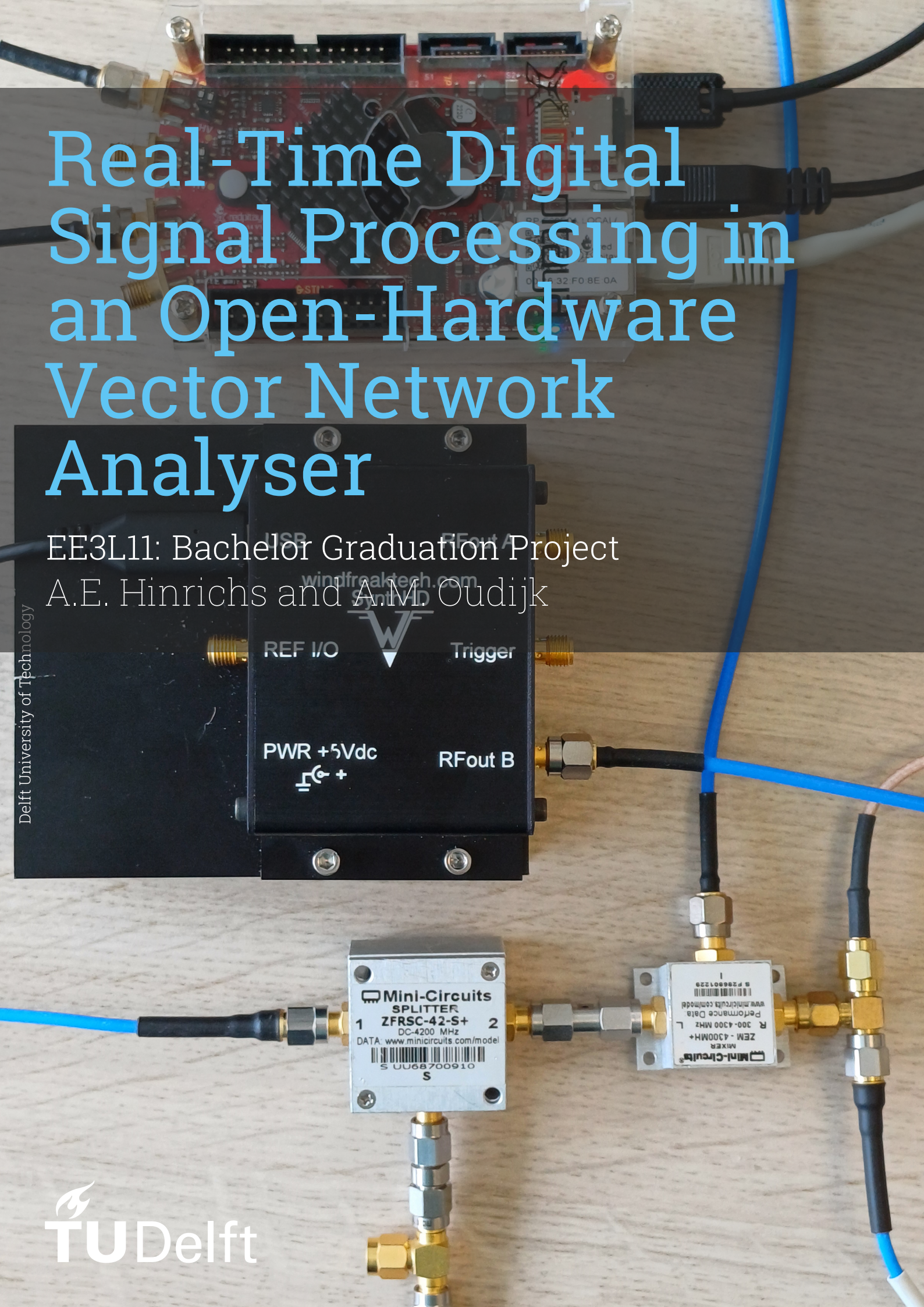

Real-Time Digital Signal Processing in an Open-Hardware Vector Network Analyser

EE3L11: Bachelor Graduation Project

A.E. Hinrichs and A.M. Oudijk

Delft University of Technology

TU Delft

Real-Time Digital Signal Processing in an Open-Hardware Vector Network Analyser

by

A.E. Hinrichs and A.M. Oudijk

in partial fulfillment of the requirements for the degree of

Bachelor of Science

in Electrical Engineering

defended on Friday June 21, 2024 at 10:30 AM.

Students:	A.E. Hinrichs	5161274
	A.M. Oudijk	5595533
Project Supervisor:	Prof.dr. G.A. Steele	TU Delft
EEMCS Supervisor:	Dr.ir. N. Haider	TU Delft
Thesis committee:	Dr.ir. N. Haider	TU Delft
	Prof.dr. G.A. Steele	TU Delft
	Dr.ir. J.S.S.M Wong	TU Delft

Cover:	Red Pitaya STEMlab 125-14, SynthHD V2 by Windfreak Technologies LLC, Mini-Circuits ZFRSC-42-S+ splitter and Mini-Circuits ZEM-4300MH+ Mixer connected by coaxial cables.
Style:	TU Delft Report Style, with modifications by Daan Zwaneveld

TU Delft

Preface

The last two months have been dedicated to working on the bachelor thesis you have in front of you. This project, conceived by Gary Steele, has involved tremendous effort to present a working prototype and to prove the concept.

We want to extend our heartfelt thanks to Gary Steele, Nadia Haider, and Stephan Wong for their invaluable support and guidance, which were instrumental in making this project a success in our eyes. We are proud of how far we have come and how much we have learned over this period. We would also like to thank the Steele Lab group members, who ensured we felt we were a real part of the team in these two months.

Additionally, we want to thank Ruben Dirkzwager, Matthijs Langenberg, Samet Öztürk, and Simon Schaap for their excellent teamwork and collaboration throughout this project.

A.E. Hinrichs and A.M. Oudijk
Delft, July 2024

Abstract

This thesis discusses the digital signal processing involved in building a Vector Network Analyser for qubit readout. Existing VNAs are aggregated and used to construct a programme of requirements for this application. An architecture is constructed and explained, and the stages IQ decomposition and data reduction are analysed mathematically. The Discrete Fourier Transform is used to extract DC signals for this application and its properties are compared to different filters. Common digital logic functions such as AXI, Direct Digital Synthesis, and Direct Memory Access are explained, as well as the implementations of custom blocks for this application such as accumulators and a sequencer. These IP blocks are demonstrated individually and integrated to be implemented on a Red Pitaya STEMLab 125-14 board containing the Zynq 7010 SoC. The implementation is tested using simulated input signals and the resulting measurements are analysed. The implementation is found to have good absolute accuracy of within 2% of expected absolute amplitude, 1% of the expected relative amplitude and 8 mrad of expected relative phase. Modulation of the input signals is tested to work as expected and no major cross-modulation is found. Future improvements are identified and the limitations of the used data reduction are discussed in relation to a Vector Signal Analyser mode.

Contents

Preface i

Abstract ii

Table of Contents iv

Nomenclature v

1 Introduction[2] **1**
- 1.1 VNA, a general overview . . . 1
- 1.2 Application in quantum research . . . 2
- 1.3 Existing solutions . . . 4
- 1.4 Functional requirements . . . 4
- 1.5 Materials . . . 5
- 1.6 Problem definition . . . 5
- 1.7 Thesis overview . . . 6

2 Programme of Requirements **7**
- 2.1 Functional requirements . . . 7
- 2.2 Non-functional requirements . . . 8

3 Design Process **9**
- 3.1 Theory of operation . . . 9
  - 3.1.1 General overview . . . 9
  - 3.1.2 First architecture . . . 10
  - 3.1.3 Second architecture . . . 11
- 3.2 Signal processing . . . 11
  - 3.2.1 Phase and amplitude, I and Q . . . 11
  - 3.2.2 Data reduction . . . 14
- 3.3 Red Pitaya and programmable logic . . . 17
  - 3.3.1 Red Pitaya . . . 19
  - 3.3.2 Protocols and pre-made blocks . . . 20
  - 3.3.3 Custom blocks . . . 24

4 Implementation and Validation **27**
- 4.1 Implementation and validation of individual components . . . 27
  - 4.1.1 Sequencer . . . 27
  - 4.1.2 Accumulator . . . 27
  - 4.1.3 Packetiser . . . 28
  - 4.1.4 FIFO . . . 28
  - 4.1.5 DMA . . . 28
- 4.2 Integration . . . 29
- 4.3 Validation of integrated system . . . 29
  - 4.3.1 Measurement 1 . . . 30
  - 4.3.2 Measurement 2 . . . 30
  - 4.3.3 Measurement 3 . . . 31

5 Discussion **33**
- 5.1 Shortcomings . . . 33
- 5.2 Future work . . . 33

[2]This chapter is shared between the three theses written by the three subteams of the project.

6 Conclusion **34**

References **35**

A Verilog Source Code **37**
A.1 Accumulator . . . 37
A.2 Accumulator trigger . . . 38
A.3 Accumulator and trigger testbench . . . 40
A.4 Sequencer . . . 42
A.5 Sequencer testbench . . . 44
A.6 Packetiser . . . 45
A.7 Packetiser testbench and top-level module . . . 47
A.8 Sample data generator for packetiser testbench . . . 49
A.9 FIFO filler . . . 50

B Implementation Block Diagrams **52**

C Implementation Testing and Results **57**
C.1 Testing parameters and code . . . 57
C.2 Test results . . . 59

Nomenclature

Abbreviations and Names

Abbreviation	Definition
ADC	Analog to Digital Converter
AMBA	Arm Advanced Microcontroller Bus Architecture
AXI	Advanced eXtensible Interface
C	A programming language
CIC/CCI	Cascaded Integrating-Comb / Cascaded Comb-Integrating (filter)
CoM	Centre of Mass
CPU	Central Processing Unit
DAC	Digital to Analog Converter
DC	Direct Current, 0 Hz frequency component
DDR	Double Data Rate
DDS	Direct Digital Synthesis
DFT	Discrete Fourier Transform
DMA	Direct Memory Access
DSP	Digital Signal Processing
DTFT	Discrete-Time Fourier Transform
DUT	Device Under Test, used for both the device and signal affected by it
EE	Electrical Engineering
EMW	ElectroMagnetic Wave
FIFO	First In First Out, type of queue
FIR	Finite Impulse Response (filter)
FPGA	Chip containing PL
GPIO	General Purpose Input/Output, exposed on physical pins
IF	Intermediate Frequency
IFBW	Intermediate Frequency BandWidth
IIR	Infinite Impulse Response (filter)
IP	Intellectual Property, functional block implemented in PL
iPython	Interactive variant of Python
IQ	In-phase and Quadrature components
LO	Local Oscillator
LPF	Low-Pass Filter
LUT	LookUp Table
MMIO	Memory-Mapped Input/Output, may be internal to PL
PL	Programmable Logic
PLL	Phase Locked Loop
PoR	Program of Requirements
PS	Processing System
PYNQ	Python library used for communication between PL and PS
Python	A programming language
RAM	Random Access Memory

Abbreviation	Definition
Red Pitaya	Red Pitaya STEMlab 125-14 development board
REF	REFerence device, used for both the device and the signal affected by it
RF	Radio Frequency
S21	Transmission/S parameter S_{21}
SDR	Software-Defined Radio
SMA	Sub-Miniature version A, RF connector
SNR	Signal-to-Noise Ratio
SoC	Chip containing both a PS and PL
SQUID	Superconducting QUantum Interference Device
USB	Universal Serial Bus
Verilog	A hardware description language
Vivado	Software used to implement Verilog modules in PL
VNA	Vector Network Analyser
VSA	Vector Signal Analyser
Xilinx	Company behind Zynq and Vivado
Zynq	Series of SoC

1

Introduction[1]

1.1. VNA, a general overview

A Vector Network Analyser (VNA) is a device that sends an electromagnetic wave (EMW) at a known frequency and amplitude through a Device under Test (DuT) or network, and records the reflected and transmitted waves[1]. The recorded waves are compared to the stimulus wave to derive a vector output, giving the change in amplitude and phase caused by the DuT.

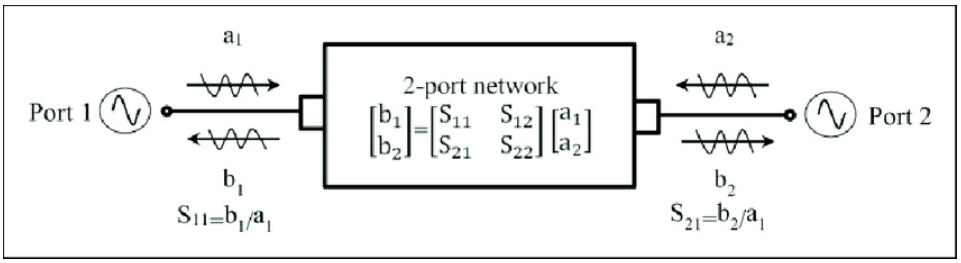

Figure 1.1: S-parameters [2]

The reflected EMW, transmitted EMW and the EMW that is sent by the VNA, which from now on can be referred to as the reference signal, can be represented using two sinusoidal waves: an in-phase cosine (I), and a sine, shifted by 90 degrees compared to I, referred to as the quadrature wave (Q). These waves are combined to form a complex mathematical IQ representation: $I + \mathrm{j}Q$.

The change caused by a DuT in its reflection and transmission of the reference signal are quantified by scattering parameters, or S-parameters, which are a form of network parameters. For a two-port DuT, the S-parameters can be put inside a 2×2-matrix [3], shown in figure 1.1. These parameters contain information about both the phase and amplitude change caused by the DuT, in a complex form. They are obtained by complex division of the reflected or transmitted signal by the reference signal, such as in equation (1.1).

$$S_{21} = \frac{b_2}{a_1} = \frac{I_{\text{trans}} + \mathrm{j}Q_{\text{trans}}}{I_{\text{ref}} + \mathrm{j}Q_{\text{ref}}} \tag{1.1}$$

For this project, this S_{21} transmission parameter is of interest, which relates the transmitted signal (b_2 in figure 1.1) to the reference signal (a_1).

VNAs have two main procedures to test a DuT. The first procedure is called frequency sweep, where an EMW is sent with a constant power and a frequency changing over a short time span in predefined steps. This procedure is used to determine the frequency dependence of the reflection and transmission parameters of the DuT. The second procedure is a power sweep, where an EMW is sent with

[1] This chapter is shared between the three theses written by the three subteams of the project.

constant frequency and a power changing over a short time span. This procedure is used to determine the power transfer of the DuT at different input powers. For this project, only the frequency sweep is of interest, and implementation of power sweeping is left to future projects.

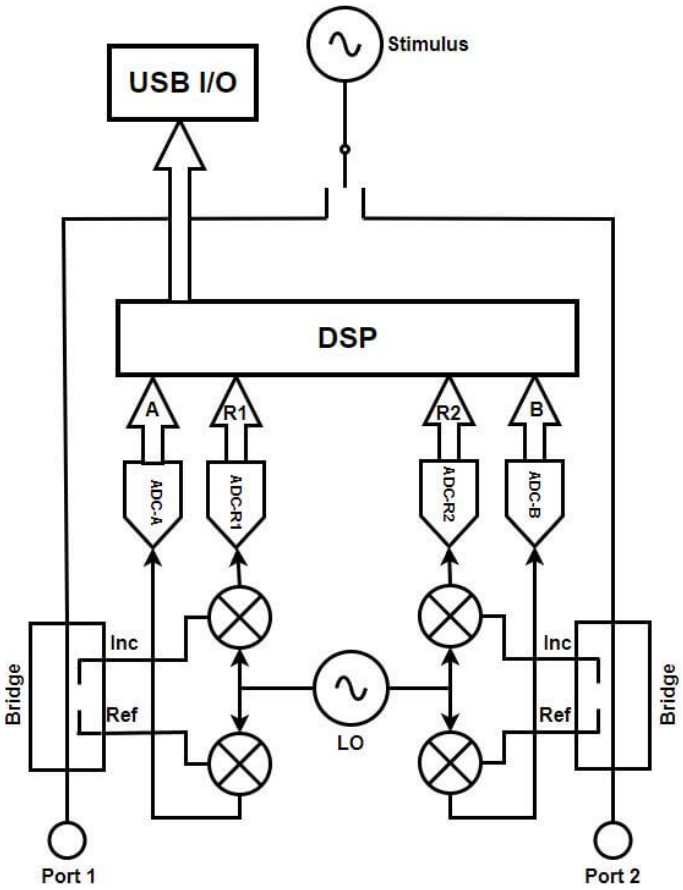

Figure 1.2: Block diagram of a simple VNA, [4]

The internal working of a general simple VNA is shown in figure 1.2. An RF stimulus coming from port 1 is provided to a DuT, which is connected between port 1 and port 2 (not shown in the figure). The stimulus is passed through a bridge (directional coupler), which splits the EMW in forward- and backward-going waves, which takes this signal as reference (Ref). This reference signal is demodulated into a lower frequency IF signal (intermediate frequency) using a mixer and a local oscillator (LO). The intermediate frequency is determined by the difference in frequency of the LO and the incoming signal. The reflected EMW coming from the DuT will be split off as the “Inc”-signal by the bridge at port 1, and the transmitted wave as the “Inc”-signal at port 2. They then go through the same process as the reference signal, to obtain two more IF signals. The same process can also be done with a RF stimulus coming from port 2, producing another reference, transmission and reflection IF signal, to study the effects of the DuT in two directions by finding other S-parameters.

All IF signals are then digitised in Analog-to-Digital Converters (ADCs) and processed in the Digital Signal Processing unit (DSP). In the DSP unit, the four S-parameters are calculated by doing complex divisions such as the one in equation 1.1. After that, the data can be retrieved via a data bus such as USB, or be immediately shown on a screen.

1.2. Application in quantum research

A Transmon qubit is a type of superconducting charge qubit. It consists of a superconducting quantum interference device (SQUID), a non-linear inductive element made of two superconductors separated

by a thin insulating barrier, and a shunting capacitor C_t. The SQUID consists of two Josephson junctions in a loop. The Josephson junctions provide the non-linear inductance necessary to create quantised energy levels with nonuniform spacing (also known as anharmonicity). Anharmonicity is the key to confining the dynamics of multi-level quantum system (such as a Transmon) to within a two-level subspace when it is driven.

Being able to confine the dynamics within a two-level subspace is important, because it simplifies the system to a manageable quantum bit, or qubit, which is the fundamental unit of information in quantum computing. This confinement allows for clear distinction between the two states, $|0\rangle$ and $|1\rangle$, necessary for reliable quantum operations and algorithms. It also reduces the likelihood of leakage into higher energy states, which can lead to errors and decoherence, thus improving the overall stability and performance of quantum circuits. The primary role of the shunting capacitor is to increase the charging energy relative to the Josephson energy, which mitigates the effects of charge noise and enhances the robustness of the qubit.

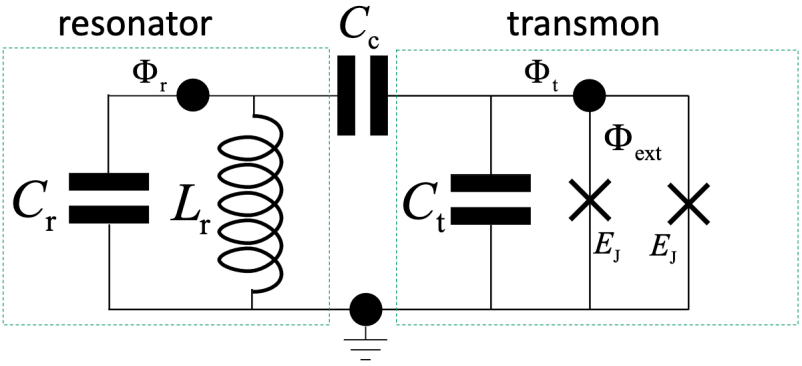

Figure 1.3: Transmon qubit coupled to a resonator [5]

Figure 1.3 shows the lumped element model of the Transmon qubit coupled to a resonator. The resonator is implemented as a waveguide (here modelled as a single inductance L_r and capacitance C_r). The resonator is the mechanism by which the qubit is read out, so it is also called the readout resonator.

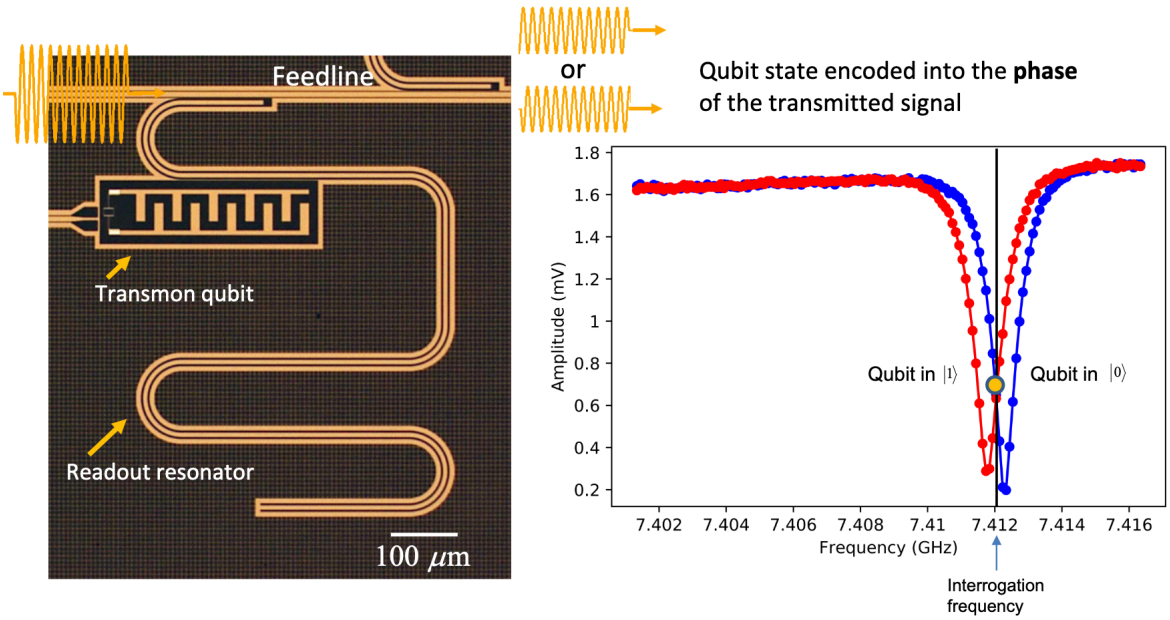

Figure 1.4: Left: image of a real Transmon qubit and the attached readout resonator. Right: amplitude of transmitted signal through the qubit as a function of applied frequency.[5]

The key to the microwave readout is sending a calibrated microwave pulse towards the resonator. This pulse is typically set at or near the resonator's base frequency ω_r , but the qubit-state-dependent frequency shift (either $\omega_r - \chi$ or $\omega_r + \chi$) affects how this pulse interacts with the resonator. The way this is done in practice is by the use of a VNA. Qubit measurement can be performed by taking the superconducting qubit circuit as the device under test (DuT) and measuring its S_{21}-parameter. This parameter helps to determine changes in the microwave signal due to the qubit-state-dependent frequency shift, thereby enabling the measurement of the qubit state.

