_2(aw_addr_2),
342     .aw_data_2(aw_data_2),
343     .ar_addr_1(ar_addr_1),
344     .ar_addr_2(ar_addr_2),
345 
346     .s_amplified_1(s_amplified_1),
347     .s_amplified_2(s_amplified_2),
348     .w_en_1(w_en_1),
349     .w_en_2(w_en_2),
350     .aw_en_1(aw_en_1),
351     .aw_en_2(aw_en_2),
352 
353     .FFT_cnt(FFT_cnt),
354     .FFT_cnt_en(FFT_cnt_en),
355     .gain_reg_en(gain_reg_en),
356     .fill_cnt_en(fill_cnt_en),
357     .VOUT_FFT(VOUT_FFT),
358     .FFT_cnt_srst(FFT_cnt_srst),
359     .FFT_stage_en(FFT_stage_en),
360     .A_prime(A_prime),
361     .B_prime(B_prime),
362     .ing1_addr(ing1_addr),
363     .ing2_addr(ing2_addr),
364     .stage_cnt(stage_cnt),
365     .conv_cnt_en(conv_cnt_en),
366     .G(G),
367     .conv_cnt(conv_cnt)
```

```
368     );
369
370     dual_port_ram #(40, 12) ram_inst (
371     .r_addr_1(r_addr_1),
372     .r_addr_2(r_addr_2),
373     .w_addr_1(w_addr_1),
374     .w_data_1(w_data_1),
375     .w_en_1(w_en_1),
376     .w_addr_2(w_addr_2),
377     .w_data_2(w_data_2),
378     .w_en_2(w_en_2),
379     .clk(clk),
380     .rst_n(rst_n),
381     .r_data_1(r_data_1),
382     .r_data_2(r_data_2)
383     );
384
385     // searching biggest value to decide a variable gain
386     logic [13:0] current_max, next_max;
387     register #(14) max_reg (
388     .data_in(next_max),
389     .clk(clk),
390     .rst_n(rst_n),
391     .srst(1'b0),
392     .reg_en(VOUT_sqrt),
393     .data_out(current_max)
394     );
395
396     comparator #(14) max_comparator (
397     .a(sigF_100k),
398     .b(current_max),
399     .max(next_max)
400     );
401
402     // selecting the gain (as a power of two) by counting the leading
        zeros of the biggest number
403     logic [3:0] gain, leading_zeros;
404     register #(4) gain_reg (
405     .data_in(leading_zeros),
406     .clk(clk),
407     .rst_n(rst_n),
408     .srst(1'b0),
409     .reg_en(gain_reg_en),
410     .data_out(gain)
411     );
412
413     leading_zeros_counter #(14) lz_counter (
414         .in_num(current_max),
415         .leading_zeros(leading_zeros)
```

```
416     );
417
418     // amplifying by removing the leading zeros
419
420
421     amplifier #(40, 4) PGA_1 (
422         .in_num(r_data_1),
423         .gain(gain),
424         .shifted_num(amplified_1)
425     );
426
427     amplifier #(40, 4) PGA_2 (
428         .in_num(r_data_2),
429         .gain(gain),
430         .shifted_num(amplified_2)
431     );
432
433     ////////////////////////////////////////////// Back to signed
        numbers ////////////////////////////////////////////////
434
435
436     assign s_amplified_1 = {amplified_1[39:20]-20'
        b10000000000000000000, amplified_1[19:0]-20'b10000000000000000000
        };
437
438     assign s_amplified_2 = {amplified_2[39:20]-20'
        b10000000000000000000, amplified_2[19:0]-20'b10000000000000000000
        };
439
440
        ////////////////////////////////////////////////////////////////////////////////
441
442     counter #(11, 2**11) FFT_cnt_inst ( // it's 2^13 samples
        including padding, so 2^12 being the sequence real, so 2^11
        butterflies
443     .clk(clk),          // Clock signal
444     .rst_n(rst_n),        // Asynchronous active low reset
445     .srst(FFT_cnt_srst),          // Asynchronous reset
446     .en(FFT_cnt_en),           // Enable signal
447     .count(FFT_cnt) // Counter output
448     );
449
450     ////////////////////////////////////// FFT
        ////////////////////////////////
451
452
453     counter #(4, 12) FFT_stage_cnt ( // it's 2^13 samples including
        padding, so 2^12 being the sequence real, so 2^11 butterflies
```