In figure 1.4, an actual picture of the Transmon qubit can be seen, together with the readout resonator and what a successful readout looks like. In figure 1.5, a more schematic representation of the readout procedure is shown.

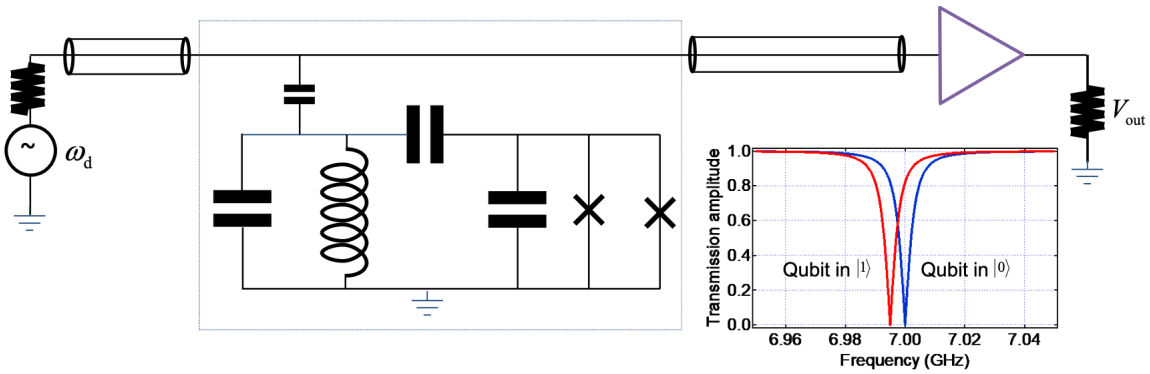

Figure 1.5: Readout of a Transmon qubit [5]

1.3. Existing solutions

Commercial VNAs from companies like Keysight and Tektronix are often quite expensive, having price tags of several tens of thousands of euros[6]. This is in large part due to their accuracy combined with a large frequency range which extends into multiple decades, which requires expensive components. Extensibility is provided with equally expensive options, but offer limited flexibility since users are limited to the offerings of the company for that specific model.

Cheaper options are available too, in the price range of hundreds to thousands of euros, but these options provide a narrower frequency range and lower accuracy [7]. Being sold in a single package, these options also do not offer much extensibility without having to study the (often open-source) documentation thoroughly.

VNAs are in the field of quantum computing sold as quantum controllers[8][9]. These systems offer most of the flexibility that are required for qubit research, but have prices in the range of hundreds of thousands of euros. This is the case because of their very high accuracy and very large frequency range.

To offer much higher flexibility than the mentioned VNAs, and low to moderate prices, there have been projects on VNAs using SDR (software defined radio) technology, which recreates (expensive) analog EMW components in software[10]. This can be done using for example a field programmable gate array (FPGA) to obtain even higher flexibility and processing speed. A hobbyist's attempt to create a VNA using SDR technology on an FPGA is well documented on internet [11]. There has also been a paper on an FPGA-based alternative for a VNA used for imaging in industry in the range of $200\ \mathrm{GHz}$ [12]. Recently, there has also been an effort to create a VNA or quantum controller using SDR technology on an FPGA [13].

1.4. Functional requirements

The requirements for the VNA of this project select the basic VNA functionality which is most useful for the application of interfacing with qubits. Omitting other functions of a commercial VNA is what makes it possible to offer a cheaper and more modular system. The system can be made using off-the-shelf

RF components, an FPGA and a RF signal generator. The layout for the FPGA and the interfacing programs are made open-source, to make the product available for free in the research sector. To make the interaction with the VNA understandable for the researchers, Python code is used for the user interface and API. The qualitative requirements of the entire VNA are shown below:

1. The system must have the ability to measure the S_{21} (transmission) parameter at different frequencies.
2. The system must be modular, so the system should work with most RF generators without any adjustments.
3. The system must be designed in such a way that it is usable by students and researchers without experience in electrical engineering.

The absolute calibration of the device is not important. It will only be used for relative measurements, because the S_{21} parameter is just a ratio between input RF signals (through-DuT or reference) and the output RF signal of the VNA.

Besides the qualitative, there are also some quantitative requirements for the system:

1. The operating frequency range must be $4-8$ GHz.
2. Integration time per measurement point:
   - upper limit: up to 1 second per point (1 Hz IF bandwidth).
   - lower limit: down to 1 millisecond per point (1 kHz IF bandwidth).
3. Transfer overhead time to transmit the data to the client must be less than 10 % of the total measurement time.
4. Spurs of the signal going to the device under test must be less than 40 dBc.

Then there are some objectives that the project should aim to achieve:

1. The system should be responsive for a human user by having a time under 100 ms between a user input/output and a physical event happening.
2. As much open-source software as possible should be used for the project.

The specific functional requirements for this subteam will be covered in chapter 2.

1.5. Materials

A Red Pitaya STEMlab 125-14 board is used for the digital section, which is described as a signal acquisition and generation platform. This device contains a Xilinx Zynq 7010 System on Chip (SoC) and several connectors, such as an ethernet port, micro-USB port, GPIO pins and RF SMA connectors. The SoC contains both Programmable Logic (PL) like that found in a Field Programmable Gate Array (FPGA), and a Processing System (PS) which contains an dual-core ARM processor.

For the RF section, SMA coaxial cables, RF mixers and power splitters from Mini-Circuits are used. Also, the following RF generators have been used:

- A SynthHD (V2) 10MHz - 15GHz Dual Channel Microwave Generator by Windfreak Technologies, LLC. With its 2 output channels it produced both the RF stimulus signal and the RF LO signal, in one package with a single API. This generator degraded to an extent which made in unusable for the VNA, which is why it was replaced halfway the project by the following 2 RF generators:
- An HMC-T2100 10 MHz - 20 GHz synthesized signal generator by Hittite Microwave Corporation (now from Analog Devices, Inc.), which was used for the stimulus signal.
- An APUASYN20 8 kHz - 20 GHz Ultra-Agile Signal Source by AnaPico AG, which was used as LO.

1.6. Problem definition

To achieve the functional requirements, several engineering problems had to be solved. For this, three teams or subgroups of two students have been formed: the RF team, the FPGA team and the software

team. The RF team had to route and downconvert the RF signals going to the DuT and REF to IF, which then could be digitised by the ADC on the Red Pitaya and used as digital input for the FPGA team. Generators, mixers and power splitters had to be chosen which would work best to achieve the requirements. Moreover, the behaviour of these components had to be measured and documented as well as the entire power budget throughout the system.

The signals that were digitised at the input of the Red Pitaya had to be converted into IQ signals by the FPGA team. Averaging was done on the FPGA to achieve the IF bandwidth requirements. Another engineering problem for the FPGA team, together with the software team, was the communication between the PL and the PS. Data from the PL had to be sent to the software team while control instructions from the software team had to be read by the PL. The software team also had to create an interface between the user and the VNA. An API and a graphical user interface (GUI) were developed for this interaction, which were part of a client program written in Python. This client also had to communicate with the PS of the Red Pitaya's SoC. A schematic of this arrangement is shown in figure 1.6.

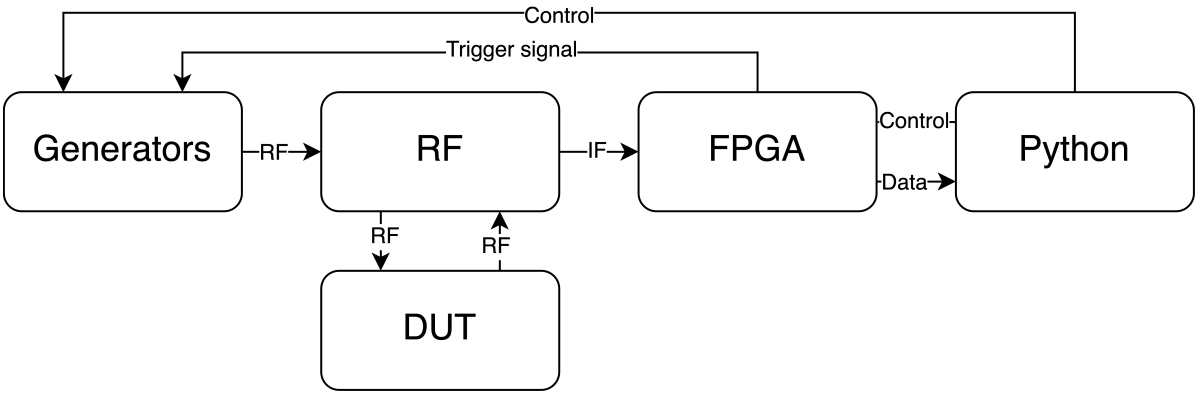

Figure 1.6: Simplified input/output scheme of each subgroup.

1.7. Thesis overview

This thesis describes the theory, design, and implementation of the Digital Signal Processing part of the Vector Network Analyser implemented in Programmable Logic, for which it is split up into several chapters. Chapter 2 describes the requirements specific to this part of the VNA that will be referenced throughout the document. Chapter 3 is split up into three sections: section 3.1 describes the theory of operation and lays out the architecture of the system, section 3.2 derives mathematical analysis and intuition behind the performed signal processing, and section 3.3 explains the digital implementation of different functions used. Chapter 4 describes the testing of the individual components as well as the system as a whole and analyses measurements made out-of-context. Finally, chapter 5 ends the thesis with a discussion of the shortcomings of the current implementation and the possibilities for future work.

2

Programme of Requirements

This programme of requirements is part of the SteeleLab VNA project, where the goal is to create a low-cost open hardware VNA which performs on such a level that it is useful for the researchers and students that are part of the SteeleLab group. This PoR regards the part of the VNA between the RF subsystem and the software subsystem, it transforms the signals from the RF domain to the digital domain in a useful way for the software system. To do this, a Red Pitaya STEMlab 125-14 was supplied by the SteeleLab group. This board has RF inputs and outputs, GPIO, ADCs, DACs and an SoC containing a processing system and programmable logic. The server is defined as the program running on the processing system of the Pitaya board, made by the software team.

2.1. Functional requirements

The qualitative requirements for the DSP implementation are as follows:

1. The digital part of the VNA system must be based on the Red Pitaya STEMlab 125-14 board.
2. The implementation must be able to digitise the IF signals coming from the analogue part.
3. The implementation must be able to IQ decompose the digitised IF signals.
4. The implementation must be able to apply data reduction to the IQ stream.
5. The implementation must be able to generate configurable triggers for the RF generators.
6. The implementation must generate a synthesized 10MHz reference clock out of the on-board DACs.
7. The server must be able to get IQ data from the implementation.
8. The server must be able to configure the timing of the acquisition cycle.

The quantitative requirements are as follows:

1. Data reduction in the programmable logic should be configurable to collect a maximum of 1 second of samples per point and a minimum of 1 millisecond of samples per point. This results in an IF BandWidth (IFBW) between 1 Hz and 1 kHz.[1]
2. Filtering as a result of the data reduction should be matched and locked to the sampling rate and have a high (more than 20 dB) harmonic rejection.
3. The filtering should also have less than 3 dB SNR reduction and less than 3 dB correlation in the data.

[1] The definition of the term IFBW varies between VNA manufacturers, in this thesis we will be defining it as the inverse of accumulation time. Note that this means the IQ sampling rate is lower than the IFBW as each point also has overhead in the form of settling time.

2.2. Non-functional requirements

1. The system should consist of open hardware and software as much as is reasonably possible.
2. The system should be documented in such a way that it is usable by students and researchers without Electrical Engineering (EE) experience.
3. The system should be documented in such a way that third-year EE bachelor students can further develop the system.

3

Design Process

In this chapter the broad theory of operation of the digital part of the VNA is laid out, followed by an explanation of the mathematics behind the signal processing and details on how this is implemented in the Programmable Logic area of the chosen FPGA board.

3.1. Theory of operation

The goal of this part of the VNA is to extract the relative amplitude and phase of two incoming Electro-Magnetic Waves (EMWs) at an Intermediate Frequency (IF), which together represent the S21 parameter of the Device Under Test (DUT). To this end, the signal processing implementation should extract the I and Q parameters of these signals which are directly related to their amplitude and phase as described in section 3.2.1. This task can be achieved in several different ways, generally referred to as architectures. A detailed breakdown of the task is given, followed by a discussion of two possible architectures, the latter of which is implemented.

3.1.1. General overview

The signal chain implemented in the Programmable Logic consists of four stages: digitisation, IQ decomposition, data reduction and data output. Alongside this signal chain, there is its control logic, and also a completely separated clock reference generator.
The incoming signals are of a fixed Intermediate Frequency, sampled at $F_s = 125$ MHz. While the choice of F_s is fixed by the design of the Red Pitaya FPGA board, the choice of IF is somewhat arbitrary as it does not significantly impact the theory of operation and performance. It was chosen to be $\omega_{IF} = \frac{2\pi}{16}$ rad/sample $= \frac{2\pi \cdot F_s}{16} = 7.8125$ MHz. This frequency was chosen as it is related to the sampling rate of the ADCs by a power of two to simplify various digital circuits in the Programmable Logic implementation. It is also low enough that RF effects and aliasing are not major issues. Most critically, it is not close to any harmonics of 10 MHz, which is a very interference-prone frequency due to laboratory equipment often using 10 MHz reference clocks. It was chosen early in the design process because the IF affects many details of the design, including the ease of implementing various DSP elements and component values in the analogue RF part handled by a different subgroup.
Below are explanations pertaining to the different sections of the implementation.

Digitisation

The digitisation stage is responsible for controlling the ADC chip available on the Red Pitaya board, namely the Analog Devices LTC2145-14[14][15]. This chip converts the incoming analogue IF signals into discrete ones sampled at 125 MHz with 14 bits per sample. The chip is connected directly to the PL and an open source IP block implemented therein that converts its data into an AXI4-Stream to be used internally. This is discussed in section 3.3.2.

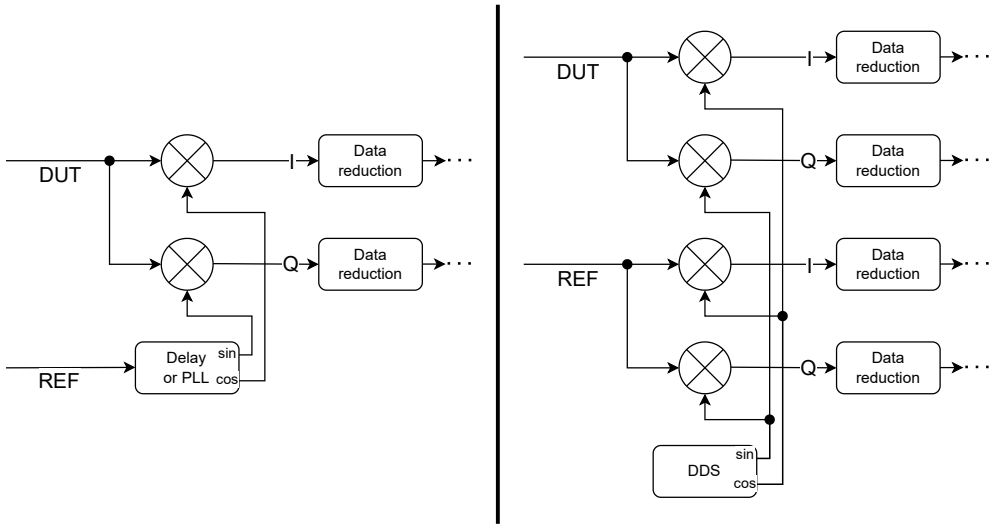

Figure 3.1: High-level block diagrams of the signal chain of both the first (left) and second (right) architectures.

IQ Decomposition

The IQ decomposition stage multiplies the digitised signals by the sine and cosine of a Local Oscillator (LO). The output of this stage is one or two discrete complex signals represented by their I and Q components. The theory of this part is described in section 3.2.1.

Data reduction

The data reduction stage takes in the I and Q stream(s) from the IQ decomposition stage, which is sampled at 125 MHz, and combines samples into points at a rate configured by the user (the IFBW). This stage also removes unwanted frequency components of the IQ signal. The theory of this part is described in section 3.2.2.

Data output

Last in the signal chain, the data output stage bundles the points from the data reduction stage into packets and stores these until they are requested by the server, at which point it writes the data into system memory where the server can read it. This is discussed in section 3.3.2.

Control logic

The control logic is responsible for configuring and coordinating the different stages of the signal chain. It takes in configuration data from the Processing System and routes this to the appropriate components. It also contains a sequencer to coordinate the acquisition cycle. This is discussed in sections 3.3.2 and 3.3.3.

Reference clock

Finally, the clock reference generator is completely isolated from all other parts except through sharing the same internal clock. It uses Direct Digital Synthesis to generate a 10 MHz sinusoid and outputs this using the on-board Analog Devices AD9767 DAC[14][16] chip to be used as a reference clock by external components such as the microwave generators. The DAC, like the ADC, is controlled by an open-source IP block. This part is discussed in section 3.3.2.

3.1.2. First architecture

The first architecture (shown on the left side of figure 3.1) utilises the incoming reference signal (REF) as the local oscillator used to decompose the test signal (DUT). This has a major benefit in that the IF does not have to be very precise, as long as the frequency on the DUT and REF inputs is still the same, and so clock-locking is not required.

The IQ decomposition can be done by directly using the acquired REF signal as the cosine and a delayed version of that same signal as the phase-shifted sine. This is not ideal, however, as the REF

signal can vary in amplitude, reducing the precision. Furthermore, because of the variable input frequency, it is not possible to delay the wave by exactly 90 degrees, causing IQ imbalance that requires software correction.
To solve these issues, a Phase Locked Loop (PLL) can be used, or rather a digital version thereof. This block should generate a sine and cosine wave that are phase-locked to the incoming reference signal. Reconstructing the wave like this solves the two disadvantages mentioned above.
Both methods have a significant flaw, however; While the outputted IQ stream contains information about the relative phase between the two signals, the amplitude is related to the product of the amplitude of the DUT and REF signals instead of the division thereof. This means separate amplitude detection circuitry is required to derive the relative amplitude which significantly increases the complexity compared to the elegance of the initial idea. Not only that, but, while theoretically possible to build, the PLL block as described would require too much time to create for this project.

3.1.3. Second architecture

The second architecture (shown on the right side of figure 3.1) treats both the DUT and REF signals identically throughout the entire signal chain, only distinguishing between them in software. Both IF signals are decomposed into IQ signals using the same LO, using complex division in software to turn these two absolute signals into a single relative signal. This architecture incurs a small amount of software overhead and takes up more space inside the Programmable Logic, but has the advantage of being very straightforward to implement. It also allows the channels to be used independently. This architecture even theoretically still allows a small IF offset while doing relative measurements, though this is hindered by the chosen implementation of data reduction as explained in section 3.2.2. This architecture is more similar to a traditional VNA.[17]

3.2. Signal processing

In the previous section, two different architectures for extracting the IQ parameters of a signal are described. This section demonstrates the mathematics behind the IQ decomposition and data reduction stages and derives the effect of several design choices on performance.

3.2.1. Phase and amplitude, I and Q

Phasors

Before the derivations, it is useful to have an intuitive framework to relate the equations to. For this topic, intuition is in the form of so-called phasors, also sometimes referred to as complex amplitude. Phasors are related to complex exponential oscillators, the generic form of which being $Ae^{j\omega t+\theta}$. Plotting the result of this function as a vector in the imaginary plane shows the vector is rotating around the origin at a constant rate (figure 3.2).[18]
A phasor is a complex number that represents a (complex) sinusoidal function of some frequency ω as a modification of this basic oscillator. They are usually denoted in polar and sometimes exponential form, $A\angle\theta = Ae^{j\theta}$, and represent the function $Ae^{j(\omega t+\theta)} = Ae^{j\theta} \cdot e^{j\omega t}$. Writing the function as a multiplication like this shows that the phasor is nothing more than a constant complex factor to the basic oscillator, defining its starting point and thus amplitude and phase. The instantaneous magnitude and angle of an oscillator can also be expressed as a phasor.
As mentioned, phasors are usually denoted in polar or exponential form as these directly show magnitude and angle in the notation. In some cases however, it is useful to denote it in the rectangular form $A\angle\theta = \mathrm{Re}\{Ae^{j\theta}\} + j \cdot \mathrm{Im}\{Ae^{j\theta}\} = I + jQ$. The visual interpretation of the parameters is shown in figure 3.3. The rest of this section will use the exponential and rectangular forms.