```
454     .clk(clk),          // Clock signal
455     .rst_n(rst_n),        // Asynchronous active low reset
456     .srst(1'b0),          // Asynchronous reset
457     .en(FFT_stage_en),          // Enable signal
458     .count(stage_cnt) // Counter output
459     );
460
461
462     logic [10:0] twiddle_addr;
463     // Instantiate the ing_addr_generator module
464     ing_addr_generator ing_addr_generator_inst (
465     .FFT_cnt(FFT_cnt),
466     .stage_cnt(stage_cnt),
467     .ing1_addr(ing1_addr),
468     .ing2_addr(ing2_addr),
469     .twiddle_addr(twiddle_addr)
470     );
471
472     logic [27:0] twiddle;
473     logic [19:0] Ar_prime, Ai_prime, Br_prime, Bi_prime;
474     logic first_stage;
475
476     assign first_stage = stage_cnt == '0;
477     // Instantiate the butterfly module
478     butterfly #(20) butt_inst (
479     .Ar(r_data_1[39:20]),
480     .Ai(r_data_1[19:0]),
481     .Br(r_data_2[39:20]),
482     .Bi(r_data_2[19:0]),
483     .Wr({twiddle[27:14],6'b0}),
484     .Wi({twiddle[13:0],6'b0}),
485     .Ar_prime(Ar_prime),
486     .Ai_prime(Ai_prime),
487     .Br_prime(Br_prime),
488     .Bi_prime(Bi_prime),
489     .first_stage(first_stage)
490     );
491
492     assign A_prime = {Ar_prime, Ai_prime};
493     assign B_prime = {Br_prime, Bi_prime};
494     // twiddle rom
495
496
497     twiddle_rom twiddle_rom_inst (
498     .r_addr(twiddle_addr),
499     .clk(clk),
500     .rst_n(rst_n),
501     .r_data(twiddle)
502     );
```

```
503
504     ////////////////////// converting to the solution of the full
        signal ///////////
505     // k1 rom
506
507     logic [27:0] k1;
508     logic [11:0] k1_addr;
509     assign k1_addr = conv_cnt;
510     k1_rom k1_rom_inst (
511     .r_addr(k1_addr),
512     .clk(clk),
513     .rst_n(rst_n),
514     .r_data(k1)
515     );
516
517     //k2 rom
518
519     logic [27:0] k2;
520     logic [11:0] k2_addr;
521     assign k2_addr = conv_cnt;
522     k2_rom k2_rom_inst (
523     .r_addr(k2_addr),
524     .clk(clk),
525     .rst_n(rst_n),
526     .r_data(k2)
527     );
528
529
530     counter #(12, 2**12) conv_cnt_inst (
531     .clk(clk),          // Clock signal
532     .rst_n(rst_n),        // Asynchronous active low reset
533     .srst(1'b0),          // Asynchronous reset
534     .en(conv_cnt_en),           // Enable signal
535     .count(conv_cnt) // Counter output
536     );
537
538     // Instantiate the converter module
539     converter #(20) conv_inst (
540     .X1r(ar_data_1[39:20]),
541     .X1i(ar_data_1[19:0]),
542     .X2r(ar_data_2[39:20]),
543     .X2i(ar_data_2[19:0]),
544     .k1r({k1[27:14], 6'b0}),
545     .k1i({k1[13:0], 6'b0}),
546     .k2r({k2[27:14], 6'b0}),
547     .k2i({k2[13:0], 6'b0}),
548     .Gr(Gr),
549     .Gi(Gi)
550     );
```