IQ decomposition

The goal of the digital part of the VNA is to extract the relative amplitude and phase between two EMWs, and the simplest way to do this is by way of IQ decomposition.
The incoming signal to be decomposed $s[n]$ has been digitised by the ADC and so is real and sampled (discrete time). It can be modelled as the real part of a complex oscillator $Ae^{j\theta} \cdot e^{j\omega_{IF}n} = Ae^{j(\omega_{IF}n+\theta)}$ where ω_{IF} is equal to the Intermediate Frequency and the objective is to extract the phasor describing the signal, namely $Ae^{j\theta} = I + jQ$.
Using Euler's formula the real oscillator can be rewritten as two complex oscillators at a positive and

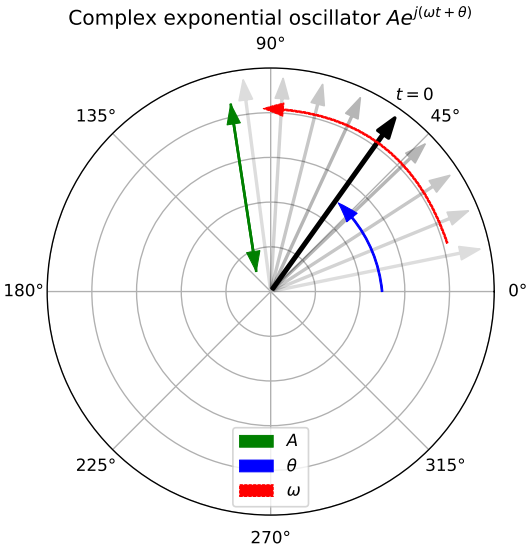

Figure 3.2: A complex exponential oscillator rotating in the imaginary plane.

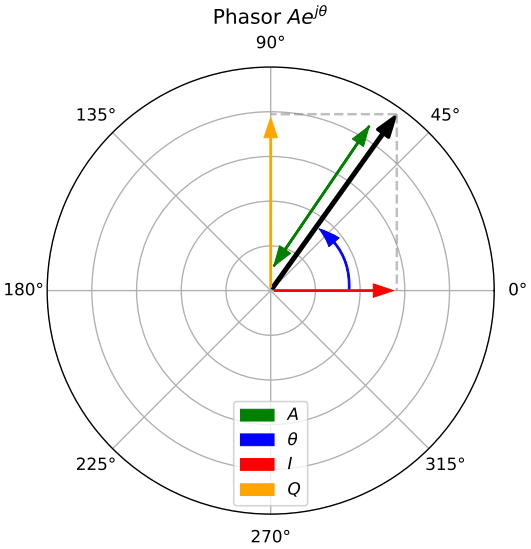

Figure 3.3: An example phasor with its parameters.

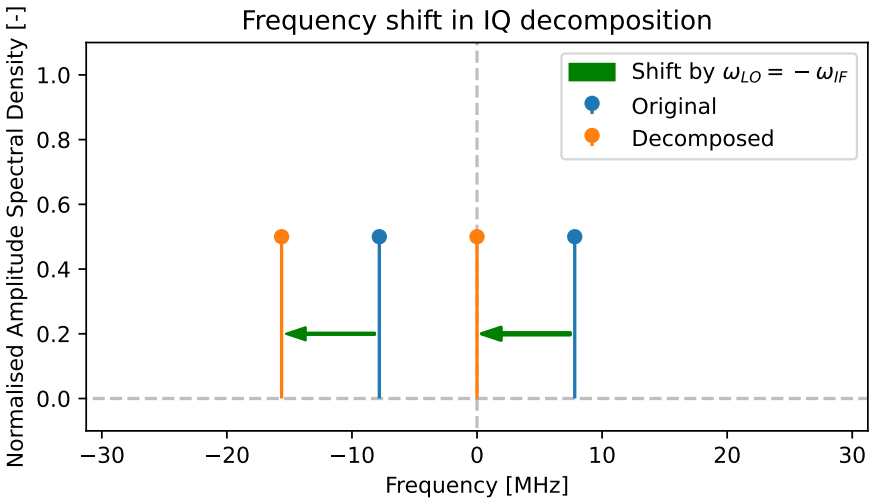

Figure 3.4: Amplitude spectrum before and after IQ decomposition.

negative frequency, shown in equation 3.2.

$$s[n] = Re\{Ae^{j(\omega_{IF}n+\theta)}\} = A\cos(\omega_{IF}n + \theta) \tag{3.1}$$

$$= \frac{1}{2}(Ae^{j(\omega_{IF}n+\theta)} + Ae^{-j(\omega_{IF}n+\theta)}) \tag{3.2}$$

Like all real signals, this signal has a frequency spectrum symmetric around 0. Multiplying this signal by a complex LO shifts these frequencies by the frequency of the LO as seen in equation 3.4 and figure 3.4. Note how because the signal is now complex the spectrum is also no longer symmetrical.

$$\frac{1}{2}((Ae^{j(\omega_{IF}n+\theta)} + Ae^{-j(\omega_{IF}n+\theta)}) * e^{j\omega_{LO}n}) = \frac{1}{2}(Ae^{j(\omega_{IF}n+\theta)} \cdot e^{j\omega_{LO}n} + Ae^{-j(\omega_{IF}n+\theta)} * e^{j\omega_{LO}n}) \tag{3.3}$$

$$= \frac{1}{2}(Ae^{j((\omega_{IF}+\omega_{LO})n+\theta)} + Ae^{-j((\omega_{IF}-\omega_{LO})n+\theta)}) \tag{3.4}$$

Just doing this multiplication has not achieved much as the desired information is still encoded into two frequency components. However, by choosing an appropriate value for ω_{LO} we can move one of these components to DC where it is easy to measure. Choosing $\omega_{LO} = \omega_{IF}$ will move both components to the right and so move the negative frequency component to DC. While this would work for the implementation, the math is slightly more elegant when shifting the other way, so $\omega_{LO} = -\omega_{IF}$ is chosen, resulting in equation 3.5 and the decomposed spectrum shown in figure 3.4.

$$\frac{1}{2}(Ae^{j((\omega_{IF}-\omega_{IF})n+\theta)} + Ae^{-j((\omega_{IF}+\omega_{IF})n+\theta)}) = \frac{1}{2}(Ae^{j\theta} + Ae^{-j(2\omega_{IF}n+\theta)}) \tag{3.5}$$

The important result of this is that the DC component of the final signal is now equal to the phasor that was used to describe the input signal. Finally, the equation can be rewritten into real and imaginary parts, shown in equations 3.6 and 3.7, to reveal the DC components to be equal to $\frac{1}{2}I$ and $\frac{1}{2}Q$ from the rectangular notation. This factor $\frac{1}{2}$ stems from the fact the IQ decomposition only moves one of the two frequency components in the original signal to DC and so half of the amplitude is lost.

$$\mathrm{Re}\{\frac{1}{2}(Ae^{j\theta} + Ae^{-j(2\omega_{IF}n+\theta)})\} = \frac{1}{2}I + \frac{1}{2}A\cos(-2\omega_{IF}n + \theta) \tag{3.6}$$

$$\mathrm{Im}\{\frac{1}{2}(Ae^{j\theta} + Ae^{-j(2\omega_{IF}n+\theta)})\} = \frac{1}{2}Q + \frac{1}{2}A\sin(-2\omega_{IF}n + \theta) \tag{3.7}$$

There is also still a second component at $2 * \omega_{IF}$ which should be removed in the data reduction stage. Expected defects of the input signal include the presence of DC offset (a 0 Hz frequency component) introduced in the ADC, and harmonics of the IF caused by nonlinearities in the analogue signal chain.

Using the same derivation as above it can be shown that these additional frequency components end up on different harmonics of the IF and should also be filtered by the data reduction stage.[1]

3.2.2. Data reduction

The IQ decomposition stage outputs sampled signals for I and Q with the same sample rate as the input signal, which is 125 MHz in this case. This is a tremendous amount of data to process and store, especially for longer sweeps, totalling over 1 GB per second. Furthermore, recall that the IQ signal contains two frequency components, one at $\omega = 0$ (DC) and one at $\omega = 2 * \omega_{IF}$, while only the DC component is relevant for this application. Lastly, because of frequency spurs, interference, and quantisation each individual point is not very precise, meaning a large part of the resultant information is just noise. All three of these problems can be addressed at once by downsampling the signals, where many samples are combined into one. Downsampling is a form of data reduction.

Different downsampling methods

To downsample a signal by an integer fraction $\frac{1}{R}$, a Low-Pass Filter (LPF) needs to be applied to the signal of which the cutoff frequency is tuned such that the passband fits inside the Nyquist frequency of the reduced sample rate, $F_{Niquist,out} = \frac{F_{s,in}}{2R}$, followed by decimation by discarding all but every R-th sample. When using an ideal filter, this method preserves all low-frequency information without any aliasing.[19] Unfortunately, ideal filters do not exist in this non-ideal world so a filter type must be chosen based on performance in this specific application.
When doing a measurement the desired result is the S21 value of the DUT, derived from the IQ values of the received EMWs, at certain precise frequencies (”points”). This is accomplished by doing a stepped sweep; configuring the RF signal generators to hold a set frequency, waiting for them to settle, acquiring data for a set period, before switching to the next frequency and repeating this process. This means each individual point in the reduced output has a certain period in time, and thus a set of samples, associated with it that should all represent a single IQ value.
Because the parameters are held constant during the acquisition period, all the samples therein contain the same amount of information and so all of their signal energy should be utilised to maximise the SNR. The requirements state only 3 dB of their energy is allowed to be lost in DSP. Furthermore, because the acquisition period for a point is the only time where the parameters are correct for that point, the samples outside of this period should not affect its final value. At most -3 dB of the value of a point is allowed to come from surrounding samples. Lastly, the acquisition time per point can vary between sweeps as the IFBW is configured by the user, which means the filter response also needs to be configurable at runtime. These three requirements constrain the selection of filters by affecting their impulse responses, which describe how much influence a given sample has on the result.

The first option is any form of Infinite Impulse Response (IIR) filter, such as the popular Butterworth filter. These filters are simple to implement and require fewer resources than other filter types to meet the same frequency-based requirements. As their name implies, however, their impulse response is infinitely long, which means the correlation between sequential points is unavoidable. They are also hard to design, which complicates changing the response at runtime. Lastly, their phase response is not linear which poses an issue if the frequency of the incoming waves is not exactly equal to ω_{IF}.[20]
The obvious solution to the infinite length of the impulse response is to use a Finite Impulse Response (FIR) filter instead. These filters also have the benefits of being easier to design and having a linear phase response, simplifying their use in this situation. However, they do not perform as well in the frequency domain as IIR filters and so need more logic resources to implement a similarly sharp filter.[20]
Cascaded Comb-Integrating (CIC/CCI) filters are a subset of FIR filters that include decimation inside the filter and can be implemented without multiplication.[21] This greatly reduces the amount of logic resources required at the cost of severely limiting the space of achievable responses. In fact, the only response achievable is moving time averages of N points. While this sounds bad, the moving time average, also known as boxcar averaging, actually optimises the two quantitative performance requirements that affect this filter, those being minimal SNR reduction and minimal inter-point correlation. This is because the moving average impulse response only considers samples inside the moving window

[1] Some very high harmonics of the IF do end up at DC due to aliasing. However, due to the IF being set at $\frac{F_s}{16}$ the first harmonic that poses a potential issue is the 16th, which is so far removed its amplitude can be neglected.

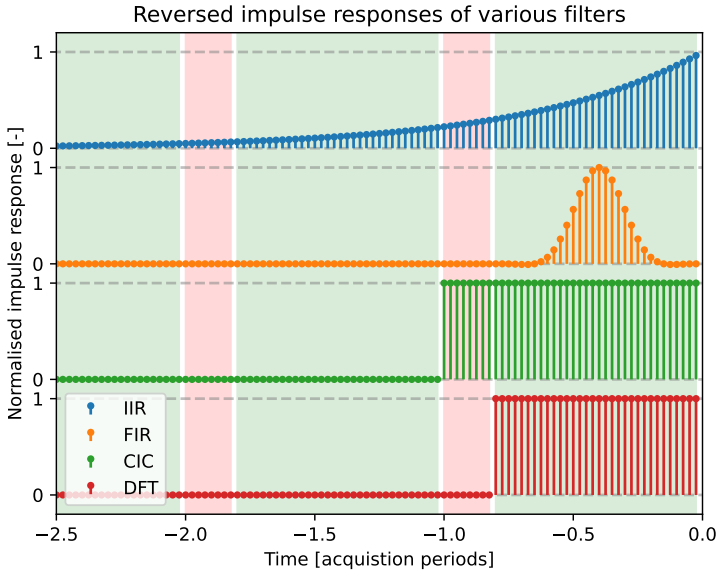

Figure 3.5: Reversed impulse response examples of various low pass filter implementations. The highlighted regions indicate the accumulation time (green) and settling time (red) per point.

and none outside, as can be seen in figure 3.5. Unfortunately, this is still not ideal when considering the entire device as a whole. This is because canonical CIC filters have a window length N equal to or larger than the downsampling ratio R, which means they also include samples from the period of time when the generators are still settling, which leads to unpredictable results. This is quite simple to solve in an FPGA implementation but that requires a different approach to model mathematically. Another way to approach the filter choice is to question if the traditional method of filtering then decimating- should be used in this application at all. That method assumes the signal to be extracted is continuous and bandlimited, however, the IQ signal that actually needs to be measured is stepped at regular intervals while being constant within these steps.
The Discrete Fourier Transform (DFT) is a transform that takes in a bounded set of N samples and outputs the frequency content of the signal in that bounded time interval, including its DC component. This is exactly what is required, but DFT is incredibly complex to implement. Luckily though, just extracting a single component is a lot simpler.
Figure 3.5 shows example impulse responses of the described filters. In practice, the exact response has to be varied during runtime to meet the configurable IFBW. Note that the impulse responses are reversed and thus show the contribution of different past samples to the output sample at $t = 0$. The highlighted regions indicate the accumulation time (green) and settling time (red) per point.

Extracting DC in an interval using DFT

The DFT is a transform that can be applied to discrete-time signals. The output is a signal in the frequency domain consisting of discrete points that represent the 'amount' of a specific frequency that is present in the original signal by way of phasors. It is defined as shown in equation 3.8.[22]

$$S[k] = \sum_{n=0}^{N-1} s[n] \cdot e^{-j2\pi \frac{k}{N} n} \tag{3.8}$$

This equation defines each point of the DFT of a signal as the summation of all the points in the original time signal multiplied by a complex oscillator with its frequency being dependent on the sample of the DFT currently being calculated. Note that the DFT is periodic in k with period N, a consequence of the input signal being discrete time. While in general this transform would require calculating many complex exponentials in parallel, in this case only the DC component needs to be extracted. For this point $k = 0$, so the exponential reduces to a simple factor of 1, reducing the transform to just simple

summation (equations 3.9 to 3.12).[2]

$$S[k] = \sum_{n=0}^{N-1} s[n] \cdot e^{-j2\pi \frac{k}{N} n} \text{ with } k = 0 \tag{3.9}$$

$$S[0] = \sum_{n=0}^{N-1} s[n] \cdot e^{-j2\pi \frac{0}{N} n} \tag{3.10}$$

$$= \sum_{n=0}^{N-1} s[n] \cdot e^{0} \tag{3.11}$$

$$= \sum_{n=0}^{N-1} s[n] \tag{3.12}$$

This is extremely simple to implement in hardware and once again optimises the two requirements for this part similar to a CIC filter. The impulse response[3] is very similar, except for its length and a factor N. This response is also shown in figure 3.5 and can be described with the following equation:

$$x[n] = \begin{cases} 1 & 0 \leq n < N \\ 0 & \text{otherwise} \end{cases}$$

However, while the impulse response optimises the energy requirements, the frequency response also needs to be considered as this is important for harmonic and spur rejection.

Frequency response of DFT

The response of a single DFT point to an input of given frequency can be derived using the Discrete Time Fourier Transform (DTFT). As the name suggests, this transform is closely related to the DFT, with the primary difference being that it applies to infinite-length time signals and as a consequence it is continuous in the frequency domain. By applying the DTFT to the impulse response of the DC point in the DFT we can obtain its continuous frequency response, shown in figure 3.6.[4] The response turns out to be a sinc function with its nulls placed at $\frac{kF_s}{N}$ for $k \in \mathbb{Z} \setminus \{0\}$. Recall that the IQ signal still contains frequency components at harmonics of the IF, $\frac{kF_s}{16}$ for $k \in \mathbb{Z} \setminus \{0\}$, that need to be cancelled. Conveniently, it is possible to choose N such that the nulls land on these harmonics, which is achieved with $N = 16k$ for $k \in \mathbb{N}^+$. In other words, the number of samples accumulated into a single point should be a multiple of 16. This also makes intuitive sense, the integral of a sinusoidal wave is equal to zero if the integration period is a multiple of the wave period, where the positive and negative parts of the wave are equal. The IF was chosen to be equal to $\frac{F_s}{16}$, which means each period of the IF is 16 cycles long, so the integration period should be a multiple of 16 cycles to cancel the harmonics of the IF.

Mismatched IF

In a previous section, it was explained that IQ decomposition relies on ω_{IF} equalling ω_{LO}. If that condition is met, the frequency components of the original signal are shifted such that the positive component ends up at DC and so the DC value of the resulting signal can be interpreted as a phasor. However, when $\omega_{IF} \neq \omega_{LO}$ the positive component is shifted to a low but non-zero frequency. This has an impact on the output value because the response of the DFT is not unity for any frequency other than DC, with the exact factor depending on the ratio $\frac{\omega_{IF} - \omega_{LO}}{IFBW}$. This can be visualised using phasors.
The DFT data reduction is equivalent to plain averaging of all the samples in the acquisition period. If the

[2] An observant reader might note that the complex exponential in the DFT that is being discarded is equivalent to the LO used in IQ decomposition. While this is true, it is not described this way in this thesis as the multiplication by LO and accumulation perform different functions of the design (IQ decomposition and data reduction). This allows choice of data reduction method as described in section 3.2.2.

[3] Because DFT is not a continuous operation like a filter it does not have an impulse response in the traditional sense, the phrase is used here to refer to the function describing the response of the (scalar) output to individual samples in the input.

[4] Similar to the impulse response discussed above, points of a DFT also do not have a traditional frequency response. It is used here to mean the (absolute) value of the point if the input is a wave of a certain frequency.

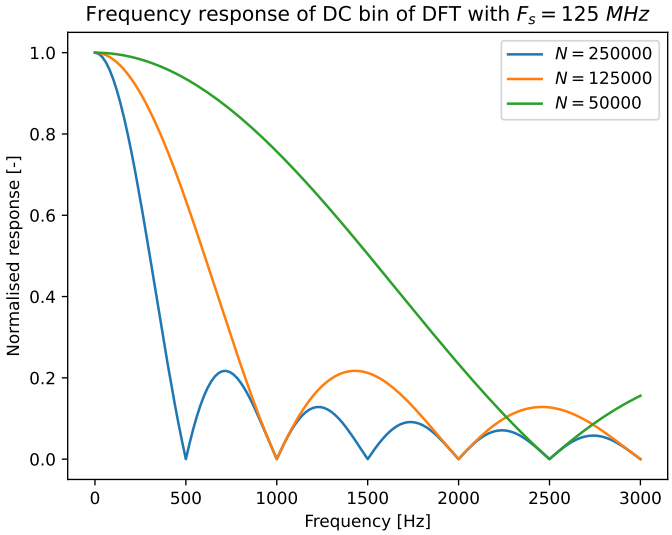

Figure 3.6: Frequency response of the DC bin of the DFT for various N at fixed sampling rate $F_s = 125\ MHz$.

phasor represented by the low-frequency component of the IQ signal is static, this will take the average of many similar samples and so return that same value while rejecting any noise present in the individual samples. When $\omega_{IF} \neq \omega_{LO}$ however, the resulting low frequency component is $Ae^{j\theta} \cdot e^{j(\omega_{IF}-\omega_{LO})n}$. This can be modelled using a complex oscillator, visualised as a slowly rotating phasor. The value output from the data reduction stage is still the average of all the samples in the acquisition period, but those now form an arc instead of a point, and so the average will be at the Centre of Mass (CoM) of the traced arc, as shown in figure 3.7.

This measured IQ point will always have a lower magnitude than the intended result as the CoM of a circular arc is always closer to its origin than any point on the arc. This response depends on the angle swept by the phasor during the acquisition period as shown in figure 3.8. This in turn is related to the ratio $\frac{\omega_{IF}-\omega_{LO}}{IFBW}$ as demonstrated in figure 3.9.

Contrary to how it initially appears, however, this has very little effect on the operation of the VNA due to the chosen architecture. Firstly, the microwave generators are frequency-locked to the Red Pitaya board through the 10 MHz reference output included in the digital design. This means the IF coming out of the analogue part of the design is extremely close to the frequency used for IQ decomposition inside the Programmable Logic. Furthermore, the S21 value the VNA measures is the relative value of the DUT signal compared to the REF signal. Because these signals are both at the same frequency and are processed the same by the signal chain, they are affected in identical ways and so the relative value theoretically does not change. In practice, the loss of signal magnitude decreases the precision and thus causes an increase in noise when the frequency offset is close to a multiple of the IFBW, as seen in figure 3.9. Whether or not this is a violation of quantitative requirement 3 is left as an exercise to the reader.