```
551
552     dual_port_ram #(40, 12) aux_ram_inst (
553     .r_addr_1(ar_addr_1),
554     .r_addr_2(ar_addr_2),
555     .w_addr_1(aw_addr_1),
556     .w_data_1(aw_data_1),
557     .w_en_1(aw_en_1),
558     .w_addr_2(aw_addr_2),
559     .w_data_2(aw_data_2),
560     .w_en_2(aw_en_2),
561     .clk(clk),
562     .rst_n(rst_n),
563     .r_data_1(ar_data_1),
564     .r_data_2(ar_data_2)
565     );
566
567     assign VOUT = VOUT_FFT;
568     //assign DOUT = {aw_addr_1, aw_addr_2, aw_data_1, aw_data_2};
569     assign DOUT = {w_data_1};
570
571 endmodule
```

Listing B.2: cheb_lpf3_lkhd.sv

```
1  module cheb_lpf3_lkhd (
2      input logic signed [13:0] DIN,
3      input logic VIN,
4      input logic rst_n,
5      input logic clk,
6
7      input logic signed [13:0] b_coefficients [1:0][6:0],
8      input logic signed [31:0] a_coefficients [1:0][6:2],
9
10     output logic signed [36:0] DOUT,
11     output logic VOUT
12 );
13
14 logic [13:0] ff_buffer [5:0];
15 logic [36:0] fb_buffer [5:0];
16 logic [36:0] y;
17
18 logic odd_evenn;
19 logic [26:0] mult_results_b [6:0];
20 logic [41:0] mult_results_a [6:2];
21
22 register #(1) VOUT_reg (
23     .data_in(VIN),
24     .clk(clk),
25     .rst_n(rst_n),
```

```
26      .srst(1'b0),
27      .reg_en(1'b1),
28      .data_out(VOUT)
29  );
30
31  register #(37) DOUT_reg (
32      .data_in(y),
33      .clk(clk),
34      .rst_n(rst_n),
35      .srst(1'b0),
36      .reg_en(VIN),
37      .data_out(DOUT)
38  );
39
40  shift_register #(14, 6) ff_buffer_sreg (
41      .data_in(DIN),        // N-bit input data to be shifted in
42      .clk(clk),            // Clock signal
43      .rst_n(rst_n),      // Asynchronous active low reset
44      .shift_en(VIN),       // Shift enable signal
45      .data_out(),
46      .parallel_out(ff_buffer)
47  );
48
49  shift_register #(37, 6) fb_buffer_sreg (
50      .data_in(y),        // N-bit input data to be shifted in
51      .clk(clk),            // Clock signal
52      .rst_n(rst_n),      // Asynchronous active low reset
53      .shift_en(VIN),       // Shift enable signal
54      .data_out(),
55      .parallel_out(fb_buffer)
56  );
57
58  counter #(1, 2) even_oddn_cnt (
59      .clk(clk),          // Clock signal
60      .rst_n(rst_n),        // Asynchronous active low reset
61      .srst(1'b0),          // Asynchronous reset
62      .en(VIN),           // Enable signal
63      .count(odd_evenn) // Counter output
64  );
65
66  multiplier #(14, 14, 27 ) multiplier (
67              .multiplicand(DIN), // Multiplicand from ff_buffer
68              .multiplier(b_coefficients[odd_evenn][0]), // Multiplier
        from b_coefficients
69              .product(mult_results_b[0]) // Product stored in
        mult_results_b
70          );
71
72  // Perform multiplication of ff_buffer entries with b_coefficients
```

```
73 genvar i;
74 generate
75     for (i = 0; i < 6; i++) begin : multiply_loop_b
76         multiplier #(14, 14, 27) multiplier_b (
77             .multiplicand(ff_buffer[i]), // Multiplicand from
    ff_buffer
78             .multiplier(b_coefficients[odd_evenn][i+1]), //
    Multiplier from b_coefficients
79             .product(mult_results_b[i+1]) // Product stored in
    mult_results_b
80         );
81     end
82 endgenerate
83
84 genvar j;
85 generate
86     for (j = 1; j < 6; j++) begin : multiply_loop_a
87         multiplier #(37, 32, 42) multiplier_a (
88             .multiplicand(fb_buffer[j]), // Multiplicand from
    ff_buffer
89             .multiplier(a_coefficients[odd_evenn][j+1]), //
    Multiplier from b_coefficients
90             .product(mult_results_a[j+1]) // Product stored in
    mult_results_b
91         );
92     end
93 endgenerate
94
95
96 // Instantiate the ff adders
97 logic [24:0] a1;
98 assign a1=mult_results_b[6][24:0];
99
100 logic [24:0] b1;
101 assign b1=mult_results_b[5][24:0];
102
103 logic [24:0] sum1;
104
105 adder #(25) adder_inst1 (
106     .a(a1),        // Connect a to input a of the adder
107     .b(b1),        // Connect b to input b of the adder
108     .sum(sum1)     // Connect sum to output sum of the adder
109 );
110
111 logic [26:0] a2;
112 assign a2={{2{sum1[24]}},sum1};
113
114 logic [26:0] b2;
115 assign b2=mult_results_b[4][26:0];
```

```
116
117 logic [26:0] sum2;
118
119 adder #(27) adder_inst2 (
120     .a(a2),         // Connect a to input a of the adder
121     .b(b2),         // Connect b to input b of the adder
122     .sum(sum2)      // Connect sum to output sum of the adder
123 );
124
125 logic [27:0] a3;
126 assign a3={{1{sum2[26]}},sum2};
127
128 logic [27:0] b3;
129 assign b3={{1{mult_results_b[3][26]}},mult_results_b[3][26:0]};
130
131 logic [27:0] sum3;
132
133 adder #(28) adder_inst3 (
134     .a(a3),         // Connect a to input a of the adder
135     .b(b3),         // Connect b to input b of the adder
136     .sum(sum3)      // Connect sum to output sum of the adder
137 );
138
139 logic [28:0] a4;
140 assign a4={{1{sum3[27]}},sum3};
141
142 logic [28:0] b4;
143 assign b4={{2{mult_results_b[2][26]}},mult_results_b[2][26:0]};
144
145 logic [28:0] sum4;
146
147 adder #(29) adder_inst4 (
148     .a(a4),         // Connect a to input a of the adder
149     .b(b4),         // Connect b to input b of the adder
150     .sum(sum4)      // Connect sum to output sum of the adder
151 );
152
153 logic [28:0] a5;
154 assign a5=sum4;
155
156 logic [28:0] b5;
157 assign b5={{2{mult_results_b[1][26]}},mult_results_b[1][26:0]};
158
159 logic [28:0] sum5;
160
161 adder #(29) adder_inst5 (
162     .a(a5),         // Connect a to input a of the adder
163     .b(b5),         // Connect b to input b of the adder
164     .sum(sum5)      // Connect sum to output sum of the adder
```

```
165 );
166
167 logic [28:0] a6;
168 assign a6=sum5;
169
170 logic [28:0] b6;
171 assign b6={{2{mult_results_b[0][26]}},mult_results_b[0][26:0]};
172
173 logic [28:0] sum6;
174
175 adder #(29) adder_inst6 (
176     .a(a6),        // Connect a to input a of the adder
177     .b(b6),        // Connect b to input b of the adder
178     .sum(sum6)     // Connect sum to output sum of the adder
179 );
180
181 logic [25:0] ff;
182 half_up_rounding #(29,3) ff_round (
183     .in(sum6),  // Original number
184     .out(ff)  // Rounded number
185 );
186
187 // Instantiate the fb adders
188 logic [39:0] a7;
189 assign a7=mult_results_a[6][39:0];
190
191 logic [39:0] b7;
192 assign b7=mult_results_a[5][39:0];
193
194 logic [39:0] sum7;
195
196 adder #(40) adder_inst7 (
197     .a(a7),        // Connect a to input a of the adder
198     .b(b7),        // Connect b to input b of the adder
199     .sum(sum7)     // Connect sum to output sum of the adder
200 );
201
202 logic [40:0] a8;
203 assign a8={{1{sum7[39]}},sum7};
204
205 logic [40:0] b8;
206 assign b8=mult_results_a[4][40:0];
207
208 logic [40:0] sum8;
209
210 adder #(41) adder_inst8 (
211     .a(a8),        // Connect a to input a of the adder
212     .b(b8),        // Connect b to input b of the adder
213     .sum(sum8)     // Connect sum to output sum of the adder
```