3.3. Red Pitaya and programmable logic

Being the interface between the RF subsystem and the software, the digital subsystem of the VNA will have to process and digitise the signals coming in at IF. It has to be able to receive IF inputs, has to contain enough processing power and has to do this processing fast and precisely enough to meet the quantitative requirements. The board chosen for this application is the Red Pitaya STEMlab 125-14 [14], shown in figure 3.10. This board has on-board ADCs and DACs and an SoC containing a processing system and programmable logic. The programmable logic can be configured using Vivado software which is made by Xilinx. Vivado also includes some pre-made logic blocks, called Intellectual Property blocks (IP blocks), that can execute common functions. These include things like direct digital synthesis to generate a sine wave, arithmetic units or slicers. It is also possible to create custom blocks using (synthesisable) Verilog. Verilog is a hardware description language, the syntax of which

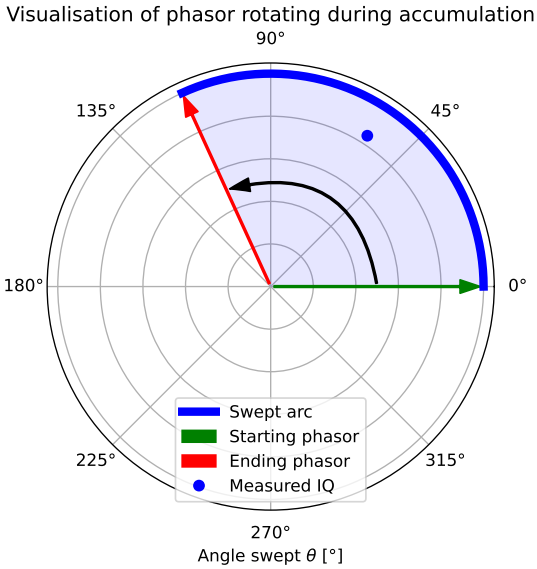

Figure 3.7: Visualisation of the result of data reduction when the low-frequency component is not at DC.

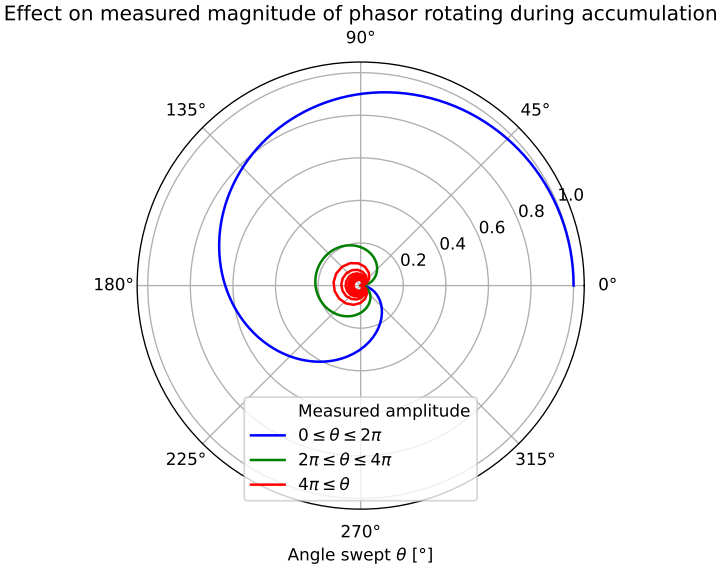

Figure 3.8: Effect of the angle swept by the IQ signal phasor on the measured magnitude.

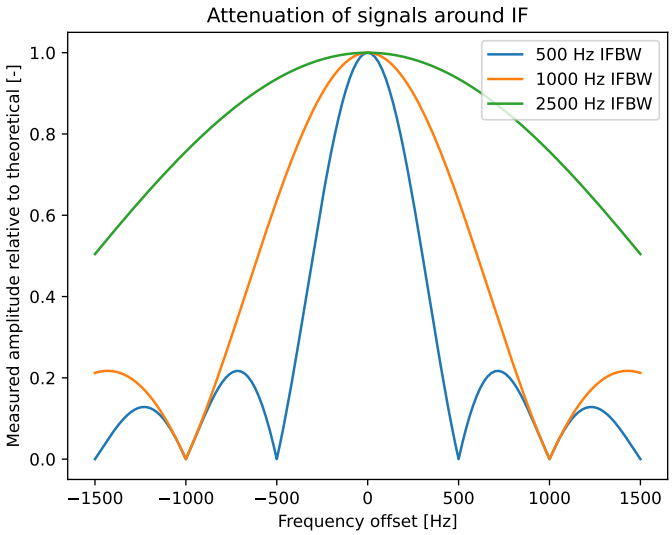

Figure 3.9: Effect of the IFBW on the magnitude response to frequencies around the IF. Note that IFBW here refers to the inverse of accumulation time and not the total time per point.

is supposed to resemble C [23]. It is a subset of the SystemVerilog language. In turn, a subset of the Verilog language is synthesisable, which means it is possible to physically realise this code and turn it into a bitstream for an FPGA or even a custom chip.
The connections between IP blocks made by Xilinx adhere to the AXI communication bus protocol. This protocol is specifically created for on-chip communication. The ADCs and DACs are chips outside the SoC on the Red Pitaya board. They can however also be controlled using a block in the Vivado software because these are connected to the SoC. The blocks to do this were created by Pavel Demin and are freely available. The following subsections will specify and explain the function of each of the components used.

3.3.1. Red Pitaya

The Red Pitaya STEMlab 125-14 development board is used to implement the design made in Vivado. The board is very versatile and is meant to develop applications related to RF. The core of the board is the Xilinx Zynq 7010 SoC. This SoC consists of two ARM Cortex A9 processing cores and programmable logic. The PL is equivalent to an Artix-7 series FPGA [24]. To indicate the size of the PL: it has approximately 28k logic cells, 18k LUTs, 35k flip-flops, 80 DSP slices and 2.1 Mb block RAM space [24]. Besides the SoC having some integrated cache, the Pitaya possesses 512 MB RAM. It also has connectivity via USB and Ethernet for remote connectivity possibilities and can run Linux from an SD card. The RF in- and outputs are SMA type and are connected to a DAC and an ADC with two channels each. The DAC can update at 125 MS/s and the ADC can sample at 125 MS/s, both using 14 bits per channel. The DAC used is the AD9767 [16] and the ADC is the LTC2145-14 [15]. Besides the RF connectors, the board has extension connectors with pins for some common communication interfaces and general-purpose IO, both digital and analogue.

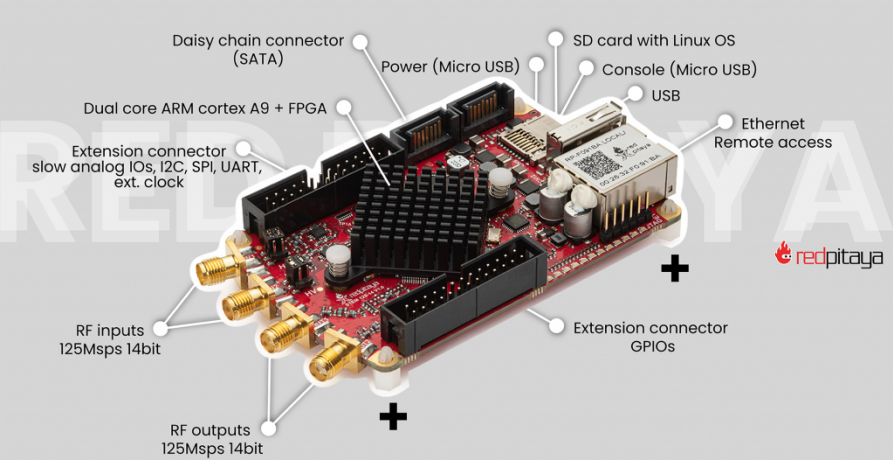

Figure 3.10: The Red Pitaya board [25].

3.3.2. Protocols and pre-made blocks

To configure the programmable logic, Vivado software is used. It is made by Xilinx and the version used for this project is available at no cost for a limited number of devices, including the SoC on the Red Pitaya. The blocks made by Xilinx and Pavel Demin all adhere to the AXI bus protocol. To interact with these modules, the custom blocks will also have to use this protocol.

AXI

The AXI (Advanced eXtensible Interface) bus protocol is designed specifically for on-chip communication. It is part of the AMBA specification, which is freely available from ARM and royalty-free. The standard is widely adopted and ensures IP components are compatible between different designers. This is important due to the open nature of the VNA project, as it will ensure that even the IP blocks on the programmable logic are easily interchangeable. Others could even use custom blocks created for this project. The specific version of AXI used most is called AXI4-Stream. This is a simplified version of the full AXI protocol and is suitable for FPGAs [26]. By AXI-Stream, AXI4-Stream is meant. If only AXI is written the information is valid for the general AXI protocol.

The complete AXI protocol includes many control and data signals. It specifies full communication between a master and a slave component. The master can send a read address to the slave and the slave will reply with data it has stored at that location via the read data channel. The master can also write data to the slave by sending a write address and data. These channels are all accompanied by many control and handshake signals. The full AXI protocol is very complex and not very relevant for this project. The only components that use this protocol are ready-made by Xilinx. It is however relevant to have a global understanding of the full protocol, to be sure the ready-made components are connected and functioning correctly. The AXI-Stream protocol, which is explained below, is used in custom components, therefore a more rigorous understanding of this protocol is required.

The AXI-Stream protocol in its simplest form consists of a transmitter and a receiver and four signals between them: a clock signal, an information bus, a valid signal and a ready signal. The information bus must be an integer number of bytes and has a standard width of 32 bits in this project. Please note that some abbreviations might still reflect the previous naming convention of 'master' and 'slave' instead of 'transmitter' and 'receiver'. For this report, the latter is used for AXI-Stream. The regular AXI protocol still uses 'master' and 'slave' naming conventions as seen before.
The transmitter sends the valid signal and the receiver sends the ready signal (`TVALID` and `TREADY`). Once both are high, the data transfer occurs on the next (low to high) clock edge. This ensures both blocks are ready for the transfer and no corrupt data is transferred. A simple handshake is visible in 3.11. As soon as the information becomes valid, the transmitter changes the valid signal to one and once the ready signal is also one by the receiver the transfer occurs on the clock edge of T3. After this, the ready and valid signals are made zero by their respective producers and the information can change again. The system is ready for the next transfer.

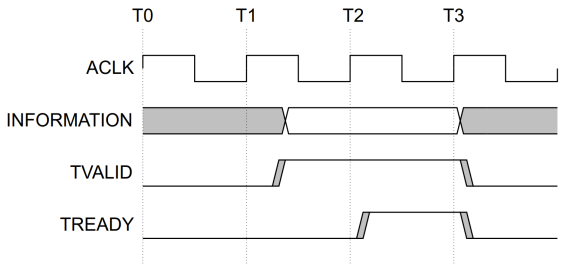

Figure 3.11: Timing diagram of AXI handshake [26]

In this protocol, it is possible to combine individual transfers into one long packet. The simplest way to do this is to transfer data like normal and then set a `TLAST` signal to high after the last individual transfer has occurred. This lets the receiver know that all transfers up to the previous `TLAST` belong to one packet [26].
The AXI-Stream protocol also includes functionality for routing different data streams through a bus. This can be done using the `TID` and `TDEST` signals. These are not relevant for this project as there is no bidirectional bus shared between all components like a CPU might have. Instead, every component's data bus is connected directly to the next component and does not need to be shared.

MMIO
To control GPIO, the processing system can use memory-mapped IO. This means certain memory addresses are linked to in- or outputs and whenever the processing system reads from or writes to this address, the signal is passed on to the GPIO. This is contrary to port-mapped IO, where the CPU will have separate instructions for reading to and writing from the IO.
MMIO can be implemented on the Red Pitaya by using an AXI bus coming from the processing system. The processor will write to a memory address corresponding to this AXI bus and this AXI bus can then be split into multiple destinations. In this project, the MMIO is only used to send signals from the processing system to the programmable logic. This means the signals stay in the SoC, the same system would apply however if the signal were connected to outside the SoC. The communication between the programmable logic and the processing systems works via a memory map. The address map of the entire memory space of a Zynq-7000 series SoC can be seen in figure 3.12. The address space is 32 bits, which allows the system to access approximately four gigabytes of memory. The first gigabyte of address space is mapped to RAM, after which two gigabytes are mapped to the programmable logic (PL) via AXI, General Purpose Port one and two. The final gigabyte is used for miscellaneous purposes.

Address Range	CPUs and ACP	AXI_HP	Other Bus Masters(1)	Notes
0000_0000 to 0003_FFFF(2)	OCM	OCM	OCM	Address not filtered by SCU and OCM is mapped low
	DDR	OCM	OCM	Address filtered by SCU and OCM is mapped low
	DDR			Address filtered by SCU and OCM is not mapped low
				Address not filtered by SCU and OCM is not mapped low
0004_0000 to 0007_FFFF	DDR			Address filtered by SCU
				Address not filtered by SCU
0008_0000 to 000F_FFFF	DDR	DDR	DDR	Address filtered by SCU
		DDR	DDR	Address not filtered by SCU(3)
0010_0000 to 3FFF_FFFF	DDR	DDR	DDR	Accessible to all interconnect masters
4000_0000 to 7FFF_FFFF	PL		PL	General Purpose Port #0 to the PL, M_AXI_GP0
8000_0000 to BFFF_FFFF	PL		PL	General Purpose Port #1 to the PL, M_AXI_GP1
E000_0000 to E02F_FFFF	IOP		IOP	I/O Peripheral registers, see Table 4-6
E100_0000 to E5FF_FFFF	SMC		SMC	SMC Memories, see Table 4-5
F800_0000 to F800_0BFF	SLCR		SLCR	SLCR registers, see Table 4-3
F800_1000 to F880_FFFF	PS		PS	PS System registers, see Table 4-7
F890_0000 to F8F0_2FFF	CPU			CPU Private registers, see Table 4-4
FC00_0000 to FDFF_FFFF(4)	Quad-SPI		Quad-SPI	Quad-SPI linear address for linear mode
FFFC_0000 to FFFF_FFFF(2)	OCM	OCM	OCM	OCM is mapped high
				OCM is not mapped high

Figure 3.12: System-level address map for the Zynq-7000 SoCs[27]

The signal now reached the PL via AXI. To convert it to IO signals in the PL, an AXI GPIO block is used. This block is made by Xilinx and converts the data sent via AXI to MMIO inside the programmable logic [28]. It is now possible to write configuration data to the PL to set parameters of different parts of the design and allow the PS to customise the acquisition and timing.

DMA

Direct memory access allows a system other than the CPU to directly access the main memory. This makes sure the CPU does not have to execute every read or write to or from the memory. It saves the CPU time and allows it to do other tasks in the meantime. Another reason direct memory access might be necessary is the speed of data coming in. It might be too high for the CPU to handle. When running an ADC for example, capturing the data using the CPU could become an all-encompassing task for it. If the ADC can just put the data directly into memory the CPU can process it at a later time making the system more flexible. For the project it's important to allow the PL to write measurement data directly to the memory, to allow the CPU to run the server. To do this, another IP block by Xilinx is used. This is an AXI DMA block that will take in an AXI-Stream bus from storage in the programmable logic, a FIFO buffer in this design, and translate this to an AXI bus for the processing system's memory controller.

DDS

Another relevant pre-existing block made by Xilinx is the DDS block. DDS stands for direct digital synthesis and is a technique for digitally synthesizing analogue waveforms. It works by using a phase accumulator, which is incremented by M every clock cycle [29]. The size of the register in the phase accumulator is n bits. This allows the programmer to set the time between overflows of the register, resulting in a frequency f_{out} controlled by the n and M parameters as seen in equation 3.13, where f_c is the clock frequency. The n and M parameters also influence the resolution of the output waveform. To enable the DAC to reconstruct a sine wave with the correct frequency, at least two samples are required between each overflow by Nyquist [29].

$$f_{out} = \frac{M \cdot f_c}{2^n} \tag{3.13}$$

The output of the phase accumulator is now a saw tooth wave. This wave can be fed to a lookup table storing a custom waveform, a sine wave for example. This lookup table then outputs the value of a sine corresponding to the phase value sent by the accumulator. When feeding the values from the lookup table to a DAC, the digital wave can be made analogue. It can also be filtered after the DAC to smooth out some of the artefacts created by the synthesizing process. Figure 3.13 shows the entire DDS process.

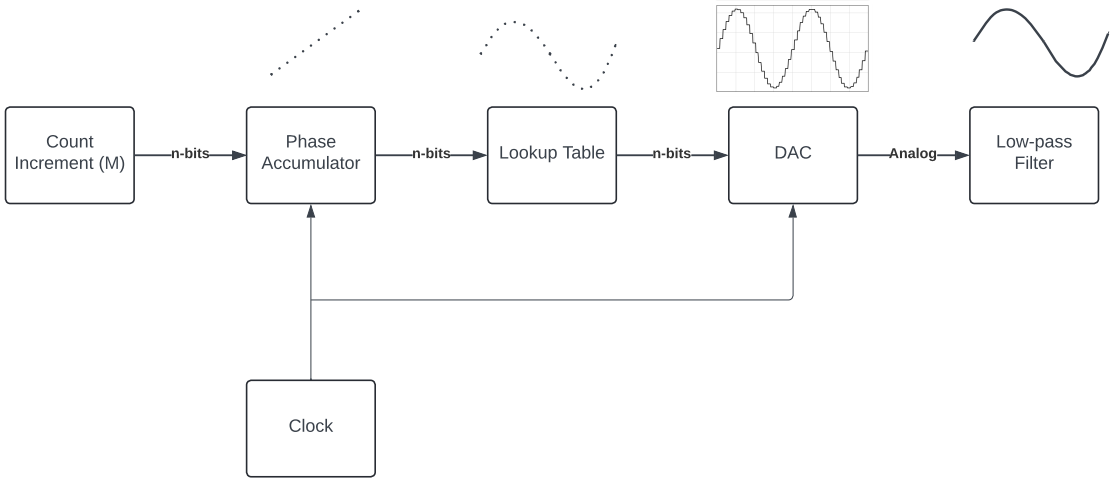

Figure 3.13: Block diagram of a simple DDS signal train.

The DDS Compiler by Xilinx integrates the phase accumulator and the lookup table. It has many different options to configure it for different applications. The increment can either be fixed internally or coming from an external source, the frequency is configurable using the n and M parameters and there is functionality to output the current phase, amongst many other things. It can only output sine and cosine waves in standard form but it is possible to use the phase accumulator part separately and connect a custom LUT [30]. The number of bits in the phase and output signal can be set by the designer and the LUT is generated specifically for this configuration. This makes sure the fidelity of the output is only dictated by the samples in each cycle and not by rounding of the data stored in the LUT.

Multiplier

The multiplier block by Xilinx is quite simple. It is one piece of pipelined combinatorial logic with two inputs and one output, the sizes of which can be configured. It implements fixed-point multiplication and can be realised using either DSP slices or lookup tables [31].

FIFO

To make sure the values coming from the accumulation system don't have to instantly be transferred into memory by the DMA, a buffer is needed. This buffer should be able to hold a certain number of samples to make sure no measurements are lost if the DMA is unavailable. The standard buffer used for such applications is a FIFO buffer. This stands for first in first out, which means that data stored first in this buffer is available first at the output, regardless of data stored in the buffer later [32]. It can be seen as a queue where data arriving can only leave when all data that arrived earlier has left. The buffer only requires one input and one output. The implementation by Xilinx is very straightforward to use [33]. An important consideration when using the buffer is its size. The buffer is synthesized using block RAMs on the programmable logic which are 512 times 32 bits in size. It would thus be logical to use a set number of block RAMs for the buffer.

Red Pitaya ADC/DAC controllers

The on-board DAC and ADC chips can be controlled by dedicated IP blocks in the Vivado software. These blocks are made by Pavel Demin[34], instead of Xilinx. The ADC block communicates with the chip outside the SoC and outputs the data into an AXI-Stream bus. Inside the ADC chips, there are two 14-bit channels that can be used as two separate ADCs. The block doesn't require any buffer, as the transformation to AXI-Stream happens instantly. The only change that happens to the signal is padding to 32 bits, the `TVALID` signal is always set high. The ADCs can sample at 125 MS/s maximum. The chip allows other sampling frequencies than 125 MHz but the version of Red Pitaya used in this project is not capable of sending it to another clock without modifications. The SNR of the ADC is 73.1 dB according to the datasheet [15].

The on-board DAC is more complicated than the ADC. This report won't go into the full details of this component, instead explaining how to use it. The DAC chip has, like the ADC, two channels of 14 bits. Each of these channels functions essentially as an independent DAC [16]. A problem that arises from this is that there is only one 14-bit connection to the two-channel DAC chip. This problem is solved by the DAC by using a double data rate (DDR) clock of 250 MHz. The DAC controller block on the programmable logic has a 32-bit input, where two 14-bit channels are padded with four bits. It will alternate between which 14 bits are sent to the DAC using the DDR clock, switching to the other channel every clock cycle. This allows the programmable logic to send data to both DACs using one 14-bit interface.

3.3.3. Custom blocks

To achieve the required functionality, not everything could be made using Xilinx IP Blocks. For this reason, custom IP blocks were designed. This can be done in the Vivado software by writing synthesisable Verilog code. Synthesisable means the code can be expressed in hardware by a synthesiser. An example of non-synthesisable Verilog are things like 'turn a signal on after 100 ns', while the synthesisable version would be 'turn a signal on after 10 cycles of this 100 MHz clock signal'. After declaring the block in a separate Verilog file, the block becomes visible in the block design editor. It can now be used in conjunction with the Xilinx blocks.

Accumulators

The accumulators will record the values coming in from the ADCs after they have been IQ decomposed. They will have to add all values together and keep track of the number of values they have added together. The division for averaging will be done outside the programmable logic to save space, as division is very hard to implement in hardware. The accumulators should be implemented in such a way that the resolution stays maximal and no data is lost, as long as this fits on the available logic. To achieve this, the following design was made.

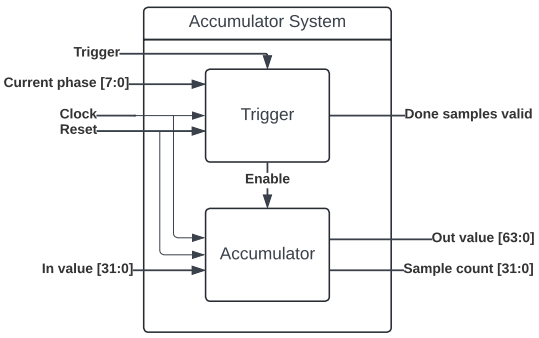

Figure 3.14: Block diagram of the accumulator system.