```
214 );
215
216 logic [41:0] a9;
217 assign a9={{1{sum8[40]}},sum8};
218
219 logic [41:0] b9;
220 assign b9={{1{mult_results_a[3][40]}},mult_results_a[3][40:0]};
221
222 logic [41:0] sum9;
223
224 adder #(42) adder_inst9 (
225     .a(a9),        // Connect a to input a of the adder
226     .b(b9),        // Connect b to input b of the adder
227     .sum(sum9)     // Connect sum to output sum of the adder
228 );
229
230 logic [41:0] a10;
231 assign a10={{1{sum9[40]}},sum9};
232
233 logic [41:0] b10;
234 assign b10={{2{mult_results_a[2][40]}},mult_results_a[2][40:0]};
235
236 logic [41:0] sum10;
237
238
239 adder #(42) adder_inst10 (
240     .a(a10),        // Connect a to input a of the adder
241     .b(b10),        // Connect b to input b of the adder
242     .sum(sum10)     // Connect sum to output sum of the adder
243 );
244
245 // Instantiate the final subtractor
246
247 logic [41:0] a11;
248 assign a11={{11{ff[25]}},ff,5'b00000};
249
250 logic [41:0] b11;
251 assign b11=sum10;
252
253 logic [41:0] diff11;
254
255
256 subtractor #(42) subtractor_inst11 (
257     .a(a11),        // Connect a to input a of the adder
258     .b(b11),        // Connect b to input b of the adder
259     .diff(diff11)     // Connect sum to output sum of the adder
260 );
261
262 // Instantiate the saturation module
```

```
saturation #(
    .input_N(42),
    .output_N(37)
) saturation_inst (
    .in(diff11),  // Connect the unsaturated output
    .out(y)              // Connect the saturated output
);

endmodule
```

Listing B.3: mixer_cosine_100k.sv

```
module mixer_cosine_100k (
    input logic clk,
    input logic VIN,
    input logic rst_n,
    input logic signed [13:0] input_signal,
    output logic signed [13:0] mixed_signal
);

    // Sine lookup table
    const logic signed [13:0] sine_lut [0:4] = '{13'd4426,
                                                 -13'd5188,
                                                 -13'd7633,
                                                 13'd471,
                                                 13'd7924};

    // Index for the sine LUT
    logic [2:0] index;

    counter #(3, 5) cnt_100k (
    .clk(clk),          // Clock signal
    .rst_n(rst_n),        // Asynchronous active low reset
    .srst(1'b0),          // Asynchronous reset
    .en(VIN),           // Enable signal
    .count(index)  // Counter output
    );

    // Mix the input signal with the sine wave
    multiplier #(14, 14, 14) multiplier_100k (
     .multiplicand(input_signal), // Multiplicand from ff_buffer
     .multiplier(sine_lut[index]), // Multiplier from b_coefficients
     .product(mixed_signal) // Product stored in mult_results_b
    );


endmodule
```

Listing B.4: sqrt.sv

```
module sqrt #(parameter int N = 6) (  // Default bit-width parameter
    input logic clk,
    input logic start,
    input logic rst_n,          // Active low reset
    input logic [N-1:0] D,      // Input number
    output logic [N/2-1:0] Q_out,  // Integer square root
    output logic [N-1:0] R_out,    // Remainder
    output logic VOUT           // Output validation signal
);

    // Internal registers
    logic [3:0] i;
    logic [1:0] D_ctrl;
    logic [N-1:0] D_reg;
    logic en;

    // For a 28 bit number, (28/2+1)=15 clk cycles are needed
    down_counter #(4, N/2) i_cnt (
    .clk(clk),          // Clock signal
    .rst_n(rst_n),        // Asynchronous active low reset
    .en(en),           // Enable signal
    .count(i) // Counter output
    );