The accumulator is split into two parts, visible in figure 3.14, the accumulator itself and the accumulator trigger. The trigger module takes the enable signal from the sequencer (named trigger in the figure) and

produces the enable signal for the accumulator while taking the current phase of the DDS used for IQ decomposition into account. It makes sure the accumulator stays enabled until a full period of the DDS is completed, as prescribed in section 3.2.2. It will remember the phase when it started accumulating and only disable the enable signal once this phase is matched. This is done by implementing the state machine visible in figure 3.15 in Verilog. It waits for a trigger in the stopped state and then transitions to the starting state once the trigger is enabled. In the starting state, the current phase is saved and the state transitions to the running state, where accumulation starts. Once the trigger is de-asserted, the state transitions to waiting, where the system will keep accumulating until the phase from the DDS matches the previously saved phase. It will transition to the ending phase and assert the valid signal to signify the data coming from the accumulators is now valid. The valid signal has the same use as the AXI-Stream `TVALID` bit, in this case signalling the packetiser there is new data available. On the next clock edge, the system will transition to the stopped state. Every state transition happens on a positive clock edge.

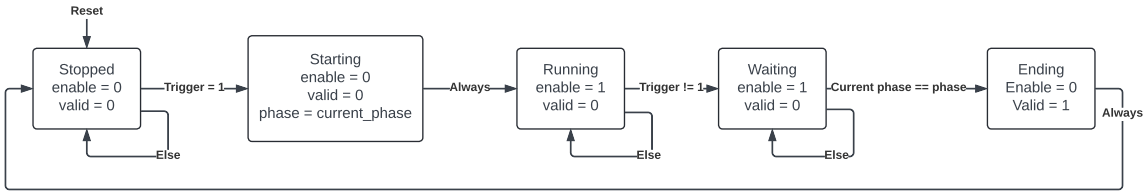

Figure 3.15: State machine for the accumulator trigger.

The accumulator itself is a simpler state machine, visible in figure 3.16. It works by waiting in the waiting state until the enable signal is high to transition into the accumulating state. Here the value coming in is added to the total and the sample count is increased by one. Once the enable is set low, the total is increased by the input value and the count is increased by one once more and sent to the output. The system then enters back into the waiting state. Please note that the count in the accumulator is named 'sample count'. The 'sample count' counts the number of samples the accumulator sums up to later divide the total by, producing one point for the rest of the system.

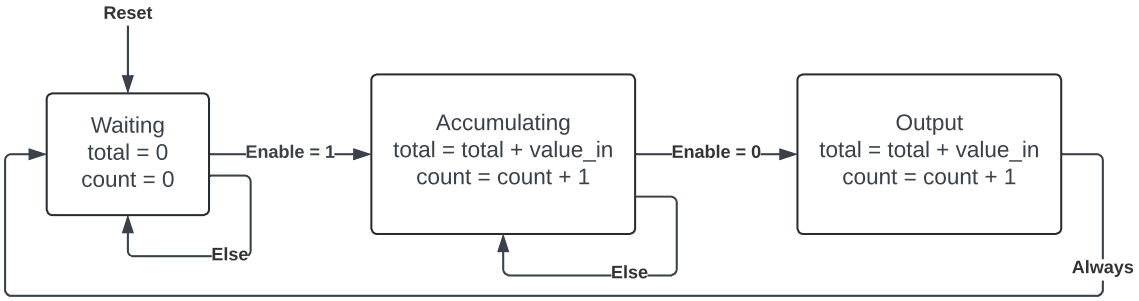

Figure 3.16: State machine for the accumulator.

Sequencer

The sequencer block is tasked with timing every sample period. It will have to trigger the generator, wait for it to settle and accumulate for a set time after that. It should then reset and start the next sample. To achieve this, a simple counter was implemented. The different signals will turn on and off for different ranges of the counter. These ranges are configurable by turning the settling time, accu time and gen end parameters. The timing is visible in figure 3.18. A previously discussed MMIO interface can be used to send timing values to this block.

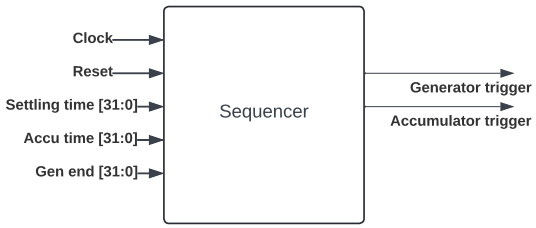

Figure 3.17: Sequencer diagram.

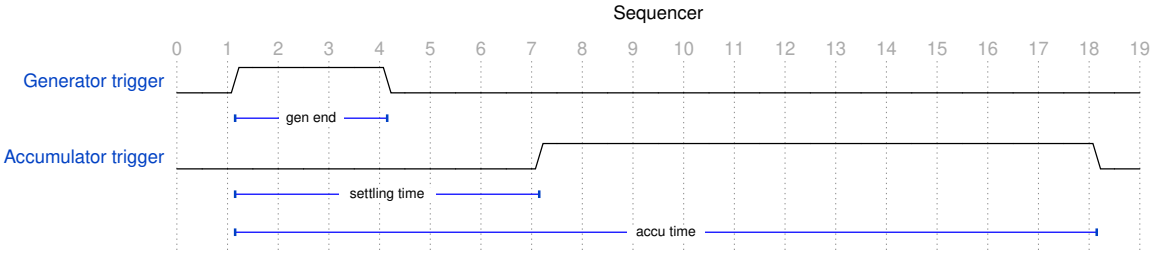

Figure 3.18: Sequencer digital timing diagram.

Packetiser

The packetiser can take values from the accumulators, both the accumulated value and the 'sample count', and turn these into an AXI-Stream packet. It does this by splitting the 64-bit outputs from the accumulator into 32-bit signals. It will then step through each of the twelve 32-bit signals, and send it to the AXI-Stream bus. This results in twelve consecutive data pieces, after which the `TLAST` signal is asserted. This signals the end of a packet to the next block. As the next block in the design is a FIFO buffer, this can get full. This is why an inhibit signal is created that will make sure the packetiser stops sending packets if the buffer is almost full so packets do not get corrupted.

4

Implementation and Validation

4.1. Implementation and validation of individual components

4.1.1. Sequencer

The sequencer module, described in section 3.3.3, was implemented in Verilog. Its code can be found in appendix A.4. A testbench for it can be found in appendix A.5. The behaviour of the system was simulated and the resulting waveform can be seen in figure 4.1. Comparing this waveform to the expected waveform in figure 3.18, the result looks very similar, with the generator trigger and accumulator trigger asserting and de-asserting at the correct time. More careful inspection of the simulated waveform reveals that every transition happens two counter increments later than it should but because this is true for every signal change, the length of the cycles is not impacted and the system functions correctly.

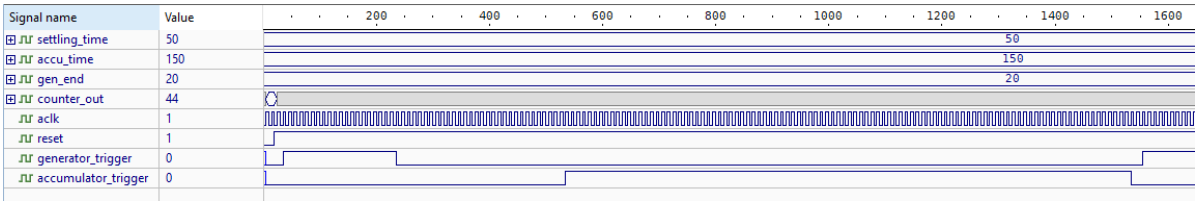

Figure 4.1: Waveform resulting from the sequencer test. (Decimal radix for all signals.)

4.1.2. Accumulator

The accumulator and accumulator trigger modules, described in section 3.3.3, were implemented in Verilog. The code can be found in appendix A.1 and A.2. To validate the function of the accumulator, a testbench was constructed that can be found in appendix A.3. This testbench initialises the system and sends random input data with looping phase data, like what would be coming from the DDS. The waveform resulting from the system behaviour can be observed in figure 4.2.

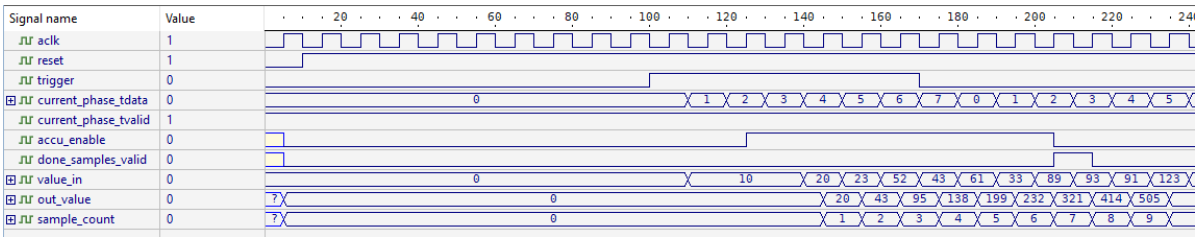

Figure 4.2: Waveform resulting from the accumulator and trigger test. (Decimal radix for all signals.)

The waveform shows after 100 ns the trigger is asserted and a little later values and phase data start

coming in. The system saves the phase and the accumulator is enabled. The output increases as expected. Once the trigger is de-asserted, the enable doesn't follow instantly but turns low once the saved phase number comes by again. This shows both the accumulator and the trigger module work.

4.1.3. Packetiser

The packetiser, described in section 3.3.3, was implemented in Verilog with the assistance of ChatGPT and can be found in appendix A.6. To test the packetiser, a testbench was made along with a sample data generator. The testbench and a top-level module can be found in appendix A.7 and the sample data generator in appendix A.8. Data is available at all input ports of the module and it has to turn this into packets while taking control signals into account. One packet transfer can be seen in figure 4.3. At the start of the packet, the different values are sent to the output and the valid signal is asserted. Then the ready signal is turned off by the testbench, simulating a receiver that's not ready to receive data. After the ready signal is turned on again, the rest of the data is transferred and after the last piece of data, `TLAST` is turned on for one clock cycle to let the receiver know the packet is finished. The packetiser works correctly.

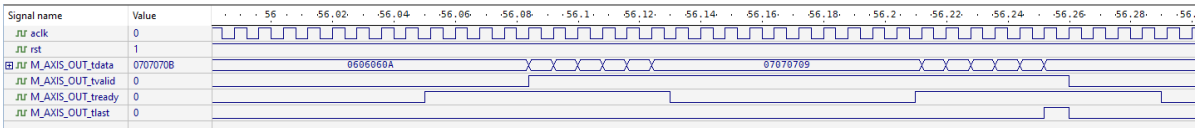

Figure 4.3: Waveform resulting from the packetiser test.

4.1.4. FIFO

The FIFO was implemented using several block RAMs available in the programmable logic. The FIFO is configured as 32,768 32-bit stages, which results in the FIFO being able to store 2,730 packets. This was estimated to be well within the limits of the server's ability to transfer data to the memory in time for the FIFO not to get full.

4.1.5. DMA

Testing the DMA using a simulation was unfortunately not practical due to its interactions with the Processing System. Unlike the other premade components it was very hard to get working and the limited debugging options and incomplete error documentation did not help with this. A system using the sample data generator A.8, also used in the previous section, and a FIFO with FIFO filler in appendix A.9 were used to understand the functioning of the DMA.

A prominent error in this process was `DMA not started`. The error was caused by the requested transfer size being too small to contain all the data until the `TLAST` signal was sent. The only recovery once this error occurred is completely re-downloading the overlay. On the other hand, a transfer larger than the packet length leaves the last part of the DMA buffer unchanged, returning old measurement data. To check if this has happened the transfer length register can be read. The error would also happen unexpectedly if the FIFO is cleared. The first four words of the next packet are buffered inside the DMA which is reset separately from the FIFO, and a cleared FIFO could thus result in the first packet being prepended with these four words and exceeding the length of the transfer.

Another common error was `Transfer size is XXXX bytes, which exceeds the maximum DMA buffer size YYY`. This was caused by micro DMA, a specific DMA implementation, only allowing single burst transfers. It was solved by not using this DMA implementation at the cost of requiring more logic resources.

`DMA not idle` was also common, it meant the last transfer hadn't finished yet. Transfers can hang for arbitrary reasons, the only recovery found is again re-downloading the overlay. This error should only happen when no data is available to the DMA but other unknown causes were suspected during testing. For this issue no fix was found, instead it seems to have been solved while working on other errors. This error also occurred in the process of finding temporary fixes for the previous two errors. Resetting only the DMA instead of re-downloading the overlay causes this error as the internal states of the DMA and the PYNQ library desynchronise.

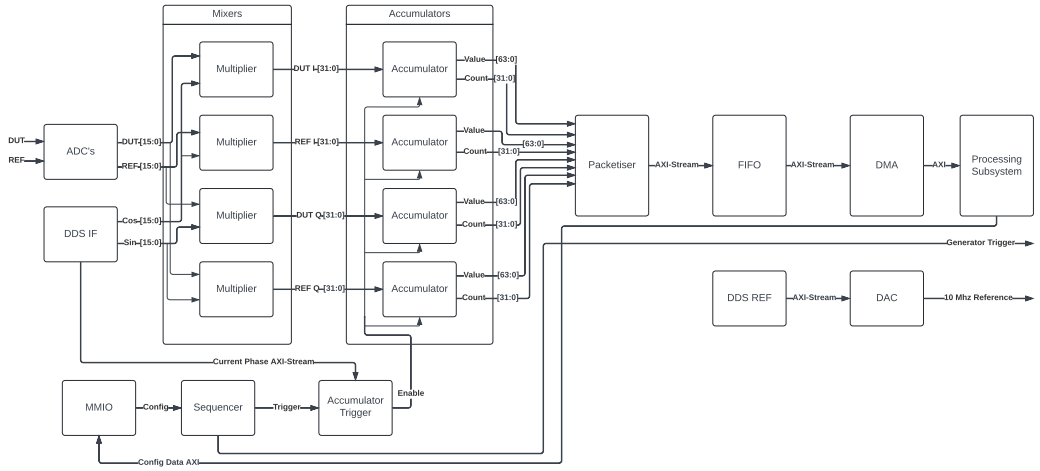

Figure 4.4: High-level and simplified block diagram of the entire logic implementation implemented in the PL.

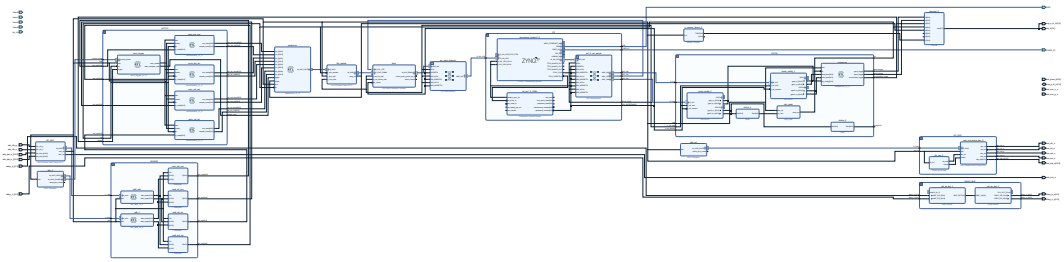

Figure 4.5: Complete block diagram of the entire logic implementation implemented in the PL as exported from Vivado. An enlarged version can be found in appendix B

4.2. Integration

All the functions described in the previous sections are implemented as Intellectual Property blocks (IP blocks) in the Vivado software used to generate a bitstream for the Programmable Logic in the SoC. These blocks combine to form the stages described in section 3.1. In turn, these stages combine to form the complete overlay. These combinations are assisted by several blocks and connections not described in detail as they function simply as ”glue logic”, performing functions such as splitting and merging busses and managing clock domains. A simplified block diagram is shown in figure 4.4, with the complete diagram as exported from Vivado shown in figure 4.5. The logic resource use of the implementation is shown in figure 4.6. This summary shows the implemented logic only takes up a small part of the available resources, leaving ample room for future work.

4.3. Validation of integrated system

The integrated system was tested using a UNI-T UTG962E arbitrary waveform generator.[35] Its two output channels were connected directly to the two input channels of the Red Pitaya board (named DUT and REF) with appropriate termination to prevent signal reflections. The generator was configured to output a 7.8125 MHz sinewave on both of its outputs, mimicking an IF signal coming from the

Name	Slice LUTs (17600)	Slice Registers (35200)	Slice (4400)	LUT as Logic (17600)	LUT as Memory (6000)	Block RAM Tile (60)	DSPs (80)
N system_wrapper	3074	5379	1656	2938	136	34.5	4

Figure 4.6: Overview of available and used logic resources.

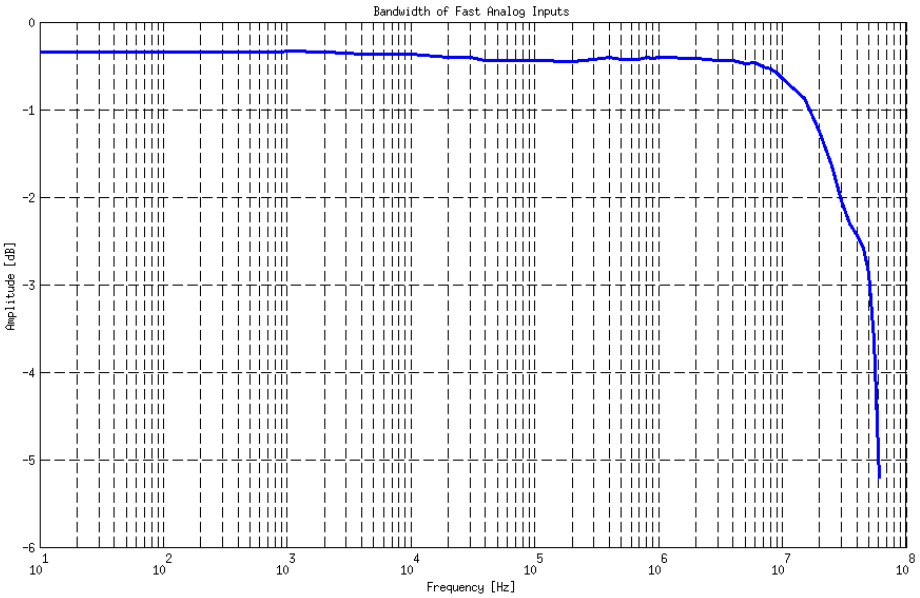

Figure 4.7: Attenuation of the analogue front-end of the Red Pitaya board.[36]

analogue part of the VNA. The amplitude and phase of both of these waves were then varied and measured with the implementation under test. The data was extracted using a simple iPython notebook based on the PYNQ library, included in appendix C.1. The notebook runs a series of tests, testing two different IFBWs[1] for each of three different configurations of the input waves. Each test acquires 1000 points spread over one second. It should be noted that the used function generator is not a precision instrument and as such it will introduce errors in the results. The following analysis takes this into account.

4.3.1. Measurement 1

The first input configuration is two identical sinusoids at 2 V_{pp} with no phase difference, described in phasor form as $\mathrm{F}_{DUT} = 1\angle 0\ V$ and $\mathrm{F}_{REF} = 1\angle 0\ V$. This resulted in the data shown in figure 4.8.
It can be seen that for both channels and both IFBWs, the measured amplitude is approximately 5% lower than expected, with the IFBW of 1 kHz being slightly more attenuated than 10 kHz. This is mostly caused by attenuation in the analogue front-end of the Red Pitaya board, shown in figure 4.7. The expected attenuation of the front-end at 7.8 MHz is approximately -0.3 dB, or about 6.7%. The remaining deviation is approximately 1.7% which is within the specifications of the ADC and function generator. The slight amplitude difference between the IFBW configurations is due to an IF mismatch between the function generator and Red Pitaya board of approximately 100 Hz, the used generator does not have a reference input and thus could not be locked to the board as would be done in use. Both of the mentioned effects affect the channels the same and so the relative amplitude measurement is very accurate, well within 1% of the expected value. The noise is also fairly low at less than -43 dB at 1 kHz IFBW, though this is only a rough estimate and more specific measurements are required to specify it fully. The noise at 10 kHz IFBW is slightly higher, as expected due to the shorter integration time.
The IF mismatch also affects the absolute phase measurements of the individual channels, rendering them unusable. The relative measurement, however, is still valid and accurate to approximately 2.5 mrad.

4.3.2. Measurement 2

The second input configuration is described in phasor form as $\mathrm{F}_{DUT} = 0.5\angle 0\ V$ and $\mathrm{F}_{REF} = 1\angle \frac{\pi}{2}\ V$. This resulted in the data shown in figure 4.9.
This measurement contains an unexpected amplitude modulation in the DUT channel which affects

[1] As a reminder, IFBW is the inverse of accumulation time, not the IQ sampling rate.

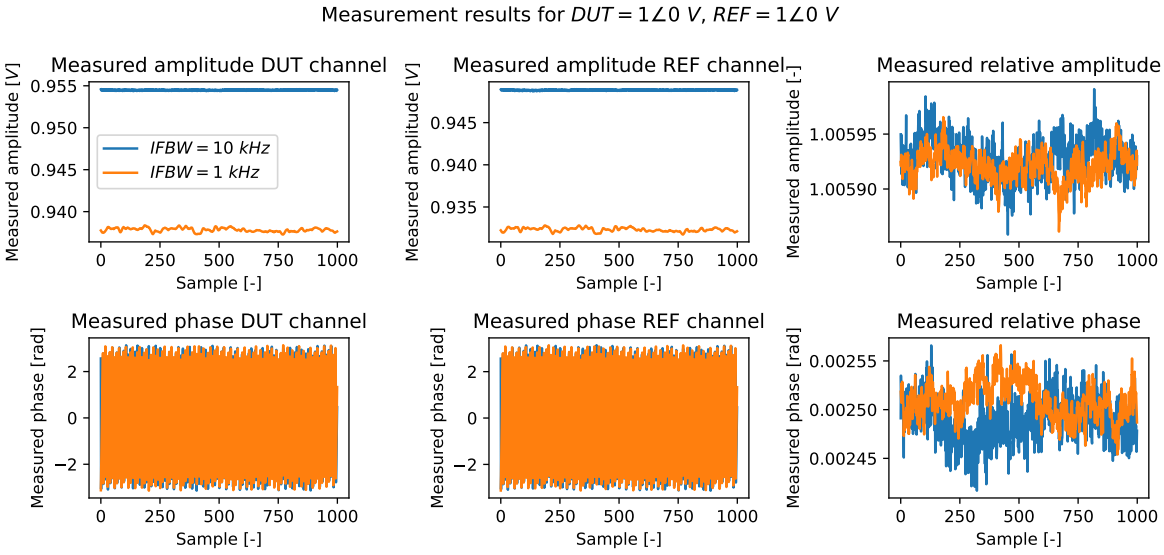

Figure 4.8: Results of measurement 1. Mind the variable vertical scales.

both its absolute measurements and the relative ones. Even still, it can be seen that the measured amplitude is approximately as expected, taking into account the same attenuation effects as in the first measurements. The phase is correctly measured, this time to within 8 mrad. Note the phase is measured as negative due to the phase offset being applied to the REF channel. This measurement also features a more significant difference in noise between the two IFBW configurations, with the lower IFBW resulting in less noise as expected.

4.3.3. Measurement 3

The last input configuration is described in phasor form as $\mathrm{F}_{DUT} = A\angle 0\ V$ and $\mathrm{F}_{REF} = 1\angle\theta\ V$, where A and θ were modulated. The amplitude was modulated using a positive-ramp sawtooth wave between 0.1 V and 1 V_{pp} at a rate of 3 Hz. The phase was modulated similarly between $-\pi$ rad and π rad at 5 Hz. This resulted in the data shown in figure 4.10.

This measurement shows modulating the signal parameters during a measurement works as expected and does not introduce any major cross-modulation effects. Note that the relative phase shows negative-ramp sawtooth modulation while the generator was configured for positive ramp. This is because the phase modulation was assigned to the REF channel instead of the DUT channel, inverting its transfer. This was done because the generator only allows one modulation per channel, so it was chosen to apply amplitude modulation to the DUT channel and phase modulation to the REF channel.

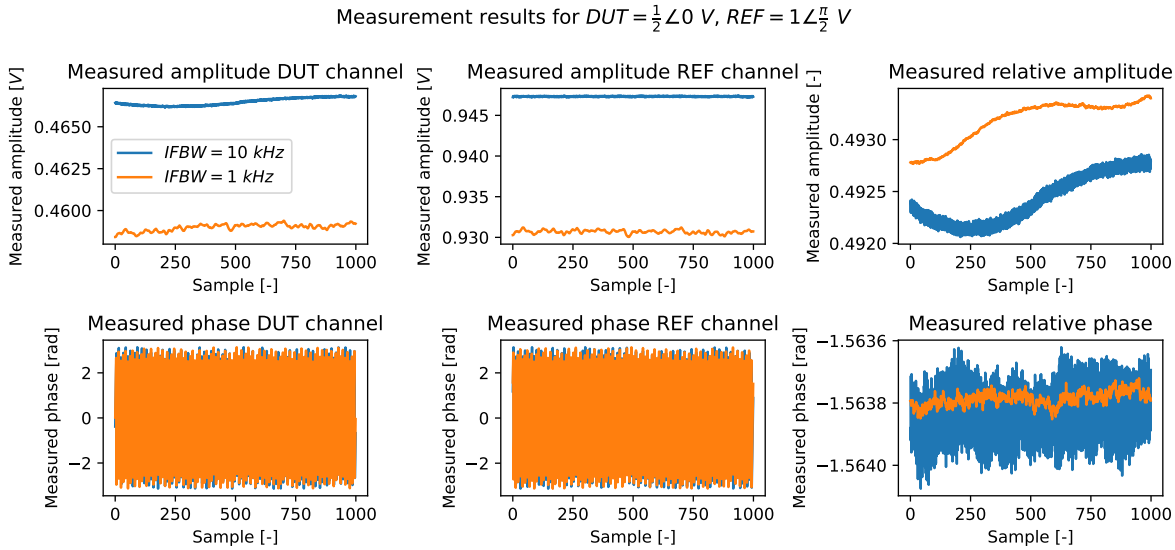

Figure 4.9: Results of measurement 2. Mind the variable vertical scales.

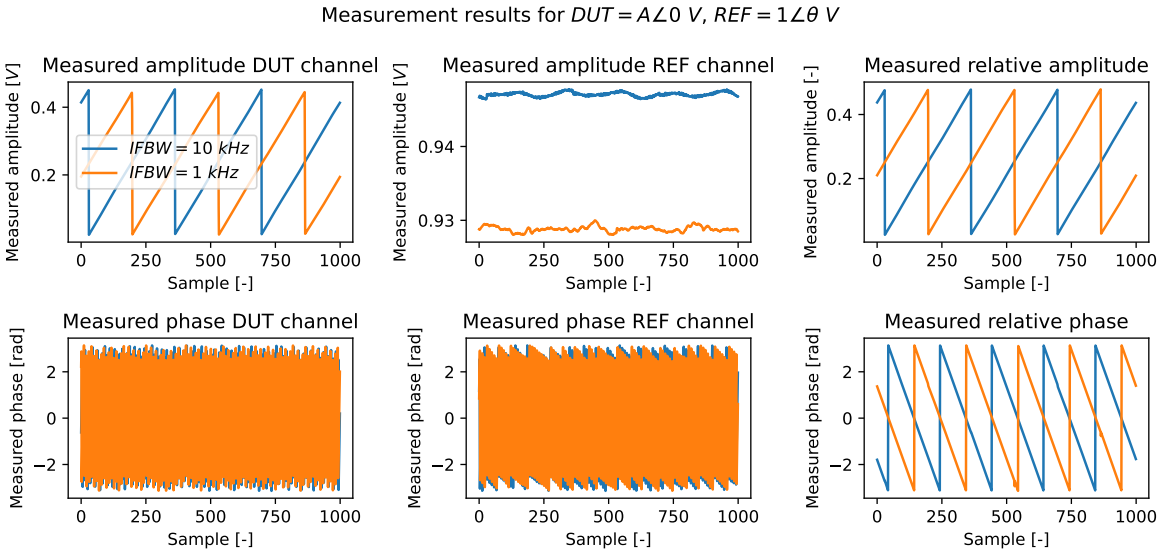

Figure 4.10: Results of measurement 3. A and θ were modulated using positive-ramp sawtooth waves. Mind the variable vertical scales.

5 Discussion

5.1. Shortcomings

The current implementation of the system satisfies all functional requirements set forth at the beginning of this project. However, unfortunately, the allotted time was too limited to finish the non-functional requirements concerning documentation of the system, both for users and future developers. The documentation should include comments in the source Verilog files, as well as a document describing the details of interfacing with the implementation from software. There is already an example iPython notebook demonstrating basic access, this could be expanded to a self-contained library and potentially ported to other languages.
Additionally, the system could not be tested in the context of the rest of the Open-Hardware VNA due to issues with the commercial microwave generators used. In-context testing provides information about performance when used as intended. More out-of-context testing is also recommended to thoroughly characterise the performance of the system and find correction factors for error sources like the attenuation of the front-end or attenuation from IF mismatch.
Both of these shortcomings are candidates for future work.

5.2. Future work

As discussed in the previous section, both documentation and testing of the system are not yet complete and are thus candidates for future work.
Furthermore, as far as tested, the current implementation does not contain any major bugs or glitches, however, there are some known smaller ones and there is potential for imperfections in areas such as cycle-accurate timings of all the state machines. Although fixing these would not have a major impact on the performance of the system, if the project is to be published as a working solution such imperfections should not be present.
Another possible improvement is allowing the configuration of the DMA packet size. During testing, it was observed that DMA transfers incur significant overhead in the Processing System, and so increasing the amount of data transferred at once would allow for faster acquisitions. This has already been implemented but could not be tested due to time constraints.
A useful feature addition for the VNA as a whole is the ability to perform power sweeps. This functionality was not explicitly considered during the design of the PL implementation, but due to the chosen architecture we expect there will be no changes needed as the PL does not need to be aware of the properties of the sweep other than its timings.
Lastly, another additional feature of the system could be a Vector Signal Analyser mode. In this mode, the two channels operate independently to measure the frequency spectrum of the input signal around a certain centre frequency. This can also already be done with the current implementation, though its performance would be very poor due to the frequency transfer of the current method of data reduction. Research into a better method for data reduction and/or switching data reduction methods during runtime would be hugely beneficial for such a feature.

6

Conclusion

In this thesis, an FPGA implementation of Real-Time Digital Signal Processing for an Open-Hardware Vector Network Analyser has been described. The user context of qubit research was identified, as well as existing commercial and open VNAs. Requirements for the DSP were set, and a theory of operation was devised. The theory of operation and design decisions of the different parts were supported mathematically and their logical implementation described. The testing of these component implementations as well as the integration and testing of the whole system were described. Several out-of-context test measurements were analysed and were found to be satisfactory to the functional requirements. The thesis concluded with a discussion of the shortcomings in the non-functional requirements of the current implementation and identification of potential future work.

References

[1] *What is a Vector Network Analyzer and How Does it Work?* [Online]. Available: https://www.tek.com/en/documents/primer/what-vector-network-analyzer-and-how-does-it-work.

[2] L. Zhong, R. Yu, and X. Hong, "Review of carbon-based electromagnetic shielding materials: film, composite, foam, textile," *Textile Research Journal*, vol. 91, p. 004 051 752 096 828, Oct. 2020. DOI: 10.1177/0040517520968282.

[3] F. Caspers, "RF engineering basic concepts: S-parameters," CERN, Tech. Rep., 2013.

[4] *What Can You Do With a VNA?* 2023. [Online]. Available: https://coppermountaintech.com/what-can-you-do-with-a-vna/.

[5] L. Dicarlo, *Introduction to circuit QED*, 2023.

[6] Keysight. "Product page: P9372A Keysight Streamline USB Vector Network Analyzer, 9 GHz." (2024), [Online]. Available: https://www.keysight.com/us/en/product/P9372A/keysight-streamline-usb-vector-network-analyzer-9-ghz.html.

[7] "About nanovna." (), [Online]. Available: https://nanovna.com.

[8] "Shfqa+ 8.5 ghz quantum analyzer." (), [Online]. Available: https://www.zhinst.com/europe/en/products/shfqa-quantum-analyzer.

[9] "Opx+: Ultra-fast quantum controller." (), [Online]. Available: https://www.quantum-machines.co/products/opx/#.

[10] A. Raza, A. Jabbar, D. A. Sehrai, H. Atiq, and R. Ramzan, "SDR Based VNA for Characterization of RF Sensors and Circuits," in *2021 1st International Conference on Microwave, Antennas & Circuits (ICMAC)*, 2021, pp. 1–4. DOI: 10.1109/ICMAC54080.2021.9678273.

[11] H. Forstén, "Improved homemade VNA," Tech. Rep., 2017. [Online]. Available: https://hforsten.com/improved-homemade-vna.html.

[12] J. Mower and Y. Kuga, "A FPGA-Based Replacement for a Network Analyzer in an Instrumentation-Based 200 GHz Radar," *High Frequency Electronics*, pp. 30–40, Sep. 2013.

[13] Y. Xu, G. Huang, N. Fruitwala, *et al.*, *QubiC 2.0: An Extensible Open-Source Qubit Control System Capable of Mid-Circuit Measurement and Feed-Forward*, 2023. arXiv: 2309.10333 [quant-ph].

[14] *STEMlab 125-14 - Red Pitaya*, en-US, Jun. 2021. [Online]. Available: https://redpitaya.com/stemlab-125-14/ (visited on 04/25/2024).

[15] *LTC2145-14 Datasheet and Product Info*. [Online]. Available: https://www.analog.com/en/products/ltc2145-14.html (visited on 04/25/2024).

[16] *AD9767 Datasheet and Product Info*. [Online]. Available: https://www.analog.com/en/products/AD9767.html (visited on 06/13/2024).

[17] C. M. Technologies. "What are the basics of a vna? - copper mountain technologies." (), [Online]. Available: https://coppermountaintech.com/what-are-the-basics-of-vna/ (visited on 09/29/2023).

[18] E. Cheever. "Phasor introduction and demo." (), [Online]. Available: https://lpsa.swarthmore.edu/BackGround/phasor/phasor.html.

[19] E. Li Tan. "Eetimes - multirate dsp, part 1: Upsampling and downsampling." (), [Online]. Available: https://www.eetimes.com/multirate-dsp-part-1-upsampling-and-downsampling/?_ga (visited on 04/21/2008).

[20] M. Zoran Milivojević. "Types-of-digital-filters - mikroe." (), [Online]. Available: https://www.mikroe.com/ebooks/digital-filter-design/types-of-digital-filters.

[21] R. Lyons. “A beginner’s guide to cascaded integrator-comb (cic) filters.” (), [Online]. Available: https://www.dsprelated.com/showarticle/1337.php (visited on 03/26/2020).

[22] S. Roberts. “Lecture 7 - the discrete fourier transform.” (), [Online]. Available: https://www.robots.ox.ac.uk/~sjrob/Teaching/SP/l7.pdf.

[23] “IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language,” *IEEE Std 1800-2023 (Revision of IEEE Std 1800-2017)*, pp. 1–1354, Feb. 2024, Conference Name: IEEE Std 1800-2023 (Revision of IEEE Std 1800-2017). DOI: 10.1109/IEEESTD.2024.10458102. [Online]. Available: https://ieeexplore-ieee-org.tudelft.idm.oclc.org/document/10458102 (visited on 06/10/2024).

[24] *Zynq-7000 SoC Product Selection Guide*. [Online]. Available: https://docs.amd.com/api/khub/documents/1L_hkh2pbc5l0Oz7tcLspA/content?Ft-Calling-App=ft%2Fturnkey-portal&Ft-Calling-App-Version=4.3.44# (visited on 06/14/2024).

[25] *4.1.1.2. STEMlab 125-14 — Red Pitaya 2.00-35 documentation*. [Online]. Available: https://redpitaya.readthedocs.io/en/latest/developerGuide/hardware/125-14/top.html (visited on 06/14/2024).

[26] *AMBA AXI-Stream Protocol Specification*. [Online]. Available: https://developer.arm.com/documentation/ihi0051/latest (visited on 06/10/2024).

[27] *Zynq-7000 SoC Technical Reference Manual*, Jan. 2018.

[28] *AXI GPIO v2.0 Product Guide*. [Online]. Available: https://docs.amd.com/api/khub/documents/0c0ItRCmnYkoHpcYUCPkEA/content?Ft-Calling-App=ft%2Fturnkey-portal&Ft-Calling-App-Version=4.3.43# (visited on 06/11/2024).

[29] *All About Direct Digital Synthesis | Analog Devices*. [Online]. Available: https://www.analog.com/en/resources/analog-dialogue/articles/all-about-direct-digital-synthesis.html (visited on 06/13/2024).

[30] *DDS Compiler*, en. [Online]. Available: https://www.xilinx.com/products/intellectual-property/dds_compiler.html (visited on 06/13/2024).

[31] *Multiplier v12.0 Product Guide*, Nov. 2015. [Online]. Available: https://docs.amd.com/api/khub/documents/idOj3Pp9ocZdMoFz3IpkjQ/content?Ft-Calling-App=ft%2Fturnkey-portal&Ft-Calling-App-Version=4.3.44# (visited on 06/15/2024).

[32] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, en. Pearson, 2015, Google-Books-ID: 9gqnngEACAAJ, ISBN: 978-0-13-359162-0.

[33] *AXI4-Stream Infrastructure IP Suite v3.0 LogiCORE IP Product Guide*, Aug. 2022. [Online]. Available: https://docs.amd.com/r/en-US/pg085-axi4stream-infrastructure/AXI4-Stream-Infrastructure-IP-Suite-v3.0-LogiCORE-IP-Product-Guide (visited on 06/15/2024).

[34] P. Demin. “Red pitaya notes.” (), [Online]. Available: https://pavel-demin.github.io/red-pitaya-notes/.

[35] UNI-T. “Utg900e - waveform generators - products - uni-t voltage meter, multimeter, oscilloscope | uni-t.” (), [Online]. Available: https://instruments.uni-trend.com/cate/22.html.

[36] R. Pitaya. “Stemlab 125-14 — red pitaya 2.00-35 documentation.” (), [Online]. Available: https://redpitaya.readthedocs.io/en/latest/developerGuide/hardware/125-14/top.html.

A

Verilog Source Code

A.1. Accumulator

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 07.05.2024 14:43:35
// Design Name:
// Module Name: accumulator
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module accumulator(
    input aclk,
    input enable,
    input reset, // Active low
    input [31:0] value_in,
    output [63:0] out_value,
    output [31:0] sample_count
    );

        // Create three states
    localparam state_width = 3;

    localparam state_waiting = 'd0;
    localparam state_accumulating = 'd1;
    localparam state_output = 'd2;

        // Create registers
    reg [state_width-1:0] state;
    reg [state_width-1:0] newstate;
    reg [63:0] accu_total;
    reg [31:0] samplecountreg;

        // Change state to new state on positive clockedge
    always @(posedge aclk) begin
      if (!reset)
```

```
        state <= state_waiting;
      else
        state <= newstate;
    end

        // State change conditions
    always @* begin
        case(state)
            state_waiting: begin
                if (enable) newstate = state_accumulating;
                else newstate = state_waiting;
            end
            state_accumulating: begin
                if (enable == 1'b0) newstate = state_output;
                else newstate = state_accumulating;
            end
            state_output: begin
                newstate = state_waiting;
            end
            default: newstate = state_waiting;
        endcase
    end

    // Outputs for every state
    always @(posedge aclk) begin
        case(state)
            state_waiting: begin
                accu_total <= 64'd0;
                samplecountreg <= 32'd0;
            end
            state_accumulating: begin
                                // Sign extend the value coming in and add it to the (64-bit) total
                                accu_total <= accu_total + {{32{value_in[31]}}, value_in[31:0]};
                samplecountreg <= samplecountreg + 1; // Increment sample count by one
            end
            state_output: begin
                accu_total <= accu_total + {{32{value_in[31]}}, value_in[31:0]};
                samplecountreg <= samplecountreg + 1;
            end
            default: begin
                accu_total <= 64'd0;
                samplecountreg <= 32'd0;
            end
        endcase
     end

         // Permanently connect the registers to the outputs
     assign sample_count = samplecountreg;
     assign out_value = accu_total;
endmodule
```

A.2. Accumulator trigger

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 07.05.2024 14:47:45
// Design Name:
// Module Name: accu_trigger
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
```

```
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module accu_trigger(
    input aclk,
    input reset,
    input trigger,
    input [31:0] current_phase_tdata,
    input current_phase_tvalid,
    output accu_enable,
    output done_samples_valid
    );

        // Declare all possible states
    localparam state_width = 3;

    localparam state_stopped = 'd0;
    localparam state_starting = 'd1;
    localparam state_running = 'd2;
    localparam state_waiting = 'd3;
    localparam state_ending = 'd4;

        // Declare registers
    reg [state_width-1:0] state;
    reg [state_width-1:0] newstate;
    reg [31:0] phase;
    reg accu_enable_reg;
    reg dsv_reg;

    // Change state to new state on positive clockedge
    always @(posedge aclk) begin
      if (!reset)
        state <= state_stopped;
      else
        state <= newstate;
    end

        // State change conditions
    always @* begin
        case(state)
            state_stopped: begin
                if (trigger) newstate = state_starting;
                else newstate = state_stopped;
                end
            state_starting: begin
                newstate = state_running;
                end
            state_running: begin
                if (!trigger) newstate = state_waiting;
                else newstate = state_running;
                end
            state_waiting: begin
                if (current_phase_tdata == phase) newstate = state_ending;
                else newstate = state_waiting;
                end
            state_ending: begin
                newstate = state_stopped;
                end
            default: newstate = state_stopped;
        endcase
    end

    // Outputs for every state
    always @(posedge aclk) begin
        case(state)
            state_stopped: begin
                accu_enable_reg <= 1'b0;
```

```
                dsv_reg <= 1'b0;
                end
            state_starting: begin
                accu_enable_reg <= 1'b0;
                dsv_reg <= 1'b0;
                phase <= current_phase_tdata;
                end
            state_running: begin
                accu_enable_reg <= 1'b1;
                dsv_reg <= 1'b0;
                end
            state_waiting: begin
                accu_enable_reg <= 1'b1;
                dsv_reg <= 1'b0;
                end
            state_ending: begin
                accu_enable_reg <= 1'b0;
                dsv_reg <= 1'b1;
                end
            default: begin
                accu_enable_reg <= 1'b0;
                dsv_reg <= 1'b0;
                end
        endcase
    end

        // Permanently connect the registers to the outputs
    assign accu_enable = accu_enable_reg;
    assign done_samples_valid = dsv_reg;

endmodule
```

A.3. Accumulator and trigger testbench

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 17.05.2024 15:26:14
// Design Name:
// Module Name: tb_accusys
// Project Name:
// Target Devices:
// Tool Versions:
// Description: Testbench for accumulator and accu_trigger modules
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module tb_accusys();

    // Clock and reset signals
    reg aclk;                       // Clock signal
    reg reset;                      // Reset signal

    // accu_trigger module signals
    reg trigger;                    // Trigger signal to start the accumulation
    reg [31:0] current_phase_tdata;  // Current phase data
    reg current_phase_tvalid;       // Valid signal for current phase data
    wire accu_enable;               // Enable signal for the accumulator (output from
        accu_trigger)
    wire done_samples_valid;        // Signal indicating done samples are valid (output from
        accu_trigger)

```