    SR_FF en_FF (
    .set(start),
    .reset(VOUT),
    .clk(clk),
    .rst_n(rst_n),
    .reg_en(1'b1),
    .Q(en)
    );

    register #(N) D_reg_inst (
    .data_in(D),
    .clk(clk),
    .rst_n(rst_n),
    .srst(1'b0),
    .reg_en(start),
    .data_out(D_reg)
    );

    logic [N/2:0] F;
    logic [N/2:0] F_reg;
    register #(N/2+1) F_reg_inst (
    .data_in(F),
    .clk(clk),
    .rst_n(rst_n),
```

```
49      .srst(VOUT),
50      .reg_en(en),
51      .data_out(F_reg)
52      );
53
54      logic [N/2:0] Q_buff;
55      logic [N/2:0] Q_buff_reg;
56      register #(N/2+1) Q_buff_reg_inst (
57      .data_in(Q_buff),
58      .clk(clk),
59      .rst_n(rst_n),
60      .srst(VOUT),
61      .reg_en(en),
62      .data_out(Q_buff_reg)
63      );
64
65      logic [N/2:0] Q;
66      logic [N/2:0] Q_reg;
67      register #(N/2+1) Q_reg_inst (
68      .data_in(Q),
69      .clk(clk),
70      .rst_n(rst_n),
71      .srst(VOUT),
72      .reg_en(en),
73      .data_out(Q_reg)
74      );
75
76      logic [N-1:0] R;
77      logic [N-1:0] R_reg;
78      register #(N) R_reg_inst (
79      .data_in(R),
80      .clk(clk),
81      .rst_n(rst_n),
82      .srst(VOUT),
83      .reg_en(en),
84      .data_out(R_reg)
85      );
86
87      // VOUT
88      assign VOUT= (i == '0);
89
90      logic [N/2:0] F_cond;
91      logic [N-1:0] R_cond;
92
93      // Combinational block to assign D_ctrl
94      always_comb begin
95          D_ctrl = D_reg[2*i + 1 -: 2];
96          R_cond = {R_reg[N-3:0], D_ctrl};
97          F_cond = {F_reg[N/2-1:0], 1'b1};
```

```
98          if (R_cond >= F_cond) begin
99              R = R_cond-F_cond;
100             Q = {Q_buff_reg[N/2-1:1], 1'b1};
101             Q_buff = {Q_buff_reg[N/2-2:1], 1'b1, 1'b0};
102             F={F_reg, 1'b1}+1'b1;
103         end else begin
104             R = R_cond;
105             Q = Q_buff_reg;
106             Q_buff = {Q_buff_reg[N/2-2:0], 1'b0};
107             F={F_reg, 1'b0};
108         end
109     end
110
111     assign Q_out = Q[N/2-1:0];
112     assign R_out = R;
113
114 endmodule
```

Listing B.5: dual_port_ram.sv

```
1     module dual_port_ram #(parameter int data_bit = 14*2, parameter
      int addr_bit = 12)(
2     input logic [addr_bit-1:0] r_addr_1,
3     input logic [addr_bit-1:0] r_addr_2,
4     input logic [addr_bit-1:0] w_addr_1,
5     input logic [addr_bit-1:0] w_addr_2,
6     input logic [data_bit-1:0] w_data_1,
7     input logic [data_bit-1:0] w_data_2,
8     input logic w_en_1,
9     input logic w_en_2,
10    input logic clk,
11    input logic rst_n,
12    output logic [data_bit-1:0] r_data_1,
13    output logic [data_bit-1:0] r_data_2
14 );
15
16 logic [data_bit-1:0] ram_matrix [(2**addr_bit)-1:0];
17
18 always_ff @(posedge clk or negedge rst_n) begin
19    // asynchronous reset
20    if (!rst_n) begin
21        // the values for all registers are resetted to 'b0
      considering a for statement
22        integer rst_addr;
23        for (rst_addr = 0; rst_addr < 2**addr_bit; rst_addr++) begin
24            ram_matrix[$unsigned(rst_addr)] = 0;
25        end
26    // @ clock posedge if reset is not active
27    end else begin
```