```
35      // accumulator module signals
36      reg [31:0] value_in;          // Input value to the accumulator
37      wire [63:0] out_value;        // Accumulated output value from the accumulator
38      wire [31:0] sample_count;     // Sample count output from the accumulator
39
40      // Instantiate the accu_trigger module
41      accu_trigger uutTrig (
42          .aclk(aclk),
43          .reset(reset),
44          .trigger(trigger),
45          .current_phase_tdata(current_phase_tdata),
46          .current_phase_tvalid(current_phase_tvalid),
47          .accu_enable(accu_enable),
48          .done_samples_valid(done_samples_valid)
49      );
50
51      // Instantiate the accumulator module
52      accumulator uutAcc (
53          .aclk(aclk),
54          .enable(accu_enable),
55          .reset(reset),
56          .value_in(value_in),
57          .out_value(out_value),
58          .sample_count(sample_count)
59      );
60
61      // Clock
62      initial begin
63          aclk = 1'b0;
64          forever #5 aclk = ~aclk;
65      end
66
67      // Initial reset sequence
68      initial begin
69          reset = 1'b0;
70          #10 reset = 1'b1;
71      end
72
73      // Test sequence
74      initial begin
75          // Initialize input signals
76          current_phase_tvalid = 1; // Set phase data valid
77          trigger = 0;              // Initialize trigger to 0
78          value_in = 32'd0;         // Initialize value_in
79          current_phase_tdata = 32'd0; // Initialize current phase data
80
81          // Wait for some time before starting the trigger
82          #100;
83
84          // Start the accumulation process with a trigger
85          trigger = 1; // Assert trigger to start accumulation
86          //#10 trigger = 0; // Deassert trigger
87
88          // Simulate accumulating values
89          #10 value_in = 32'd10; // First value to accumulate
90          current_phase_tdata = 32'd1; // Change phase data
91
92          #10 value_in = 32'd10; // Second value to accumulate
93          current_phase_tdata = 32'd2; // Change phase data
94
95          #10 value_in = 32'd10; // Third value to accumulate
96          current_phase_tdata = 32'd3; // Change phase data
97
98          #10 value_in = 32'd20;
99          current_phase_tdata = 32'd4;
100
101                 #10 value_in = 32'd23;
102         current_phase_tdata = 32'd5;
103
104                 #10 value_in = 32'd52;
105         current_phase_tdata = 32'd6;
```

```
106
107                 #10 value_in = 32'd43;
108         current_phase_tdata = 32'd7;
109                         trigger = 0; // Deassert trigger
110
111                 #10 value_in = 32'd61;
112         current_phase_tdata = 32'd0;
113
114                 #10 value_in = 32'd33;
115         current_phase_tdata = 32'd1;
116
117         #10 value_in = 32'd89;
118         current_phase_tdata = 32'd2;
119
120         #10 value_in = 32'd93;
121         current_phase_tdata = 32'd3;
122
123         #10 value_in = 32'd91;
124         current_phase_tdata = 32'd4;
125
126                 #10 value_in = 32'd123;
127         current_phase_tdata = 32'd5;
128
129                 #10 value_in = 32'd32;
130         current_phase_tdata = 32'd6;
131
132                 #10 value_in = 32'd22;
133         current_phase_tdata = 32'd7;
134
135                 #10 value_in = 32'd12;
136         current_phase_tdata = 32'd0;
137
138                 #10 value_in = 32'd0;
139
140         // Wait and observe the output
141         #100;
142
143         // Print results
144         $display("out_value: %d", out_value);
145         $display("sample_count: %d", sample_count);
146         $display("accu_enable: %d", accu_enable);
147         $display("done_samples_valid: %d", done_samples_valid);
148
149         // End simulation
150         #20 $stop;
151     end
152
153 endmodule
```

A.4. Sequencer

```
1  `timescale 1ns / 1ps
2  //////////////////////////////////////////////////////////////////////////////////
3  // Company:
4  // Engineer:
5  //
6  // Create Date: 24.05.2024 08:35:10
7  // Design Name:
8  // Module Name: sequencer
9  // Project Name:
10 // Target Devices:
11 // Tool Versions:
12 // Description:
13 //
14 // Dependencies:
15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
```

```
////////////////////////////////////////////////////////////////////////////////


module sequencer(
    input aclk,
    input rst,
    input [31:0] dead_time,
    input [31:0] point_time,
    input [31:0] trig_config,
    output gen1_trigger,
    output gen2_trigger,
    output accumulator_trigger
    );

    reg [31:0] counter;
    reg first_sample;

    reg [31:0] dead_time_reg;
    reg [31:0] point_time_reg;
    // Trigger config input is split into multiple registers
    reg [31:0] trig_time_reg;
    reg trig1_invert_reg;
    reg trig1_first_reg;
    reg trig1_rest_reg;
    reg trig2_invert_reg;
    reg trig2_first_reg;
    reg trig2_rest_reg;

    reg accumulator_trigger_reg;
    reg generator_trigger_reg;

    wire gen1_first, gen1_rest;
    wire gen2_first, gen2_rest;

    // Sequential logic to count clock cycles and handle reset
    always @(posedge aclk) begin
        if (!rst) begin
            // Reset state
            counter <= 32'b0;
            first_sample <= 1'b1;
            accumulator_trigger_reg <= 1'b0;
            generator_trigger_reg <= 1'b0;

            // Parse and latch inputs
                        dead_time_reg <= dead_time;
                        point_time_reg <= point_time;
                        trig_time_reg <= {8'b0, trig_config[23:0]};
                        trig1_invert_reg <= trig_config[24];
                        trig1_first_reg <= trig_config[25];
                        trig1_rest_reg <= trig_config[26];
                        trig2_invert_reg <= trig_config[28];
                        trig2_first_reg <= trig_config[29];
                        trig2_rest_reg <= trig_config[30];
        end
        else begin
            // Increment counter on every clock cycle
            counter <= counter + 1;

            // Check if we should be outputting generator trigger
            if (counter < trig_time_reg) begin
                generator_trigger_reg <= 1'b1;
            end
            else begin
                generator_trigger_reg <= 1'b0;
            end

            // Check if we should be outputting accumulator trigger
            // gt instead of ge means we lose a cycle but that is required for packetiser
                trigger
            if (counter > dead_time_reg) begin
                accumulator_trigger_reg <= 1'b1;
```

```
            end
            else begin
                accumulator_trigger_reg <= 1'b0;
            end

            // End of sample
            if (counter >= point_time_reg) begin
                // Reset counter but clear first_sample
                counter <= 32'd0;
                first_sample <= 1'b0;
                // Don't latch inputs again, a sweep should have constant settings
            end
        end
    end

//      assign gentrig_point = generator_trigger_reg;
//      assign gentrig_sweep = first_sample ? generator_trigger_reg : 1'b0;
//      assign gentrig_not_first = first_sample ? 1'b0 : generator_trigger_reg;
//      assign gen1_trigger = trig1_invert_reg ^ (trig1_type_reg ? (trig1_first_reg ? gentrig_point : gentrig_not_first) : gentrig_sweep);

        assign gen1_first = trig1_first_reg & generator_trigger_reg;
        assign gen1_rest = trig1_rest_reg & generator_trigger_reg;
        assign gen1_trigger = trig1_invert_reg ^ (first_sample ? gen1_first : gen1_rest);

        assign gen2_first = trig2_first_reg & generator_trigger_reg;
        assign gen2_rest = trig2_rest_reg & generator_trigger_reg;
        assign gen2_trigger = trig2_invert_reg ^ (first_sample ? gen2_first : gen2_rest);

        assign accumulator_trigger = accumulator_trigger_reg;

endmodule
```

A.5. Sequencer testbench

```
`timescale 1ns / 1ps

module sequencer_tb;

    // Testbench signals
    reg [31:0] settling_time;
    reg [31:0] accu_time;
    reg [31:0] gen_end;
        reg [31:0] counter_out;
    reg aclk;
    reg reset;

    wire generator_trigger;
    wire accumulator_trigger;

    // Instantiate the sequencer module
    sequencer uut (
        .settling_time(settling_time),
        .accu_time(accu_time),
        .gen_end(gen_end),
        .aclk(aclk),
        .reset(reset),
        .generator_trigger(generator_trigger),
        .accumulator_trigger(accumulator_trigger),
                .counter_out(counter_out)
    );

    // Clock generation
    initial begin
        aclk = 0;
        forever #5 aclk = ~aclk; // 10ns period clock
    end

    // Stimulus process
    initial begin
```

```
        // Initialize inputs
        reset = 0;
                // It will act two clock cycles later
        settling_time = 32'd50;
        accu_time = 32'd150;
        gen_end = 32'd20;

        // Wait for a few clock cycles
        #20;

        // Deassert reset
        reset = 1;

        // Wait for some time to observe the behavior
        #1500;

        // Assert reset again to observe the reset behavior
        //reset = 1;
        //#20;
        //reset = 0;

        // Wait and then finish simulation
        #2000;
        $finish;
    end

    // Monitor the outputs
    initial begin
        //$monitor("Time: %0d, reset: %b, counter: %d, generator_trigger: %b,
            accumulator_trigger: %b",
        //         $time, reset, uut.counter, generator_trigger, accumulator_trigger);
    end

endmodule
```

A.6. Packetiser

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 22.05.2024 11:58:55
// Design Name:
// Module Name: packetiser
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

// Made by ChatGPT without modification..

module packetiser(
    input [63:0] val_1,
    input [31:0] cnt_1,
    input [63:0] val_2,
    input [31:0] cnt_2,
    input [63:0] val_3,
    input [31:0] cnt_3,
    input [63:0] val_4,
    input [31:0] cnt_4,
    input trigger,
```

```
    input aclk,
    input rst,
    output reg [31:0] M_AXIS_OUT_tdata,
    output reg M_AXIS_OUT_tvalid,
    input M_AXIS_OUT_tready,
    output reg M_AXIS_OUT_tlast
    );

    // State machine states
    localparam IDLE = 2'b00;
    localparam SEND_WORDS = 2'b01;

    reg [1:0] state, next_state;

    // Registers to hold the data
    reg [63:0] val_reg_1, val_reg_2, val_reg_3, val_reg_4;
    reg [31:0] cnt_reg_1, cnt_reg_2, cnt_reg_3, cnt_reg_4;
    reg [3:0] word_count;

    // Registering input values on trigger
    always @(posedge aclk) begin
        if (!rst) begin
            val_reg_1 <= 64'b0;
            cnt_reg_1 <= 32'b0;
            val_reg_2 <= 64'b0;
            cnt_reg_2 <= 32'b0;
            val_reg_3 <= 64'b0;
            cnt_reg_3 <= 32'b0;
            val_reg_4 <= 64'b0;
            cnt_reg_4 <= 32'b0;
        end else if (trigger) begin
            val_reg_1 <= val_1;
            cnt_reg_1 <= cnt_1;
            val_reg_2 <= val_2;
            cnt_reg_2 <= cnt_2;
            val_reg_3 <= val_3;
            cnt_reg_3 <= cnt_3;
            val_reg_4 <= val_4;
            cnt_reg_4 <= cnt_4;
        end
    end

    // State machine for packetizing data
    always @(posedge aclk) begin
        if (!rst) begin
            state <= IDLE;
        end else begin
            state <= next_state;
        end
    end

    always @(*) begin
        next_state = state;
        case (state)
            IDLE: begin
                if (trigger) begin
                    next_state = SEND_WORDS;
                end
            end
            SEND_WORDS: begin
                if (word_count == 4'd11 && M_AXIS_OUT_tready) begin
                    next_state = IDLE;
                end
            end
        endcase
    end

    always @(posedge aclk) begin
        if (!rst) begin
            word_count <= 4'b0;
            M_AXIS_OUT_tdata <= 32'b0;
```

```
            M_AXIS_OUT_tvalid <= 1'b0;
            M_AXIS_OUT_tlast <= 1'b0;
        end else if (state == SEND_WORDS) begin
            if (M_AXIS_OUT_tready) begin
                word_count <= word_count + 4'b1;
                M_AXIS_OUT_tvalid <= 1'b1;

                case (word_count)
                    4'd0: M_AXIS_OUT_tdata <= val_reg_1[31:0];
                    4'd1: M_AXIS_OUT_tdata <= val_reg_1[63:32];
                    4'd2: M_AXIS_OUT_tdata <= cnt_reg_1;
                    4'd3: M_AXIS_OUT_tdata <= val_reg_2[31:0];
                    4'd4: M_AXIS_OUT_tdata <= val_reg_2[63:32];
                    4'd5: M_AXIS_OUT_tdata <= cnt_reg_2;
                    4'd6: M_AXIS_OUT_tdata <= val_reg_3[31:0];
                    4'd7: M_AXIS_OUT_tdata <= val_reg_3[63:32];
                    4'd8: M_AXIS_OUT_tdata <= cnt_reg_3;
                    4'd9: M_AXIS_OUT_tdata <= val_reg_4[31:0];
                    4'd10: M_AXIS_OUT_tdata <= val_reg_4[63:32];
                    4'd11: begin
                        M_AXIS_OUT_tdata <= cnt_reg_4;
                        M_AXIS_OUT_tlast <= 1'b1;
                    end
                endcase
            end
        end else begin
            word_count <= 4'b0;
            M_AXIS_OUT_tvalid <= 1'b0;
            M_AXIS_OUT_tlast <= 1'b0;
        end
    end

endmodule
```

A.7. Packetiser testbench and top-level module

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 23.05.2024 14:32:10
// Design Name:
// Module Name: tb_tb_packetiser
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module tb_top_module;

    reg aclk;
    reg rst;
    wire [31:0] M_AXIS_OUT_tdata;
    wire M_AXIS_OUT_tvalid;
    reg M_AXIS_OUT_tready;
    wire M_AXIS_OUT_tlast;

    // Instantiate the top module
    top_module uut (
        .aclk(aclk),
```

```
        .rst(rst),
        .M_AXIS_OUT_tdata(M_AXIS_OUT_tdata),
        .M_AXIS_OUT_tvalid(M_AXIS_OUT_tvalid),
        .M_AXIS_OUT_tready(M_AXIS_OUT_tready),
        .M_AXIS_OUT_tlast(M_AXIS_OUT_tlast)
    );

    // Clock generation
    initial begin
        aclk = 0;
        forever #4 aclk = ~aclk;  // 125 MHz clock -> 8ns period -> 4ns half period
    end

    // Reset generation
    initial begin
        rst = 0;
        #20 rst = 1;  // Assert reset for 20ns
    end

    // Stimulate M_AXIS_OUT_tready
    initial begin
        M_AXIS_OUT_tready = 0;
        #50 M_AXIS_OUT_tready = 1;
        // Keep M_AXIS_OUT_tready asserted
        forever #80 M_AXIS_OUT_tready = ~M_AXIS_OUT_tready;
    end

    // Simulation duration
    initial begin
        #100000 $stop;  // Run the simulation for 100us
    end

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 23.05.2024 14:28:39
// Design Name:
// Module Name: tb_packetiser
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

// ChatGPT without modification

module top_module (
    input aclk,
    input rst,
    output [31:0] M_AXIS_OUT_tdata,
    output M_AXIS_OUT_tvalid,
    input M_AXIS_OUT_tready,
    output M_AXIS_OUT_tlast
    );

    wire [63:0] val_1, val_2, val_3, val_4;
    wire [31:0] cnt_1, cnt_2, cnt_3, cnt_4;
    wire trigger;

    // Instantiate the test_packetiser module
```

```
    test_packetiser test_inst (
        .aclk(aclk),
        .rst(rst),
        .val_1(val_1),
        .cnt_1(cnt_1),
        .val_2(val_2),
        .cnt_2(cnt_2),
        .val_3(val_3),
        .cnt_3(cnt_3),
        .val_4(val_4),
        .cnt_4(cnt_4),
        .trigger(trigger)
    );

    // Instantiate the packetiser module
    packetiser packet_inst (
        .val_1(val_1),
        .cnt_1(cnt_1),
        .val_2(val_2),
        .cnt_2(cnt_2),
        .val_3(val_3),
        .cnt_3(cnt_3),
        .val_4(val_4),
        .cnt_4(cnt_4),
        .trigger(trigger),
        .aclk(aclk),
        .rst(rst),
        .M_AXIS_OUT_tdata(M_AXIS_OUT_tdata),
        .M_AXIS_OUT_tvalid(M_AXIS_OUT_tvalid),
        .M_AXIS_OUT_tready(M_AXIS_OUT_tready),
        .M_AXIS_OUT_tlast(M_AXIS_OUT_tlast)
    );

endmodule
```

A.8. Sample data generator for packetiser testbench

```

// Made by ChatGPT with modifications

module test_packetiser(
    input aclk,
    input rst,
    output reg [63:0] val_1,
    output reg [31:0] cnt_1,
    output reg [63:0] val_2,
    output reg [31:0] cnt_2,
    output reg [63:0] val_3,
    output reg [31:0] cnt_3,
    output reg [63:0] val_4,
    output reg [31:0] cnt_4,
    output reg trigger
    );

    reg [31:0] cycle_count;

    always @(posedge aclk) begin
        if (!rst) begin
            cycle_count <= 32'b0;
            trigger <= 1'b0;
            val_1 <= 64'h12345678ABCDEF01;
            cnt_1 <= 32'h00000001;
            val_2 <= 64'h23456789BCDEF012;
            cnt_2 <= 32'h00000002;
            val_3 <= 64'h3456789ACDEF0123;
            cnt_3 <= 32'h00000003;
            val_4 <= 64'h456789ABDEF01234;
            cnt_4 <= 32'h00000004;
        end else begin
            if (cycle_count < 32'd1000) begin
```

```
                cycle_count <= cycle_count + 1;
                trigger <= 1'b0;
            end else if (cycle_count == 32'd1000) begin
                cycle_count <= 32'd0;
                //cycle_count <= cycle_count + 1;
                trigger <= 1'b1;
                // Change values for each trigger if needed
                val_1 <= val_1 + 64'h0101010101010101;
                cnt_1 <= cnt_1 + 32'h01010101;
                val_2 <= val_2 + 64'h0101010101010101;
                cnt_2 <= cnt_2 + 32'h01010101;
                val_3 <= val_3 + 64'h0101010101010101;
                cnt_3 <= cnt_3 + 32'h01010101;
                val_4 <= val_4 + 64'h0101010101010101;
                cnt_4 <= cnt_4 + 32'h01010101;
//              end else begin
//                  cycle_count <= cycle_count;
//                  trigger <= 1'b0;
            end
        end
    end

endmodule
```

A.9. FIFO filler

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 17.05.2024 11:26:46
// Design Name:
// Module Name: fifo_filler
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module fifo_filler #
(
    parameter integer AXIS_TDATA_WIDTH = 32,
    parameter integer VALUE_COUNT = 8
)
(
    input aclk,
    input rst,
    output [AXIS_TDATA_WIDTH-1:0] m_axis_tdata,
    output m_axis_tvalid,
    input m_axis_tready,
    output m_axis_tlast
);

    reg [AXIS_TDATA_WIDTH-1:0]  counter, next_count;
    reg                         done, next_done;

    always @(posedge aclk) begin
        if (!rst) begin
            counter = 10;
            done = 0;
        end
```

```
45          if (m_axis_tready && !done) begin
46              counter = next_count;
47              done = next_done;
48          end
49      end
50
51      always @* begin
52          next_count = counter + 1;
53          next_done = next_count >= (VALUE_COUNT+10);
54      end
55
56      assign m_axis_tdata = counter;
57      assign m_axis_tvalid = !done;
58      assign m_axis_tlast = next_done && !done;
59
60  endmodule
```

B

Implementation Block Diagrams

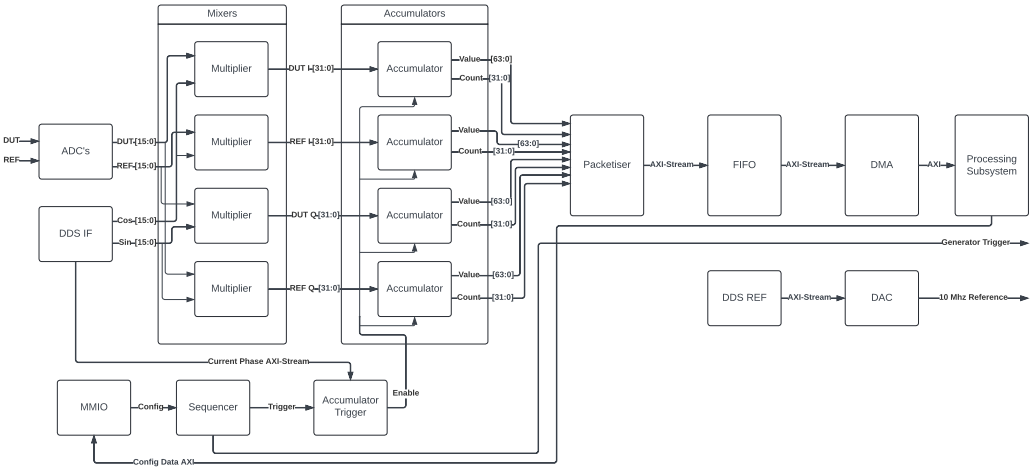

Figure B.1: Simplified block diagram of the entire logic implementation implemented in the PL.