```
28          // handle the write operation, if the request is for
        addr_force_0 then it's ignored
29          if (w_en_1) begin
30              ram_matrix[$unsigned(w_addr_1)] = w_data_1;
31          end
32          if (w_en_2) begin
33              ram_matrix[$unsigned(w_addr_2)] = w_data_2;
34          end
35      end
36  end
37
38
39  always_comb begin
40
41      r_data_1 <= ram_matrix[$unsigned(r_addr_1)];
42      r_data_2 <= ram_matrix[$unsigned(r_addr_2)];
43
44  end
45
46  endmodule
```

Listing B.6: butterfly.sv

```
1  module butterfly #(parameter N = 14) (
2      input  logic signed [N-1:0] Ar, Ai, Br, Bi, Wr, Wi,
3      input  logic first_stage,
4      output logic signed [N-1:0] Ar_prime, Ai_prime, Br_prime,
       Bi_prime
5  );
6
7  logic signed [2*N:0] Ar_temp, Ai_temp, Br_temp, Bi_temp; //
       intermediate results
8  parameter N_reduced = N-1;
9
10 // Sign-extend inputs before arithmetic operations
11 logic signed [N+1:0] Ar_ext, Ai_ext;
12 assign Ar_ext = {{2{Ar[N-1]}}, Ar};
13 assign Ai_ext = {{2{Ai[N-1]}}, Ai};
14
15 // Perform multiplications
16 logic signed [2*N-2:0] BrWr, BiWi, BrWi, BiWr;
17 assign BrWr = Br * Wr;
18 assign BiWi = Bi * Wi;
19 assign BrWi = Br * Wi;
20 assign BiWr = Bi * Wr;
21
22 // Sign-extend inputs before arithmetic operations
23 logic signed [2*N:0] BrWr_ext, BiWi_ext, BrWi_ext, BiWr_ext;
24 assign BrWr_ext = {{2{BrWr[2*N-2]}}, BrWr};
```

```
25 assign BiWi_ext = {{2{BiWi[2*N-2]}}, BiWi};
26 assign BrWi_ext = {{2{BrWi[2*N-2]}}, BrWi};
27 assign BiWr_ext = {{2{BiWr[2*N-2]}}, BiWr};
28
29 // Add multiplication results to temp variables
30 assign Ar_temp = {Ar_ext, {N_reduced{1'b0}}} + BrWr_ext - BiWi_ext;
31 assign Ai_temp = {Ai_ext, {N_reduced{1'b0}}} + BrWi_ext + BiWr_ext;
32 assign Br_temp = {Ar_ext, {N_reduced{1'b0}}} - BrWr_ext + BiWi_ext;
33 assign Bi_temp = {Ai_ext, {N_reduced{1'b0}}} - BrWi_ext - BiWr_ext;
34
35 // Mux to select proper output based on `first_stage`
36 logic signed [2*N:0] Ar_mux, Ai_mux, Br_mux, Bi_mux;
37 assign Ar_mux = first_stage ? Ar_temp : {Ar_temp[2*N-1:0], 1'b0};
38 assign Ai_mux = first_stage ? Ai_temp : {Ai_temp[2*N-1:0], 1'b0};
39 assign Br_mux = first_stage ? Br_temp : {Br_temp[2*N-1:0], 1'b0};
40 assign Bi_mux = first_stage ? Bi_temp : {Bi_temp[2*N-1:0], 1'b0};
41
42 // Rounding modules
43 half_up_rounding #(2*N+1, N+1) Ar_round (
44     .in(Ar_mux),  // Original number
45     .out(Ar_prime)  // Rounded number
46 );
47
48 half_up_rounding #(2*N+1, N+1) Ai_round (
49     .in(Ai_mux),  // Original number
50     .out(Ai_prime)  // Rounded number
51 );
52
53 half_up_rounding #(2*N+1, N+1) Br_round (
54     .in(Br_mux),  // Original number
55     .out(Br_prime)  // Rounded number
56 );
57
58 half_up_rounding #(2*N+1, N+1) Bi_round (
59     .in(Bi_mux),  // Original number
60     .out(Bi_prime)  // Rounded number
61 );
62
63 endmodule
```

Listing B.7: ing_addr_generator.sv

```
1     module ing_addr_generator (
2     input logic [10:0] FFT_cnt,
3     input logic [3:0] stage_cnt,
4     output logic [11:0] ing1_addr,
5     output logic [11:0] ing2_addr,
6     output logic [10:0] twiddle_addr
7 );
```

```
8
9     // Define local variables
10    logic [10:0] mask1, mask1_tmp;
11    logic [10:0] mask2;
12    logic [10:0] twiddle_temp, twiddle_temp2;
13
14    // Assign masks based on input 'i'
15    assign mask1_tmp = ((1 << stage_cnt) - 1); // Mask for the first
      i elements
16
17    always_comb begin
18    for (int i = 0; i < 11; i++) begin
19      mask1[i] = mask1_tmp[10-i];
20    end
21    end
22
23    assign mask2 = ~mask1; // Mask for the last elements from i+1 to
      N-1
24
25
26    // Logic to create modified_FFT_count
27    assign ing1_addr = {(FFT_cnt & mask1), 1'b0} | (0 << 11-stage_cnt
      ) | {1'b0, (FFT_cnt & mask2)};
28    assign ing2_addr = {(FFT_cnt & mask1), 1'b0} | (1 << 11-stage_cnt
      ) | {1'b0, (FFT_cnt & mask2)};
29
30    assign twiddle_temp = (FFT_cnt & mask1);
31
32    always_comb begin
33    for (int i = 0; i < 11; i++) begin
34      twiddle_temp2[i] = twiddle_temp[10-i];
35    end
36    end
37
38    assign twiddle_addr = (twiddle_temp2 << 11-stage_cnt);
39
40 endmodule
```

Bibliography

[1] S. S. Afzal, W. Chen, and F. Adib. «3D-BLUE: Backscatter Localization for Underwater Robotics». In: *IEEE Journal of Oceanic Engineering* 48.1 (2023), pp. 123–134 (cit. on p. 1).

[2] D. Koulouris, A. Menychtas, and I. Maglogiannis. «Augmented Reality for Indoor Localization and Navigation: The Case of UNIPI AR Experience». In: *Lecture Notes in Computer Science*. Vol. 14185. 2023, pp. 233–243 (cit. on p. 1).

[3] Emanuele Grossi, Hedieh Taremizadeh, and Luca Venturino. «Radar Target Detection and Localization Aided by an Active Reconfigurable Intelligent Surface». In: *IEEE Signal Processing Letters* PP (Jan. 2023), pp. 1–5. DOI: 10.1109/LSP.2023.3296372 (cit. on p. 2).

[4] Q. Luo, K. Yang, X. Yan, J. Li, C. Wang, and Z. Zhou. «An Improved Trilateration Positioning Algorithm with Anchor Node Combination and K-Means Clustering». In: *Sensors*. Vol. 22. 16. 2022, p. 6085 (cit. on p. 3).

[5] Wikipedia contributors. *Butterworth filter*. Accessed: 2024-09-11. Sept. 2024. URL: https://en.wikipedia.org/wiki/Butterworth_filter (cit. on p. 15).

[6] Cishen Zhang and Lihua Xie. «Periodic stabilization of look-ahead filters in VLSI implementation». In: *IEEE Transactions on Automatic Control* 47.8 (2002), pp. 1362–1366. DOI: 10.1109/TAC.2002.801201 (cit. on p. 23).

[7] Rachmad Vidya Wicaksana Putra. «A novel fixed-point square root algorithm and its digital hardware design». In: *International Conference on ICT for Smart Society*. 2013, pp. 1–4. DOI: 10.1109/ICTSS.2013.6588110 (cit. on p. 32).

[8] Robert Matusiak. «Implementing Fast Fourier Transform Algorithms of Real-Valued Sequences With the TMS 320 DSP Platform». In: 2002. URL: https://api.semanticscholar.org/CorpusID:11262963 (cit. on p. 35).