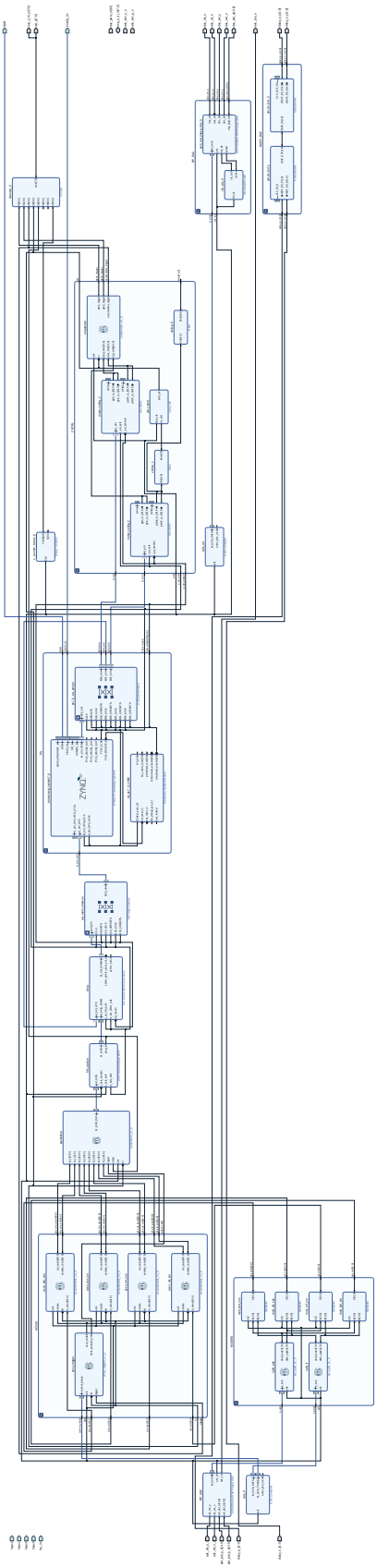

Figure B.2: An enlarged version of the complete block diagram of the entire logic implementation implemented in the PL as exported from Vivado.

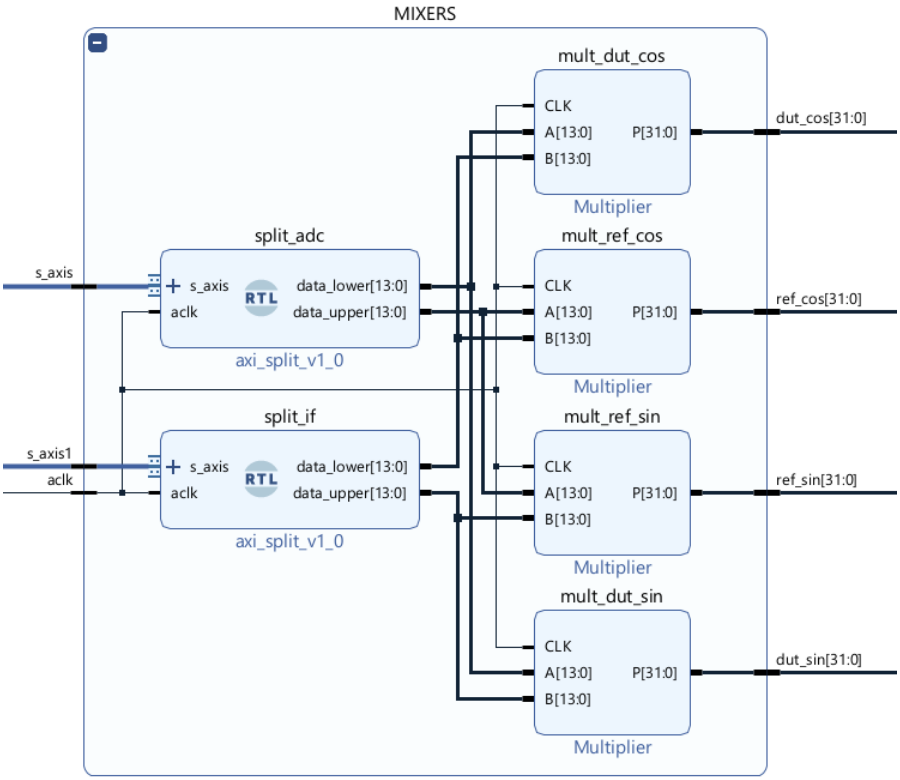

Figure B.3: The IQ decomposition stage implementation.

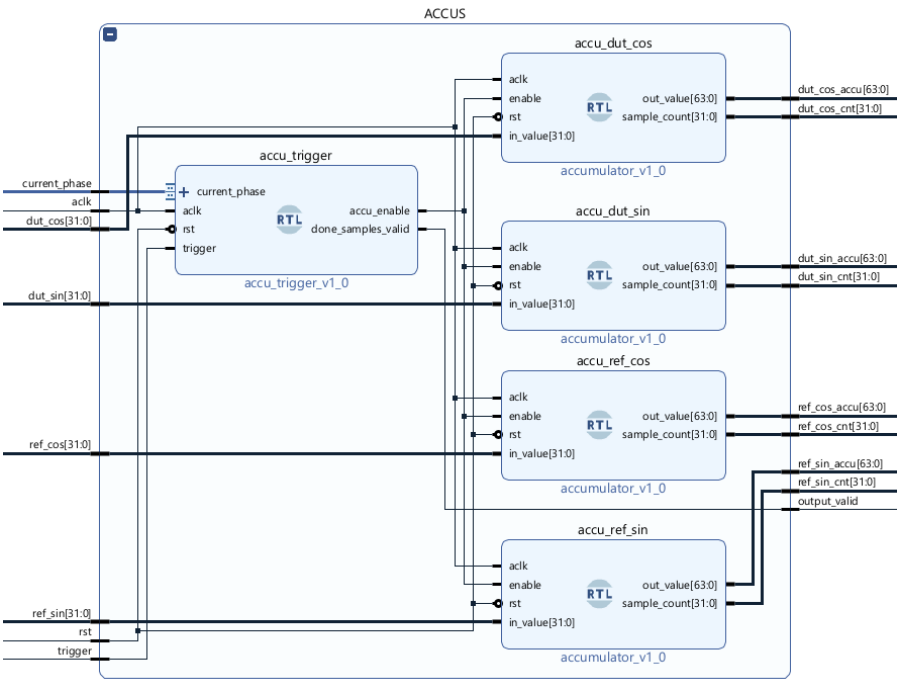

Figure B.4: The data reduction stage implementation.

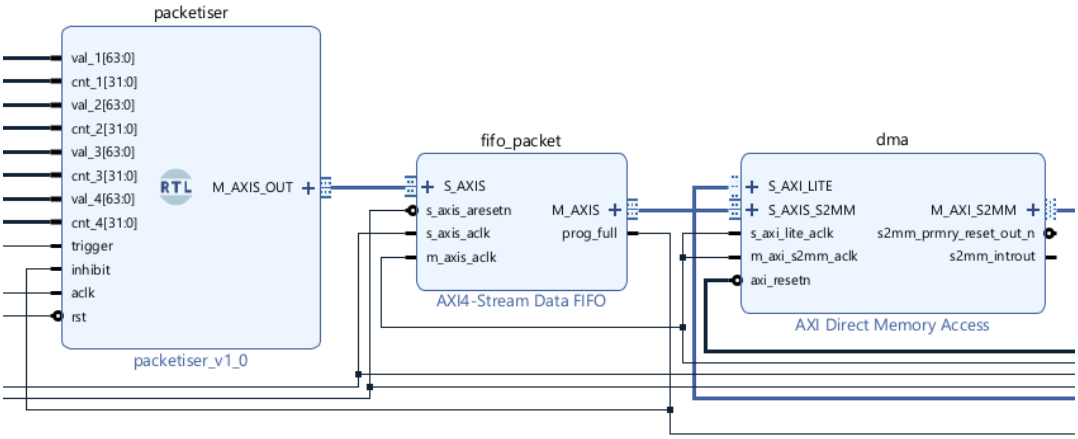

Figure B.5: The data output stage implementation.

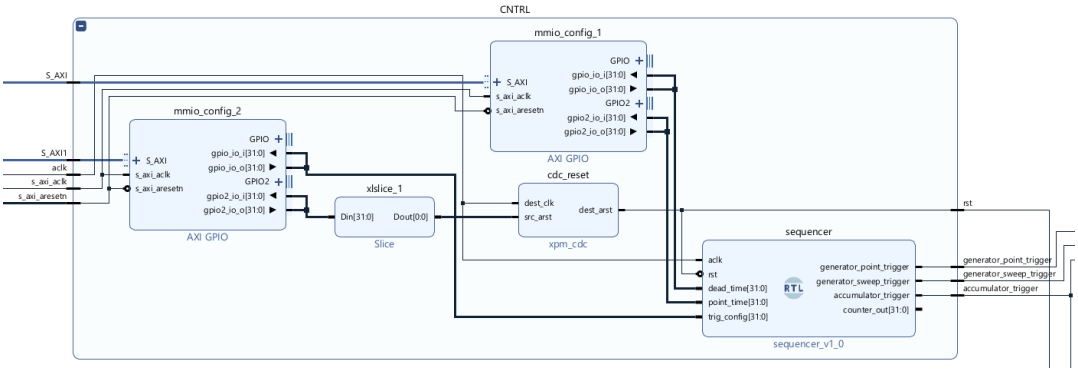

Figure B.6: The control section implementation.

C

Implementation Testing and Results

C.1. Testing parameters and code

```
import numpy as np
from pynq import MMIO, Overlay, allocate
import matplotlib.pyplot as plt
import time

# ----- NEXT CELL -----

RAW_TO_VOLTS = 2**-25   # Fixed point to floating point

def uint64_to_signed_int(unsigned):
    """Converts 64-bit unsigned integer to signed integer. By Bit Twiddling Hacks; see
    https://stackoverflow.com/questions/1375897/how-to-get-the-signed-integer-value-of-a-long-in-python.
    """
    unsigned &= (1<<64) - 1  # Keep only the lowest 64 bits.
    return (unsigned ^ (1<<63)) - (1<<63)  # Swap and shift down.

def buffer_to_volts(buffer):
    """Divides integer I and Q values by sample count. Buffer is an array containing a multiple of three
    elements: I value, Q value, count. The I and Q values are divided by count
    and multiplied by a conversion factor to get the unit of volts.
    """
    volts = [
        (
            uint64_to_signed_int(int(buffer[i])          # to Python integer(first entry: unsigned 32-bit integer
            + (int(buffer[i + 1]) << 32))                #     adding second unsigned integer shifted left 32 bits)
            / buffer[i + 2]                              # dividing by third entry (count)
            * RAW_TO_VOLTS                               # scaling to units of volts
        )
        for i in range(0, 12, 3)                         # i = 0, 3, 6, 9
    ]
    return volts

def volts_to_phasors(volts):
    """Interpret the 4 voltage values as phasors
    """
    dut = volts[0] + 1j*volts[1]
    ref = volts[2] + 1j*volts[3]
    rel = dut/ref
    return dut, ref, rel

# ----- NEXT CELL -----

# Set configuration bits using MMIO
```

```
44
45 ADC_FREQ = 125_000_000
46 cyc = lambda x: round(x*ADC_FREQ)
47 RESET_BIT = 0b1
48
49 # Trigger configuration, eg trigger1_conf = TRIG_POS+TRIG_SWEEP
50 TRIG_POS = 0b0000
51 TRIG_NEG = 0b0001
52 TRIG_SWEEP = 0b0010
53 TRIG_POINTS = 0b0100
54
55 mmio_dead_time = MMIO(0x42000000)       # Dead time in ADC samples
56 mmio_point_time = MMIO(0x42000008)      # Total point time in ADC samples (dead time +
       accumulation time)
57 mmio_trigger_conf = MMIO(0x41200000)    # Trigger config
58 mmio_general_conf = MMIO(0x41200008)    # General config
59
60 def set_config(dead_time = 300E-6, point_time = 1E-3, trigger_length = 10E-6, trigger1_conf =
       0, trigger2_conf = 0):
61 #     assert trigger_length <= dead_time <= point_time, "Trigger length should be less than
       settling time, less than total point time!"
62
63     mmio_dead_time.write(0, cyc(dead_time))              # Only one value in register,
           overwrite completely
64     mmio_point_time.write(0, cyc(point_time))            # Idem
65     mmio_trigger_conf.write(0, cyc(trigger_length) + (trigger1_conf << 24) + (trigger2_conf
           << 28))
66
67 def read_status():
68     return mmio_general_conf.read(0)
69
70 def start_acq(dma_recv):
71     curr = mmio_general_conf.read(0)
72     mmio_general_conf.write(0, curr | RESET_BIT)
73
74     # dma request of 16 words to get rid of misformed packet
75     buffer = allocate(shape=(16,), dtype=np.uint32)
76     dma_recv.transfer(buffer)
77     dma_recv.wait()
78     del buffer
79
80 def stop_acq():
81     curr = mmio_general_conf.read(0)
82     mmio_general_conf.write(0, curr & ~RESET_BIT)
83
84 # ----- NEXT CELL -----
85
86 # Program the overlay onto the PL, configure it and get a handle for the DMA
87
88 ol = Overlay("/home/xilinx/bit/vna_v1_7.bit")
89
90 dma = ol.dma
91 dma_recv = dma.recvchannel
92
93 # ----- NEXT CELL -----
94
95 # Allocate input buffer and show it's empty
96
97 data_size = 12
98 buffer = allocate(shape=(data_size,), dtype=np.uint32)
99
100 for i in range(data_size):
101     print(f'0x{format(buffer[i],␣"02x")},␣', end='')
102
103 # ----- NEXT CELL -----
104
105 # Transfer data and show it (can repeat indefinitely)
106
107 set_config(dead_time=250E-6, point_time=1E-3, trigger_length=10E-6, trigger1_conf=TRIG_POS+
       TRIG_SWEEP, trigger2_conf=TRIG_POS+TRIG_POINTS)
108
```

```
# Acquire data
start_acq(dma_recv)
dma_recv.transfer(buffer)
dma_recv.wait()
stop_acq()

# Convert it to human units
volts = buffer_to_volts(buffer)
dut, ref, rel = volts_to_phasors(volts)

# Print results
for i in range(data_size):
    print(f'0x{format(buffer[i], "02x")}, ', end='')
print()
print(volts)
print(f'DUT: {np.abs(dut):.3f}V, {np.angle(dut):.3f}r, ' +
      f'REF: {np.abs(ref):.3f}V, {np.angle(ref):.3f}r, ' +
      f'REL: {np.abs(rel):.3f}, {np.angle(rel):.3f}r')

# ----- NEXT CELL -----

from pickle import Pickler

TESTS = [(1000E-6, 900E-6, 1000, "2vpp 0d, 2vpp 0d"),(1000E-6, 0.5E-6, 1000, "idem"),
         (1000E-6, 900E-6, 1000, "1vpp 0d, 2vpp 90d"),(1000E-6, 0.5E-6, 1000, "idem"),
         (1000E-6, 900E-6, 1000, "0.1-1vpp 3hz 0d, 2vpp -90-90d 5hz"),(1000E-6, 0.5E-6, 1000,
              "idem"),
        ]

results = []

for TIME, DEADTIME, POINTS, INSTRUCTIONS in TESTS:
    input(INSTRUCTIONS)

    set_config(dead_time=DEADTIME, point_time=TIME, trigger_length=0.5E-6, trigger1_conf=
        TRIG_NEG+TRIG_SWEEP, trigger2_conf=TRIG_POS+TRIG_POINTS)
    values = np.ndarray((POINTS, data_size), dtype=np.uint32)

    start_acq(dma_recv)
    start = time.perf_counter()
    for i in range(POINTS):
        dma_recv.transfer(buffer)
        dma_recv.wait()
        values[i] = buffer
    end = time.perf_counter()
    stop_acq()
    print(f"Took {end-start:.4f}s")

    # Calculate after the loop to allow faster IFBWs (above 2k instead of barely 1k)
    all_values = np.ndarray((POINTS, 3), dtype=complex)
    for i, buf in enumerate(values):
        volts = buffer_to_volts(buf)
        all_values[i, :] = volts_to_phasors(volts)

    results.append(all_values)

Pickler(open("thesis_measurements.pickle", "wb")).dump(("time per point, deadtime, points, 
    instructions", TESTS, results))
```

C.2. Test results

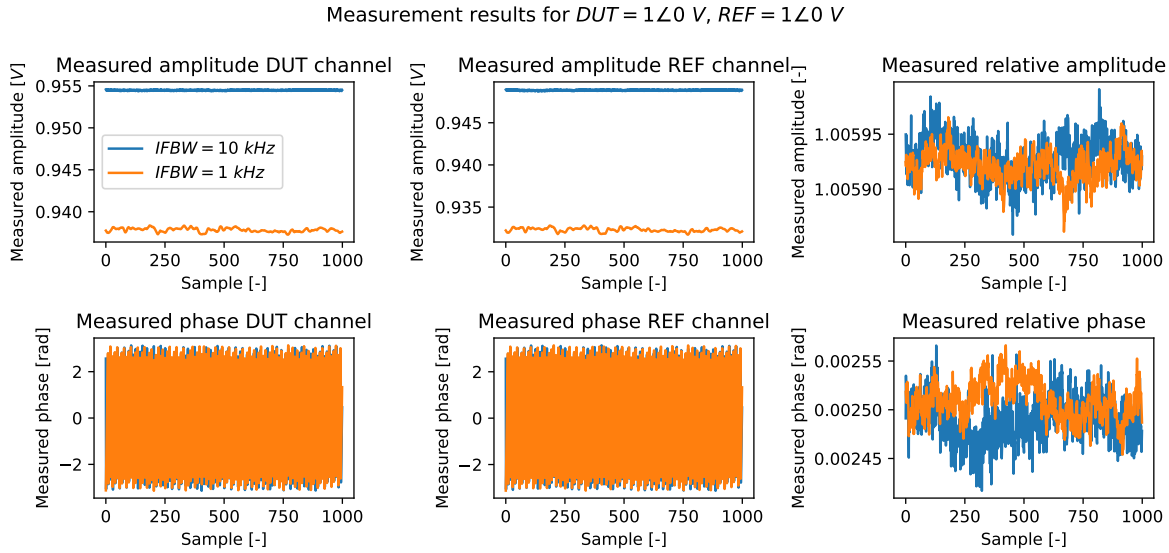

Figure C.1: Results of experiment 1. Mind the variable vertical scales.

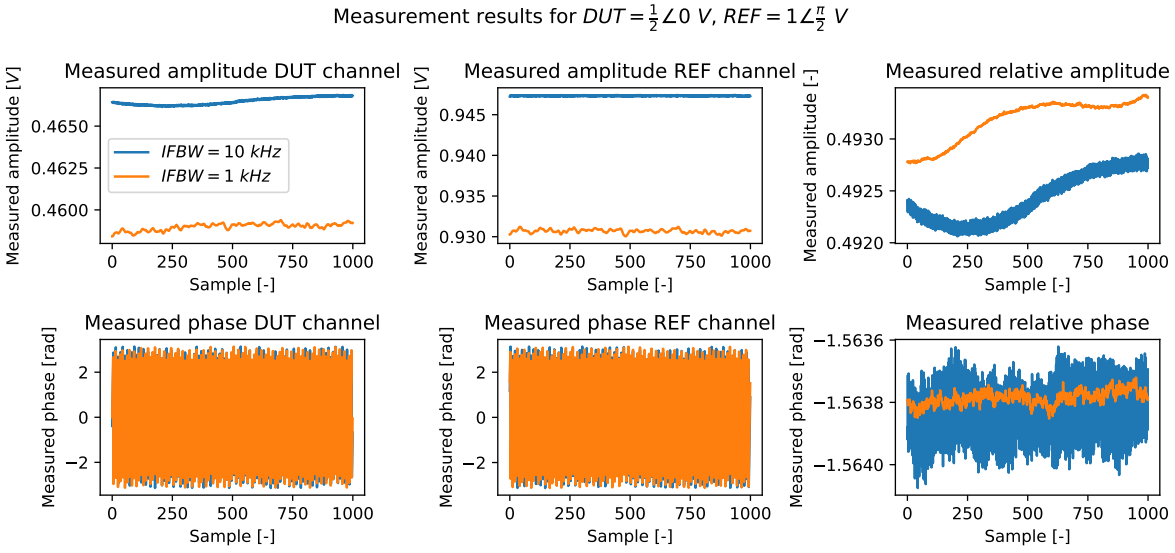

Figure C.2: Results of experiment 2. Mind the variable vertical scales.

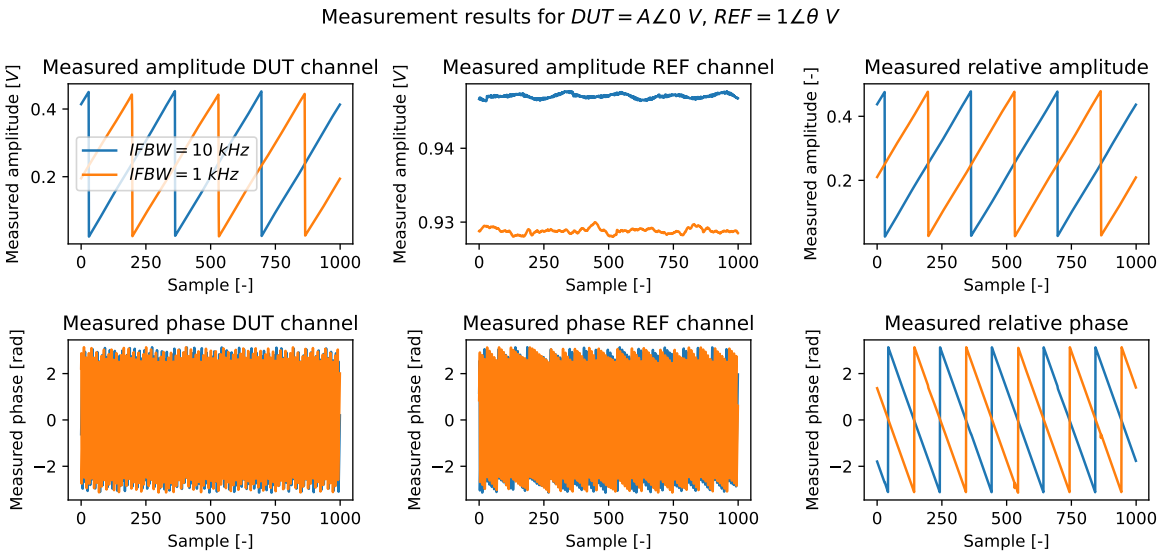

Figure C.3: Results of experiment 3. A and θ were modulated using positive-ramp sawtooth waves. Mind the variable vertical scales.